Internet of Things and Health: SmartWalker

A Major Qualifying Project submitted to the Faculty of Worcester Polytechnic Institute in partial fulfillment of the requirements for the degree in Bachelor of Science in Electrical and Computer Engineering

Submitted by:
Tyler Ducharme
Brian Mitchell
Justin Trott

Submitted to:
Professor Xinming Huang, Advisor
Professor Edward Clancy, Co-Advisor

Abstract

The Internet of Things (IoT) is an emerging field of technology that utilizes sensors and wireless data communication to allow for improvements in quality and usability of conventional devices. The healthcare field is a dynamic environment that can greatly benefit from the inclusion of IoT devices and technologies. Two specific recurring problems in the healthcare field are those of wandering and preventative falls that often occur in the Alzheimer's and Dementia population. In this project, an IoT device was designed and created to replace a conventional walker, named the SmartWalker, which attempts to prevent these problems from occurring by monitoring a patient's walking behavior and location. The intent of the SmartWalker is to alert a nearby caregiver if a patient is exhibiting behaviors that lead to falls, so that the care provider can intervene and prevent a fall from occurring altogether. The SmartWalker integrates various sensors and Bluetooth Low Energy (BLE) communication to successfully achieve this task.

Acknowledgements

The completion of this project was made possible by Professor Xinming Huang, PhD for providing the project with a direction and a focus as well as technical experience, Professor Edward Clancy, PhD for providing great technical experience, and William Appleyard for his help in obtaining the resources needed for construction of the project. Finally, we would like to express our gratitude to the WPI Electrical and Computer Engineering Department for financing the project as well as providing us with the wonderful opportunity to be creative without boundaries.

Table of Contents

Table of Figures

1. Introduction

A growing problem in the United States is the growth of Alzheimer’s disease and other forms of Dementia within the elderly population. Currently, more than 5 million Americans live with Alzheimer’s disease. Alzheimer’s disease is also the 6th leading cause of death in the U.S. [1]. Alzheimer’s Patients tend to be aged 65 and older, meaning that there are likely a significant number of patients who use assistive mobility devices such as walkers, canes, or wheelchairs.

The use of these devices involves the risk of a patient falling over due to factors related the disease itself, medications, or even improper use. Often times, these problems are allayed by having the patient reside at an assisted living facility or nursing home, where the patients are under constant supervision for preventative reasons. Alternatively, Alzheimer’s patients have the option of receiving in-home care via a nurse or other certified caretaker to assist in easing symptoms and watching over patients. However, fall prevention measures are more difficult to adopt in this type of circumstance. Fall prevention in an in-home setting could be greatly improved with the use of the latest technology.

In recent years, the wireless communication industry has grown immensely. Improvements in the technology as a whole, especially in power consumption, and lower costs of Bluetooth and Wi-Fi enabled products have allowed the Internet of Things (IoT) to flourish. The IoT is defined as the network of physical devices including but not limited to motor vehicles, home appliances, mobile sensors, and healthcare devices that are enabled to send and receive data wirelessly. As this network has grown, so have the potential applications that can utilize its features.

The goal of this project was to create an IoT device that can simultaneously improve task efficiency of an in-home caretaker and promote a safer environment for their patients. To do so, the following design constraints were considered:

1. The device must be an all-in-one assistive mobility device.
2. The device must utilize a number of sensors that can measure the heart rate of a patient, monitor the patient's movement performance, and track the patient's location.
3. The device must be able to collect data discreetly, as the patient should not be able to alter the performance of the sensors.
4. The device must be able to wirelessly connect to a smartphone and transmit the sensor data which are then processed and utilized by an application which will serve as the user interface.
5. The device must operate using a battery and at a low power setting to ensure long battery life.

In order to meet these constraints, this project explored the concepts of IoT, the effects of Alzheimer's disease, and the risks of unstable walking patterns within the elderly population, which ultimately resulted in a device that we coined the "SmartWalker". The SmartWalker utilizes a number of sensors attached to an assistive mobility walker in order to monitor the overall health, performance, and location of an elderly patient under the care of an on-site caretaker. These sensors include a heart rate monitor to sense irregularities in the patient's pulse, an accelerometer to sense dangerous motion of the device, pressure sensors to monitor the hand grip of the patient, and a GPS unit to track the patient's location. The data collected from the device's sensors are then processed using an Arduino Uno microcontroller and transmitted to an

Android smartphone, operated by the caretaker, via Bluetooth Low Energy. The data is then utilized by an Android application which alerts the caretaker if the patient has irregular heart activity, irregular movement such as falling, or has moved out of a designated area of set GPS coordinates. The device will be powered by a rechargeable battery.

The following Background section discusses the process of creating the SmartWalker, beginning with an exploration of health concerns in the elderly community, previous solutions implemented to meet the health needs of the elderly community, and important considerations regarding the protocol and requirements of in-home caretaker services and nursing homes. Next, the Methodology section explains in detail the overall system design and its motivation; this includes the physical placement of sensors, circuit design, processing algorithms, and application and user interface design. Each of these design choices are then evaluated after comprehensive testing and data collection described in the Results chapter. Lastly, conclusions and recommendations are presented in order to improve the design in a future solution and further the concepts explored by such a design.

2 Background

According to the 2015 United States census, people of age 65 and older accounted for 14.9 percent of the total US population. This demographic is projected to increase significantly by 2060, comprising nearly one in four US residents [2]. A survey conducted by the Centers for Disease Control and Prevention (CDC) states that the 65 years and over demographic has the highest percentage of respondents reporting fair or poor health at 21.8 percent [3]. As a result, over 2.5 million people in the United States receive long-term care services from adult day services, nursing homes, or residential care communities [4].

2.1 Common Issues Prevalent within the Elderly Community

Wandering

Wandering is a problem that affects many people within the elderly community, especially those suffering from Dementia. The Alzheimer's Association states that six in ten people with Dementia may wander [5]. People with memory problems and the ability to move around can become disoriented or confused by their surroundings, leading them to wander. This can be especially dangerous if the person becomes lost and is away from their caregiver.

Heart health

One's heart health generally degrades with age which can cause many issues for older people, some of which can be fatal. Heart disease is the leading cause of death among persons aged 65 and older in the United States [6]. An aging heart, when worked harder than normal, may not be able to sufficiently pump enough blood to supply all parts of the body. This will cause a number of problems, including angina, abnormal heart rhythms, congestive heart failure, and strokes [7].

Reduced Level of Mobility

Reduced level of mobility is a common issue reported by people aged 65 and older and is characterized by an inability to walk as far or as long as the person was previously able, without experiencing fatigue, aches, and/or pains. Several health conditions can contribute to a reduced level of mobility; some examples of which are strokes, arthritis, heart disease, and osteoporosis. However, some elderly people experience reduced mobility without any pre-existing condition [8]. According to a survey done by the CDC, there are 18.2 million adults in the United States that are unable or find it very difficult to walk a quarter mile [9].

Falling

Falling is a serious problem that affects millions of people age 65 and older each year. The Centers for Disease Control and Prevention states that more than one in four older people falls each year, though less than half inform their doctor. While most falls are innocuous, it is estimated that one fifth of falls cause a serious injury such as a broken bone or a traumatic head injury. As a result, 2.8 million older people are treated in emergency departments for fall related injuries each year. The elderly demographic is more susceptible to falls due to their predisposition to lower body weakness, difficulty with balance, and vision problems [10].

The problems outlined above can become exacerbated simply by a delayed response from a caretaker. This can occur when a caretaker does not have immediate knowledge of a situation, or has their attention drawn elsewhere, especially in a nursing home environment.

2.2 Previous Solutions

There have been several solutions to the common issues addressed above already in the market. While simple solutions to individual problems are fairly common, it is clear that there lacks a solution that can effectively detect multiple issues autonomously. For the purpose of research, we have analyzed several of these solutions and outlined their purpose, usage, and effectiveness below.

<u>Personal Emergency Response Systems</u>

These are basically alarm devices (as shown in Figure 1) meant to be worn on person and generally have a single button that, when pressed, will wirelessly connect the user to an operator at the company's command center who will assess the situation and call for help.

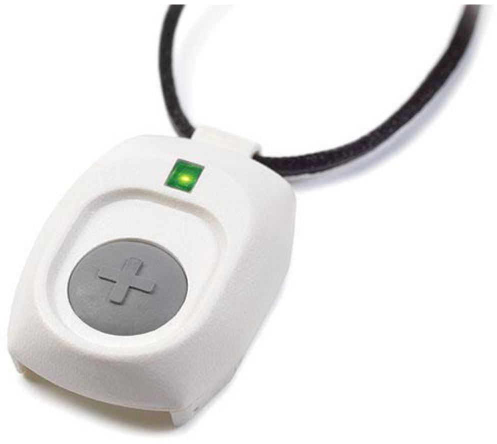


Figure 1: Personal Emergency Response Device

This solution is helpful for older people who live alone and helps decrease response time for falls and other medical emergencies where the user is still capable of pressing the button. These types of products are often not capable of predicting when an emergency had occurred, needing user input in order to function. Further, these products tend to require a subscription payment that can be cost preventative for an older demographic.

<u>Walking Assistance Devices</u>

Devices such as canes, walkers, and wheelchairs are designed to help people who have trouble walking. There are different types of walking assistance devices for different levels of mobility, however this introduces the possibility of people using the wrong device based on their needs. This can lead to lowered effectiveness – if the device used is meant for a higher level of mobility than the user is capable of – or further ability degradation – if the device used is meant for a lower level of mobility than that of the user. The latter is a common issue in assisted living situations, where caregivers often place patients that could use canes or walkers in wheelchairs because of the reduced fall risk (Gavin-Dreschnack, D & Volicer, Ladislav & Morris, C. (2010). Prevention of overuse of wheelchairs in nursing homes. Annals of Long-Term Care. 18. 34-38.).

On Person Trackers

Such devices (as shown in Figure 2) utilize GPS in order to keep track of the location of people who have a tendency to wander. These products are worn by the patients will send location alerts to caregivers either periodically or after the user has left a preset boundary.

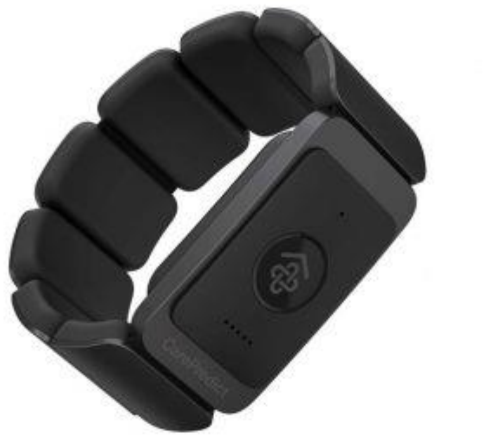


Figure 2: On-Person Tracker Device

A common issue with this type of product is that users, especially those with Alzheimer's or other forms of Dementia, will often remove the tracking device from their wrists because they do not know what it is. Some products disguise the tracking device in an attempt to combat this issue.

2.3 Important Considerations

In order to ensure that our solution will be appropriate for a possible implementation in long-term care centers, there are several considerations that must be made for the safety of our product's users. These considerations are outlined below.

Patient Confidentiality

The healthcare industry maintains a very important obligation towards patient confidentiality. A patient's personal health information must be kept private unless consent is provided by the patient. In order to ensure that the privacy of our product's users is not infringed upon, there are several methods we can use. First, the product cannot provide any numerical data to the connected phone, only providing to the caregiver whether or not an emergency is currently taking place. Secondly, our product can delete stored data when it is no longer needed. Not storing data for long periods of time can significantly decrease the possibility of a user's privacy being violated.

Ease of Operations

The operation of the device must be ubiquitous such that users at any level of technology awareness can easily handle it, for both patients and caregivers. In order to achieve this, our design should be naturally integrated with a device or apparatus that the patient already uses, such as a walker. The product should also be able to run autonomously without requiring any input from the patient. This will eliminate any learning curve for the patients as our product will be functionally identical to that of a walker, which they already use. For the caregiver, the smartphone app should make Bluetooth connection to nearby walkers intuitive, have a graphical interface that clearly and concisely displays the status of a connected walker, and push notifications to the smartphone whenever a problem is detected by a connected walker – even when the app is not currently open.

Cost Effectiveness

The device must be affordable. Our target demographics of caregivers and assisted living centers will not be able to afford to purchase expensive devices for each elderly person within their care. In order to keep the price of our product low, the final product should have all of its main components streamlined to a single printed circuit board (PCB). Unfortunately this is beyond the scope of the prototyping done within this project and our prototype will make use of microcontroller boards that include circuitry that would be unnecessary for a final product.

3. Methodology

3.1 SmartWalker Design Overview

The goal of this project was to develop an IoT device that assists patients or elderly people who use a walker. The device is intended to monitor various aspects of the user's health and usage of the walker through several sensors, and report the status of the person to the caregiver's smartphone app via Bluetooth Low Energy (BLE).

As mentioned in Section 2.1, many elderly people experience preventable falls while using a walker due to their conditions, some of which can be deadly. To address this issue, our device is aimed to detect various signals of a person's behavior that can lead to falling and alert someone – specifically a caregiver – that the person may need assistance. Alternatively, in a worst case scenario, alert a nearby person or emergency number if a fall does occur. To detect a preventable fall before it happens, several aspects of the experience using a walker were considered, and sensors were chosen to monitor these aspects. As expanded on in Sections 3.2.2 through 3.2.6 (Sensors), the sensors chosen for this task are pressure sensors to detect irregular pressure placed on the handles of the walker, an accelerometer to detect irregular movement patterns of the walker, and a heartbeat sensor that measures irregularities in the user's heart rate. The heart rate is measured through two nodes each placed on a handle – similar to the systems used in treadmills and other cardio workout equipment. These sensors are expected to be sufficient in detecting patterns that occur before a person falls.

The device samples data from each of these sensors while the walker is in use and processes on a central processing unit (CPU) which determines whether a fall is likely to happen or has already happened. This unit has Bluetooth Low Energy capabilities and will further export the results via BLE to be analyzed by an app running on the smartphone of a nearby caregiver.

Depending on the settings of the app, an alert will be sent to the phone to notify the care provider, or even automatically alert an emergency care system similar to Personal Emergency Response Systems (Section 2.2).

Another problem that the SmartWalker device addresses is wandering by patients with Alzheimer's disease, or any other form of Dementia. This problem is described in further detail in Section 2.1. In order to detect wandering, the SmartWalker is equipped with a GPS module that is able to detect whether or not the walker is in use within a specified range. In a case where the walker is detected outside of the specified range, the app installed on the patient's own smartphone will be able to send a message to a primary caregiver alerting them of the patient's location. However, this functionality is not currently implemented in the prototype SmartWalker because it does not fall within the scope of this project.

This section aims to provide an explanation of the components we used, the strategies we employed, and the testing methods we utilized in order to create our version of such a SmartWalker device.

3.1.1 Initial Design

The original design of the device included the use of Texas Instruments' CC2650 programmable microprocessor and Launchpad (Figure 3).

Figure 3: TI CC2650 Launchpad

This chip was intended to serve as the processing unit and BLE transmission node because of its built in BLE integration. This chip is likely a superior component for the device due to its low power usage and integrated BLE capabilities.

Unfortunately, the CC2650 is one of the newest processors in Texas Instruments' RF communications line, and therefore the useful and relevant guides, documentation, and resources for its use and programming are difficult to find online, and some of its capabilities do not currently have fully working drivers available (for example, the CC2650 does not have working I2C drivers, which was needed for this design). As a result, we discontinued our work with this chip during the course of the project, as it required greater expertise in embedded software development, and demanded considerably more time to implement into the project than we had available.

3.1.2 Modified Design

As we moved on from using the TI CC2650, we had to take the deadline of the project into consideration, and therefore decided to continue our work with the simpler Arduino Uno platform. This board is simpler to use and easier to implement into projects than the TI CC2650, but unfortunately that simplicity comes with a trade-off. The Arduino Uno does not possess innate low power or BLE capabilities. The Arduino Uno is a great board under certain circumstances and for certain applications, but we believe it is not the optimal board for the implementation of the SmartWalker device. However, due to time and resource constraints, we concluded that for the purpose of prototyping our SmartWalker, the Arduino Uno was ideal, but for future designs or modifications to the SmartWalker device, the TI CC2650 should be considered as a superior option.

3.1.3 Bluetooth Low Energy Implementation

In order to implement the transfer of data from the Arduino Uno to an app running on an Android smartphone device, a BLE environment between these devices was created. In a typical BLE system, one or multiple nodes are considered Peripherals, denoted as such by their ability to advertise a connection and provide a GATT profile, but also by their inability to request a connection to another node in the system. Peripherals in a BLE system are equivalent to slave devices in serial communication protocols. The other type of node relevant to our implementation is a Central node. Central devices are also able to maintain a GATT profile, but are also able to initiate a connection between itself and another node, as well as request data from another nodes characteristics. Central nodes are equivalent to master nodes in serial communication protocols. In this project’s design, an Arduino Uno serves as a peripheral device which collects, processes, and aggregates the sensor data into a characteristic, and an Android smartphone device serves as a central device which connects to the peripheral and requests the data stored in the characteristic.

The Arduino BLE library, RBL_nRF8001, made for convenient use with the RedBear BLE shield (see Section 3.2.7), automatically creates a BLE peripheral service with a characteristic having Write/Notify functionality. Additionally, the library provides code functions that allow updating the data bytes written to the characteristic, as well as code functions that read the data written to the characteristic by the Central device. For the purpose of this project, the ‘notify’ aspect of this characteristic was greatly utilized, and the ‘write’ aspect was not utilized for the project’s current prototype state. This allowed a very simple implementation of a BLE environment, where the sensor data is aggregated and encoded into a single characteristic, and sent via BLE communication to the Android Device, which decodes the data and notifies the phone user if necessary. A representation of this strategy is shown below in Figure 4.

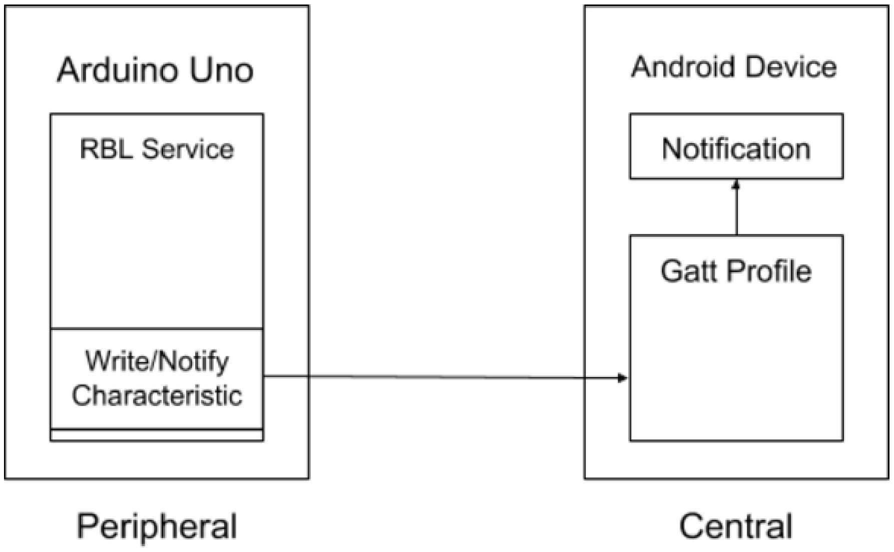


Figure 4: SmartWalker BLE data path

We chose to use a one-characteristic implementation instead of a multiple-characteristic (assuming one characteristic per sensor) implementation, for multiple reasons. One major reason being that having only one characteristic allows the Android app to only be required to decode and analyze one data value instead of reading and processing multiple values. We made this decision due to the fact that none of the members of this project team have a strong foundation in software development, and a more robust Android app can be an improvement made by a team with more software background in the future. The other reason we chose a one-characteristic implementation is that the Arduino library we use (the RBL_nRF8001 library) easily supports a one-characteristic model, while using it to create a multi-characteristic environment would be markedly more complex and time-consuming.

3.2 Hardware

The intention of this section is to give an overview of each individual hardware component used in the prototype. Each component is analyzed based on several aspects, including its required purpose within the prototype, technical specifications and limitations, the overhead of its implementation in code, and possible improvements that could be made.

3.2.1 Arduino Uno

The Arduino Uno, briefly introduced in Section 3.1.2 and displayed below in Figure 5, is a general purpose microcontroller popular for its ease of use within personal "do-it-yourself" style projects.

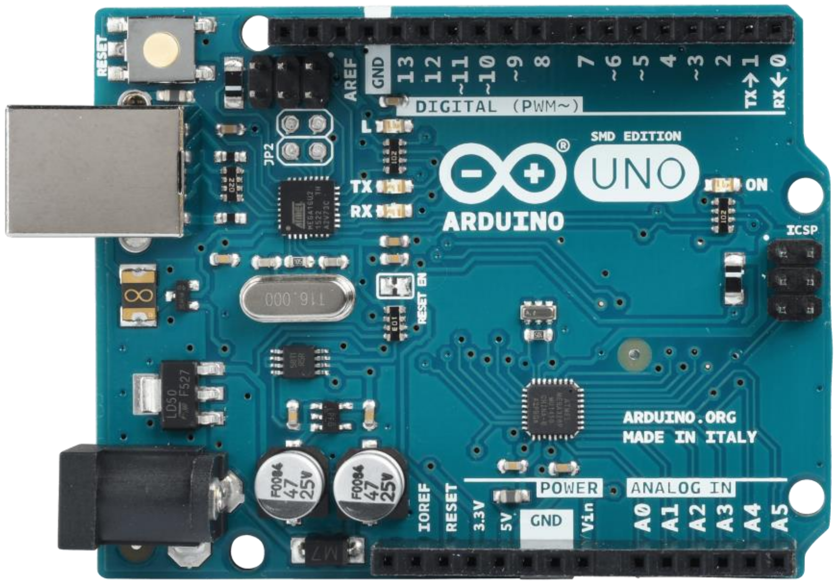


Figure 5: Arduino Uno

It lends itself to this reputation by having many usable I/O pins with various purposes, which are easy to attach to peripherals and immediately utilize and test within a project. It also has a distinct coding environment in which the controller's setup and optimizations are concealed from the user, and the user is given an interface in which it is easy to modify the controller's functionality and upload code to the board for testing.

The Arduino Uno contains several pins that were specifically relevant in the context of this project; multiple analog pins that are connected to a five channel multiplexed 16-bit analog-to-digital converter, serial data pins that support I2C and SPI communication protocols, serial data pins that support UART communication protocols, and both 3.3V and 5V power outputs to support peripherals of both power levels. The use of each of these functionalities will be explained in the following sections about each sensor (Sections 3.2.2-3.2.6), but it is due to these capabilities (and other reasons described in Section 3.1.1-3.1.2) that the Arduino Uno was chosen as the microcontroller to be used in the SmartWalker prototype.

The processor used by the Arduino Uno (the ATmega328) is an 8-bit CPU, has a 16MHz clock rate, 2KB SRAM, and 32KB flash storage. The coding environment uses a C based language with the board's initialization and setup being handled largely by the processor itself, or the included libraries as opposed to being customizable by the user.

There are two particular downsides to using this processor in the SmartWalker prototype. The first being the 16 MHz clock rate, which ultimately translates to an analog sample rate of only 9600 Hz. This sample rate is adequate for the purpose of the prototype, but a higher clock speed (i.e. the 48 MHz speed possible with the TI CC2650) would lend itself to higher sample speed and therefore greater accuracy of hazard detection. The second downside is the Arduino Uno's lack of inherent low-power capabilities. This downside is much more impactful because once the Arduino is powered on, it will be running its routines, constantly drawing approximately a 25mA current. The actual processing done by the Arduino is not intensive, meaning that between uses of the SmartWalker or even between sampling periods, the processor should be able to enter a low power mode and conserve energy. These are the tradeoffs that were made to ensure ease of use during the prototype stage of this project, and a more robust processor should likely be used in any future improvements made to the SmartWalker.

3.2.2 AD8232 Single Lead Heart Rate Monitor

After researching a number of different instrumentation amplifiers, we decided to use Analog Device's AD8232 single-lead, heart rate monitor integrated circuit for our pulse detection application.

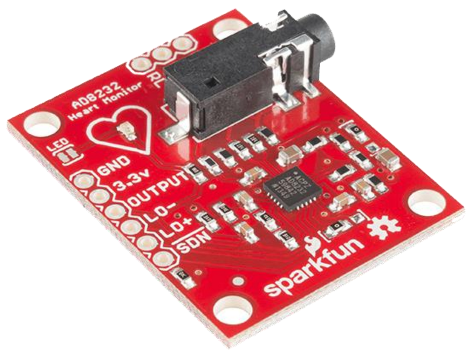


Figure 6: AD8232 Heart Rate Monitor with Sparkfun Breakout

According to the device datasheet, the AD8232 supports two or three electrode configurations, has a high signal gain of 100, and can be powered between 2.0V and 3.5V. For our application, we needed a system with two sensing electrodes via hand grips, a high gain in order to be able to process the heart rate signal, and a low powered device in order to standardize supply voltages between other devices needed for the SmartWalker. Shown below is the functional block diagram of the AD8232:

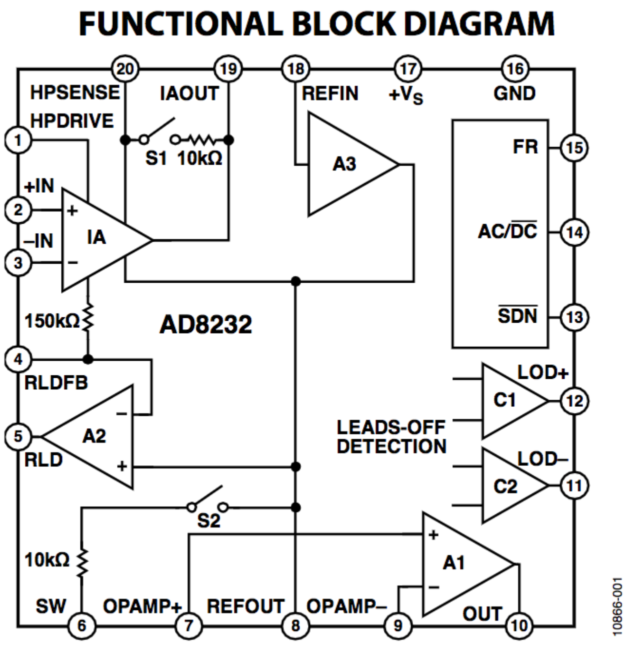


Figure 7: AD8232 Block Diagram

The signal path begins at nodes 2 and 3, where the left hand is connected to the positive input to the instrumentation amplifier (IA) labeled node 2, and the right hand is connected to the inverting input labeled node 3. For our application, we decided to leave node 5, the right leg drive input of A2, disconnected. The right leg drive inverts the common-mode signal at the input of the instrumentation amplifier and injects its output current into the user in order to improve common-mode rejection of the system. The resulting inverting amplifier output signal is less noisy due to rejection of common-mode voltages, typically a requirement for capturing Electrocardiogram signals.

In order for the SmartWalker to measure the user's heart rate discreetly, the electrodes cannot be physically attached to the user directly. Additionally, our goal was not to capture ECG, but to instead count the peaks of the user's heart rate signal and convert them to a beats per minute measurement. Conveniently, the right leg drive is connected to the right leg drive feedback input (node 4) on the Sparkfun breakout, allowing us to leave it disconnected with no penalty. The Sparkfun breakout board also includes an internally connected low-pass filter configured using the internal general purpose op-amp (A1) included in the AD8232 IC. The board then outputs the analog signal where it can then be processed by our Arduino microcontroller. The basic processing algorithm involved setting threshold values in order to detect heartbeat peak values, calculating BPM from processing the peaks, and deciding if the BPM is too high or too low which then sends a flag to the Android App.

3.2.3 Hand Held Heart Rate Grips

In order to create a two-node ECG circuit for the purpose of measuring heart rate, two conducting nodes needed to be placed on the SmartWalker in locations that the user is required to make contact with. Placing the nodes on the handles was the most logical choice, and Hand Held Heart Rate (HHHR) Grips were chosen to act as those nodes. The HHHR Grips, shown below in Figures 8 and 9, are very common components used in modern exercise equipment.

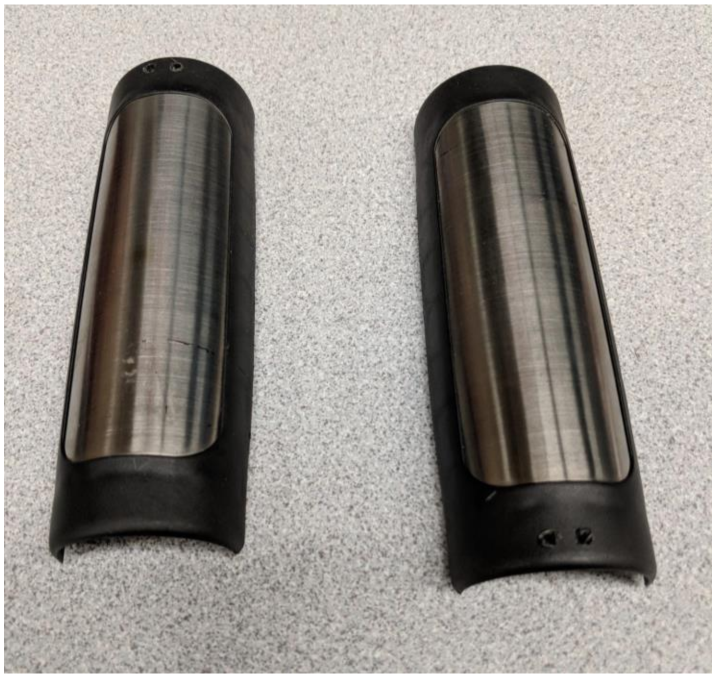


Figure 8: HHHR Grips

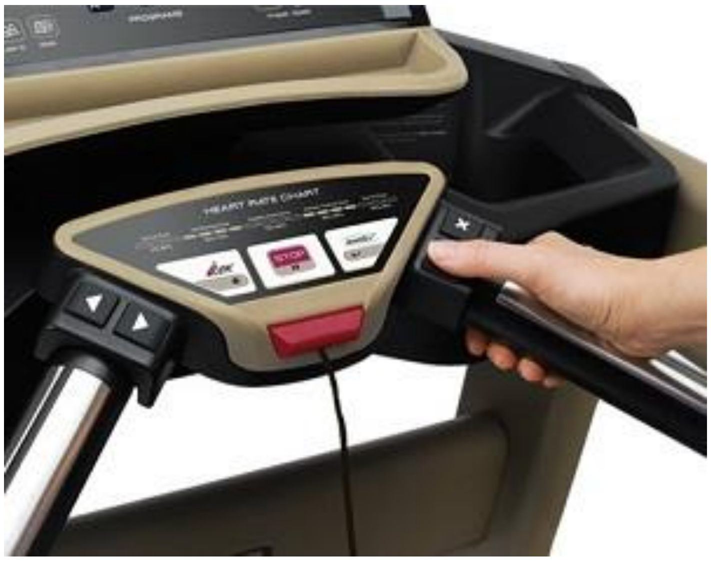


Figure 9: HHHR Grips on the handles of a treadmill

The HHHR Grips are convenient to use as nodes because they easily conform to the shape of the walker's handles, and the simplicity of soldering the leads of the ECG circuit to the underside of the grips. One downside of using this method is that using only two leads to measure a heart rate is less accurate than using several leads. Due to this, the ECG circuit is only able to measure heart rate, and unable to further measure or process any other aspects of the signal obtained by the circuit.

