Multi-Sensor BLE Platform Using TI Wireless MCU and Mobile Application

by

Ronald Yarwood

Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science in Engineering

in the

Electrical Engineering

Program

YOUNGSTOWN STATE UNIVERSITY

May 2022

Multi-Sensor BLE Platform Using TI Wireless MCU and Mobile Application

Ronald Yarwood

Signature:

Ronald Yarwood, Student Date

Approvals:

Dr. Pedro Cortes, Thesis Advisor Date

Dr. Eric MacDonald, Committee Member Date

Dr. Frank X. Li, Committee Member Date

Dr. Salvatore A. Sanders, Dean of Graduate Studies Date

Abstract

The need for small versatile sensors is ever-growing [1]. The demand to create sensors that are more versatile and resilient while still improving in size and speed is only growing. As this technology improves, we find more innovative applications. These new applications are bringing this technology out of strictly the academic and industrial realm and bringing it into everyday life. People are now able to run important medical tests on the fly without anything more than a watch or ring [2]. Athletes are able to extract important statistics data from practice to improve their craft and increase safety.

This research aims to create a universal platform to interface with a group of different sensors that is easy to use, versatile and expandable. To make the platform easy to adapt, it should be developed to work with existing infrastructure that people are already familiar with and have easy access to. Bluetooth Low Energy is a technology that is present on nearly all mobile devices [3], which makes it a great mode of communication. A sensor platform system is developed from the hardware level all the way to the application layer, reading data on a mobile device. This system also avoids the problems of other implementations with limited adaptabilities.

List of Figures

Nomenclature

ADC Analog to Digital Converter

API Application Programming Interface

BLE Bluetooth Low Energy

I2C Inter-Integrated Circuit

IOT Internet-of-Things

MCU Microcontroller Unit

PCB Printed Circuit Board

TI Texas Instruments

1.0 Introduction

Bluetooth Low Energy (BLE) is a wireless standard that prioritizes low power operation while maintaining flexibility to allow for new product development [4]. BLE is a feature that is present in nearly all mobile electronics that communicate with other devices. This makes it an obvious choice for use as a means of communication on a platform with a goal of being both universal and versatile [5]. In order to ensure that the platform is realizable, development is done