3.2.4 Force Sensitive Resistors

To detect irregular pressure placed on the handles of the SmartWalker, Force Sensitive Resistors (FSRs) are used (shown in Figure 10 below). The FSRs chosen for the project are 0.5 inches in diameter, and therefore it is possible to fit two FSRs on each handle of the walker.

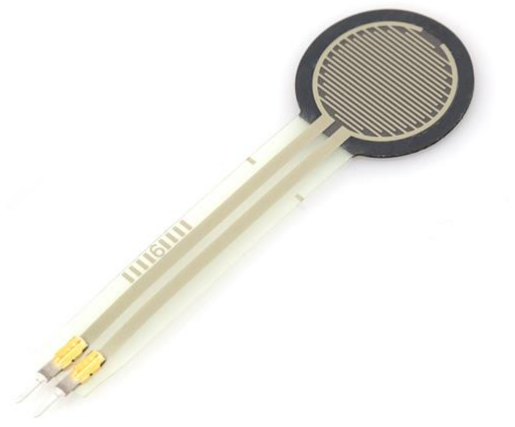


Figure 10: 0.5" Force Sensitive Resistor

With no pressure applied to the surface, an FSR acts as an open circuit, and theoretically acts as a 5kΩ resistor when about 100g of pressure is applied. The resistance further decreases as more pressure is applied in a logarithmic pattern as shown in Figure 11 below.

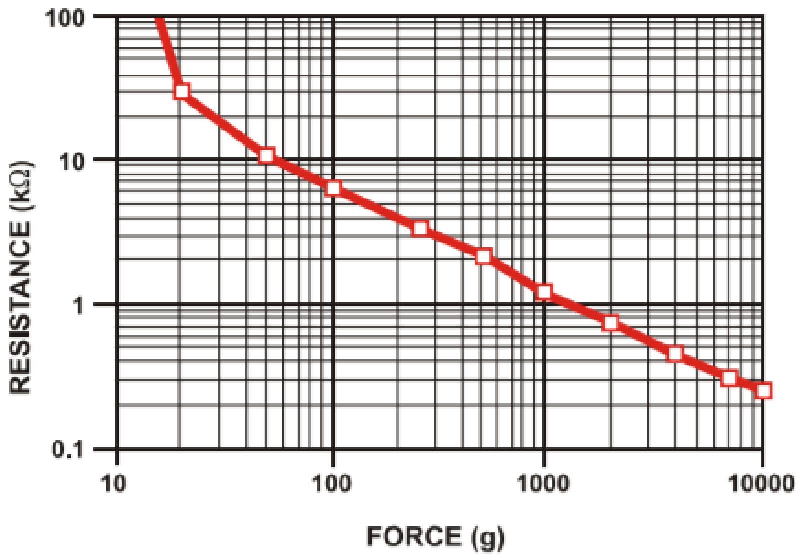


Figure 11: FSR Resistance vs. Force plot

However, this graph is closer to an approximate model rather than a precise one. Due to the FSRs' form factor and convenient nature, the force to resistance translation is very imprecise, meaning the FSRs work better in a detection method (i.e. is there pressure being applied or not) rather than actually measuring force. This does not contradict the purpose of the component on the SmartWalker, because the walker only needs to detect irregular pressure distributions between two handles. To implement this, two FSRs are used in a single voltage divider circuit with a 12 kΩ resistor that is used for a single handle on the walker. A circuit diagram of this voltage divider circuit is shown below in Figure 12.

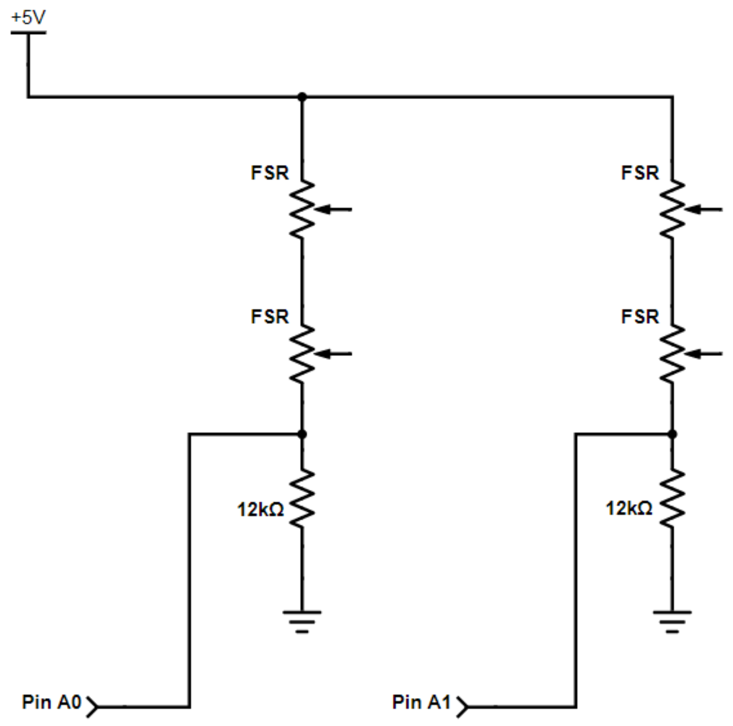


Figure 12: FSR voltage divider circuit diagram

This implementation works because with only a single FSR in the voltage divider circuit, the output voltage is too imprecise, making the output voltage only 0v with no pressure applied, or 5v with some pressure applied. With two FSRs in the voltage divider, and the pressure spread across both components, the voltage output better represents the pressure applied.

The output of each voltage divider circuit is connected to an ADC channel on the Arduino Uno. This way, the Arduino is able to sample each voltage, and then process the values to determine if the pressure applied to each handle is uneven enough to cause an alert. The processing that decides these values is further explained in Section 3.3.1 (Code).

It may seem useful to use a different component that has greater accuracy to sense pressure on the handles of the SmartWalker. However, the FSRs perform the required task and are simple to integrate into the device, therefore making them our preferred component to measure pressure.

3.2.5 ADXL345 Accelerometer with Adafruit Breakout

The Adafruit ADXL345 Accelerometer breakout component is perfect for measuring the acceleration of the SmartWalker.

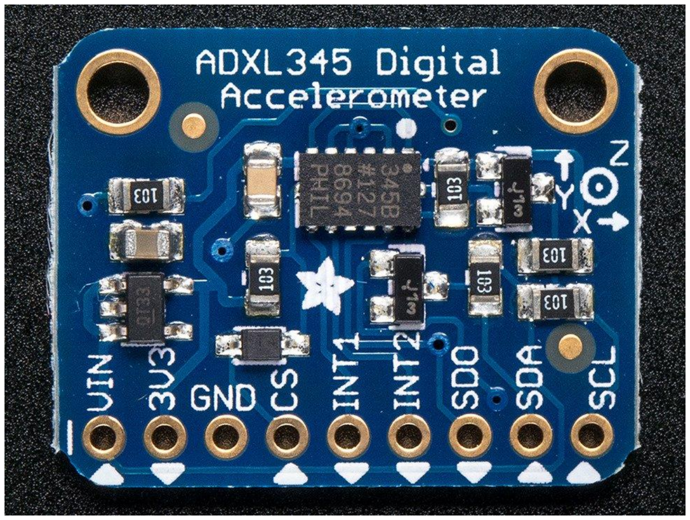


Figure 13: ADXL345 Triple-Axis Accelerometer

The device is a triple-axis accelerometer with various resolution settings. The breakout includes a 3.3V voltage regulator, therefore supporting both 3V and 5V outputs of the Arduino Uno. Also, the ADXL345 supports both I2C and SPI digital serial output protocols. The SmartWalker integrates this component using I2C communication, sampling the current acceleration once per iteration, and determining whether the acceleration is within safe ranges. It could prove useful to also include a triple-axis gyroscope to the motion sensing aspect of the SmartWalker, however we decided that the inclusion of a triple-axis accelerometer is sufficient for the purpose of prototyping the device.

3.2.6 Adafruit Ultimate GPS with Adafruit Breakout

To monitor the GPS coordinates of the SmartWalker, the FGPMMOPA6H (Adafruit Ultimate GPS) was chosen, as it is able to collect coordinate data as well as communicate its data via a serial UART connection.

Figure 14: FGPMMOPA6H GPS with Adafruit Breakout

Although it is possible to communicate to multiple slave devices on the same I2C/SPI master data line, we decided to use a component that communicates via UART to be able to simultaneously communicate with the accelerometer component and the GPS component.

In order to implement the GPS functionality in the SmartWalker, a set of coordinates is used as ‘boundaries’ that the walker must remain within. If the walker is located outside the given boundaries, an alert is issued saying that the user of the SmartWalker has begun wandering. This design is acceptable if the user is not supposed to leave a given premise, but an improved design can be realized with further utilization of the component. For example, it is possible to allow the SmartWalker to calibrate the GPS, creating a new boundary at a given radius from its current location. This design would require further development of both the processor code and the smartphone app code, but is theoretically possible.

3.2.7 RedBear BLE Shield V2.1

The final piece of hardware included in the SmartWalker prototype is the RedBear BLE Shield V2.1 (shown in Figure 15).

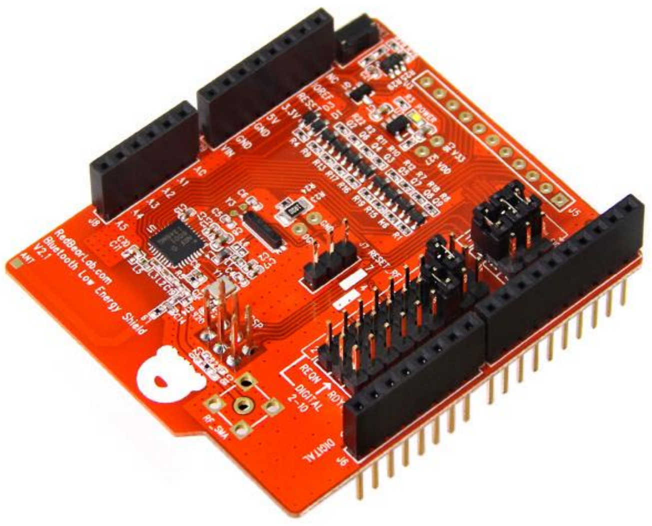


Figure 15: RedBear BLE Shield V2.1

The RedBear BLE Shield uses the Nordic nRF8001 IC to perform Bluetooth Low Energy v4.0, and combines the required connections and circuitry to integrate the Nordic chip with Arduino processors. The shape and form of the BLE Shield is made to directly plug in to the breakout pins provided on the Arduino Uno, and extend those breakout connections so that other peripherals are able to connect to the Arduino through the connections on the BLE Shield. This design makes it very simple to integrate into the SmartWalker prototype, allowing space to be saved and making the physical wiring less confusing.

The RedBear BLE Shield also provides its own Arduino code library that can be found online. The library is convenient in that, when using the RedBear BLE Shield, the functions automatically work and interface properly with the shield. Although the library makes BLE operations simple, it obfuscates the ability to create and implement additional GATT services and characteristics. Another downside to the RedBear Shield is that it lacks certain functionality that allows it to work unmodified with different Arduino libraries. For example, the RedBear shield is incompatible with the "SandeepMistry" BLE library, which also provides intuitive BLE functionality.

As stated before in Sections 3.1.1 and 3.1.2, if the design of the SmartWalker prototype had remained unchanged, and continued to use the TI CC2650 instead of the Arduino Uno, then the prototype would not need to use an external BLE chip, as BLE functionality is built-in to the CC2650.

3.2.8 Android App

Along with the physical SmartWalker prototype, we have also developed an Android OS application that should be used in conjunction with the SmartWalker. The purpose of the app is to alert a caregiver that the user is or may have difficulties walking. In this sense, the SmartWalker and app are used in a preventive manner. The app also alerts the caregiver if the user has already fallen or had another similar problem, and therefore also works by giving a more immediate response to an emergency.

For the purpose of the prototype, the application was developed in the Android Studio environment using the Java programming language. The app is supported on phones running Android KitKat Operating System (Android OS 4.4) or later, which includes over 92% of Android devices in use today.

The functionality of the app is intentionally left simple. When the app is turned on, it is able to scan for nearby Bluetooth Low Energy devices (Figure 16).

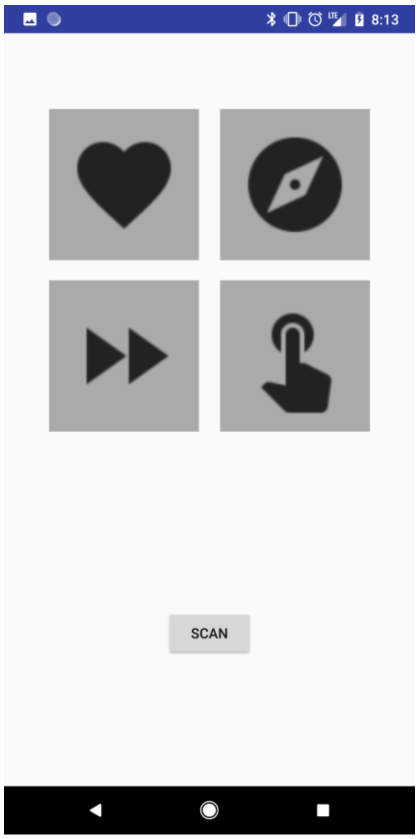


Figure 16: Android App ready to scan for SmartWalker

If the app detects the SmartWalker's BLE signal, it is able to connect and begin receiving data updates from the walker. The app's UI then changes to display an icon for each aspect of the walker that is being monitored (Figure 17).

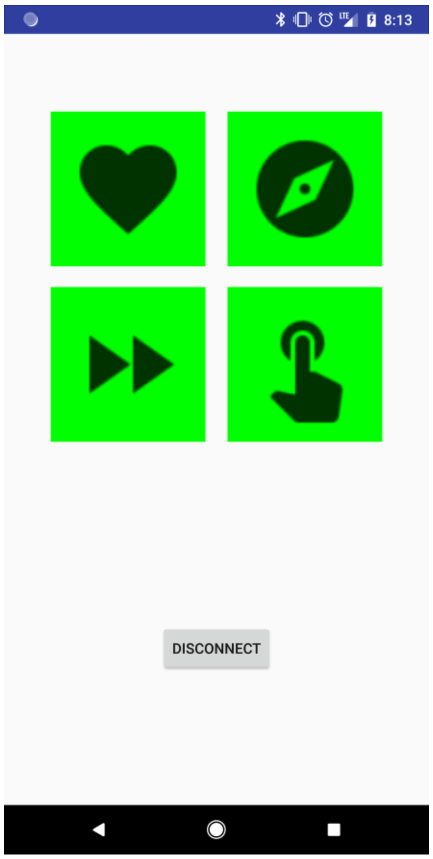


Figure 17: Android App monitoring the SmartWalker

If any of the aspects change to the "alert" status, then the icon for the corresponding aspect changes to red, and the phone will send a notification to the user to alert them of the situation.

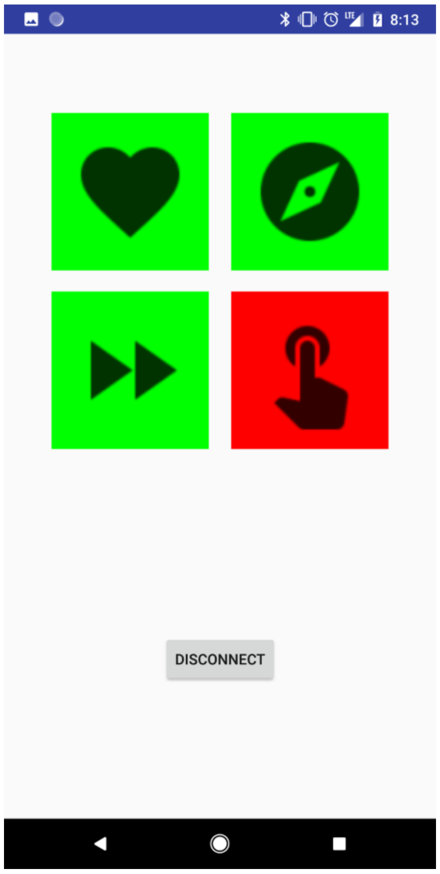


Figure 18: Android App notifying the user of a problem

The icons, however, are only included as a visual component of the app. In practice, the app will be continuously connected to the walker, and then be run in the background of the phone. This way, the caregiver can still use their phone, and simultaneously be alerted if the walker user requires attention.

3.2.9 Power and Charging Circuit

A final component to the SmartWalker that must be considered is a rechargeable power unit. During creation and testing, the SmartWalker receives its power from a USB connection with a computer. However, this power method is impractical for legitimate use of the device, and therefore a circuit that both provides power, and a method of recharging must be included.

For the purpose of the prototype, we decided to include a power circuit that uses a 9V lithium-ion battery as a power source. A circuit diagram of this power circuit is shown in Figure 19 below.

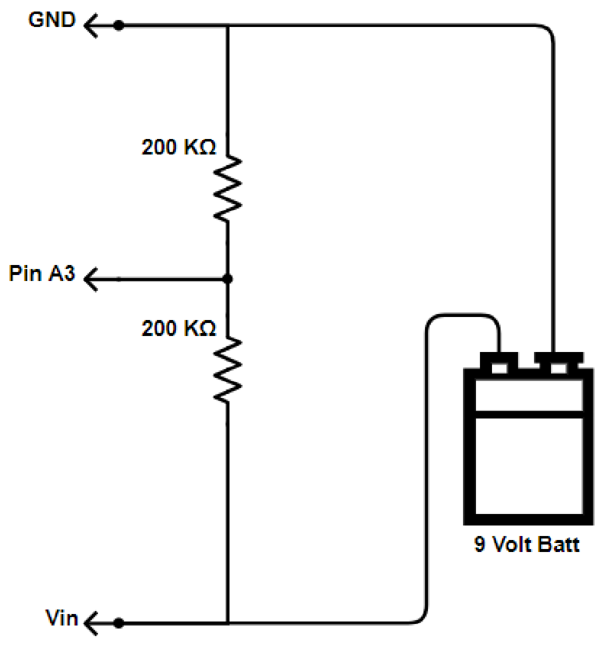


Figure 19: Power Circuit Diagram

Such a battery typically contains 600 mAh of power, meaning, with the 25mA of current drawn to the Arduino Uno when it is running, that a single battery in this system should be able to last about 24 hours of consistent use. A lithium-ion battery also has the capability of being rechargeable, allowing a user to be able to recharge the batteries used on the SmartWalker device instead of continuously have to purchase new batteries.

The power circuit also connects to the last remaining analog input pin on the Arduino Uno, which in turn allows the Arduino to measure the voltage output by the battery, and activate an on-board LED to alert the user that the battery is low.

An improvement can be made to this design in a future implementation of the SmartWalker, where the battery used to power the device has a much greater capacity and is capable of being charged directly on the walker via a USB power cable. An implementation like this is more modern, as most rechargeable devices in the present day use this method, and it would be more convenient for the user by allowing them to charge their SmartWalker directly.

3.2.10 Full Prototype Integration

In the full prototype, each sensor is connected to the Arduino Uno via the pins that correspond to the sensor’s data communication type.

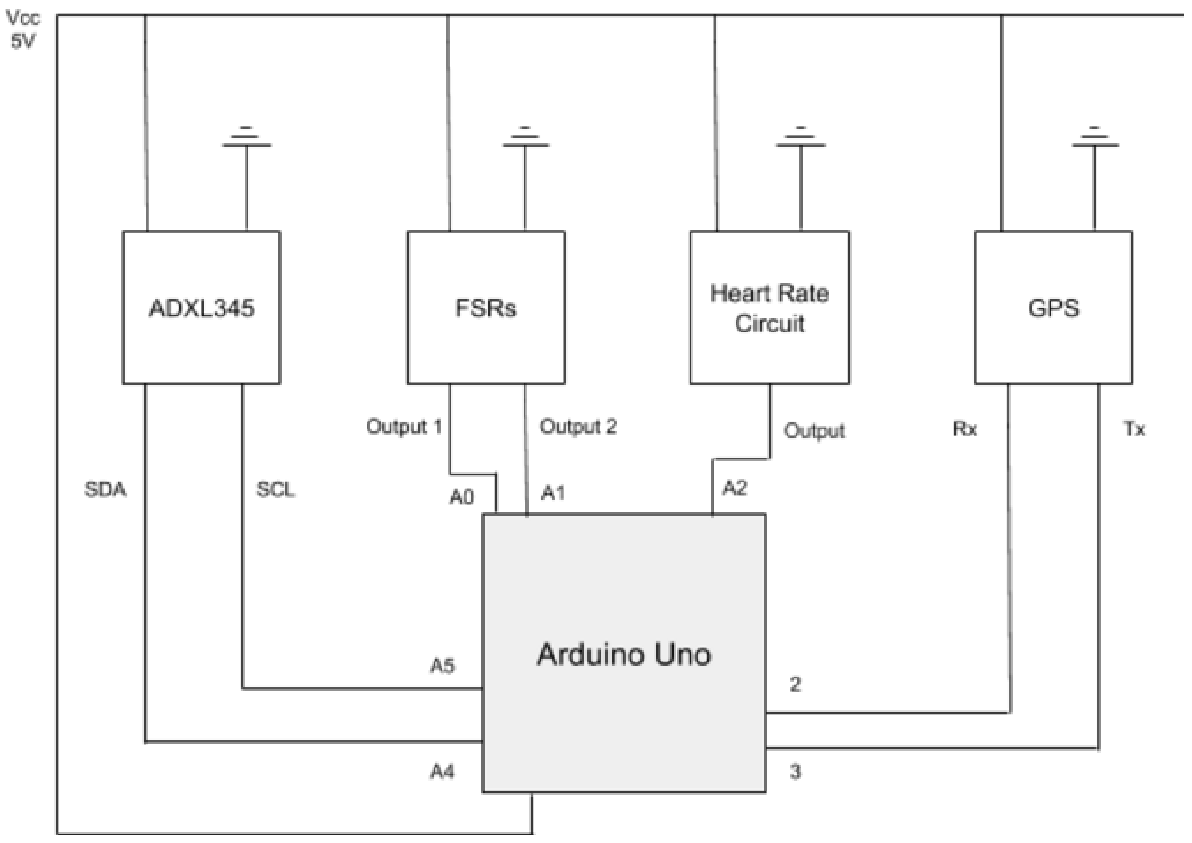


Figure 20: Arduino Pinout Circuit Diagram

As shown in Figure 20 above, the outputs of the two pressure sensor, voltage divider circuits are connected to pins A0 and A1 respectively. The output of the ECG circuit is connected to ADC pin A2. The SDA and SCL connections of the ADXL345 accelerometer are connected to pins 18 and 19 respectively. Finally, the Tx UART node of the GPS module is connected to pin 3 and the Rx node is connected to pin 2. Each component either innately supports a 5V DC voltage input, or contains a voltage regulator that changes the 5V input into a 3.3V input, and therefore, a single connection is made between the 5V output from the Arduino Uno to the input of each individual sensor. Also, the ground output from the Arduino is used as a common ground across each individual sensor. Further information about the process of building the SmartWalker prototype is portrayed in Section 3.4: Prototype Construction.

3.3 Code Implementation

This section will briefly outline the strategies and methods used to code both the Arduino and the Android app used in this iteration of the SmartWalker prototype. The language used by the Arduino is C++, and the environment coded in is Arduino's own IDE. Arduino code is structured very intuitively where there are two main functions that must be modified. The 'setup' function is used to enact initializations that should only happen once at the beginning of the code, and the 'loop' function is used to run processes repeatedly while the Arduino is functioning. The Android app, in contrast, uses the Java programming language, and is also written in its own IDE, Android Studio. Although Java and C++ are both object-oriented languages, the way C++ is implemented in the Arduino code is exceedingly different than the way Java is implemented in the Android code. The Android code requires several different structures, called 'activities', which interact using standard Android Package Kit (APK) libraries. Some activities, directly control what appears on the phone screen that is running the app, while others only control certain functions (i.e. Bluetooth communication). The use of these activities and their individual functions will be explained in the section dedicated to the Android code (Section 3.3.2).

3.3.1 Arduino Code

<u>ADC Sensors</u>

The code written to acquire data from the sensors that output analog voltages is the simplest part of the Arduino code. Certain variables are declared at the beginning of the code that have values corresponding to the ADC pins being used. These variables are then used with a function from a built-in Arduino library that sample the current voltages on the ADC pins, convert the voltages into digital values, and save the values in variables in the code. For the pressure sensors, the values of the two voltage divider circuits are then compared to determine if there is an irregular weight distribution on the handles of the walker. If an irregular distribution is found, then a byte long flag is set to 1, indicating a problem.

For the heart rate monitor, a separate function is used to sample the voltage value many times consecutively, and then process that signal to return a heart rate value. This value is compared to the previously obtained values, and, if determined to be irregular, a flag is set describing the irregularity. Both this code and the code that reads the pressure sensor values are repeated a single time in the 'loop' function, therefore updating their values every iteration of the loop.

I2C Communication

The ADXL345 accelerometer breakout comes equipped with its own Arduino library that easily implements the communication necessary to sample the accelerometer. The library allows an object to be declared, which holds all the functions and variables needed to run the accelerometer. After this object is created and the settings initialized in the 'setup' function, the accelerometer is polled for its values using a library specific function in the 'loop' code. Following this function, the x and y accelerations are obtained from the object that was declared in the beginning. The code then analyzes the two values to determine if the walker is moving in an irregular manner. Again, if this is found to be true, another flag is set to the value of 1, indicating a problem.

Serial UART

The GPS sensor requires more complicated code than the other sensors. The sensor again comes with its own library, however the functions and processing that need to be used are much more complex. To keep this section brief, most of the detail relating to the functionality will be skimmed over, however the code is included in full in Appendix A. In essence, the code initializes the GPS module in the ‘setup’ function. The code here also specifies that the GPS will be sampled within the ‘loop’ function instead of in the background by using interrupts. Then, in the loop function, a section of code samples the GPS coordinates and compares the coordinates to a predefined boundary created by another, hardcoded set of coordinates. If the current coordinates fall outside of this boundary, a final flag is set to 1 indicating a problem.

Bluetooth Low Energy

As explained in Section 3.2.7, the RedBear BLE Shield is able to run with its own library, which integrates with the Nordic nRF8001 Bluetooth chip used by the shield. To implement this, a GATT profile is initialized and named in the ‘setup’ function. This then allows the Arduino to read values from, or write values to the characteristics built into the default GATT service. In the ‘loop’ function, functions were used to write the values of the four flags, described previously in this section, to the characteristic to be able to send them to the central device (the app). Each flag is a single byte of data because the function used to write the data requires a byte of data as a parameter. Once all four flags are written to the characteristic, a function is used which updates the values in the characteristic all at once, allowing the app to receive the updated values all at once. This functionality is included once at the end of the ‘loop’ function, after each flag has had a chance to update its value based on its corresponding sensor.

3.3.2 Android Application Code

The code needed to implement BLE communication into an Android App far exceeds the software development background that any of our group members have. Therefore, much of the code used to implement BLE is reproduced examples on how to code a BLE app with slight modifications to cause the app to operate as it is intended, and to create certain UI designs specific to our SmartWalker app. Again, the code in full is included in Appendices A and B, as most of the detail will be overlooked here for the sake of brevity and comprehension.

Essentially, when a button within the app is pressed, the app launches the various BLE activities. These activities scan the surroundings for potential BLE pairing partners. After a fixed time of scanning, a list of the potential connections is given to the user, and after the user chooses from the list, the app will attempt to connect to the device using standard BLE protocols. If the SmartWalker is chosen from the list, then the connection will complete successfully, and a new activity will begin which reads the data values stored in the peripheral's characteristic. With more time and knowledge, the app can be further refined to automatically search for and connect to the SmartWalker specifically.

Once the connection is established, a looping process occurs which reads the encoded flag values, checks to see if any of the flags have been set, and updates the UI on the screen to reflect which flag has been set. When this happens, the app also launches a separate activity which uses the phone's notification system to alert the user that there is a problem with the SmartWalker whether or not the app was running in the phone's foreground.

The app intentionally does not keep any data given to it by the SmartWalker, as the values are overwritten every time the Arduino updates its characteristic values. The reason for this decision is that it allows for the app to process and use the information quicker because it does not need to launch new activities for the purpose of storing the data given to it. This also allows for simpler coding, as none of the group members for this project are very experienced with software development. Finally, this also preserves the privacy of the patient by not allowing the confidential data to be obtained from anywhere in the system.

Some very clear improvements can be made to the app. The first would be the functionality to input settings (like the boundaries of the GPS) into the app, which would then be able to send them back to the Arduino, changing the way the SmartWalker works. Another improvement could allow the Arduino to send the data directly to the app, instead of sending a flag value indicating if there was a problem or not. This would allow the app to perform its own, more robust processing, and also display some of this data to the end user.

3.4 Prototype Construction

The construction of the SmartWalker prototype required specific steps that we followed to ensure proper functionality of the prototype as a whole. The three steps met in the development of the project were separate sensor integration, app development, and full prototype integration.

3.4.1 Separate Sensor Integration

The first step in building the full SmartWalker prototype was integrating each sensor with the Arduino individually. This was done to ensure that each component was working properly and appropriately filled its purpose in the prototype. The code was written in separate modules at this point as well, meaning that each component's code still needed to be tested to determine if it could work with the other components' code. Once each sensor worked individually, implementation of the Bluetooth connectivity was included into the code of the pressure sensors. This resulted in enough functionality to begin working on a separate part of the prototype – the Android app.

3.4.2 Android App Development

Once the Arduino was capable of BLE communication through the RedBear BLE Shield, work began on the Android app. The first main reason for switching our focus to the app was to ensure that the BLE integration on the Arduino was working properly. Before the app was coded, the only way we could test the BLE integration was through free apps on the app store that scan the area for valid Bluetooth connections. Developing the app to confirm this functionality allowed progress to continue on two separate stages of the prototype at the same time.

To begin work on the Android app, a great amount of research went into the basics of app development, the basics of BLE, and how BLE is commonly integrated into an Android app.

Many tutorials and example code fragments were utilized to create a new app with the basic functionality of BLE connectivity. Throughout the process, many bugs appeared in the code, which were dealt with as they presented themselves. One notable bug occurred because the app was attempting to read the data stored in the BLE characteristic too soon after the connection was formed, which caused the app to crash. To deal with this particular bug, a delay was added between connecting to the device and polling the characteristic to read its data. Other bugs similar to this one were dealt with before a fully functional app was working.

At this point, the app development was set aside to work on the main prototype further, but the app was eventually finished with data decoding, processing, and some UI elements being added later in development.

3.4.3 Fully Integrated Prototype

With the app largely already developed, the attention of the project returned to building the full circuit prototype. To test this, each sensor was wired to the Arduino simultaneously, and a code module was created that included all the critical sections of each individual sensor.

The code would not immediately run, however, because the code that implemented the accelerometer interfered with the code that implemented the BLE shield. After this bug was resolved, though, and the alert flags were accurately being sent to the Android app, the prototype was ready to be fully combined. During this step, the leads to the ECG circuit were soldered to the backs of the HHHR Grips, and the 4 FSRs were soldered to various leads of a Cat 5 cable (Figure 21).

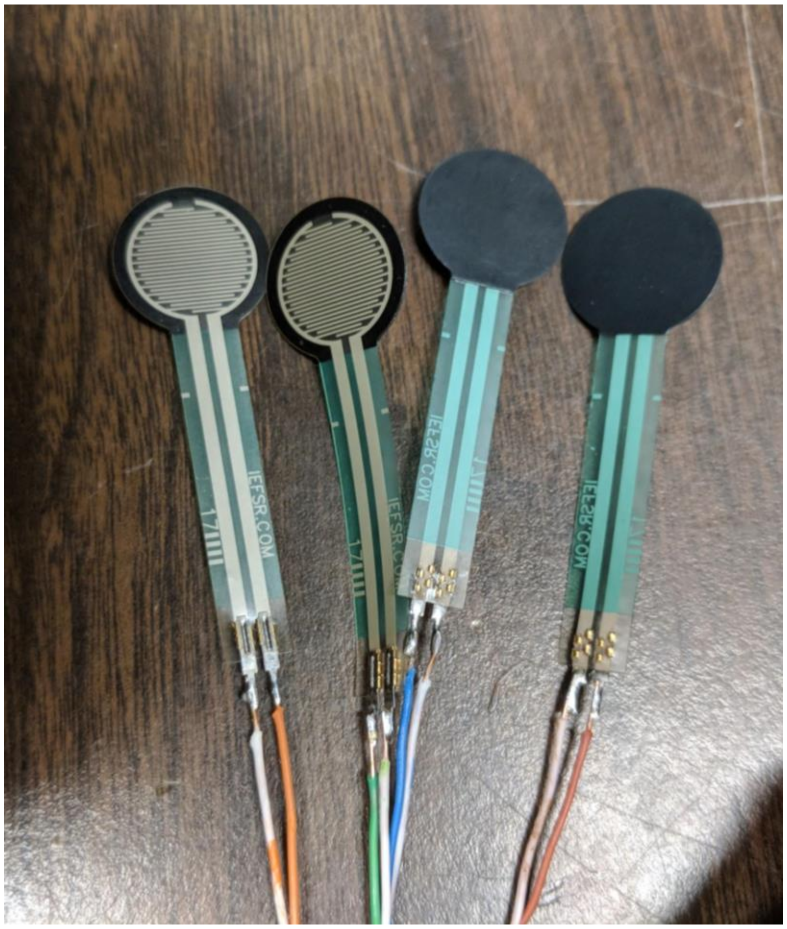


Figure 21: FSRs soldered to leads of a Cat 5 cable

This was done to ensure a good connection between the FSRs and the Arduino, as well as extend the length of the wires used while still insulating the connections. Now the circuit was ready to be mounted onto a generic walker to create the SmartWalker (shown in Figure 22).

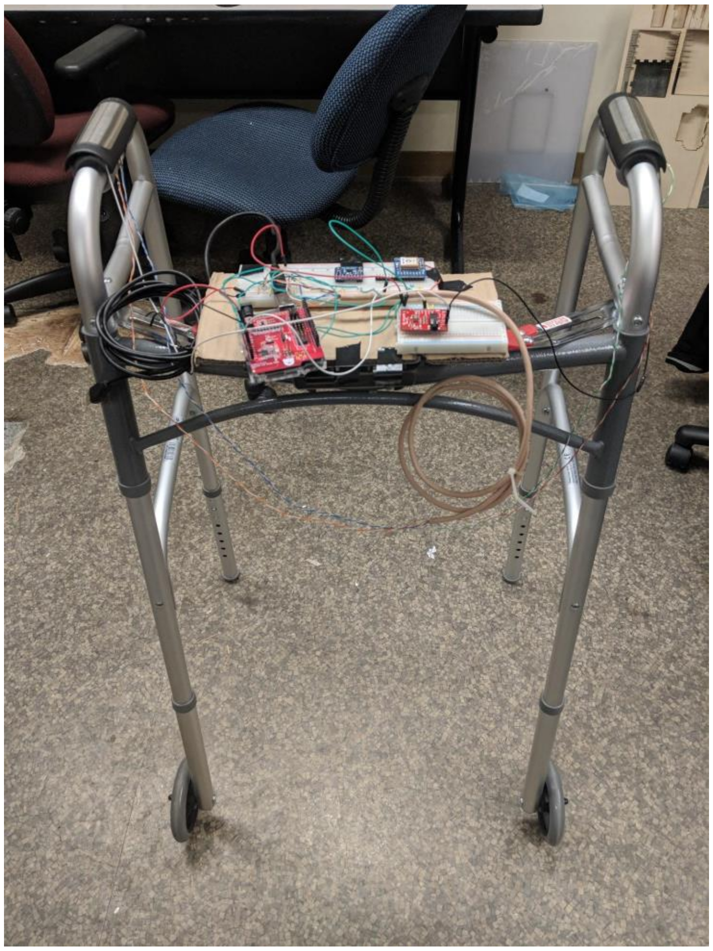


Figure 22: Mounted SmartWalker Prototype

In this state, the thresholds used to detect irregularities in a user's walking pattern were ready to be tested.

3.5 Testing

With the SmartWalker prototype fully functional, several aspects of the device underwent testing. Specifically, some of the sensors were tested to ensure that the processing done by the Arduino accurately set the alert flag in the correct situation. To test the pressure sensors and the accelerometer, we used the SmartWalker with several walking patterns that ranged from normal walking to irregular and dangerous walking (i.e. might lead to a fall). We varied aspects of our walking like how quickly we turned, how quickly we started and stopped walking, and how much pressure we put on each handle to simulate a person leaning to one side or the other. The results of these tests can be found in Section 4 (Results).

Next, to test the GPS, we brought the SmartWalker to the edge of the boundary we set in the Arduino code, and tested whether or not the 'wandering' flag was set when we brought the walker outside the boundary. This test succeeded as the flag was set once the GPS recognized that we had left the boundary.

Finally, we tested the heart rate monitor, which returned the most speculative results. As we are unable to accurately simulate an active heart problem, we first tested that the walker was able to register a valid heart rate signal. Then, we intentionally increased our heart rate by performing moderate exercise, and then allowed it to fall back down through rest. Using these states of heart activity as benchmarks, we tweaked the threshold within the code to set the 'alert' flag when the user's heart rate increases or decreases too rapidly. We concluded that our results for this test (described in more detail in Section 4.1.3) can understandably be seen as subjective, and possibly require more changes to the threshold. An even better solution can include the SmartWalker "learning" the user's average heart rate, and then being able to notify the care provider if the user's heart rate changes dramatically from that average.

4. Results

In order to determine whether the SmartWalker is able to effectively prevent falls and injuries sustained during walker use, the SmartWalker was tested in various conditions that can be considered dangerous and can lead to falls. To do this, each sensor was tested individually to learn how to adjust each respective detection algorithm. The response time of the entire system was also tested. This was done to ensure that the app responds fast enough to allow a care provider to take action in time if the SmartWalker user is about to fall. Thus, if the response time of the system is determined to be sufficient, then it can be concluded that the SmartWalker is an effective system.

4.1 Sensor Testing

The testing methods described in Section 3.5 were used to refine the algorithm used to detect abnormalities by each sensor. Each of the following sections describe how the sensor was initially implemented, and then describes what needed to change about its algorithm to make it both more effective and accurate in its problem detection.

4.1.1 Pressure Sensors

Originally, the pressure sensors would set the ‘alert’ flag if the voltages input by each pressure sensor circuit were different by over 50%. This algorithm was far too lenient as testing revealed that the flag could easily be set accidentally without intending to hold either handle with more pressure than the other.

Using the methods outlined in Section 3.5, pressure sensor data was collected and compiled to provide insight on how to modify the flagging algorithms. The first step taken before the testing began, however, was to change the algorithm already in use from one that compares the magnitude of the two sensors, to one that calculates the difference in outputs from each sensor. This method provides valuable data regardless of the pressures exerted to each individual handle.

A preliminary test indicated that the maximum difference in pressure observable is approximately 900 units. This measurement was taken by exerting a large force onto one handle, allowing its circuit to output the maximum 5 volts, while exerting no pressure onto the other handle. After this absolute maximum was established, the walker was used in two different circumstances. First, the walker was used for a period of time with a normal walking pattern. The values of pressure difference were recorded and graphed in order to observe what normal pressure on each handle looks like graphically. The resulting graph is provided below in Figure 23.

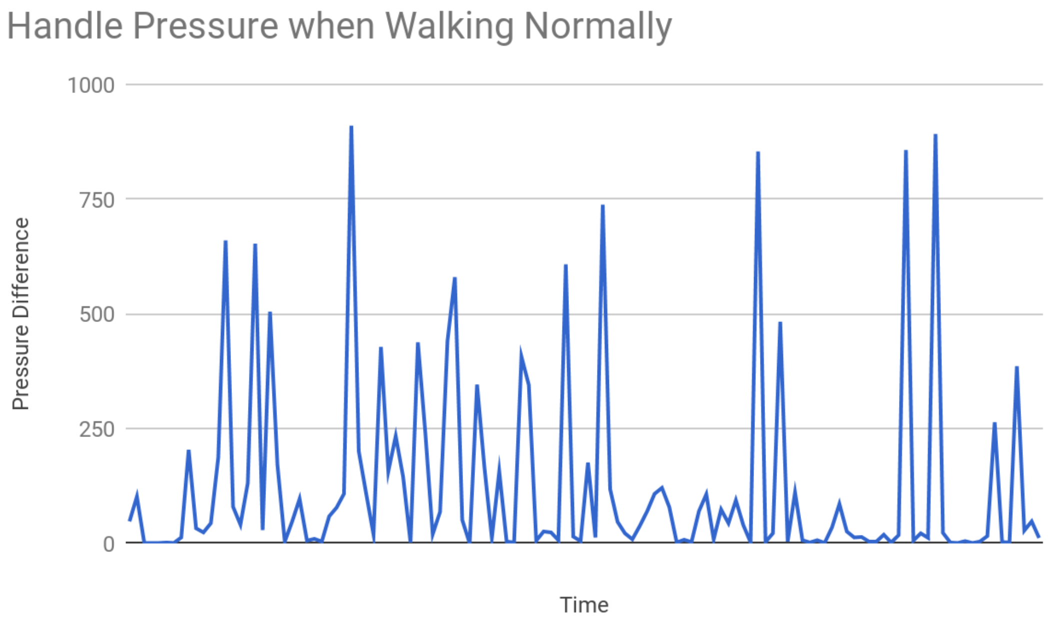


Figure 23: Handle Pressure When Walking Normally

It can be seen in the graph above that when walking normally, the pressure difference between each handle is reasonably volatile. Therefore, an algorithm that sets the alert flag whenever the pressure difference exceeds a certain value would inevitably generate false alarms frequently.

Another test was subsequently run in which the walker was used in a “dangerous” manner. This test involves two specific walking patterns considered dangerous. The first pattern resembles a person walking with a considerable majority of their weight placed on one handle of the walker rather than the other. This pattern is considered dangerous because it can lead to the walker becoming imbalanced, thus causing the walker to swerve unpredictably, or possibly tip over entirely. The second behavior that was monitored was when a person stands still while using the walker, but leans to one side to maintain balance. Speculatively, either of these usage patterns can be exhibited by users when experiencing fatigue while walking, or when experiencing a serious medical incident (for example, a stroke). The graph made from the data collected during these tests is displayed in Figure 24 below.

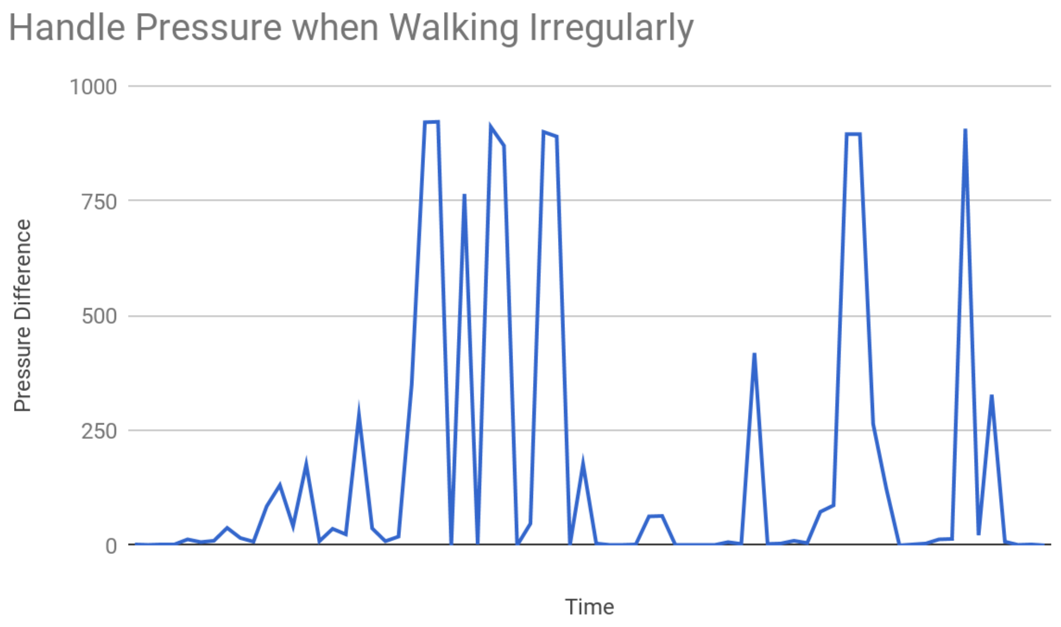


Figure 24: Handle Pressure When Walking Irregularly

There is a clear difference between the data collected when walking regularly, and walking irregularly with respect to pressure on the handles. It can be seen that when walking regularly, the difference in pressure between the handles is greatly varied, but also greatly volatile. In contrast, when walking irregularly, the pressure difference on the handles also varies, but more consistently maintains a great difference for longer periods of time. Using this analysis, it was determined that the threshold of pressure difference to use in the flagging algorithm should be 500 units. This value was decided on because every instance of unsafe pressure habits displayed a value distinctly greater than 500, but unreliably greater than any other value. It was also concluded that alongside a pressure difference threshold, the algorithm includes a delay system, such that the SmartWalker must detect at least 3 samples of pressure difference greater than the threshold within close proximity to one another. This addition to the algorithm prevents “false alarms” that could occur when walking normally because the handle pressure is much more volatile, and should presumably never maintain such a high pressure difference for that period of time.

4.1.2 Accelerometer

In order to create an effective algorithm to set the ‘alert’ flag due to irregularities detected by the accelerometer, the acceleration values in the X and Y directions were considered. When the accelerometer is mounted flat on the SmartWalker, the X direction corresponds to the walker’s left and right, and the Y direction corresponds to the walker’s forward and backward. The Z direction provided by the accelerometer is not considered in the algorithm as it corresponds to the walker’s up and down direction, and, while at rest, provides a constant -9.8 m/s^2 denoted by the acceleration of the object due to gravity. Using these parameters, a threshold that denotes ‘irregular’ motion needed to be designated.

Similarly to the procedure performed to create the pressure sensor algorithm, the walker was first used in a regular manner to observe the acceleration data sampled by the accelerometer. A graph of this data is shown in Figure 25 below.

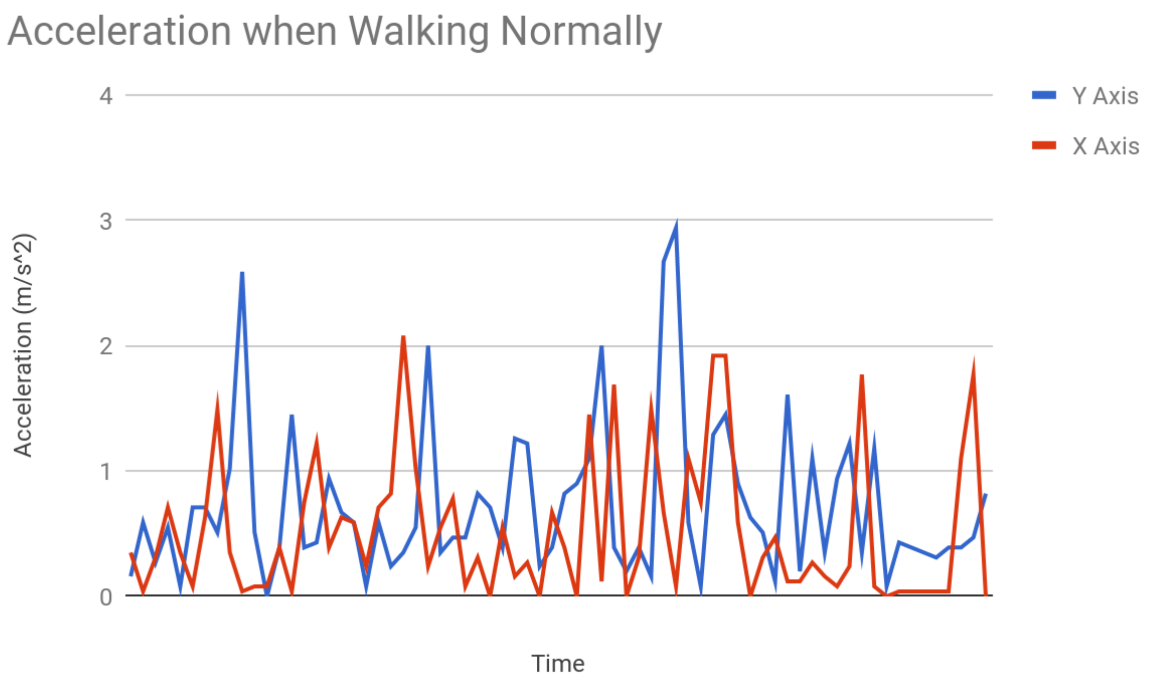


Figure 25: Acceleration When Walking Normally

As it is seen in the graph, the acceleration of both axes remains low in magnitude, specifically remaining below 3 m/s^2 throughout the entire test run. Another aspect of the data to note, is that the readings are very volatile, meaning that the acceleration of the walker at almost any point in time is distinct from the readings around it. This detail is consistent with the way the motion of the walker should theoretically behave. If the acceleration of the walker was shown to be less volatile, it would mean that the walker is speeding up or slowing down at a consistent rate, rather than speeding up or slowing down instantaneously; for a person using a walker to increase their speed at a consistent rate, it would require them to be going fast enough to be running after a couple seconds, which is an unrealistic walking pattern for someone who is using a walker.

Therefore, volatile data that remains under a certain acceleration is in line with the expected outcome of this first test.

With this baseline data collected, another test was run that simulated irregular walking patterns that lead to a fall. The walking pattern used consisted of a user using the walker while speeding up, simulating a stumbling motion, and losing control of the walker itself. A critical moment was pinpointed from the data collected during this test, and is shown as a graph in the figure below.

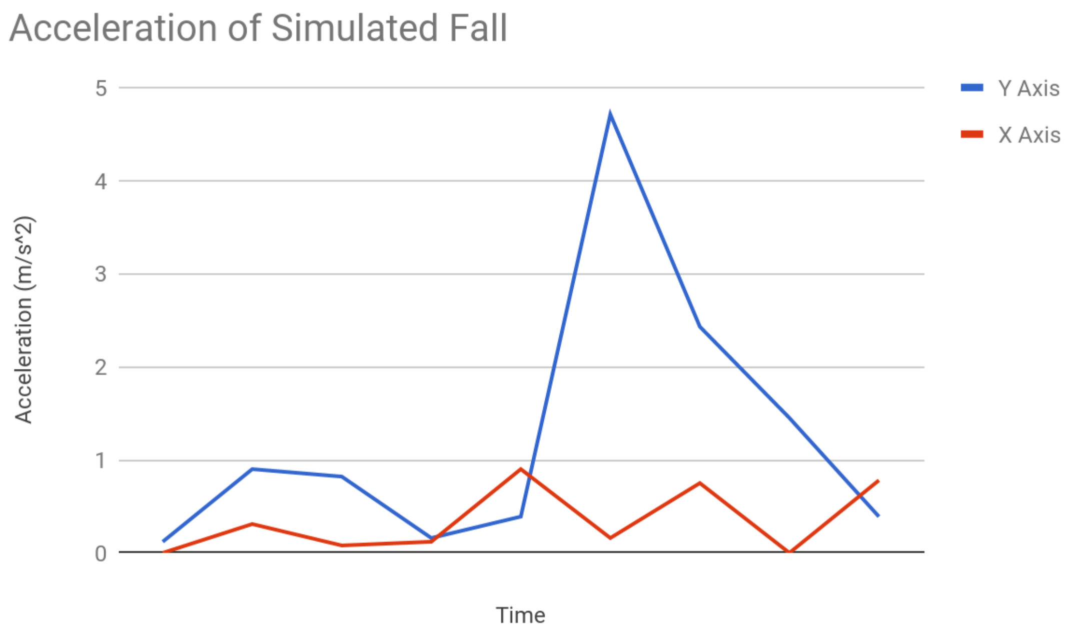


Figure 26: Acceleration of a Simulated Fall

The critical moment displayed in the above graph occurred when the user simulated a stumbling motion that would likely lead to a fall in a realistic situation. As it is shown in the graph, during the stumble, the acceleration in the Y (forward/backward) direction spiked well above 4 m/s^2 for a single sample.

Using these two tests, a threshold of 3 m/s^2 was chosen to be the upper limit of normal walking activity. This particular value was chosen because when the walker is under normal use, the acceleration in either direction never exceeds 3 m/s^2, while it does exceed this magnitude when the user exhibits a dangerous walking pattern. Another point to note is that even during the dangerous pattern, the acceleration value does not exceed 3 m/s^2 for more than a single sample likely due to the physical behavior of the motion of a walker described previously. Due to this, the algorithm was modified so that the acceleration value only has to exceed 3 m/s^2 for a single sample before the ‘alert’ flag is set.

4.1.3 Heart Rate Sensor

In order to create an algorithm to capture a heart rate signal and calculate the signal’s beats per minute, a test program was implemented in order to see peak values of the heart rate in real time. SparkFun provides a test module on their website, which reads the raw output of the AD8232 into an analog pin of an Arduino (analog pin A0 in our case). The Arduino analog read function reads in analog values and converts them to a digital value from 0 to 1023. Shown below is a plot of raw data captured by Arduino analog pin A0 from the output of the AD8232 breakout board:

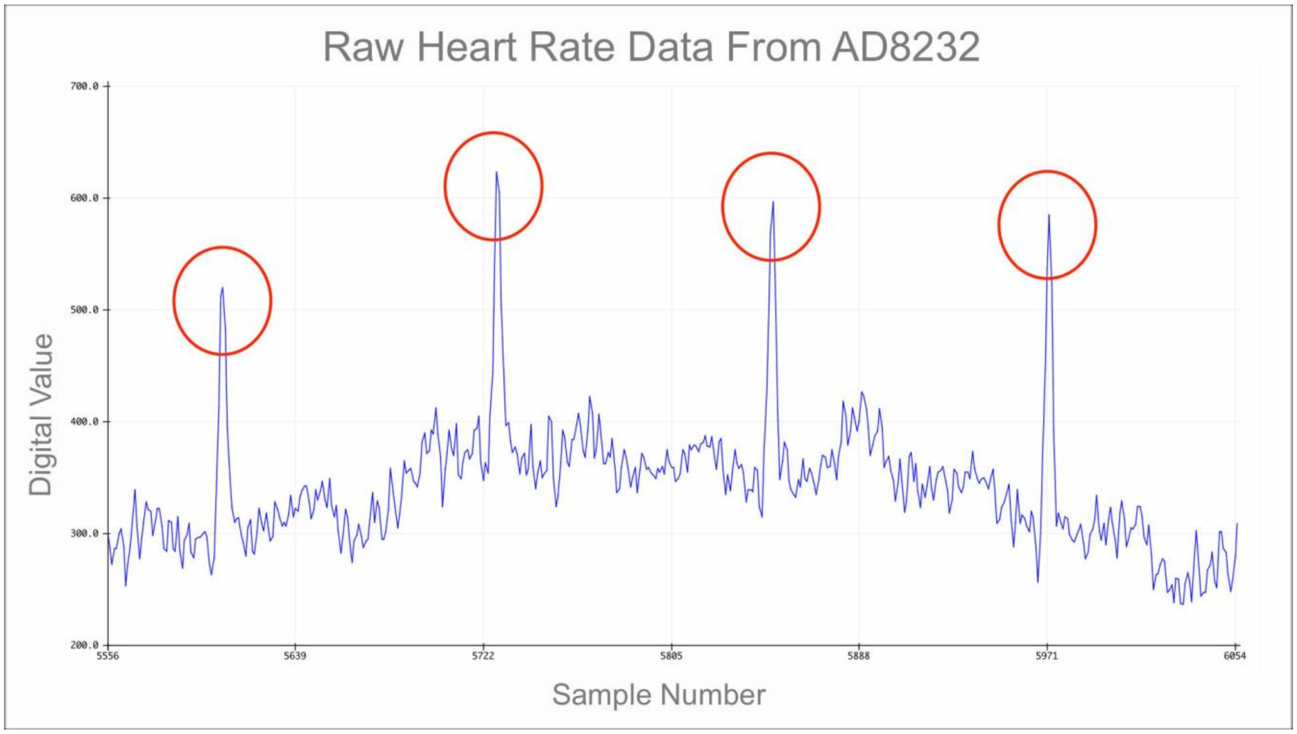
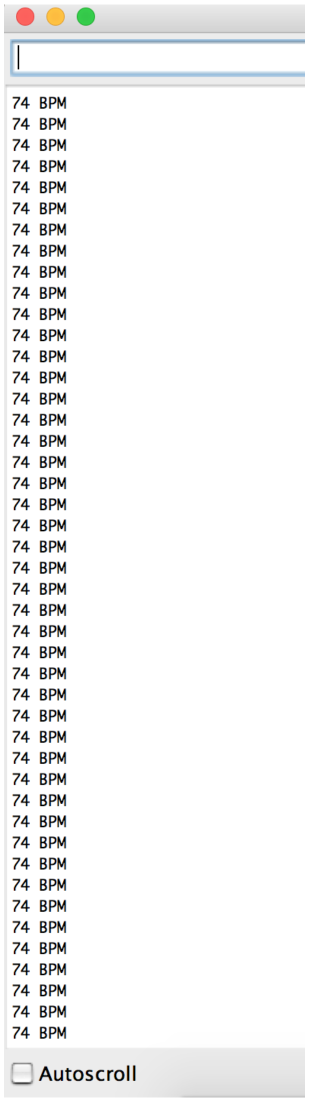


Figure 27: Raw Heart Rate Data Captured Over Time

By observing this data visually, it was clear that the majority of peak values exceeded 500, which helped in choosing correct threshold values in order to detect peaks in the final algorithm. The final heart rate function, *takeHR(),* was designed to read a raw signal, calculate beats per minute if a raw value is greater than the set upper threshold value and if a beat was read, and display the previously calculated beats per minute value if a raw value is less than a lower threshold value. After observing raw peak values over time, the upper and lower threshold values were set at values 445 and 425 respectively in order to properly detect peaks. Once a peak is detected, the BPM is calculated by subtracting the time of the previous peak from the time of the current peak. This is accomplished by keeping track of time using the Arduino *millis()* function, which returns the number of milliseconds elapsed since the program started running. The variable *newTime* is set to the return value of *millis()* if a peak is detected while the value of *LastTime* is set to the value of newTime when a peak is not detected. The Boolean value of the pulse time check variable *BPMTime* is set to ‘false’ after BPM is calculated and is set to ‘true’ once it is checked again. If a read signal is less than the lower threshold value and the Boolean value of *BPMTime* is set to ‘true’, then the BPM value is displayed serially for debugging purposes

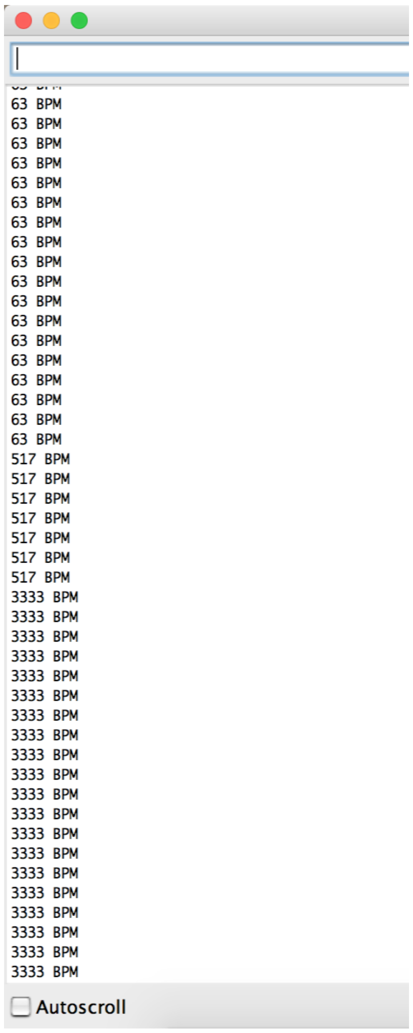
The value of the BPM is returned by *takeHR()* and is processed further in order to send a flag to the Android app if the heart rate is too high or too low. To do so, the program checks if the value of *takeHR()* is less than 160, as the human heart rate should not exceed 160 BPM, and if it is, then the heart rate is ‘valid’. Then, the BPM is checked if it is between 45 and 100 BPM and if it is, then no flag is sent to the Android App and the heart rate monitor remains green. If the heart rate is equal to 0 or less than 45, then a flag is sent, as these BPM values indicate a heart rate that is abnormally slow or not present. This also applies when the heart rate is greater than 100, but only if the heart rate is non-valid. This was to ensure that if the user was not holding the hand grips, no flag would be sent (See Appendix A for full functionality). Shown below is BPM data captured serially:



```
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
74 BPM
```

Figure 28: Serial BPM Data of Resting Heart Rate

In this case, a flag was not sent to the Android App, which allows the heart rate icon on the app remain ‘green’. Shown below is a simulated situation where the BPM rises too high:



```
63 BPM
63 BPM
63 BPM
63 BPM
63 BPM
63 BPM
63 BPM
63 BPM
63 BPM
63 BPM
63 BPM
63 BPM
63 BPM
63 BPM
63 BPM
63 BPM
63 BPM
63 BPM
517 BPM
517 BPM
517 BPM
517 BPM
517 BPM
517 BPM
517 BPM
3333 BPM
3333 BPM
3333 BPM
3333 BPM
3333 BPM
3333 BPM
3333 BPM
3333 BPM
3333 BPM
3333 BPM
3333 BPM
3333 BPM
3333 BPM
3333 BPM
3333 BPM
3333 BPM
3333 BPM
3333 BPM
3333 BPM
3333 BPM
```

Figure 29: Serial BPM Data of Heart Rate above Upper Threshold

In this situation, a resting heart rate of 63 BPM changes to 517 BPM, an abnormally fast heart rate. The program is designed to detect three abnormal values before sending a flag to the app. Thusly, after the third reading of 517 BPM, the heart rate icon on the app turns ‘red’. However, once the BPM value changes to 3333 BPM, it is assumed that the user has removed their hands from the grips, resulting in a removal of the previous flag and changing the heart rate icon back to ‘green’.

4.1.4 GPS

When the walker is taken beyond the predetermined boundaries shown by the red area around Worcester Polytechnic Institute in Figure 31 below, the SmartWalker system correctly alerts the app that the patient has begun wandering.

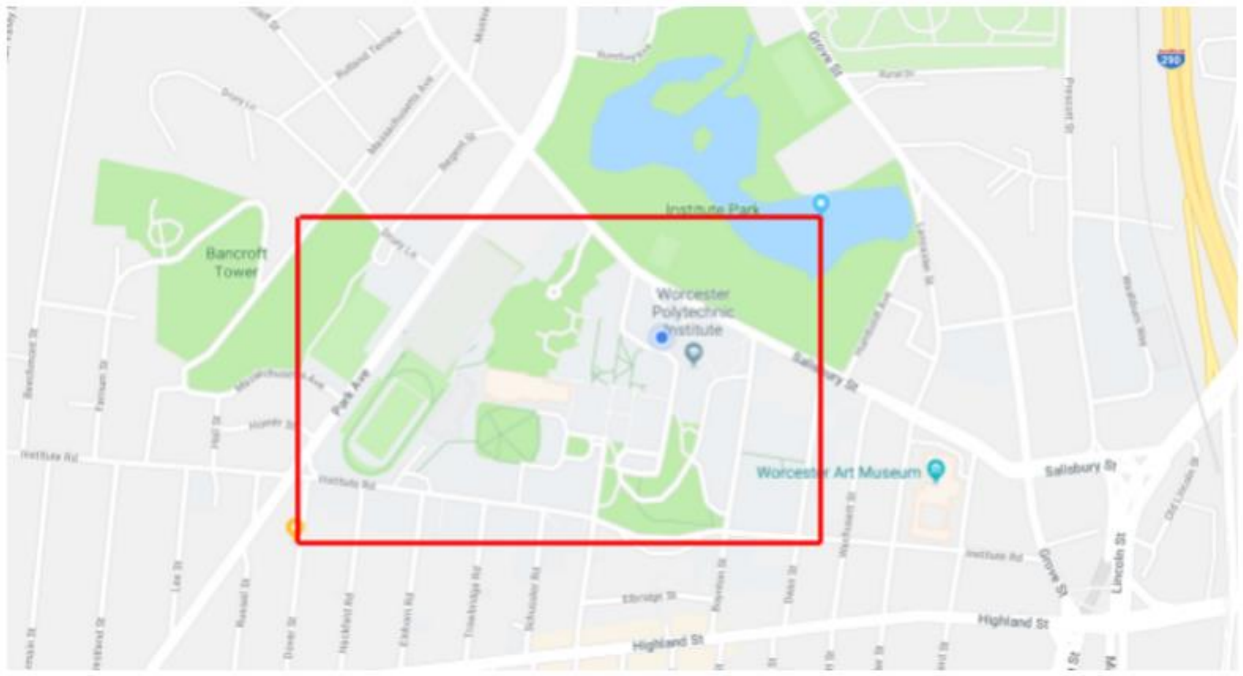


Figure 30: Location Boundaries that Indicate Wandering

The GPS module was not tested beyond this initial testing because this demonstration is enough to prove that the GPS module of the SmartWalker is successful at detecting wandering, and thus satisfies the design requirement.

4.2 Response time testing

After the testing of each sensor was performed, and the algorithm used for each flag was created and proven to be effective, the response time of the system as a whole was tested. To do this, we again tested the SmartWalker system using irregular walking patterns, and recorded the time that elapsed before the app alerted the user that there was a problem. This method was repeated for both types of movement that can be considered dangerous--both uneven pressure on the handles and unsafe movement of the walker as a whole. It was much more difficult to test the response time of the system when detecting a heart rate problem, because we were unable to accurately simulate a heart problem directly when needed; therefore, we did not measure the response time of the system when detecting a heart problem. The response time of the GPS module was also not tested because when a patient is wandering, they are not necessarily in any immediate danger, so as long as the app is alerted as soon as the patient leaves the boundaries, then the module can be considered successful.

The first set of response time tests were performed to examine the alert generated by the pressure sensor. The tests were conducted by having one group member start a timer when another group member started walking with an irregular distribution of weight placed on each walker handle. The timer was then stopped as soon as the SmartWalker app generated a notification indicating that there was a problem. This test was run 10 times, and the average of the elapsed time was calculated to be 3.21 seconds. This means that on average, when a person uses the SmartWalker system and starts applying irregular and dangerous weight distribution, the system will take about 3.21 seconds to alert a care provider of the problem. Due to this, one is able to conclude that 3.21 seconds is well within an acceptable time range in which a care provider needs to be alerted. This is because when a person starts exhibiting dangerous weight distribution, it does not necessarily mean that they are going to fall soon after, but rather it shows that the person is at increased risk of falling, and should probably sit down or tend to whatever is causing them to walk in this pattern.

The second set of response time tests were performed to examine the alert generated by the accelerometer. Similarly to the last set of tests, one group member started a timer when another group member using the walker began to simulate stumbling, and stopped the timer when the SmartWalker app generated a notification about it. Again, this test was repeated 10 times, and the average response time in this instance was 1.10 seconds. This result is less obviously within an acceptable range of response times, but it is still effective given the system its working within. This response time is expected and required to be shorter than the one for the pressure sensors because when a person using a walker begins to stumble, they are far more likely to fall as an immediate result of the loss of balance. Therefore, if a person begins to lose their balance, the care provider should be alerted immediately to attempt to prevent the person from falling, or helping the person immediately after falling if necessary. Although a hypothetical perfect response time for a person losing their balance would be instantly, 1.10 seconds is approximately as short as the response time for any alert can get due to the sample rate of the program, and the transfer rate of the data over BLE. Therefore, we conclude that this short response time is adequate in proving the SmartWalker system is effective in this regard.

5. Conclusion

Based on the results collected and presented in the previous section, it is beneficial to evaluate the current functionality of the SmartWalker in terms of the required design constraints outlined in the Introduction. The design constraints are reproduced below:

1. The device must be an all-in-one assistive mobility device.
2. The device must utilize a number of sensors that can measure the heart rate of a patient, monitor the patient's movement performance, and track the patient's location.
3. The device must be able to collect data discreetly, as the patient should not be able to alter the performance of the sensors.
4. The device must be able to wirelessly connect to a smartphone and transmit the sensor data which are then processed and utilized by an application which will serve as the user interface.
5. The device must operate using a battery and at a low power setting to ensure long battery life.

The first two of these constraints are inherently met by the definition of the SmartWalker design. The prototype we have built can be used entirely without the BLE functionality and retain the same assistive functionalities as a conventional walker. Also, with the inclusion of the heart rate monitor, pressure sensors, accelerometer, and GPS module, the SmartWalker is able to monitor the user's behavior discreetly, meeting both the second and third design constraints. The fourth design constraint is met with the inclusion of the Android app described in Section 3.2.8. Finally, with the inclusion of rechargeable batteries in the power circuit, the SmartWalker should maintain power for 24 hours of constant use before requiring a battery change, thus accomplishing the fifth design constraint.

In its current model, the SmartWalker has the capability to alleviate some problems affecting the Alzheimer's and Dementia population. The SmartWalker is able to both warn a caretaker of a patient's irregular behavior, as well as promptly alert the caretaker of an accident that has occurred. This allows the caretaker to be more efficient in their service, and can reduce the number of preventable falls that occur in general. The SmartWalker is also capable of reducing the wandering problem seen in the same population, and allow a caretaker to relocate a lost patient immediately if necessary.

Improvements

As the SmartWalker is currently in the prototype stage, there are undoubtedly many improvements that could be made to its design that would increase its usability. One such improvement would be to replace the Arduino ATmega328 processor with one that has a higher clock rate and better performance in general. The TI CC2650 processor described in Section 3.1.1 is a favorable replacement, as it has integrated BLE functionality, as well as a low power mode.

Another improvement that can be made requires increasing the SmartWalker's low power capabilities. In its current form, the SmartWalker operates in full when in use, and does not operate at all when turned off. A better solution would be to include interrupt based low-power features that perform actions when needed to decrease power usage while operating, and allow the walker to automatically turn off and on when it begins or ends use respectively. This improvement would greatly improve its battery life.

The SmartWalker would also benefit from an improvement to its data processing. The current method of data processing, described in full in Section 3.3, just allows the device to set a flag when there is a problem. An improvement can be made to this that allows the device to detect and assert a more specific problem rather than one solely based on which component is identifying the problem. Also, more of the data received from the sensors can theoretically be placed in other BLE characteristics and transferred to the Android app for use in the app.

A final, larger, improvement that can be made involves adding Wi-Fi capabilities to the SmartWalker. This improvement would be a much greater change to the SmartWalker's design in general, but would allow the received data to be transmitted over Wi-Fi to a server on the internet, instead of being solely sent over BLE communication. This would also allow for data retrieval at a much greater distance from the walker itself, and removes some other limitations present in a Bluetooth implementation.

Incorporating some of these ideas to the current state of the SmartWalker prototype would immensely increase the device's usability and scope. However, even in its current state, the SmartWalker certainly meets the desired goals for this project. In conclusion, we believe that the SmartWalker has the potential to address the problems found in the Alzheimer's and Dementia population, and the current state of the SmartWalker device exemplifies the work done throughout this project to achieve these intended goals.

6. References

[1] Latest Alzheimer's Facts and Figures. (2016, March 30). Retrieved from https://www.alz.org/facts/overview.asp

[2] United States of America, Census Bureau. (2017, April 10). Facts for Features: Older Americans Month: May 2017. Retrieved from https://www.census.gov/newsroom/facts-for-features/2017/cb17-ff08.html

[3] United States of America, Department of Health and Human Services, Centers for Disease Control and Prevention. (2017). *Health, United States, 2016: With Chartbook on Long-term Trends in Health*. Hyattsville, MD. Retrieved from https://www.cdc.gov/nchs/data/hus/hus16.pdf#045

[4] Harris-Kojetin L, Sengupta M, Park-Lee E, et al. *Long-term care providers and services users in the United States: Data from the National Study of Long-Term Care Providers,* 2013–2014. National Center for Health Statistics. Vital Health Stat 3(38). 2016. Retrieved from https://www.cdc.gov/nchs/data/series/sr_03/sr03_038.pdf

[5] Alzheimer’s Association. Wandering and Getting Lost. Retrieved from https://www.alz.org/care/alzheimers-dementia-wandering.asp.

[6] United States of America, Center for Disease Control and Prevention. (2017, May 3). *FastFacts: Older Persons' Health*. Retrieved from https://www.cdc.gov/nchs/fastats/older-american-health.htm

[7] United States of America, National Library of Medicine. (2018, March 5). *Aging Changes in the Heart and Blood Vessels*. Retrieved from https://medlineplus.gov/ency/article/004006.htm

[8] Physio.co.uk. Reduced Mobility. (n.d.). Retrieved from http://www.physio.co.uk/what-we-treat/elderly/reduced-mobility.php

[9] United States of America, Center for Disease Control and Prevention. (2017, May 3). *FastFacts: Disability and Functioning (Noninstitutionalized Adults Aged 18 and Over)*. Retrieved from https://www.cdc.gov/nchs/fastats/older-american-health.htm

[10] United States of America, Center for Disease Control and Prevention. (2017, February 10). *Important Facts about Falls*. Retrieved from https://www.cdc.gov/homeandrecreationalsafety/falls/adultfalls.html

7. Appendices

Appendix A: Arduino Code

```
//"services.h/spi.h/boards.h" is needed in every new project
#include <SPI.h>
#include <boards.h>
#include <RBL_nRF8001.h>
#include <services.h>
#include <Adafruit_GPS.h>
#include <SoftwareSerial.h>
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_ADXL345_U.h>


#define UpperThreshold 445 // upper threshold value to check for valid signal
#define LowerThreshold 425 // lower threshold value to check for valid signal

//initialize the accelerometer object
Adafruit_ADXL345_Unified accel = Adafruit_ADXL345_Unified(12345);

//initialize GPS variables and functions
SoftwareSerial mySerial(3, 2);
Adafruit_GPS GPS(&mySerial);

#define GPSECHO  true

boolean usingInterrupt = false;
void useInterrupt(boolean);

//characteristic uuid: 713d0000-503e-4c75-ba94-3148f18d941e
//service uuid:        713d0002-503e-4c75-ba94-3148f18d941e

//initialize ADC pins
int adc0 = A0;
int adc1 = A1;
int hrSensor = A2;


//initialize variables used throughout the Loop function
int press0, press1;    //results of pressure sensor reads
int pressCounter = 0;  //counter used in pressure processing
int accelCounter = 0;  //counter used in accelerometer processing
float accelx, accely;  //results of accelerometer read
unsigned char pressByte = 0;  //pressure alert flag
unsigned char HRByte = 0;     //heart rate alert flag
unsigned char accelByte = 0;  //acceleration alert flag
unsigned char GPSByte = 0;    //gps alert flag
int x = 0; // counter
unsigned long LastTime = 0; // previous time in ms
bool BPMTime = false; // pulse time check
bool BeatComplete = false; // checks if BPM has been read successfully
int BPM = 0; // holds value of BPM
int HRreturn;  //holds the BPM value returned by takeHR
int HRcount = 0; //counter that determines when the app should be alerted of a HR problem
bool HRvalid = false; //used to decide if BPM values are valid
```

```
void setup()
{
  //setup the broadcast name of the BLE profile
  ble_set_name("SmartWalk");

  // Init. and start BLE library.
  ble_begin();
  Serial.begin(57600);

  //Initialize and begin the GPS functions
  GPS.begin(9600);
  GPS.sendCommand(PMTK_SET_NMEA_OUTPUT_RMCGGA);
  GPS.sendCommand(PMTK_SET_NMEA_UPDATE_1HZ);
  GPS.sendCommand(PGCMD_ANTENNA);
  useInterrupt(false);
  delay(1000);
  mySerial.println(PMTK_Q_RELEASE);


  //initialize and start the accelerometer library
  if(!accel.begin())
  {
    Serial.println("Ooops, no ADXL345 detected ... Check your wiring!");
  }

  accel.setRange(ADXL345_RANGE_4_G);
}

unsigned char buf[16] = {0};
unsigned char len = 0;

//coordinates used for boundaries of GPS
uint32_t timer = millis();
float homeLatNorth = 42.276601;
float homeLatSouth = 42.272326;
float homeLonWest = -71.804063;
float homeLonEast = -71.813483;



//Function that decides whether or not to use interrupts for the GPS functionality
//This project has them set to be unused
void useInterrupt(boolean v) {
  if (v) {
    // Timer0 is already used for millis() - we'll just interrupt somewhere
    // in the middle and call the "Compare A" function above
    OCR0A = 0xAF;
    TIMSK0 |= _BV(OCIE0A);
    usingInterrupt = true;
  } else {
    // do not call the interrupt function COMPA anymore
    TIMSK0 &= ~_BV(OCIE0A);
    usingInterrupt = false;
  }
}
```

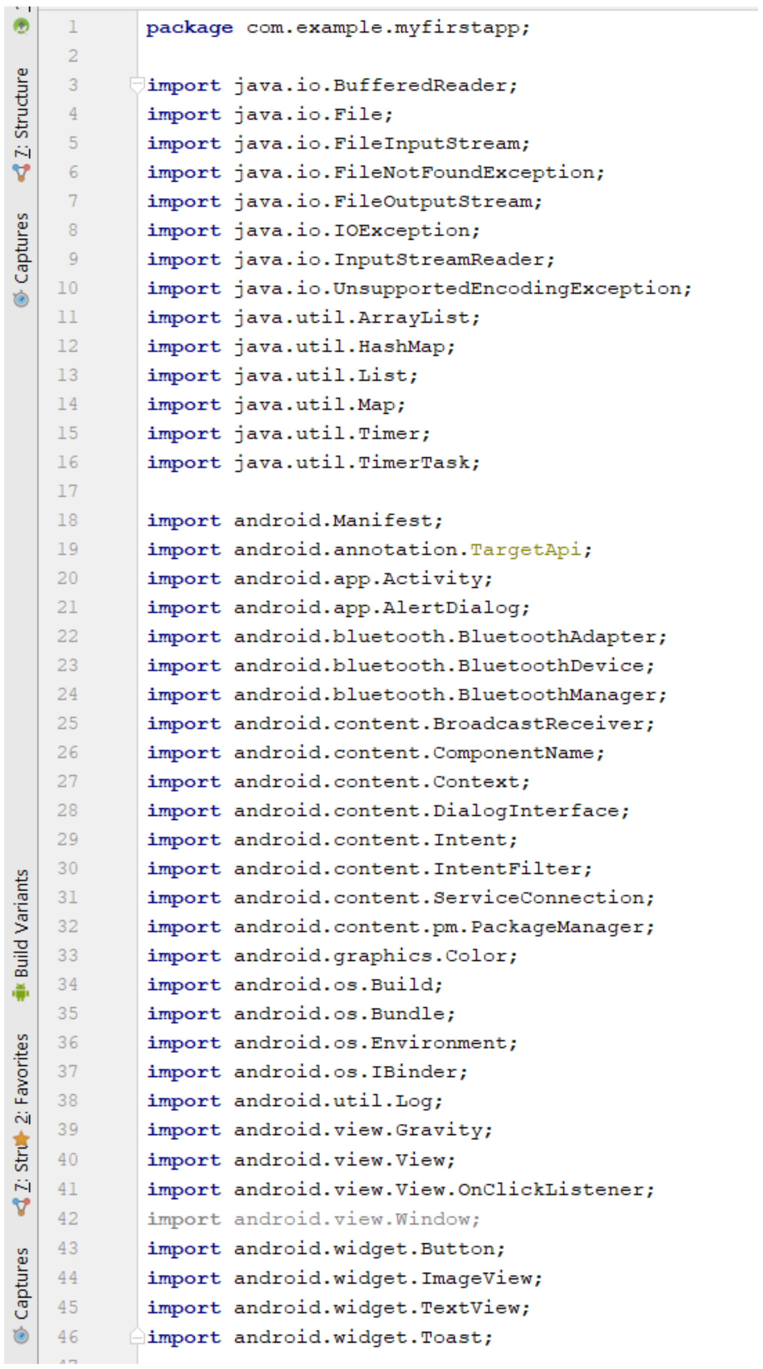
```
void loop()
{
  //below is all the GPS polling functionality
  //At the end of this section, the GPS flag gets set to 1 if the location is outside of the boundaries
  //-----------------------------------------------------------------------------------
  if (! usingInterrupt) {
    // read data from the GPS in the 'main loop'
    char c = GPS.read();
    // if you want to debug, this is a good time to do it!
    if (GPSECHO)
      if (c) Serial.print(c);
  }

  // if a sentence is received, we can check the checksum, parse it...
  if (GPS.newNMEAreceived()) {
    // a tricky thing here is if we print the NMEA sentence, or data
    // we end up not listening and catching other sentences!
    // so be very wary if using OUTPUT_ALLDATA and trytng to print out data
    //Serial.println(GPS.lastNMEA());   // this also sets the newNMEAreceived() flag to false

    if (!GPS.parse(GPS.lastNMEA()))   // this also sets the newNMEAreceived() flag to false
      return;  // we can fail to parse a sentence in which case we should just wait for another
  }

  // if millis() or timer wraps around, we'll just reset it
  if (timer > millis())  timer = millis();

  // approximately every 2 seconds or so, print out the current stats
  if (millis() - timer > 2000) {
    timer = millis(); // reset the timer

    if (GPS.fix) {
        if ((GPS.latitudeDegrees > homeLatNorth) || (GPS.latitudeDegrees < homeLatSouth)
            || (GPS.longitudeDegrees > homeLonEast) || (GPS.longitudeDegrees < homeLonWest)) {
          GPSByte = 1;
        }
    }
  }
  //----------------------------------------------------------------------------------


  //read the analog values of the two pressure sensor circuits
  press0 = analogRead(adc0);
  press1 = analogRead(adc1);


  //run the function that measures heart rate
  HRreturn = takeHR();

  //processing done to determine if the app should be alerted
  if(HRreturn < 160)
    HRvalid = true;
  else
    HRvalid = false;

  if(HRreturn != 0 && (HRreturn < 45 || HRreturn > 100) && HRvalid == true) {
    if(HRreturn != lastHR) {
      HRcount++;
    }
    lastHR = HRreturn;
  }
  else if(HRreturn != 0 && HRvalid == true) {
    if(HRcount > 0)
      HRcount--;
  }
  else if(HRvalid == false) {
    HRcount == 0;
  }
```

```
//retrieve the accelerometer data through an event struct
sensors_event_t event;
accel.getEvent(&event);

//save the acceleration values to local variables
accelx = event.acceleration.x;
accely = event.acceleration.y;


//calculate the difference in pressure, and set positive if its a negative value
press1 = press1 - press0;
if(press1 < 0)
  press1 = 0 - press1;

//if the difference is over 500 units, increment a counter
if(press1 > 500) {
  pressCounter++;
}
//if the difference is less than 500 units, decrement the counter
else if(pressCounter != 0)
  pressCounter--;

//if the counter is greater than two, set the pressure flag
if(pressCounter > 2)
  pressByte = 1;
//if not, then the flag remains unset
else
  pressByte = 0;

//checks to see if BPM is in a safe range, sets the HR flag if not
if(HRcount > 2 && HRvalid == true) {
  HRByte = 1;
}
else
  HRByte = 0;

//set both acceleration values to positive
if(accelx < 0)
  accelx = 0 - accelx;
if(accely < 0)
  accely = 0 - accely;


//if either value is greater than 3 m/s^2 in magnitude, then set the flag
if(accelx > 3 || accely > 3) {
  accelByte = 1;
  accelCounter = 3;
}
//accelCounter is used to allow the alert to remain in the app for multiple data transfers rather than just one
else {
  if(accelCounter)
    accelCounter--;
  else
    accelByte = 0;
}
```

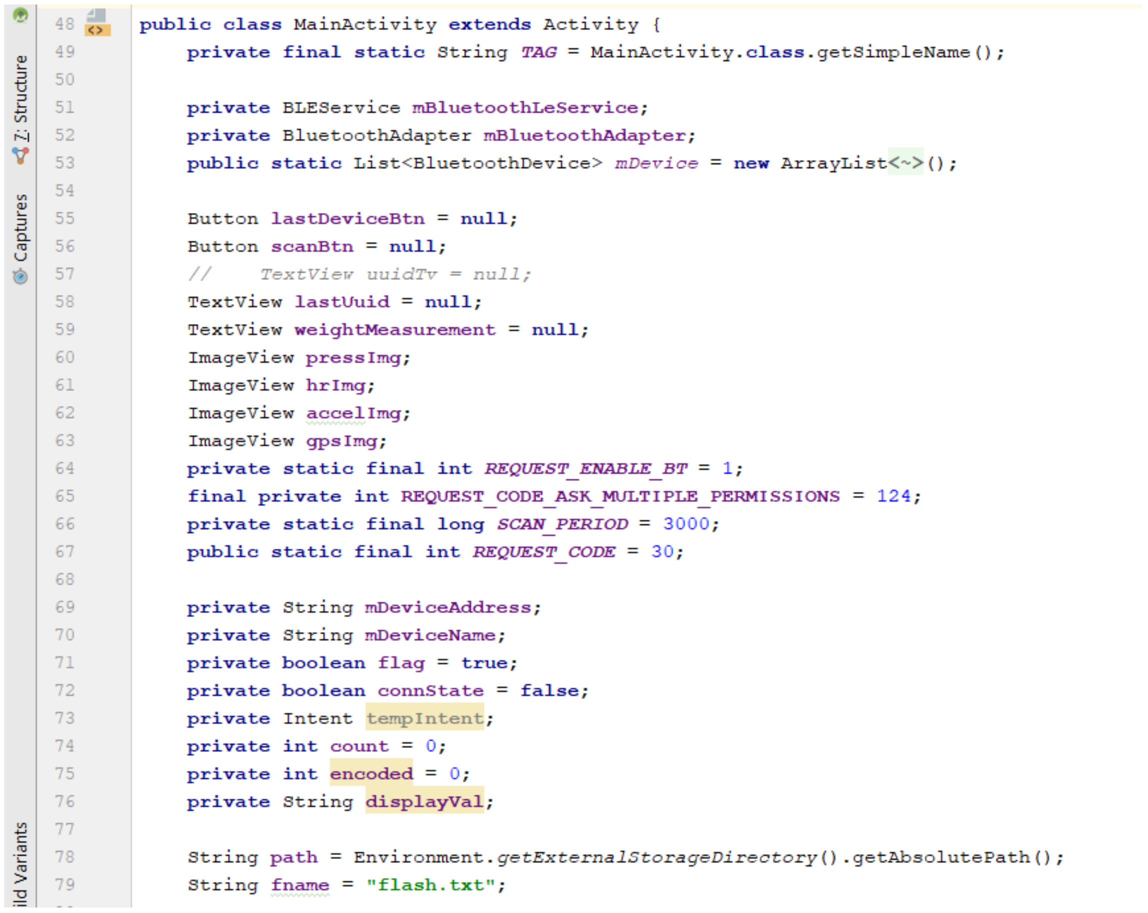
```
  //if there is an active BLE connection, write all four flag bytes to the characteristic
  if ( ble_connected() )
  {
    ble_write(pressByte);
    ble_write(HRByte);
    ble_write(accelByte);
    ble_write(GPSByte);
  }
  //update the characteristic with the new value after all four are written
  ble_do_events();

  //check for written data to the BLE characteristic
  //this section is unused by the current prototype
  if ( ble_available() )
  {
    while ( ble_available() )
    {
      Serial.write(ble_read());
    }

    Serial.println();
  }

  GPSByte = 0;
}


int takeHR() {
    int falseFlag = 0;
    int newTime;
    int signal = analogRead(hrSensor); // read signal
    // if signal value read is greater than upper threshold, check for a beat
    if (signal > UpperThreshold) {
      if (BeatComplete) {
        newTime = millis();
        BPM = newTime - LastTime;
        BPM = int(60 / (float(BPM) / 1000));
        BPMTime = false;
        BeatComplete = false;
      }
      // if time is false, set LastTime to time since last reset in ms
      if (BPMTime == false) {
        LastTime = newTime;
        falseFlag = 1;
        BPMTime = true; // set pulse time to true
      }
    }
    // if read signal value is less than lower threshold and pulse time is set, display BPM
    if ((signal < LowerThreshold) & (BPMTime))
      BeatComplete = true;
      // display bpm
      Serial.print(BPM);
      Serial.println(" BPM");
      x++; // increment counter

      if(falseFlag)
        return 0;
      else
        return BPM;
}
```

Appendix B: Android App Code

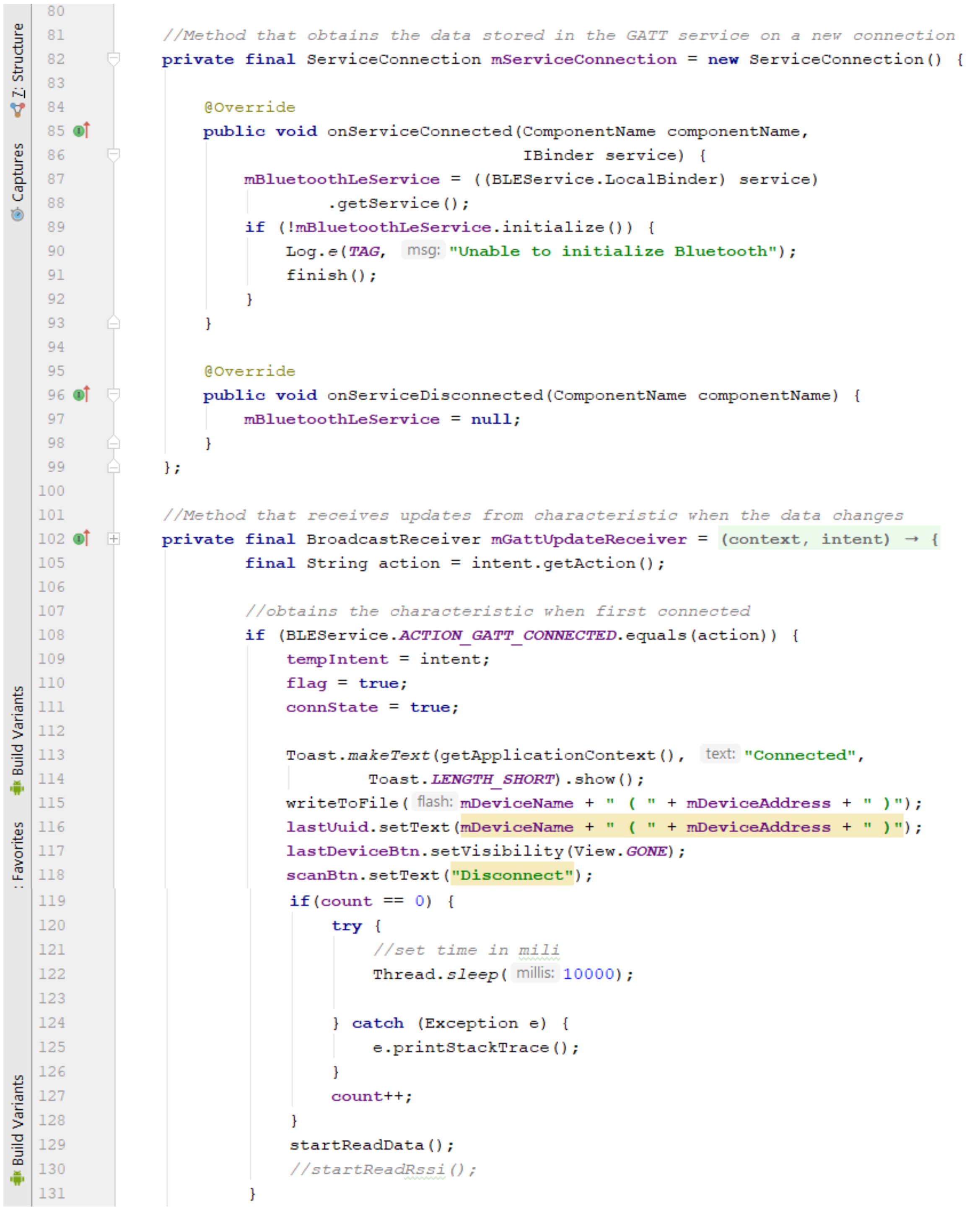
MainActivity.java



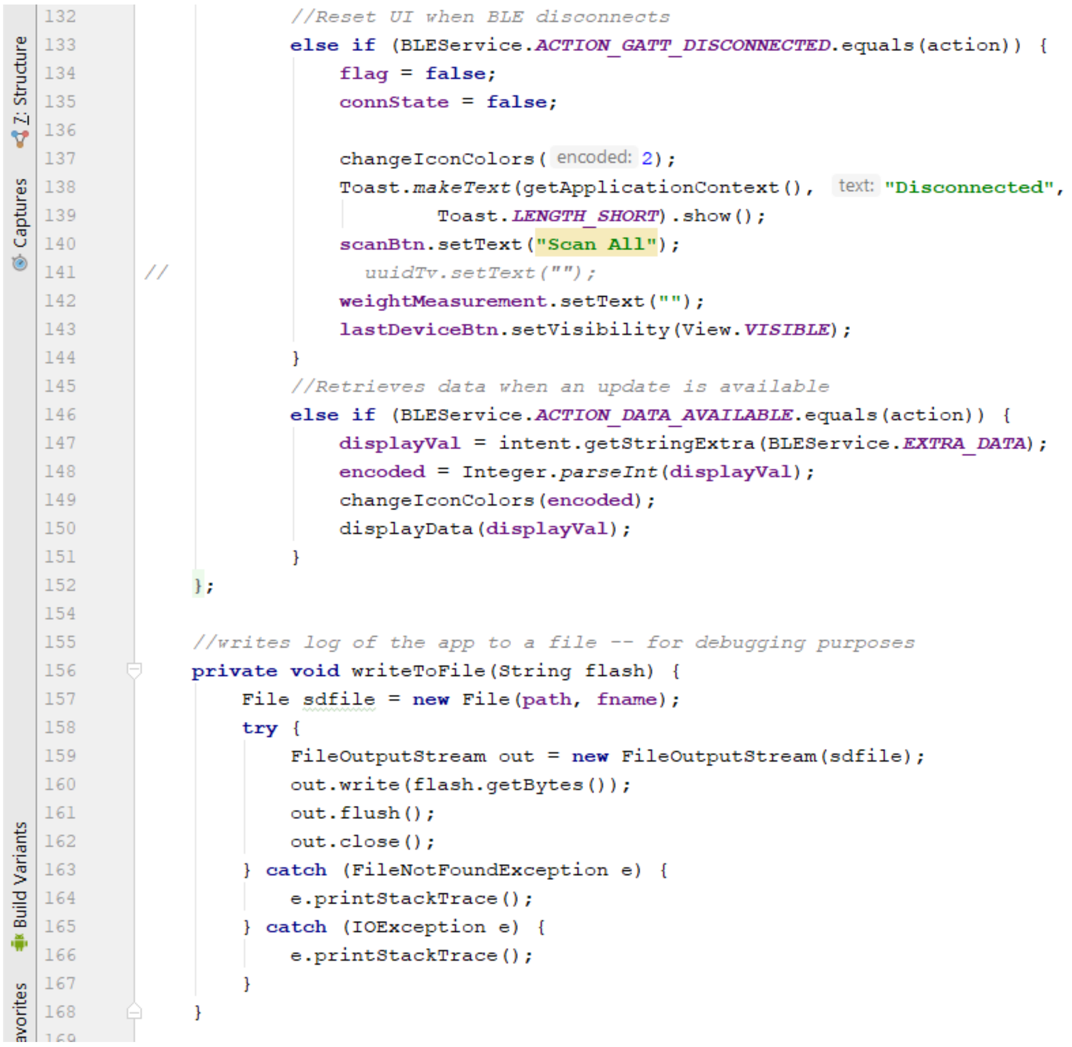
```java
package com.example.myfirstapp;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.UnsupportedEncodingException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Timer;
import java.util.TimerTask;

import android.Manifest;
import android.annotation.TargetApi;
import android.app.Activity;
import android.app.AlertDialog;
import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothManager;
import android.content.BroadcastReceiver;
import android.content.ComponentName;
import android.content.Context;
import android.content.DialogInterface;
import android.content.Intent;
import android.content.IntentFilter;
import android.content.ServiceConnection;
import android.content.pm.PackageManager;
import android.graphics.Color;
import android.os.Build;
import android.os.Bundle;
import android.os.Environment;
import android.os.IBinder;
import android.util.Log;
import android.view.Gravity;
import android.view.View;
import android.view.View.OnClickListener;
import android.view.Window;
import android.widget.Button;
import android.widget.ImageView;
import android.widget.TextView;
import android.widget.Toast;
```



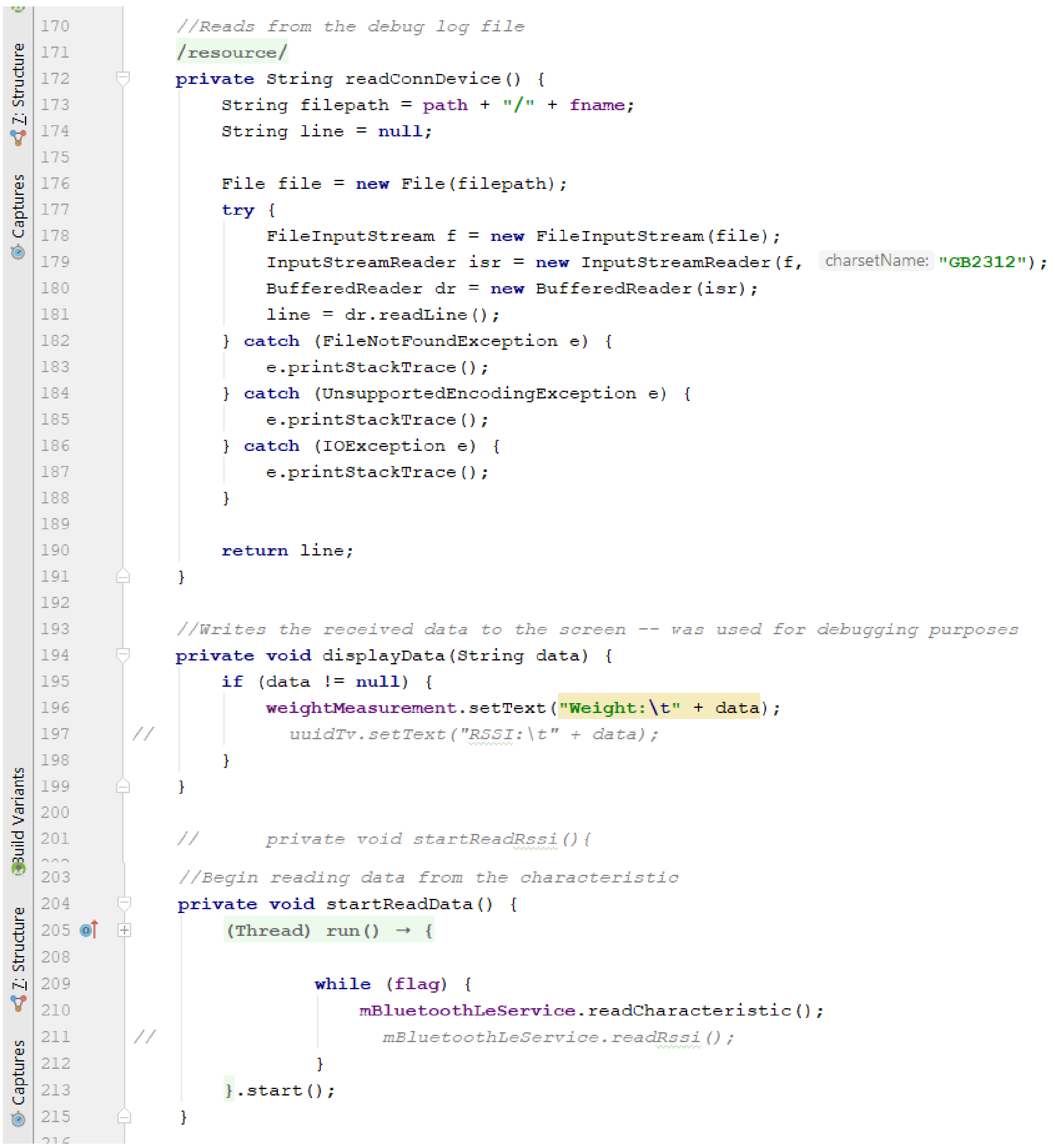
```java
public class MainActivity extends Activity {
    private final static String TAG = MainActivity.class.getSimpleName();

    private BLEService mBluetoothLeService;
    private BluetoothAdapter mBluetoothAdapter;
    public static List<BluetoothDevice> mDevice = new ArrayList<~>();

    Button lastDeviceBtn = null;
    Button scanBtn = null;
    //     TextView uuidTv = null;
    TextView lastUuid = null;
    TextView weightMeasurement = null;
    ImageView pressImg;
    ImageView hrImg;
    ImageView accelImg;
    ImageView gpsImg;
    private static final int REQUEST_ENABLE_BT = 1;
    final private int REQUEST_CODE_ASK_MULTIPLE_PERMISSIONS = 124;
    private static final long SCAN_PERIOD = 3000;
    public static final int REQUEST_CODE = 30;

    private String mDeviceAddress;
    private String mDeviceName;
    private boolean flag = true;
    private boolean connState = false;
    private Intent tempIntent;
    private int count = 0;
    private int encoded = 0;
    private String displayVal;

    String path = Environment.getExternalStorageDirectory().getAbsolutePath();
    String fname = "flash.txt";
```

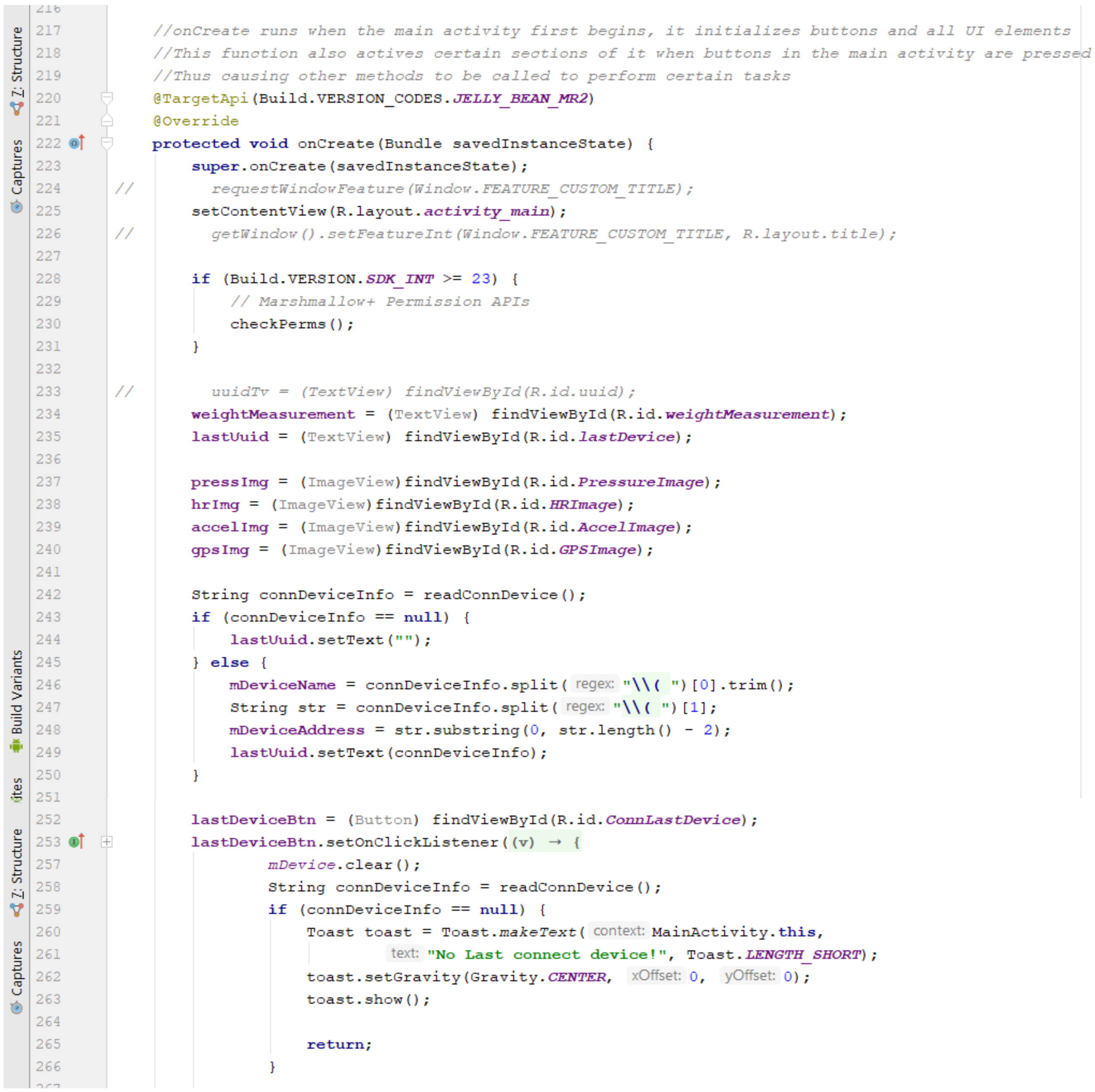


```
//Method that obtains the data stored in the GATT service on a new connection
private final ServiceConnection mServiceConnection = new ServiceConnection() {

    @Override
    public void onServiceConnected(ComponentName componentName,
                                   IBinder service) {
        mBluetoothLeService = ((BLEService.LocalBinder) service)
                .getService();
        if (!mBluetoothLeService.initialize()) {
            Log.e(TAG,  msg: "Unable to initialize Bluetooth");
            finish();
        }
    }

    @Override
    public void onServiceDisconnected(ComponentName componentName) {
        mBluetoothLeService = null;
    }
};

//Method that receives updates from characteristic when the data changes
private final BroadcastReceiver mGattUpdateReceiver = (context, intent) → {
        final String action = intent.getAction();

        //obtains the characteristic when first connected
        if (BLEService.ACTION_GATT_CONNECTED.equals(action)) {
            tempIntent = intent;
            flag = true;
            connState = true;

            Toast.makeText(getApplicationContext(),  text: "Connected",
                    Toast.LENGTH_SHORT).show();
            writeToFile( flash: mDeviceName + " ( " + mDeviceAddress + " )");
            lastUuid.setText(mDeviceName + " ( " + mDeviceAddress + " )");
            lastDeviceBtn.setVisibility(View.GONE);
            scanBtn.setText("Disconnect");
            if(count == 0) {
                try {
                    //set time in mili
                    Thread.sleep( millis: 10000);

                } catch (Exception e) {
                    e.printStackTrace();
                }
                count++;
            }
            startReadData();
            //startReadRssi();
        }
```

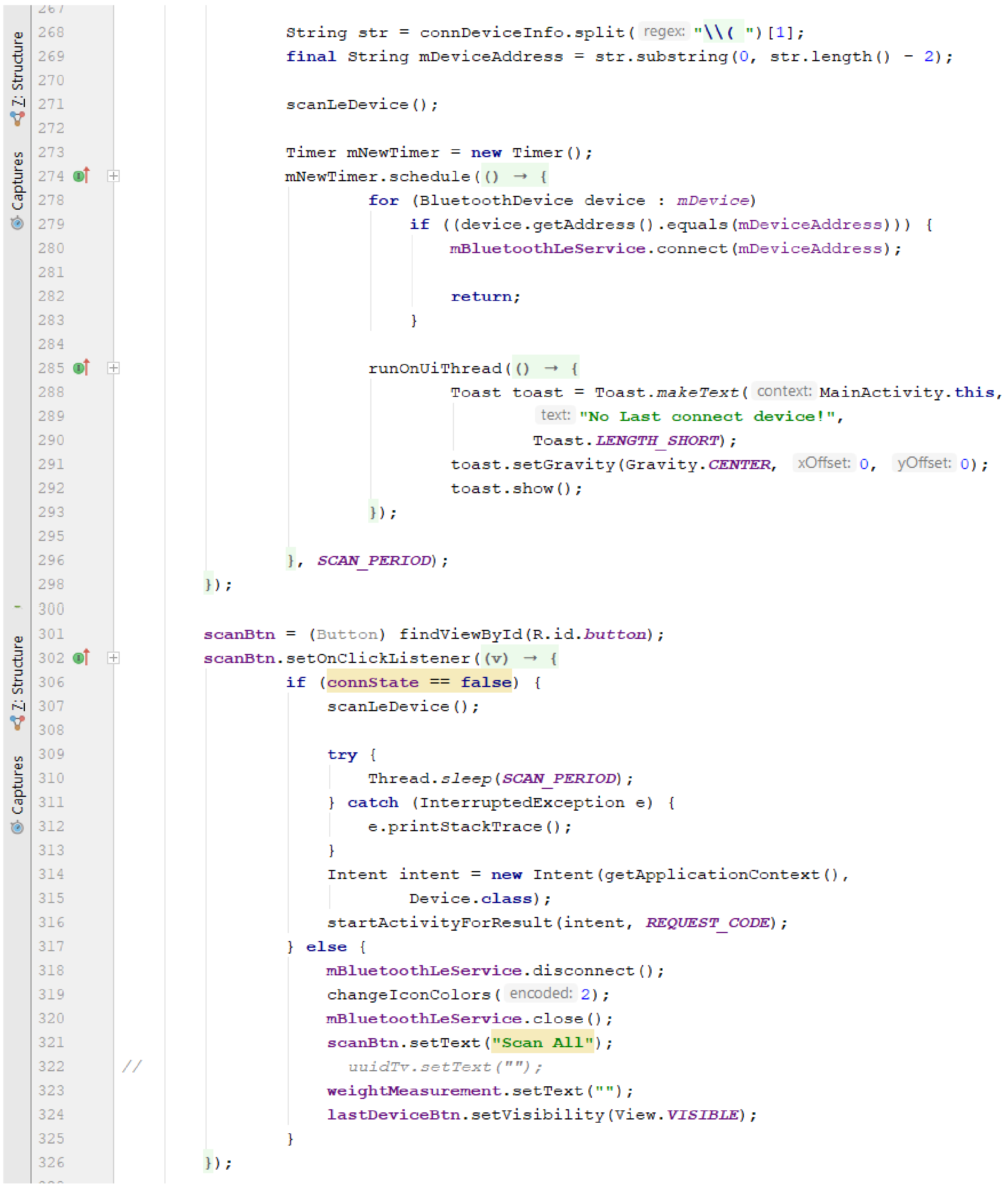


```
        //Reset UI when BLE disconnects
        else if (BLEService.ACTION_GATT_DISCONNECTED.equals(action)) {
            flag = false;
            connState = false;

            changeIconColors( encoded: 2);
            Toast.makeText(getApplicationContext(),  text: "Disconnected",
                    Toast.LENGTH_SHORT).show();
            scanBtn.setText("Scan All");
//                uuidTv.setText("");
            weightMeasurement.setText("");
            lastDeviceBtn.setVisibility(View.VISIBLE);
        }
        //Retrieves data when an update is available
        else if (BLEService.ACTION_DATA_AVAILABLE.equals(action)) {
            displayVal = intent.getStringExtra(BLEService.EXTRA_DATA);
            encoded = Integer.parseInt(displayVal);
            changeIconColors(encoded);
            displayData(displayVal);
        }
    };

    //writes log of the app to a file -- for debugging purposes
    private void writeToFile(String flash) {
        File sdfile = new File(path, fname);
        try {
            FileOutputStream out = new FileOutputStream(sdfile);
            out.write(flash.getBytes());
            out.flush();
            out.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

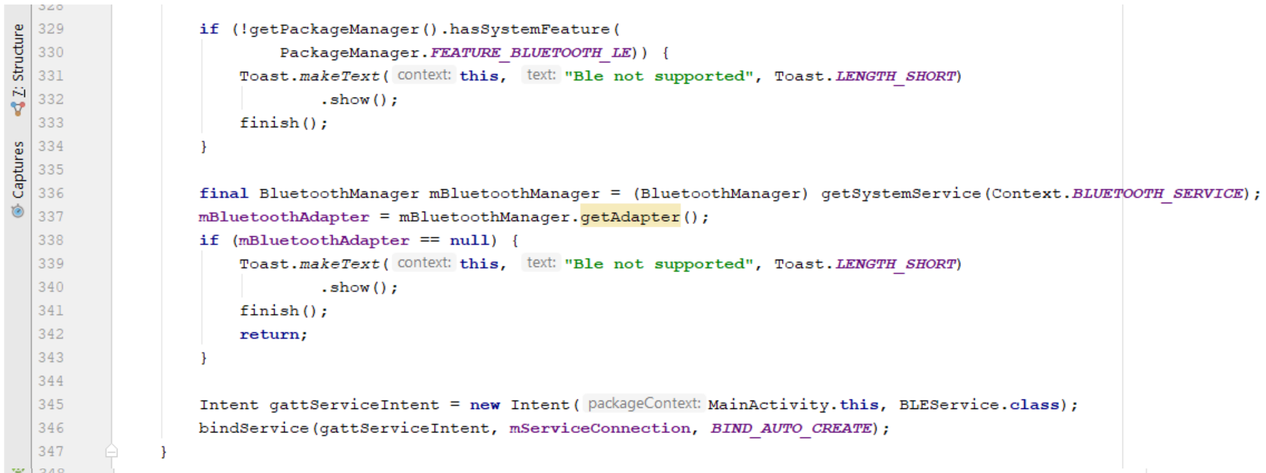

```
    //Reads from the debug log file
    /resource/
    private String readConnDevice() {
        String filepath = path + "/" + fname;
        String line = null;

        File file = new File(filepath);
        try {
            FileInputStream f = new FileInputStream(file);
            InputStreamReader isr = new InputStreamReader(f,  charsetName: "GB2312");
            BufferedReader dr = new BufferedReader(isr);
            line = dr.readLine();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        return line;
    }

    //Writes the received data to the screen -- was used for debugging purposes
    private void displayData(String data) {
        if (data != null) {
            weightMeasurement.setText("Weight:\t" + data);
//            uuidTv.setText("RSSI:\t" + data);
        }
    }

    //      private void startReadRssi(){
    //Begin reading data from the characteristic
    private void startReadData() {
        (Thread) run() → {

                while (flag) {
                    mBluetoothLeService.readCharacteristic();
//                    mBluetoothLeService.readRssi();
                }
        }.start();
    }
```



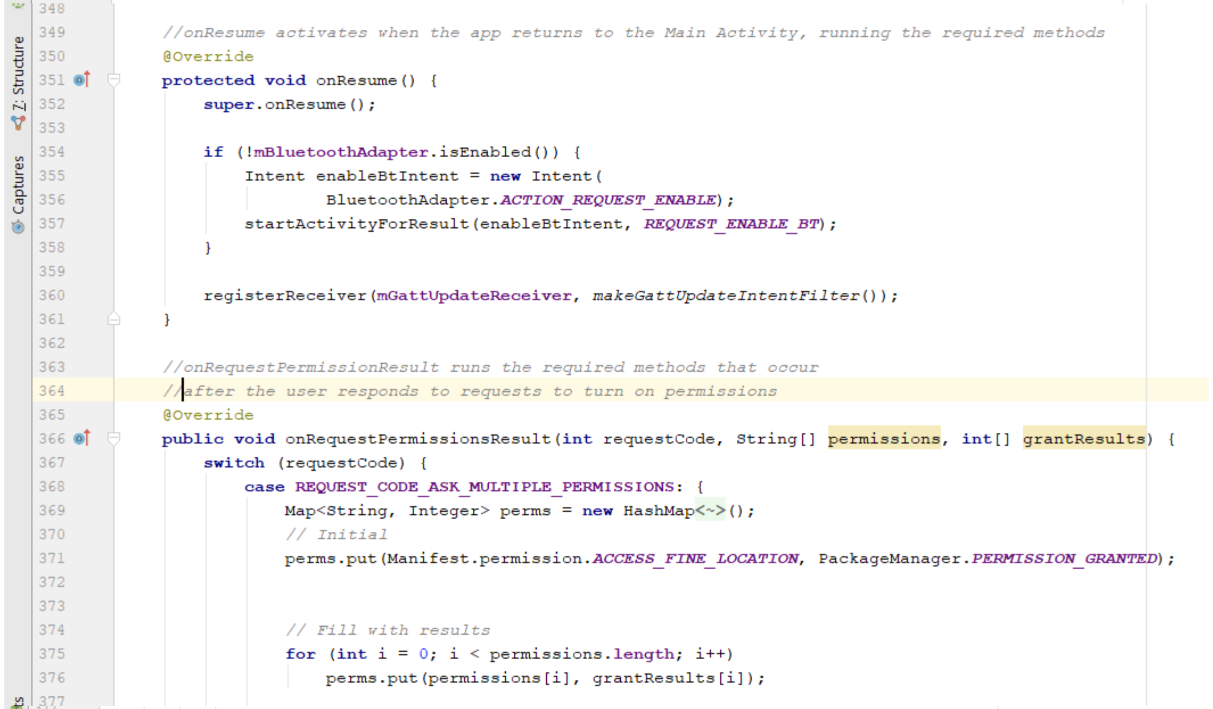
```
    //onCreate runs when the main activity first begins, it initializes buttons and all UI elements
    //This function also actives certain sections of it when buttons in the main activity are pressed
    //Thus causing other methods to be called to perform certain tasks
    @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR2)
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
//        requestWindowFeature(Window.FEATURE_CUSTOM_TITLE);
        setContentView(R.layout.activity_main);
//        getWindow().setFeatureInt(Window.FEATURE_CUSTOM_TITLE, R.layout.title);

        if (Build.VERSION.SDK_INT >= 23) {
            // Marshmallow+ Permission APIs
            checkPerms();
        }

//        uuidTv = (TextView) findViewById(R.id.uuid);
        weightMeasurement = (TextView) findViewById(R.id.weightMeasurement);
        lastUuid = (TextView) findViewById(R.id.lastDevice);

        pressImg = (ImageView)findViewById(R.id.PressureImage);
        hrImg = (ImageView)findViewById(R.id.HRImage);
        accelImg = (ImageView)findViewById(R.id.AccelImage);
        gpsImg = (ImageView)findViewById(R.id.GPSImage);

        String connDeviceInfo = readConnDevice();
        if (connDeviceInfo == null) {
            lastUuid.setText("");
        } else {
            mDeviceName = connDeviceInfo.split( regex: "\\( ")[0].trim();
            String str = connDeviceInfo.split( regex: "\\( ")[1];
            mDeviceAddress = str.substring(0, str.length() - 2);
            lastUuid.setText(connDeviceInfo);
        }

        lastDeviceBtn = (Button) findViewById(R.id.ConnLastDevice);
        lastDeviceBtn.setOnClickListener((v) → {
                mDevice.clear();
                String connDeviceInfo = readConnDevice();
                if (connDeviceInfo == null) {
                    Toast toast = Toast.makeText( context: MainActivity.this,
                             text: "No Last connect device!", Toast.LENGTH_SHORT);
                    toast.setGravity(Gravity.CENTER,  xOffset: 0,  yOffset: 0);
                    toast.show();

                    return;
                }
```



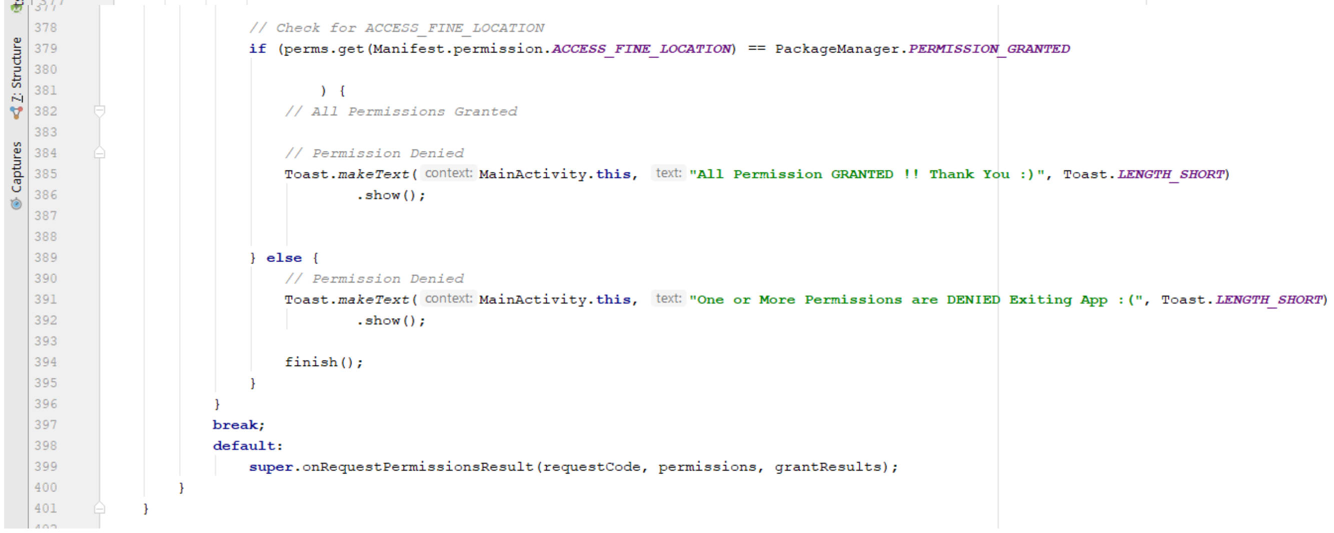
```
                String str = connDeviceInfo.split( regex: "\\( ")[1];
                final String mDeviceAddress = str.substring(0, str.length() - 2);

                scanLeDevice();

                Timer mNewTimer = new Timer();
                mNewTimer.schedule(() → {
                        for (BluetoothDevice device : mDevice)
                            if ((device.getAddress().equals(mDeviceAddress))) {
                                mBluetoothLeService.connect(mDeviceAddress);

                                return;
                            }

                        runOnUiThread(() → {
                                Toast toast = Toast.makeText( context: MainActivity.this,
                                        text: "No Last connect device!",
                                        Toast.LENGTH_SHORT);
                                toast.setGravity(Gravity.CENTER,  xOffset: 0,  yOffset: 0);
                                toast.show();
                        });

                }, SCAN_PERIOD);
        });

        scanBtn = (Button) findViewById(R.id.button);
        scanBtn.setOnClickListener((v) → {
                if (connState == false) {
                    scanLeDevice();

                    try {
                        Thread.sleep(SCAN_PERIOD);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    Intent intent = new Intent(getApplicationContext(),
                            Device.class);
                    startActivityForResult(intent, REQUEST_CODE);
                } else {
                    mBluetoothLeService.disconnect();
                    changeIconColors( encoded: 2);
                    mBluetoothLeService.close();
                    scanBtn.setText("Scan All");
//                    uuidTv.setText("");
                    weightMeasurement.setText("");
                    lastDeviceBtn.setVisibility(View.VISIBLE);
                }
        });
```



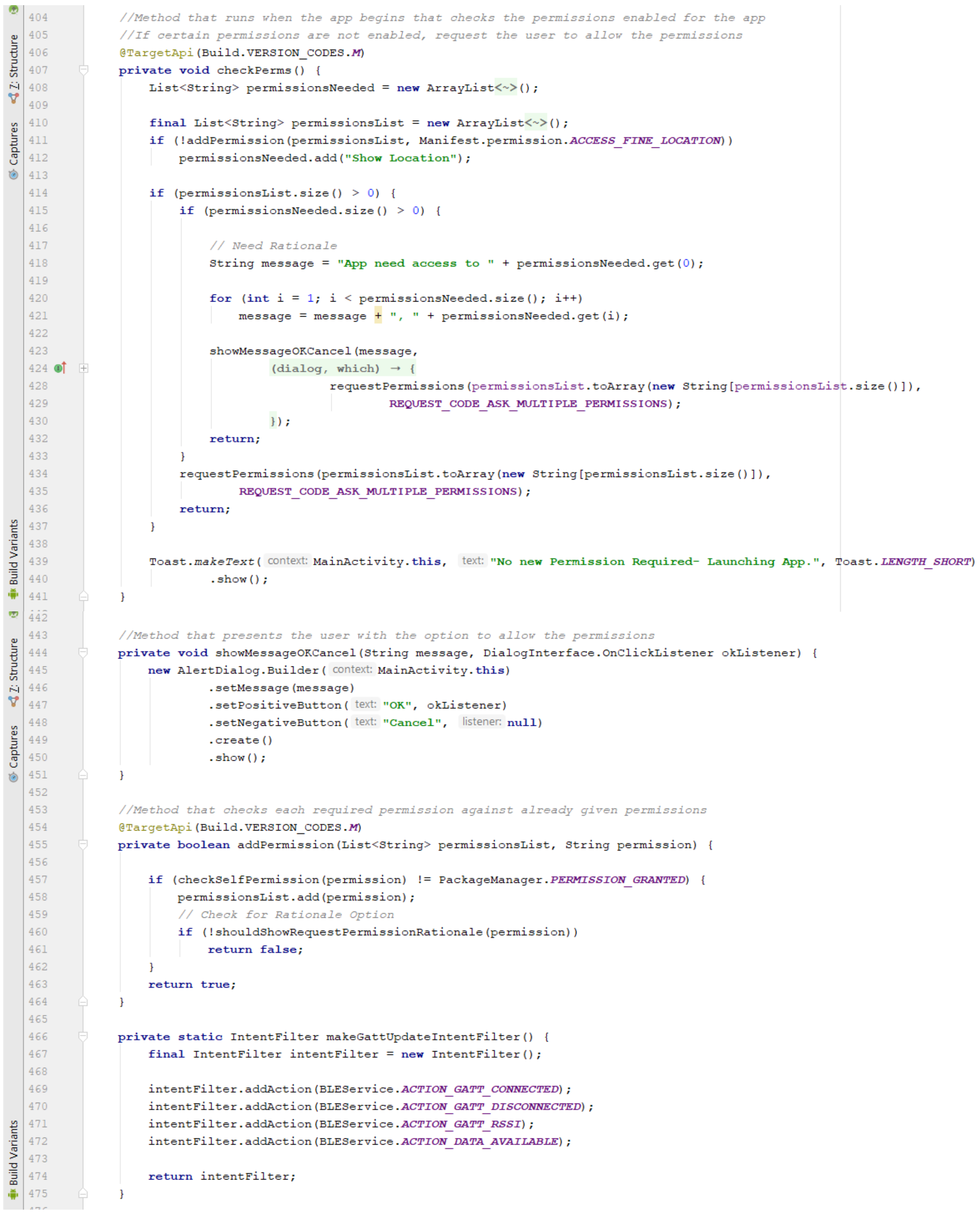
```
        if (!getPackageManager().hasSystemFeature(
                PackageManager.FEATURE_BLUETOOTH_LE)) {
            Toast.makeText( context: this,  text: "Ble not supported", Toast.LENGTH_SHORT)
                    .show();
            finish();
        }

        final BluetoothManager mBluetoothManager = (BluetoothManager) getSystemService(Context.BLUETOOTH_SERVICE);
        mBluetoothAdapter = mBluetoothManager.getAdapter();
        if (mBluetoothAdapter == null) {
            Toast.makeText( context: this,  text: "Ble not supported", Toast.LENGTH_SHORT)
                    .show();
            finish();
            return;
        }

        Intent gattServiceIntent = new Intent( packageContext: MainActivity.this, BLEService.class);
        bindService(gattServiceIntent, mServiceConnection, BIND_AUTO_CREATE);
    }
```



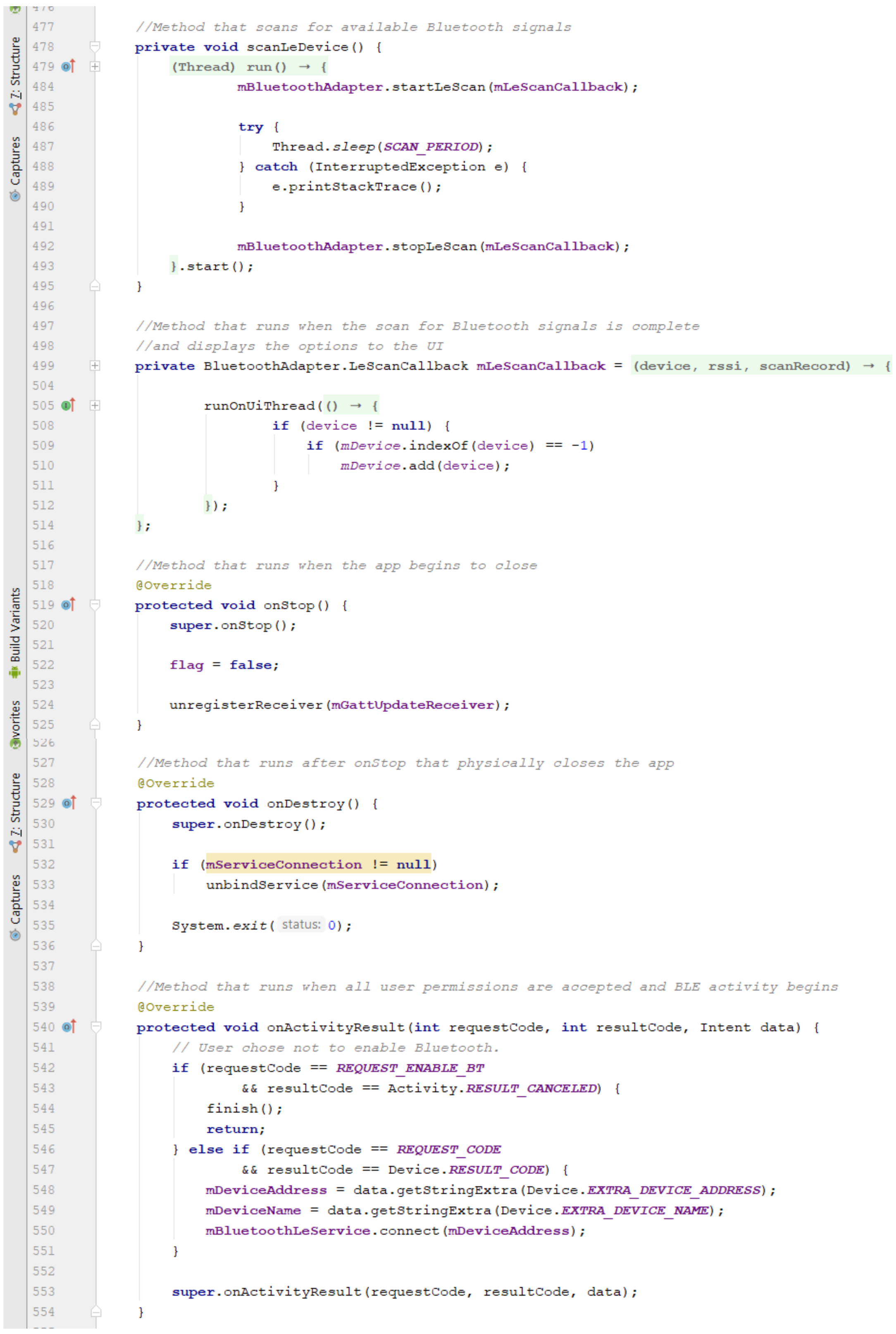
```
    //onResume activates when the app returns to the Main Activity, running the required methods
    @Override
    protected void onResume() {
        super.onResume();

        if (!mBluetoothAdapter.isEnabled()) {
            Intent enableBtIntent = new Intent(
                    BluetoothAdapter.ACTION_REQUEST_ENABLE);
            startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
        }

        registerReceiver(mGattUpdateReceiver, makeGattUpdateIntentFilter());
    }

    //onRequestPermissionResult runs the required methods that occur
    //after the user responds to requests to turn on permissions
    @Override
    public void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults) {
        switch (requestCode) {
            case REQUEST_CODE_ASK_MULTIPLE_PERMISSIONS: {
                Map<String, Integer> perms = new HashMap<~>();
                // Initial
                perms.put(Manifest.permission.ACCESS_FINE_LOCATION, PackageManager.PERMISSION_GRANTED);


                // Fill with results
                for (int i = 0; i < permissions.length; i++)
                    perms.put(permissions[i], grantResults[i]);
```



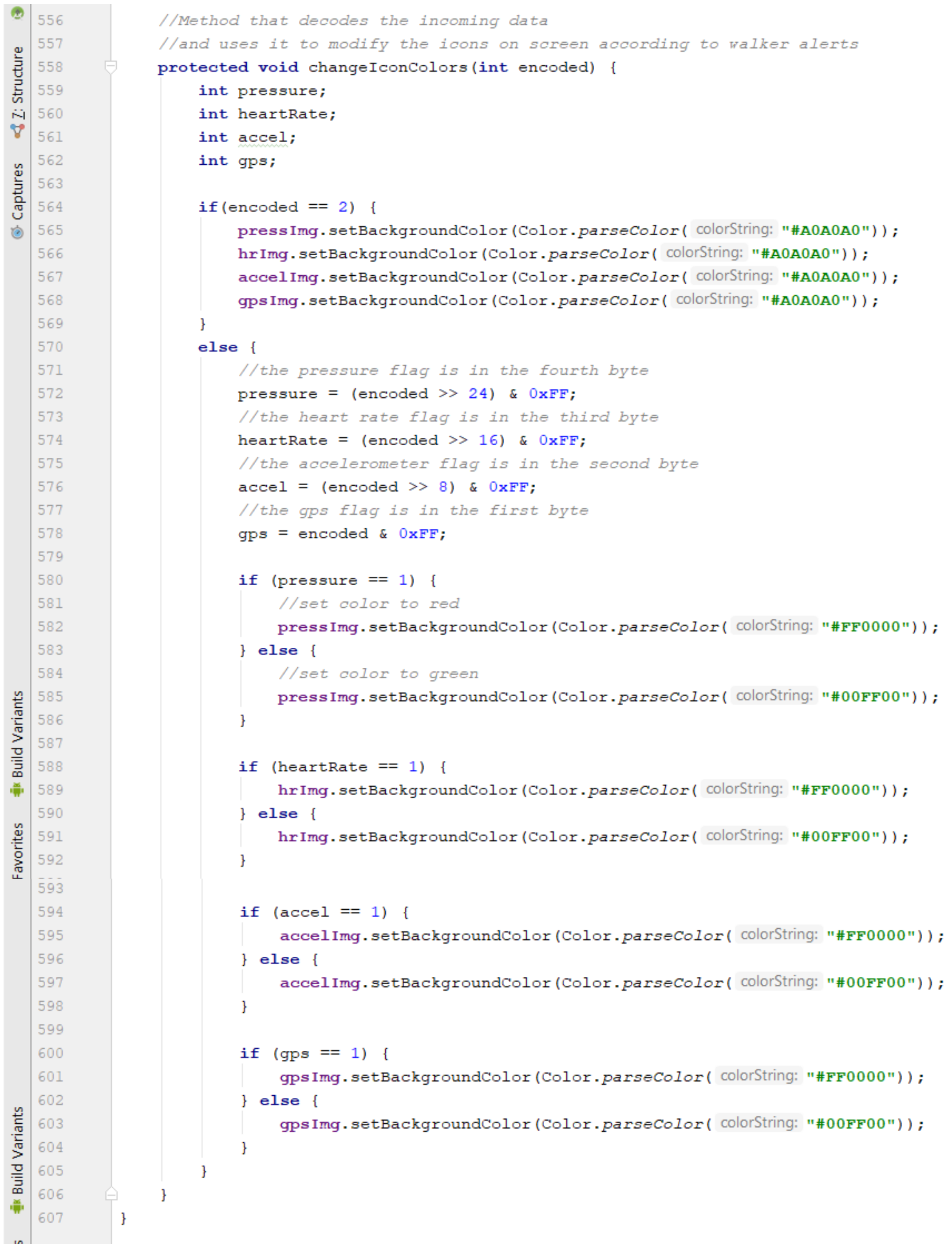
```
                // Check for ACCESS_FINE_LOCATION
                if (perms.get(Manifest.permission.ACCESS_FINE_LOCATION) == PackageManager.PERMISSION_GRANTED

                        ) {
                    // All Permissions Granted

                    // Permission Denied
                    Toast.makeText( context: MainActivity.this,  text: "All Permission GRANTED !! Thank You :)", Toast.LENGTH_SHORT)
                            .show();


                } else {
                    // Permission Denied
                    Toast.makeText( context: MainActivity.this,  text: "One or More Permissions are DENIED Exiting App :(", Toast.LENGTH_SHORT)
                            .show();

                    finish();
                }
            }
            break;
            default:
                super.onRequestPermissionsResult(requestCode, permissions, grantResults);
        }
    }
```



```java
//Method that runs when the app begins that checks the permissions enabled for the app
//If certain permissions are not enabled, request the user to allow the permissions
@TargetApi(Build.VERSION_CODES.M)
private void checkPerms() {
    List<String> permissionsNeeded = new ArrayList<~>();

    final List<String> permissionsList = new ArrayList<~>();
    if (!addPermission(permissionsList, Manifest.permission.ACCESS_FINE_LOCATION))
        permissionsNeeded.add("Show Location");

    if (permissionsList.size() > 0) {
        if (permissionsNeeded.size() > 0) {

            // Need Rationale
            String message = "App need access to " + permissionsNeeded.get(0);

            for (int i = 1; i < permissionsNeeded.size(); i++)
                message = message + ", " + permissionsNeeded.get(i);

            showMessageOKCancel(message,
                    (dialog, which) → {
                            requestPermissions(permissionsList.toArray(new String[permissionsList.size()]),
                                    REQUEST_CODE_ASK_MULTIPLE_PERMISSIONS);
                    });
            return;
        }
        requestPermissions(permissionsList.toArray(new String[permissionsList.size()]),
                REQUEST_CODE_ASK_MULTIPLE_PERMISSIONS);
        return;
    }

    Toast.makeText( context: MainActivity.this,  text: "No new Permission Required- Launching App.", Toast.LENGTH_SHORT)
            .show();
}

//Method that presents the user with the option to allow the permissions
private void showMessageOKCancel(String message, DialogInterface.OnClickListener okListener) {
    new AlertDialog.Builder( context: MainActivity.this)
            .setMessage(message)
            .setPositiveButton( text: "OK", okListener)
            .setNegativeButton( text: "Cancel",  listener: null)
            .create()
            .show();
}

//Method that checks each required permission against already given permissions
@TargetApi(Build.VERSION_CODES.M)
private boolean addPermission(List<String> permissionsList, String permission) {

    if (checkSelfPermission(permission) != PackageManager.PERMISSION_GRANTED) {
        permissionsList.add(permission);
        // Check for Rationale Option
        if (!shouldShowRequestPermissionRationale(permission))
            return false;
    }
    return true;
}

private static IntentFilter makeGattUpdateIntentFilter() {
    final IntentFilter intentFilter = new IntentFilter();

    intentFilter.addAction(BLEService.ACTION_GATT_CONNECTED);
    intentFilter.addAction(BLEService.ACTION_GATT_DISCONNECTED);
    intentFilter.addAction(BLEService.ACTION_GATT_RSSI);
    intentFilter.addAction(BLEService.ACTION_DATA_AVAILABLE);

    return intentFilter;
}
```

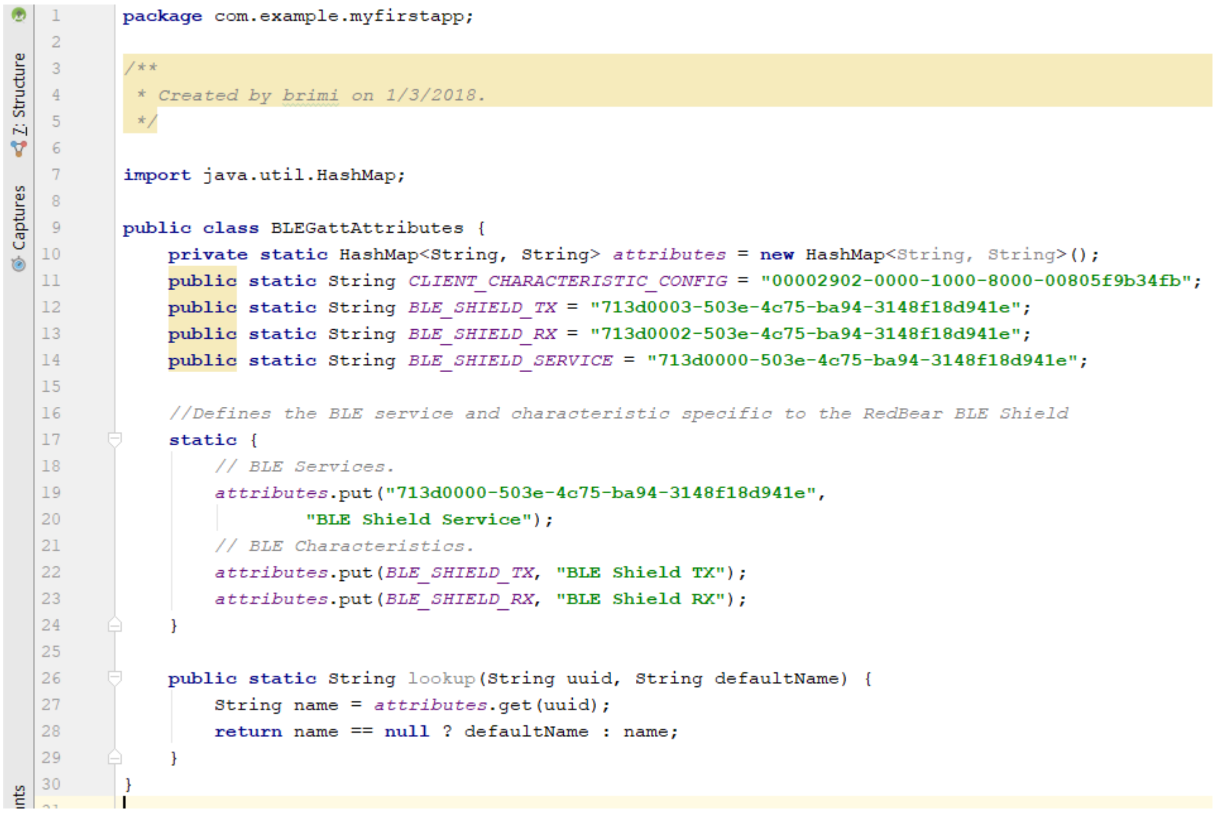


```java
//Method that scans for available Bluetooth signals
private void scanLeDevice() {
    (Thread) run() → {
            mBluetoothAdapter.startLeScan(mLeScanCallback);

            try {
                Thread.sleep(SCAN_PERIOD);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            mBluetoothAdapter.stopLeScan(mLeScanCallback);
    }.start();
}

//Method that runs when the scan for Bluetooth signals is complete
//and displays the options to the UI
private BluetoothAdapter.LeScanCallback mLeScanCallback = (device, rssi, scanRecord) → {

        runOnUiThread(() → {
                if (device != null) {
                    if (mDevice.indexOf(device) == -1)
                        mDevice.add(device);
                }
        });
};

//Method that runs when the app begins to close
@Override
protected void onStop() {
    super.onStop();

    flag = false;

    unregisterReceiver(mGattUpdateReceiver);
}

//Method that runs after onStop that physically closes the app
@Override
protected void onDestroy() {
    super.onDestroy();

    if (mServiceConnection != null)
        unbindService(mServiceConnection);

    System.exit( status: 0);
}

//Method that runs when all user permissions are accepted and BLE activity begins
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    // User chose not to enable Bluetooth.
    if (requestCode == REQUEST_ENABLE_BT
            && resultCode == Activity.RESULT_CANCELED) {
        finish();
        return;
    } else if (requestCode == REQUEST_CODE
            && resultCode == Device.RESULT_CODE) {
        mDeviceAddress = data.getStringExtra(Device.EXTRA_DEVICE_ADDRESS);
        mDeviceName = data.getStringExtra(Device.EXTRA_DEVICE_NAME);
        mBluetoothLeService.connect(mDeviceAddress);
    }

    super.onActivityResult(requestCode, resultCode, data);
}
```



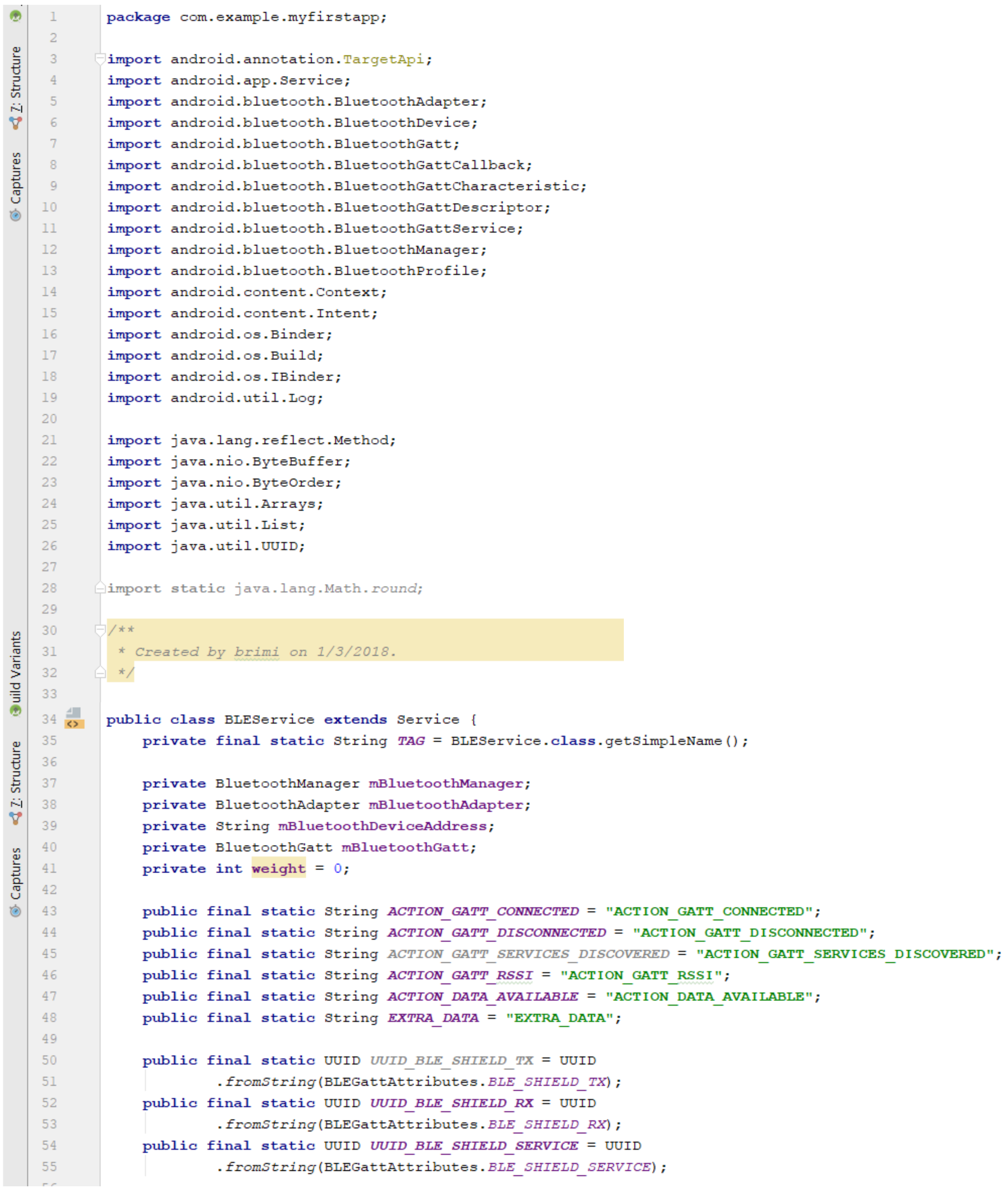
```java
    //Method that decodes the incoming data
    //and uses it to modify the icons on screen according to walker alerts
    protected void changeIconColors(int encoded) {
        int pressure;
        int heartRate;
        int accel;
        int gps;

        if(encoded == 2) {
            pressImg.setBackgroundColor(Color.parseColor( colorString: "#A0A0A0"));
            hrImg.setBackgroundColor(Color.parseColor( colorString: "#A0A0A0"));
            accelImg.setBackgroundColor(Color.parseColor( colorString: "#A0A0A0"));
            gpsImg.setBackgroundColor(Color.parseColor( colorString: "#A0A0A0"));
        }
        else {
            //the pressure flag is in the fourth byte
            pressure = (encoded >> 24) & 0xFF;
            //the heart rate flag is in the third byte
            heartRate = (encoded >> 16) & 0xFF;
            //the accelerometer flag is in the second byte
            accel = (encoded >> 8) & 0xFF;
            //the gps flag is in the first byte
            gps = encoded & 0xFF;

            if (pressure == 1) {
                //set color to red
                pressImg.setBackgroundColor(Color.parseColor( colorString: "#FF0000"));
            } else {
                //set color to green
                pressImg.setBackgroundColor(Color.parseColor( colorString: "#00FF00"));
            }

            if (heartRate == 1) {
                hrImg.setBackgroundColor(Color.parseColor( colorString: "#FF0000"));
            } else {
                hrImg.setBackgroundColor(Color.parseColor( colorString: "#00FF00"));
            }

            if (accel == 1) {
                accelImg.setBackgroundColor(Color.parseColor( colorString: "#FF0000"));
            } else {
                accelImg.setBackgroundColor(Color.parseColor( colorString: "#00FF00"));
            }

            if (gps == 1) {
                gpsImg.setBackgroundColor(Color.parseColor( colorString: "#FF0000"));
            } else {
                gpsImg.setBackgroundColor(Color.parseColor( colorString: "#00FF00"));
            }
        }
    }
}
```

BLEGattAttributes.java

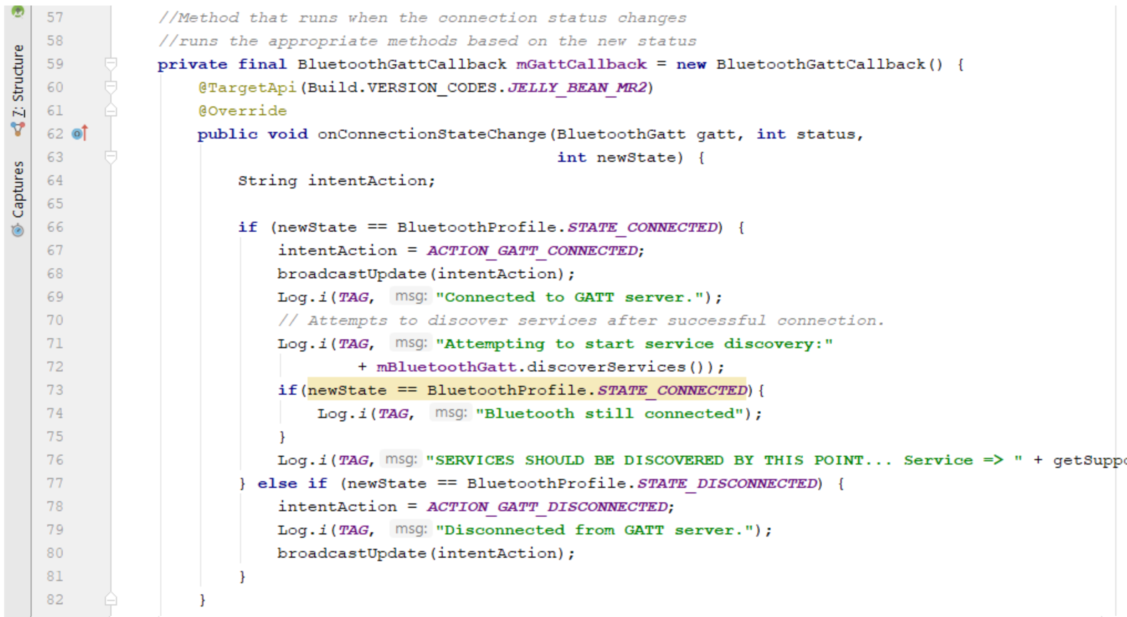


```
package com.example.myfirstapp;

/**
 * Created by brimi on 1/3/2018.
 */

import java.util.HashMap;

public class BLEGattAttributes {
    private static HashMap<String, String> attributes = new HashMap<String, String>();
    public static String CLIENT_CHARACTERISTIC_CONFIG = "00002902-0000-1000-8000-00805f9b34fb";
    public static String BLE_SHIELD_TX = "713d0003-503e-4c75-ba94-3148f18d941e";
    public static String BLE_SHIELD_RX = "713d0002-503e-4c75-ba94-3148f18d941e";
    public static String BLE_SHIELD_SERVICE = "713d0000-503e-4c75-ba94-3148f18d941e";

    //Defines the BLE service and characteristic specific to the RedBear BLE Shield
    static {
        // BLE Services.
        attributes.put("713d0000-503e-4c75-ba94-3148f18d941e",
                "BLE Shield Service");
        // BLE Characteristics.
        attributes.put(BLE_SHIELD_TX, "BLE Shield TX");
        attributes.put(BLE_SHIELD_RX, "BLE Shield RX");
    }

    public static String lookup(String uuid, String defaultName) {
        String name = attributes.get(uuid);
        return name == null ? defaultName : name;
    }
}
```

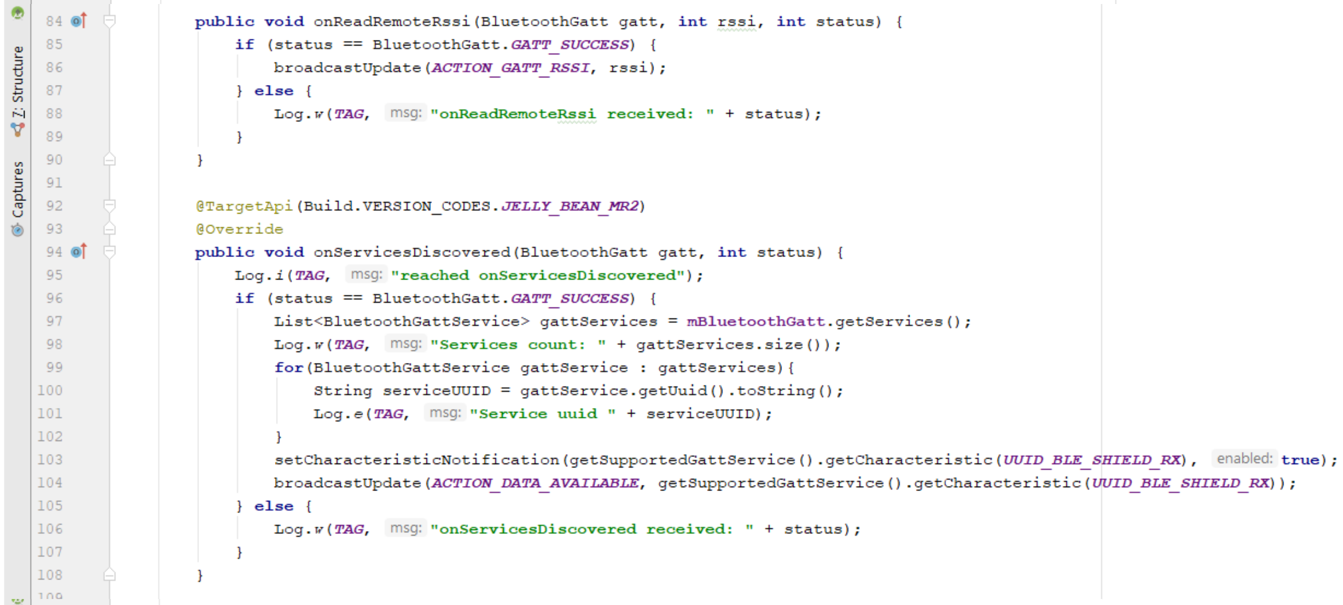
BLEService.java



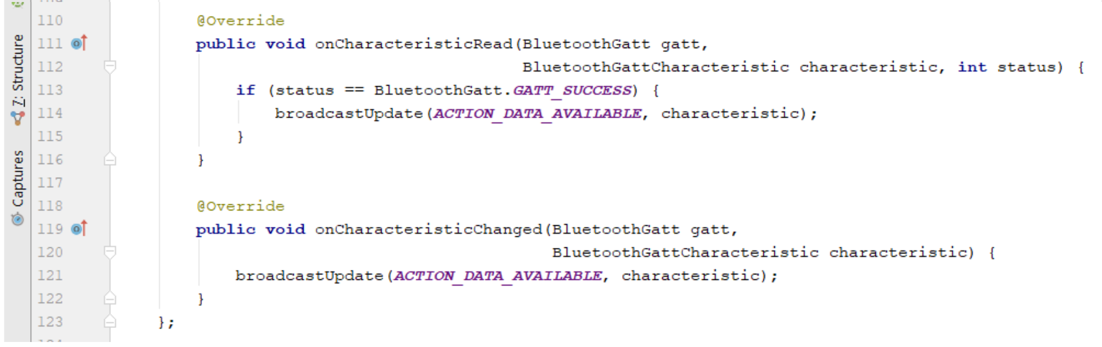
```java
package com.example.myfirstapp;

import android.annotation.TargetApi;
import android.app.Service;
import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothGatt;
import android.bluetooth.BluetoothGattCallback;
import android.bluetooth.BluetoothGattCharacteristic;
import android.bluetooth.BluetoothGattDescriptor;
import android.bluetooth.BluetoothGattService;
import android.bluetooth.BluetoothManager;
import android.bluetooth.BluetoothProfile;
import android.content.Context;
import android.content.Intent;
import android.os.Binder;
import android.os.Build;
import android.os.IBinder;
import android.util.Log;

import java.lang.reflect.Method;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.util.Arrays;
import java.util.List;
import java.util.UUID;

import static java.lang.Math.round;

/**
 * Created by brimi on 1/3/2018.
 */

public class BLEService extends Service {
    private final static String TAG = BLEService.class.getSimpleName();

    private BluetoothManager mBluetoothManager;
    private BluetoothAdapter mBluetoothAdapter;
    private String mBluetoothDeviceAddress;
    private BluetoothGatt mBluetoothGatt;
    private int weight = 0;

    public final static String ACTION_GATT_CONNECTED = "ACTION_GATT_CONNECTED";
    public final static String ACTION_GATT_DISCONNECTED = "ACTION_GATT_DISCONNECTED";
    public final static String ACTION_GATT_SERVICES_DISCOVERED = "ACTION_GATT_SERVICES_DISCOVERED";
    public final static String ACTION_GATT_RSSI = "ACTION_GATT_RSSI";
    public final static String ACTION_DATA_AVAILABLE = "ACTION_DATA_AVAILABLE";
    public final static String EXTRA_DATA = "EXTRA_DATA";

    public final static UUID UUID_BLE_SHIELD_TX = UUID
            .fromString(BLEGattAttributes.BLE_SHIELD_TX);
    public final static UUID UUID_BLE_SHIELD_RX = UUID
            .fromString(BLEGattAttributes.BLE_SHIELD_RX);
    public final static UUID UUID_BLE_SHIELD_SERVICE = UUID
            .fromString(BLEGattAttributes.BLE_SHIELD_SERVICE);
```



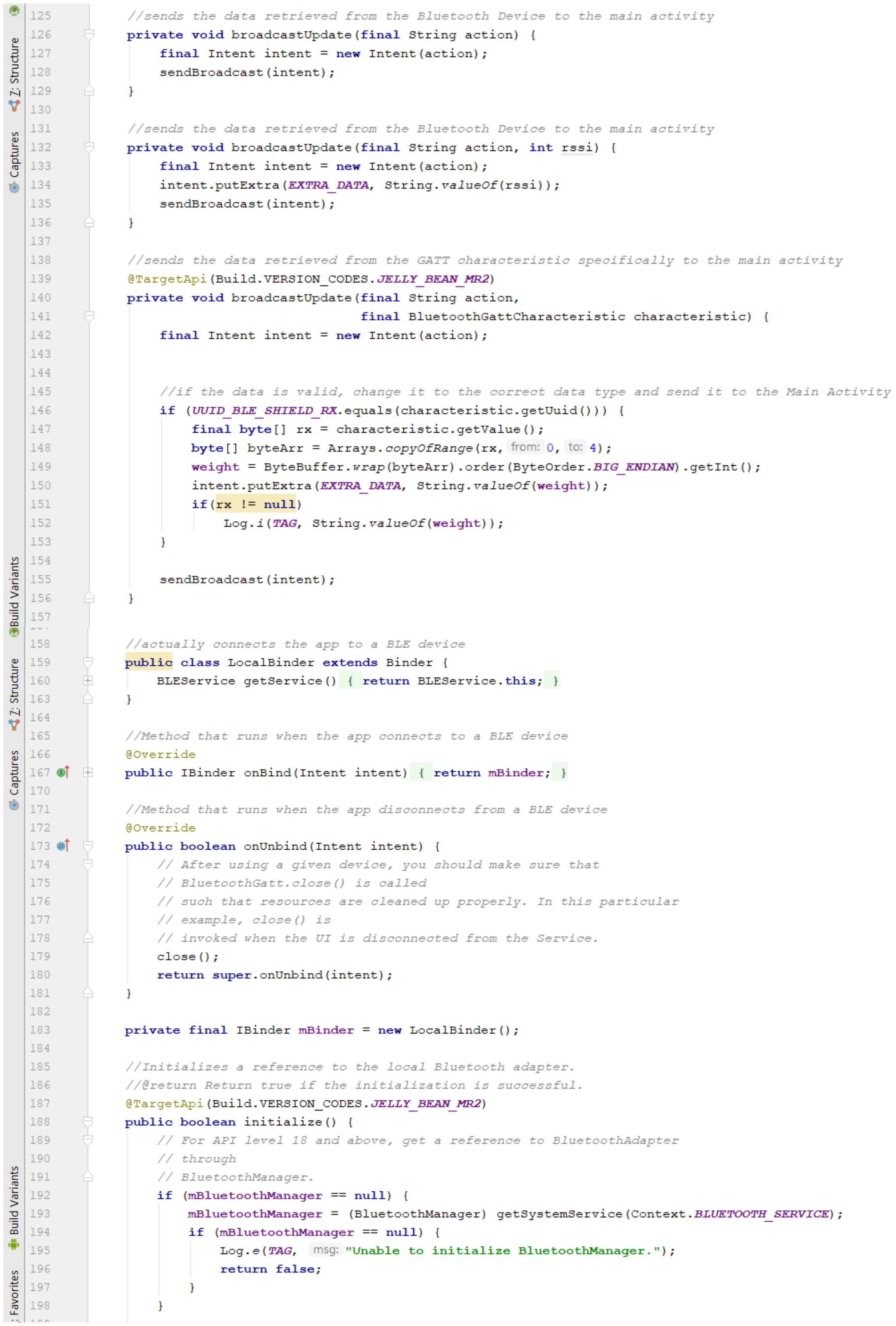
```java
//Method that runs when the connection status changes
//runs the appropriate methods based on the new status
private final BluetoothGattCallback mGattCallback = new BluetoothGattCallback() {
    @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR2)
    @Override
    public void onConnectionStateChange(BluetoothGatt gatt, int status,
                                        int newState) {
        String intentAction;

        if (newState == BluetoothProfile.STATE_CONNECTED) {
            intentAction = ACTION_GATT_CONNECTED;
            broadcastUpdate(intentAction);
            Log.i(TAG,  msg: "Connected to GATT server.");
            // Attempts to discover services after successful connection.
            Log.i(TAG,  msg: "Attempting to start service discovery:"
                    + mBluetoothGatt.discoverServices());
            if(newState == BluetoothProfile.STATE_CONNECTED){
                Log.i(TAG,  msg: "Bluetooth still connected");
            }
            Log.i(TAG, msg: "SERVICES SHOULD BE DISCOVERED BY THIS POINT... Service => " + getSupportedGattService());
        } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
            intentAction = ACTION_GATT_DISCONNECTED;
            Log.i(TAG,  msg: "Disconnected from GATT server.");
            broadcastUpdate(intentAction);
        }
    }
```



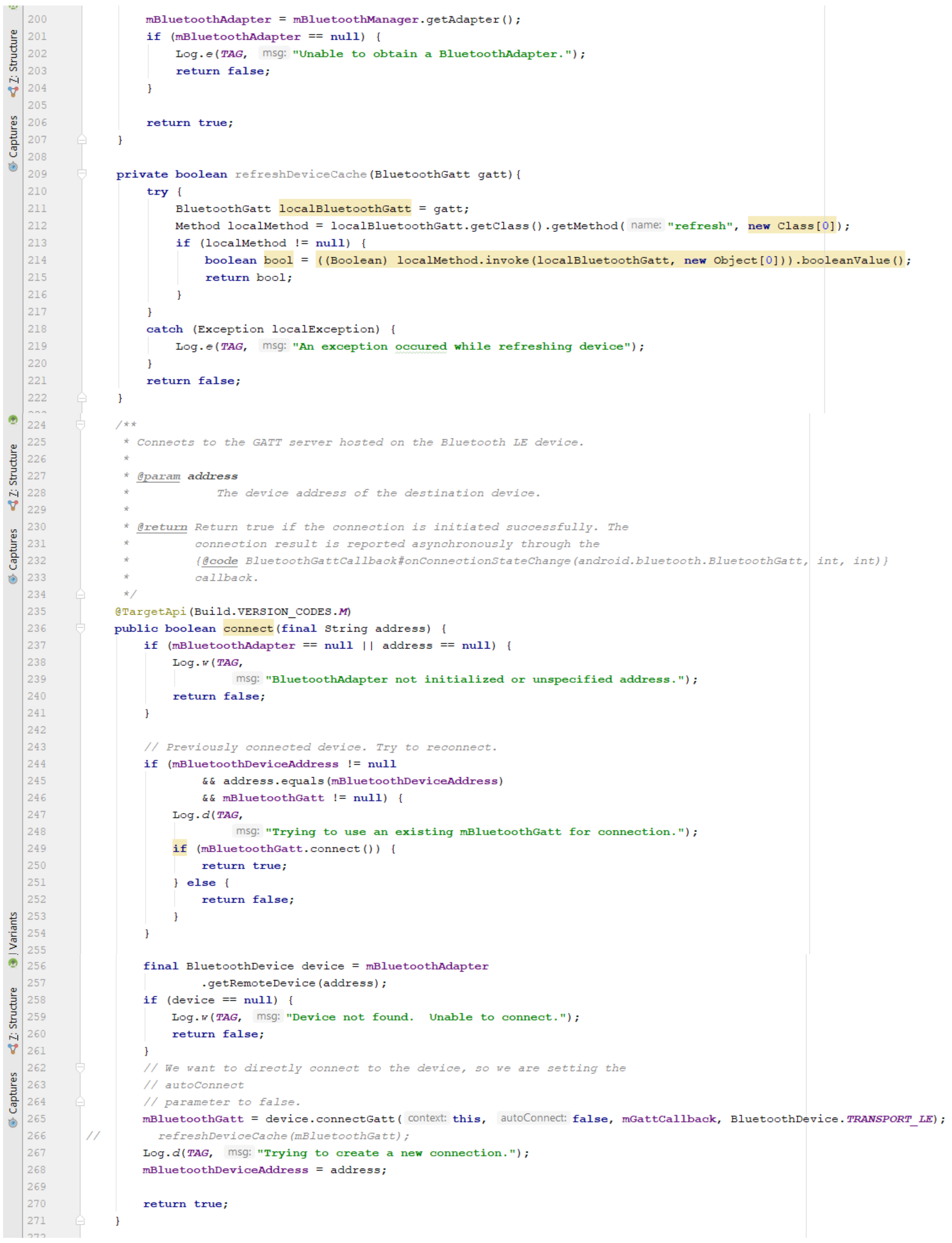
```java
    public void onReadRemoteRssi(BluetoothGatt gatt, int rssi, int status) {
        if (status == BluetoothGatt.GATT_SUCCESS) {
            broadcastUpdate(ACTION_GATT_RSSI, rssi);
        } else {
            Log.w(TAG,  msg: "onReadRemoteRssi received: " + status);
        }
    }

    @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR2)
    @Override
    public void onServicesDiscovered(BluetoothGatt gatt, int status) {
        Log.i(TAG,  msg: "reached onServicesDiscovered");
        if (status == BluetoothGatt.GATT_SUCCESS) {
            List<BluetoothGattService> gattServices = mBluetoothGatt.getServices();
            Log.w(TAG,  msg: "Services count: " + gattServices.size());
            for(BluetoothGattService gattService : gattServices){
                String serviceUUID = gattService.getUuid().toString();
                Log.e(TAG,  msg: "Service uuid " + serviceUUID);
            }
            setCharacteristicNotification(getSupportedGattService().getCharacteristic(UUID_BLE_SHIELD_RX),  enabled: true);
            broadcastUpdate(ACTION_DATA_AVAILABLE, getSupportedGattService().getCharacteristic(UUID_BLE_SHIELD_RX));
        } else {
            Log.w(TAG,  msg: "onServicesDiscovered received: " + status);
        }
    }
```



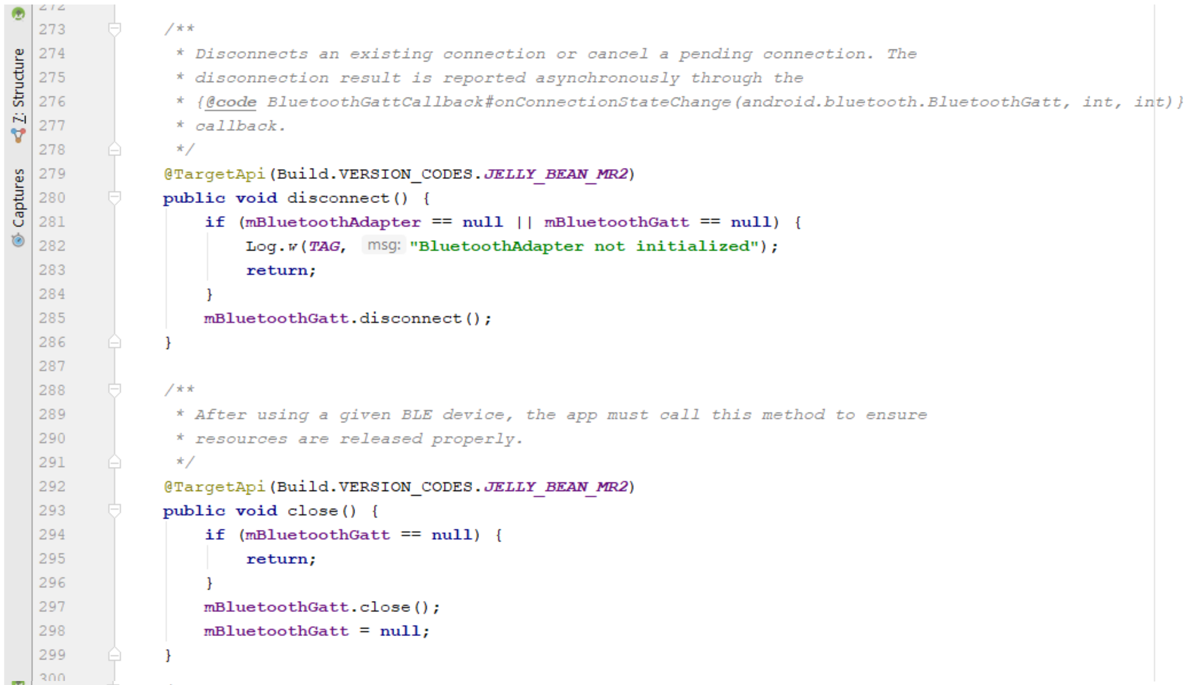
```java
    @Override
    public void onCharacteristicRead(BluetoothGatt gatt,
                                     BluetoothGattCharacteristic characteristic, int status) {
        if (status == BluetoothGatt.GATT_SUCCESS) {
            broadcastUpdate(ACTION_DATA_AVAILABLE, characteristic);
        }
    }

    @Override
    public void onCharacteristicChanged(BluetoothGatt gatt,
                                        BluetoothGattCharacteristic characteristic) {
        broadcastUpdate(ACTION_DATA_AVAILABLE, characteristic);
    }
};
```



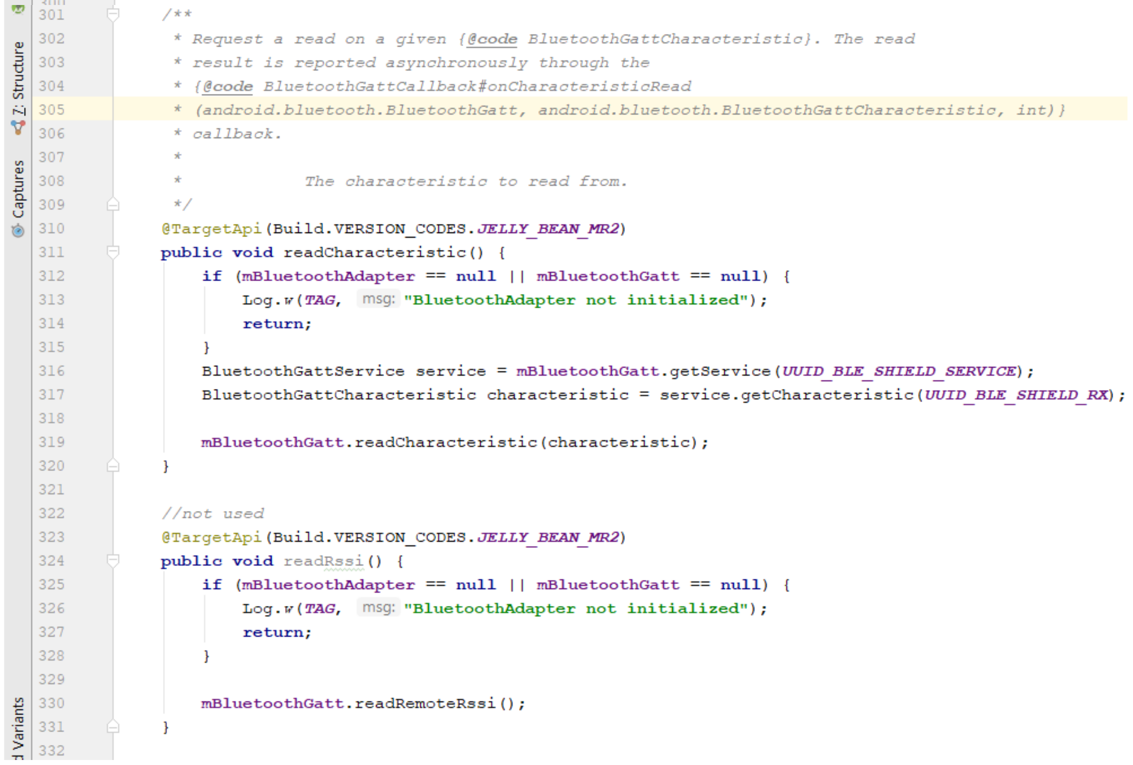
```
    //sends the data retrieved from the Bluetooth Device to the main activity
    private void broadcastUpdate(final String action) {
        final Intent intent = new Intent(action);
        sendBroadcast(intent);
    }

    //sends the data retrieved from the Bluetooth Device to the main activity
    private void broadcastUpdate(final String action, int rssi) {
        final Intent intent = new Intent(action);
        intent.putExtra(EXTRA_DATA, String.valueOf(rssi));
        sendBroadcast(intent);
    }

    //sends the data retrieved from the GATT characteristic specifically to the main activity
    @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR2)
    private void broadcastUpdate(final String action,
                                 final BluetoothGattCharacteristic characteristic) {
        final Intent intent = new Intent(action);


        //if the data is valid, change it to the correct data type and send it to the Main Activity
        if (UUID_BLE_SHIELD_RX.equals(characteristic.getUuid())) {
            final byte[] rx = characteristic.getValue();
            byte[] byteArr = Arrays.copyOfRange(rx, from: 0, to: 4);
            weight = ByteBuffer.wrap(byteArr).order(ByteOrder.BIG_ENDIAN).getInt();
            intent.putExtra(EXTRA_DATA, String.valueOf(weight));
            if(rx != null)
                Log.i(TAG, String.valueOf(weight));
        }

        sendBroadcast(intent);
    }

    //actually connects the app to a BLE device
    public class LocalBinder extends Binder {
        BLEService getService() { return BLEService.this; }
    }

    //Method that runs when the app connects to a BLE device
    @Override
    public IBinder onBind(Intent intent) { return mBinder; }

    //Method that runs when the app disconnects from a BLE device
    @Override
    public boolean onUnbind(Intent intent) {
        // After using a given device, you should make sure that
        // BluetoothGatt.close() is called
        // such that resources are cleaned up properly. In this particular
        // example, close() is
        // invoked when the UI is disconnected from the Service.
        close();
        return super.onUnbind(intent);
    }

    private final IBinder mBinder = new LocalBinder();

    //Initializes a reference to the local Bluetooth adapter.
    //@return Return true if the initialization is successful.
    @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR2)
    public boolean initialize() {
        // For API level 18 and above, get a reference to BluetoothAdapter
        // through
        // BluetoothManager.
        if (mBluetoothManager == null) {
            mBluetoothManager = (BluetoothManager) getSystemService(Context.BLUETOOTH_SERVICE);
            if (mBluetoothManager == null) {
                Log.e(TAG, msg: "Unable to initialize BluetoothManager.");
                return false;
            }
        }
```



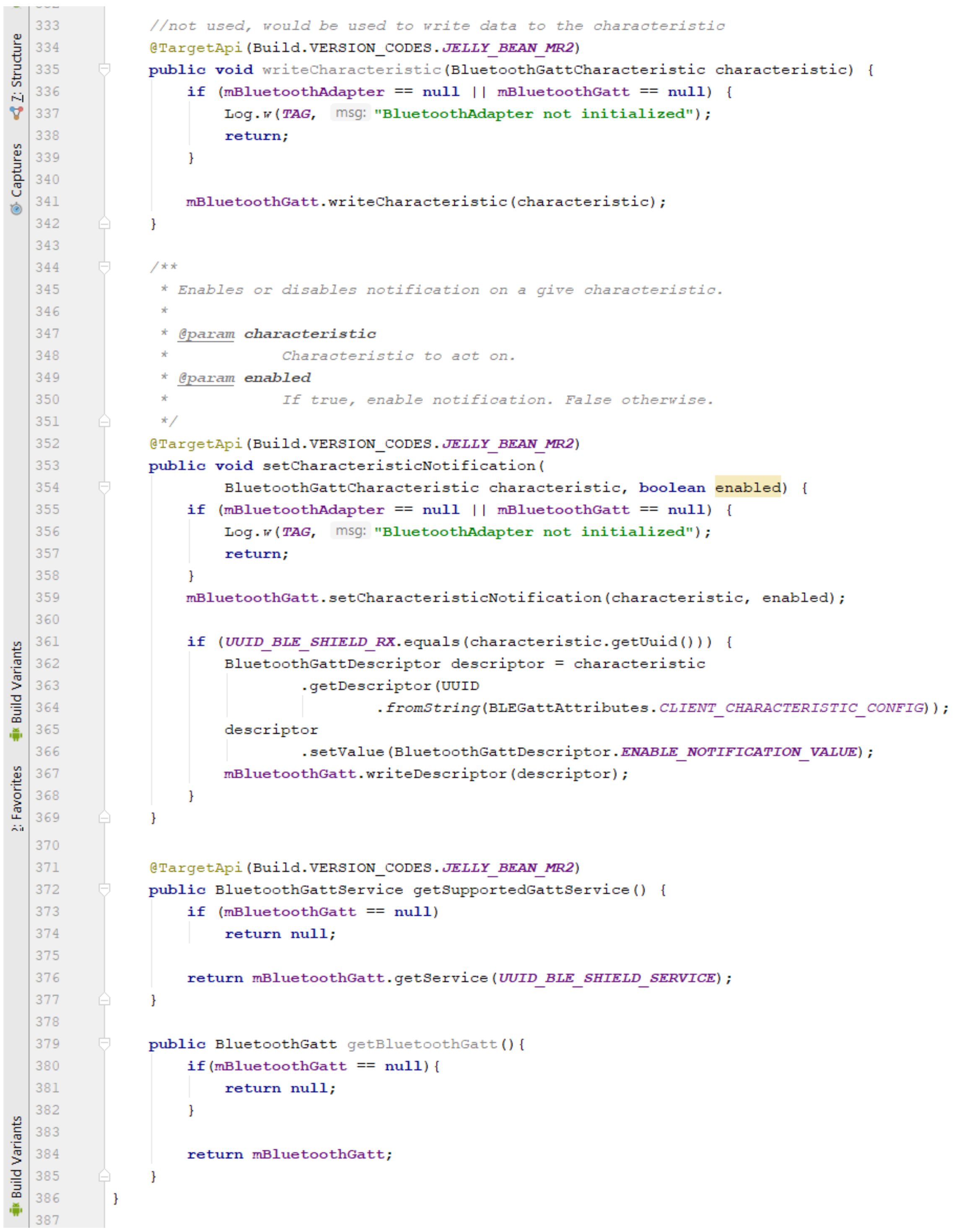
```java
        mBluetoothAdapter = mBluetoothManager.getAdapter();
        if (mBluetoothAdapter == null) {
            Log.e(TAG,  msg: "Unable to obtain a BluetoothAdapter.");
            return false;
        }

        return true;
    }

    private boolean refreshDeviceCache(BluetoothGatt gatt){
        try {
            BluetoothGatt localBluetoothGatt = gatt;
            Method localMethod = localBluetoothGatt.getClass().getMethod( name: "refresh", new Class[0]);
            if (localMethod != null) {
                boolean bool = ((Boolean) localMethod.invoke(localBluetoothGatt, new Object[0])).booleanValue();
                return bool;
            }
        }
        catch (Exception localException) {
            Log.e(TAG,  msg: "An exception occured while refreshing device");
        }
        return false;
    }

    /**
     * Connects to the GATT server hosted on the Bluetooth LE device.
     *
     * @param address
     *            The device address of the destination device.
     *
     * @return Return true if the connection is initiated successfully. The
     *         connection result is reported asynchronously through the
     *         {@code BluetoothGattCallback#onConnectionStateChange(android.bluetooth.BluetoothGatt, int, int)}
     *         callback.
     */
    @TargetApi(Build.VERSION_CODES.M)
    public boolean connect(final String address) {
        if (mBluetoothAdapter == null || address == null) {
            Log.w(TAG,
                     msg: "BluetoothAdapter not initialized or unspecified address.");
            return false;
        }

        // Previously connected device. Try to reconnect.
        if (mBluetoothDeviceAddress != null
                && address.equals(mBluetoothDeviceAddress)
                && mBluetoothGatt != null) {
            Log.d(TAG,
                     msg: "Trying to use an existing mBluetoothGatt for connection.");
            if (mBluetoothGatt.connect()) {
                return true;
            } else {
                return false;
            }
        }

        final BluetoothDevice device = mBluetoothAdapter
                .getRemoteDevice(address);
        if (device == null) {
            Log.w(TAG,  msg: "Device not found.  Unable to connect.");
            return false;
        }
        // We want to directly connect to the device, so we are setting the
        // autoConnect
        // parameter to false.
        mBluetoothGatt = device.connectGatt( context: this,  autoConnect: false, mGattCallback, BluetoothDevice.TRANSPORT_LE);
//        refreshDeviceCache(mBluetoothGatt);
        Log.d(TAG,  msg: "Trying to create a new connection.");
        mBluetoothDeviceAddress = address;

        return true;
    }
```



```
    /**
     * Disconnects an existing connection or cancel a pending connection. The
     * disconnection result is reported asynchronously through the
     * {@code BluetoothGattCallback#onConnectionStateChange(android.bluetooth.BluetoothGatt, int, int)}
     * callback.
     */
    @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR2)
    public void disconnect() {
        if (mBluetoothAdapter == null || mBluetoothGatt == null) {
            Log.w(TAG, msg: "BluetoothAdapter not initialized");
            return;
        }
        mBluetoothGatt.disconnect();
    }

    /**
     * After using a given BLE device, the app must call this method to ensure
     * resources are released properly.
     */
    @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR2)
    public void close() {
        if (mBluetoothGatt == null) {
            return;
        }
        mBluetoothGatt.close();
        mBluetoothGatt = null;
    }
```

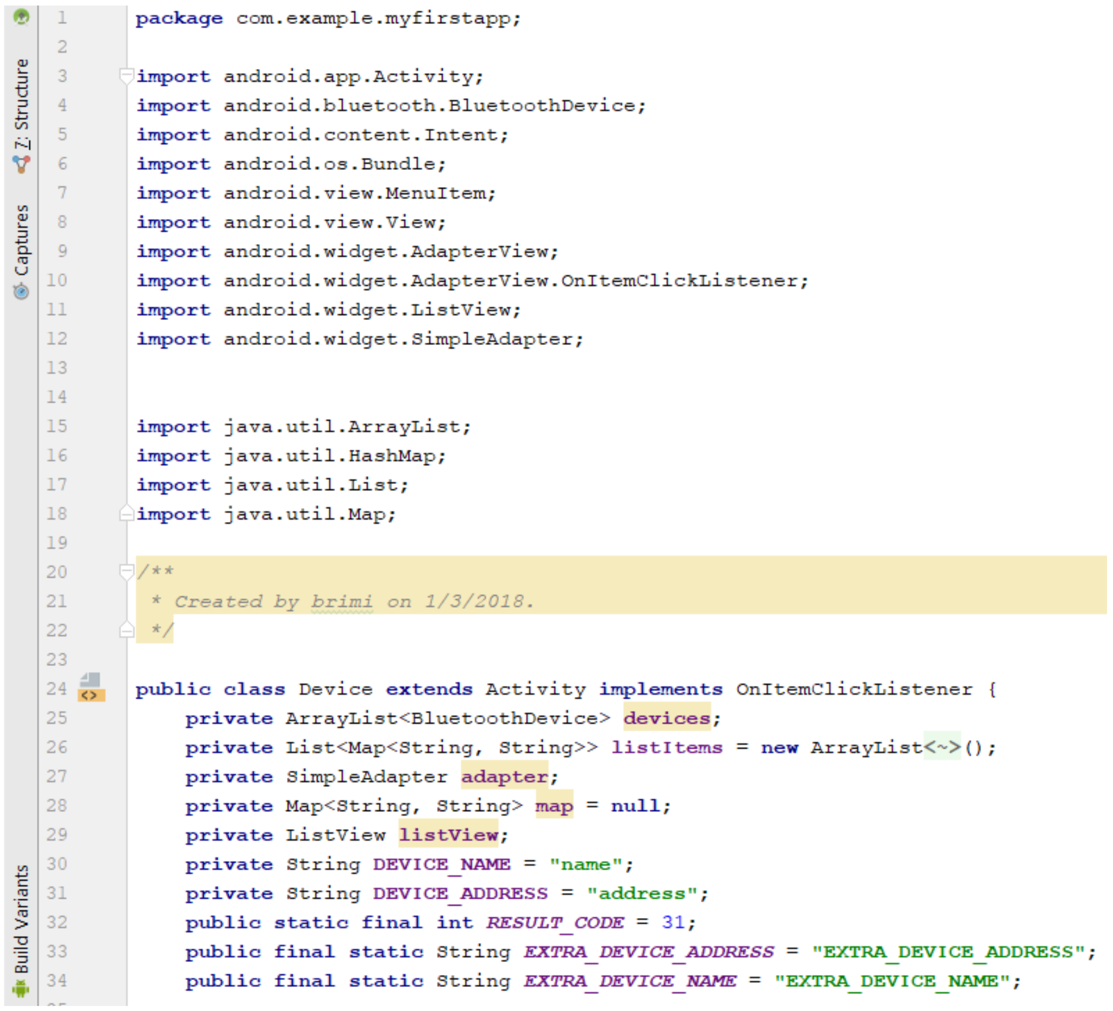


```
    /**
     * Request a read on a given {@code BluetoothGattCharacteristic}. The read
     * result is reported asynchronously through the
     * {@code BluetoothGattCallback#onCharacteristicRead
     * (android.bluetooth.BluetoothGatt, android.bluetooth.BluetoothGattCharacteristic, int)}
     * callback.
     *
     *             The characteristic to read from.
     */
    @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR2)
    public void readCharacteristic() {
        if (mBluetoothAdapter == null || mBluetoothGatt == null) {
            Log.w(TAG, msg: "BluetoothAdapter not initialized");
            return;
        }
        BluetoothGattService service = mBluetoothGatt.getService(UUID_BLE_SHIELD_SERVICE);
        BluetoothGattCharacteristic characteristic = service.getCharacteristic(UUID_BLE_SHIELD_RX);

        mBluetoothGatt.readCharacteristic(characteristic);
    }

    //not used
    @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR2)
    public void readRssi() {
        if (mBluetoothAdapter == null || mBluetoothGatt == null) {
            Log.w(TAG, msg: "BluetoothAdapter not initialized");
            return;
        }

        mBluetoothGatt.readRemoteRssi();
    }
```

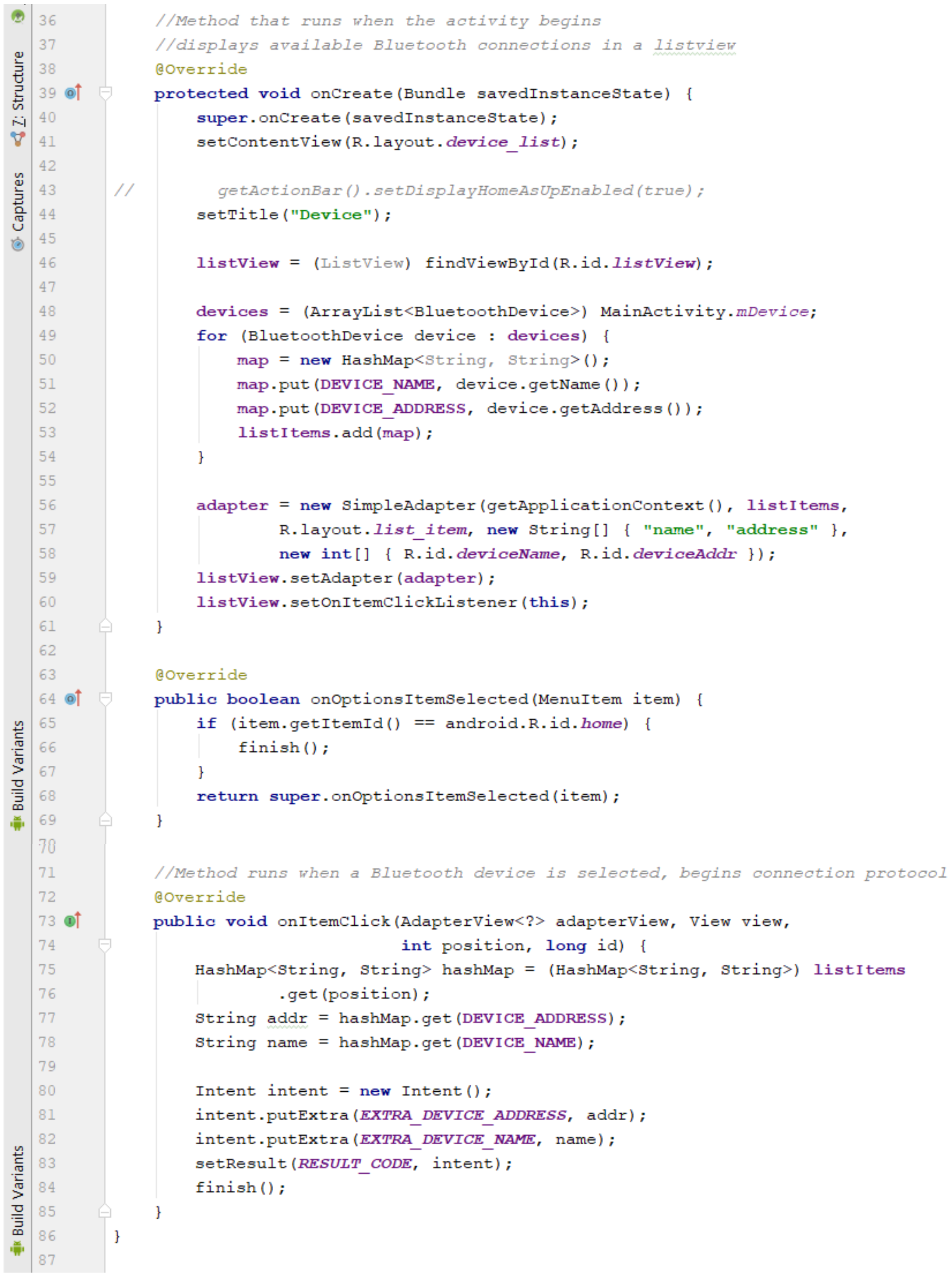


```java
    //not used, would be used to write data to the characteristic
    @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR2)
    public void writeCharacteristic(BluetoothGattCharacteristic characteristic) {
        if (mBluetoothAdapter == null || mBluetoothGatt == null) {
            Log.w(TAG,  msg: "BluetoothAdapter not initialized");
            return;
        }

        mBluetoothGatt.writeCharacteristic(characteristic);
    }

    /**
     * Enables or disables notification on a give characteristic.
     *
     * @param characteristic
     *            Characteristic to act on.
     * @param enabled
     *            If true, enable notification. False otherwise.
     */
    @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR2)
    public void setCharacteristicNotification(
            BluetoothGattCharacteristic characteristic, boolean enabled) {
        if (mBluetoothAdapter == null || mBluetoothGatt == null) {
            Log.w(TAG,  msg: "BluetoothAdapter not initialized");
            return;
        }
        mBluetoothGatt.setCharacteristicNotification(characteristic, enabled);

        if (UUID_BLE_SHIELD_RX.equals(characteristic.getUuid())) {
            BluetoothGattDescriptor descriptor = characteristic
                    .getDescriptor(UUID
                            .fromString(BLEGattAttributes.CLIENT_CHARACTERISTIC_CONFIG));
            descriptor
                    .setValue(BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE);
            mBluetoothGatt.writeDescriptor(descriptor);
        }
    }

    @TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR2)
    public BluetoothGattService getSupportedGattService() {
        if (mBluetoothGatt == null)
            return null;

        return mBluetoothGatt.getService(UUID_BLE_SHIELD_SERVICE);
    }

    public BluetoothGatt getBluetoothGatt(){
        if(mBluetoothGatt == null){
            return null;
        }

        return mBluetoothGatt;
    }
}
```

Device.java

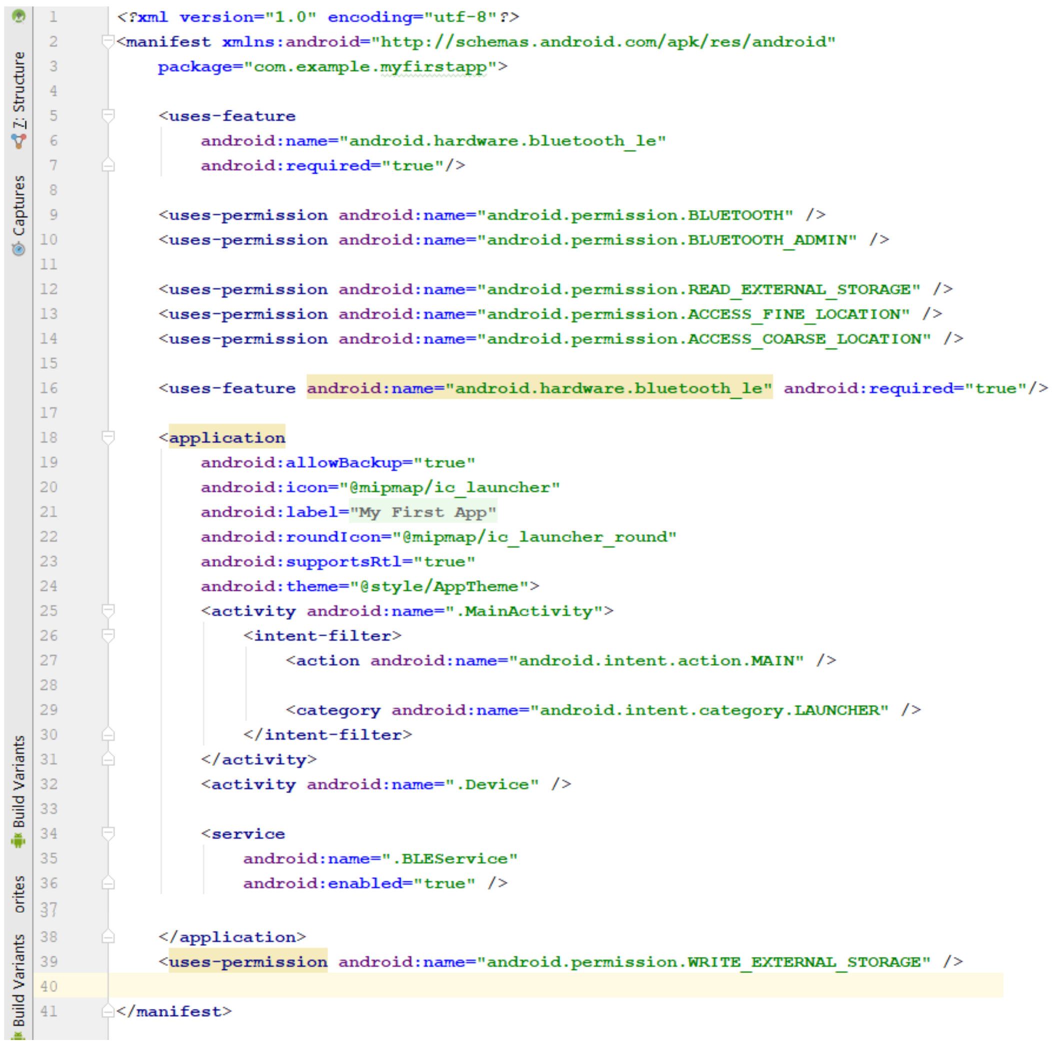


```java
package com.example.myfirstapp;

import android.app.Activity;
import android.bluetooth.BluetoothDevice;
import android.content.Intent;
import android.os.Bundle;
import android.view.MenuItem;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ListView;
import android.widget.SimpleAdapter;


import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * Created by brimi on 1/3/2018.
 */

public class Device extends Activity implements OnItemClickListener {
    private ArrayList<BluetoothDevice> devices;
    private List<Map<String, String>> listItems = new ArrayList<~>();
    private SimpleAdapter adapter;
    private Map<String, String> map = null;
    private ListView listView;
    private String DEVICE_NAME = "name";
    private String DEVICE_ADDRESS = "address";
    public static final int RESULT_CODE = 31;
    public final static String EXTRA_DEVICE_ADDRESS = "EXTRA_DEVICE_ADDRESS";
    public final static String EXTRA_DEVICE_NAME = "EXTRA_DEVICE_NAME";
```



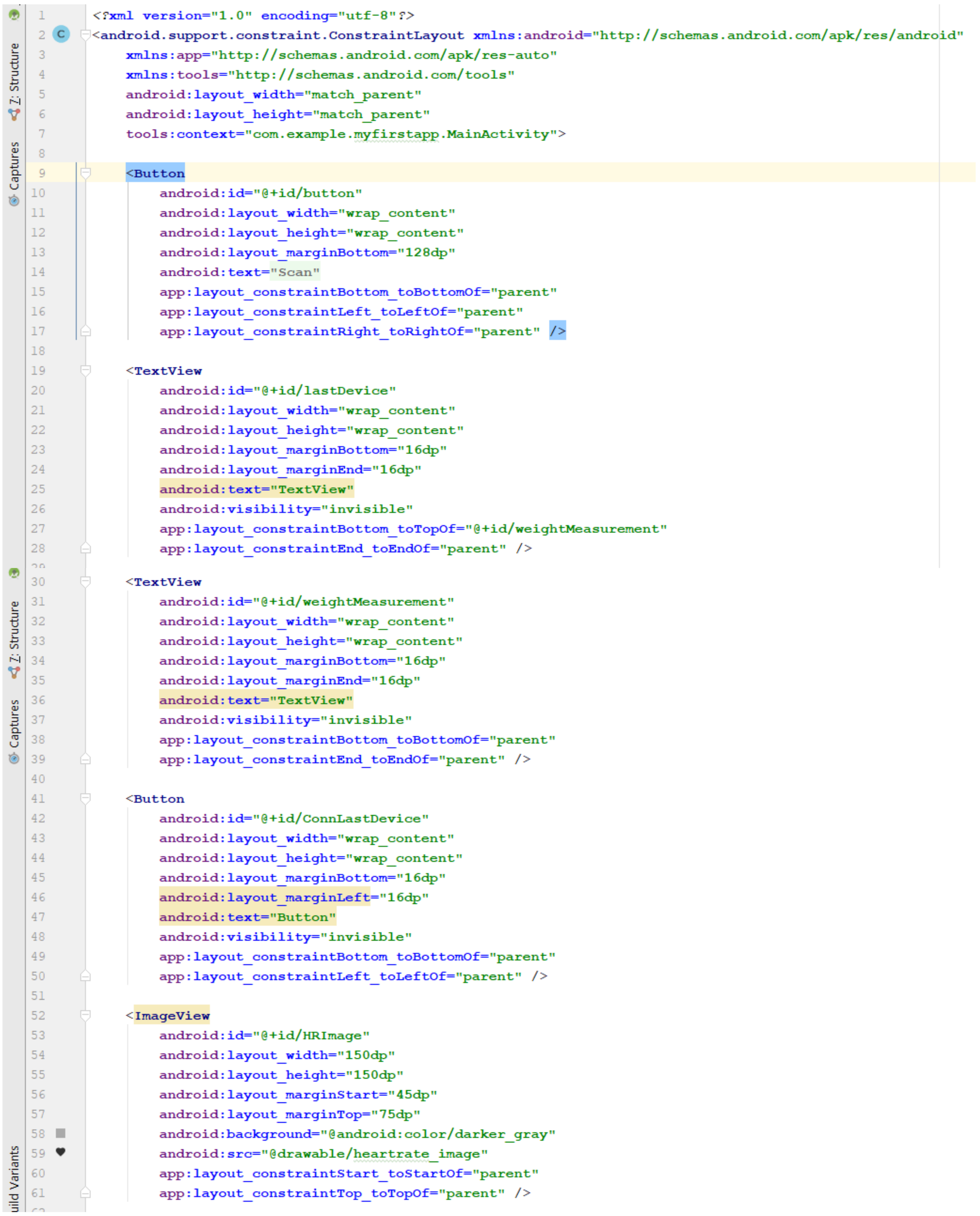
```java
    //Method that runs when the activity begins
    //displays available Bluetooth connections in a listview
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.device_list);

//        getActionBar().setDisplayHomeAsUpEnabled(true);
        setTitle("Device");

        listView = (ListView) findViewById(R.id.listView);

        devices = (ArrayList<BluetoothDevice>) MainActivity.mDevice;
        for (BluetoothDevice device : devices) {
            map = new HashMap<String, String>();
            map.put(DEVICE_NAME, device.getName());
            map.put(DEVICE_ADDRESS, device.getAddress());
            listItems.add(map);
        }

        adapter = new SimpleAdapter(getApplicationContext(), listItems,
                R.layout.list_item, new String[] { "name", "address" },
                new int[] { R.id.deviceName, R.id.deviceAddr });
        listView.setAdapter(adapter);
        listView.setOnItemClickListener(this);
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        if (item.getItemId() == android.R.id.home) {
            finish();
        }
        return super.onOptionsItemSelected(item);
    }

    //Method runs when a Bluetooth device is selected, begins connection protocol
    @Override
    public void onItemClick(AdapterView<?> adapterView, View view,
                            int position, long id) {
        HashMap<String, String> hashMap = (HashMap<String, String>) listItems
                .get(position);
        String addr = hashMap.get(DEVICE_ADDRESS);
        String name = hashMap.get(DEVICE_NAME);

        Intent intent = new Intent();
        intent.putExtra(EXTRA_DEVICE_ADDRESS, addr);
        intent.putExtra(EXTRA_DEVICE_NAME, name);
        setResult(RESULT_CODE, intent);
        finish();
    }
}
```

AndroidManifest.xml

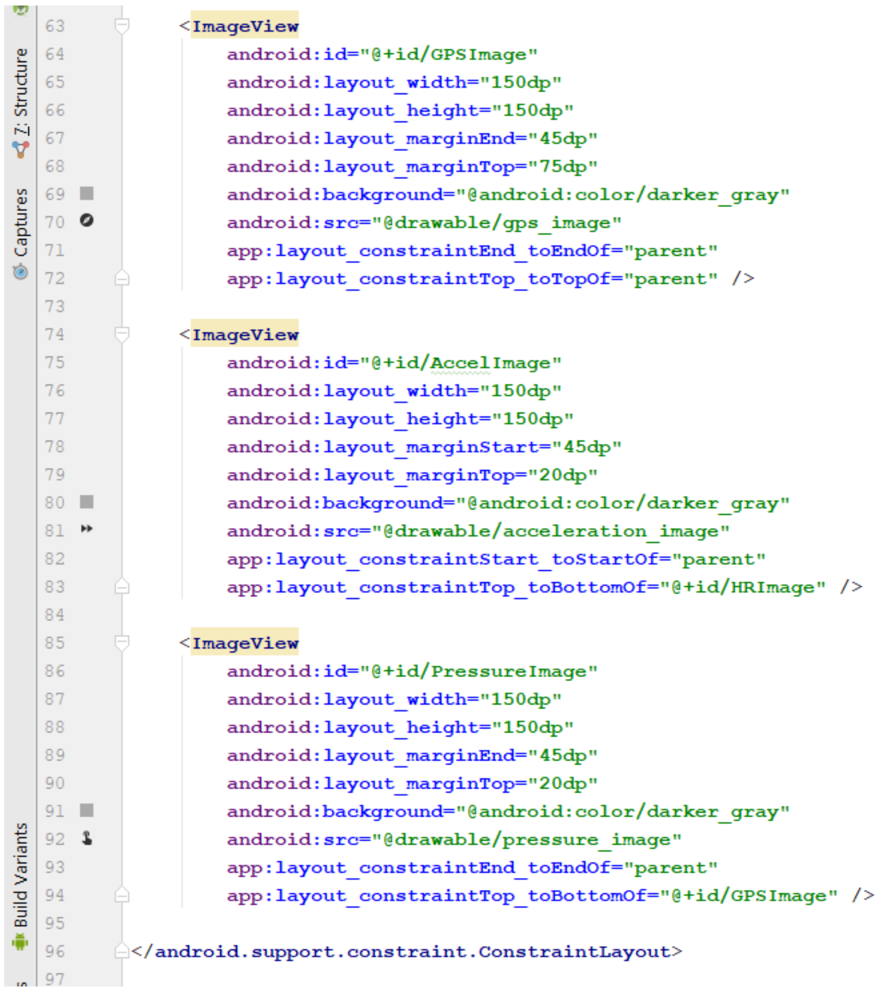


```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myfirstapp">

    <uses-feature
        android:name="android.hardware.bluetooth_le"
        android:required="true"/>

    <uses-permission android:name="android.permission.BLUETOOTH" />
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />

    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />

    <uses-feature android:name="android.hardware.bluetooth_le" android:required="true"/>

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="My First App"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".Device" />

        <service
            android:name=".BLEService"
            android:enabled="true" />

    </application>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

</manifest>
```

activity_main.xml

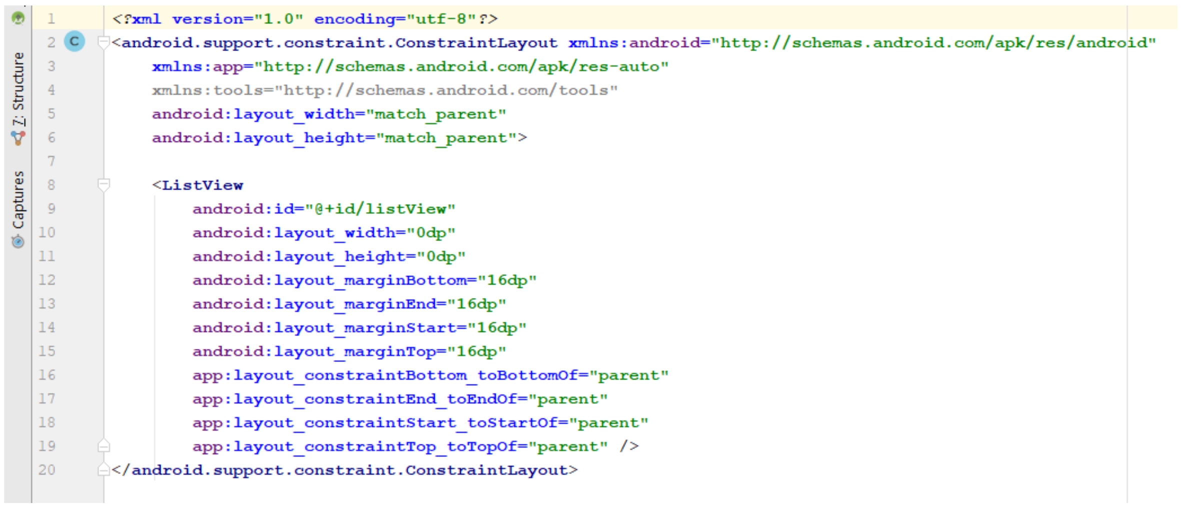


```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.example.myfirstapp.MainActivity">

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="128dp"
        android:text="Scan"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent" />

    <TextView
        android:id="@+id/lastDevice"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="16dp"
        android:layout_marginEnd="16dp"
        android:text="TextView"
        android:visibility="invisible"
        app:layout_constraintBottom_toTopOf="@+id/weightMeasurement"
        app:layout_constraintEnd_toEndOf="parent" />

    <TextView
        android:id="@+id/weightMeasurement"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="16dp"
        android:layout_marginEnd="16dp"
        android:text="TextView"
        android:visibility="invisible"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent" />

    <Button
        android:id="@+id/ConnLastDevice"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="16dp"
        android:layout_marginLeft="16dp"
        android:text="Button"
        android:visibility="invisible"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent" />

    <ImageView
        android:id="@+id/HRImage"
        android:layout_width="150dp"
        android:layout_height="150dp"
        android:layout_marginStart="45dp"
        android:layout_marginTop="75dp"
        android:background="@android:color/darker_gray"
        android:src="@drawable/heartrate_image"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
```



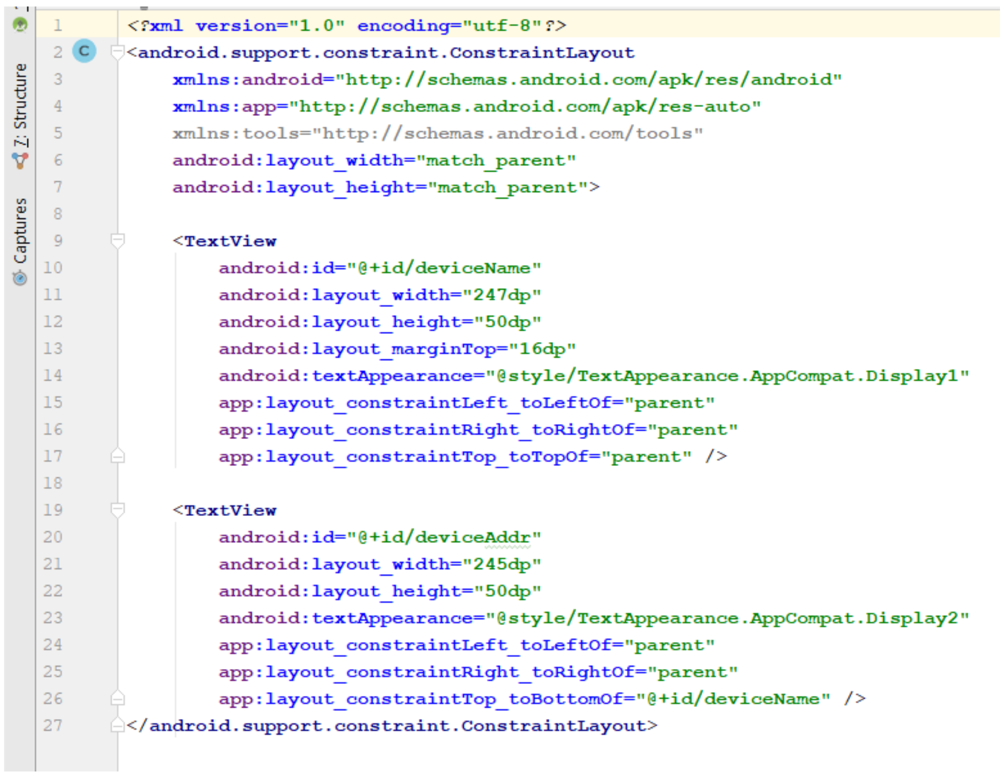
```
    <ImageView
        android:id="@+id/GPSImage"
        android:layout_width="150dp"
        android:layout_height="150dp"
        android:layout_marginEnd="45dp"
        android:layout_marginTop="75dp"
        android:background="@android:color/darker_gray"
        android:src="@drawable/gps_image"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <ImageView
        android:id="@+id/AccelImage"
        android:layout_width="150dp"
        android:layout_height="150dp"
        android:layout_marginStart="45dp"
        android:layout_marginTop="20dp"
        android:background="@android:color/darker_gray"
        android:src="@drawable/acceleration_image"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/HRImage" />

    <ImageView
        android:id="@+id/PressureImage"
        android:layout_width="150dp"
        android:layout_height="150dp"
        android:layout_marginEnd="45dp"
        android:layout_marginTop="20dp"
        android:background="@android:color/darker_gray"
        android:src="@drawable/pressure_image"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/GPSImage" />

</android.support.constraint.ConstraintLayout>
```

device_list.xml



```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ListView
        android:id="@+id/listView"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:layout_marginBottom="16dp"
        android:layout_marginEnd="16dp"
        android:layout_marginStart="16dp"
        android:layout_marginTop="16dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</android.support.constraint.ConstraintLayout>
```

list_item.xml



```xml
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/deviceName"
        android:layout_width="247dp"
        android:layout_height="50dp"
        android:layout_marginTop="16dp"
        android:textAppearance="@style/TextAppearance.AppCompat.Display1"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/deviceAddr"
        android:layout_width="245dp"
        android:layout_height="50dp"
        android:textAppearance="@style/TextAppearance.AppCompat.Display2"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/deviceName" />
</android.support.constraint.ConstraintLayout>
```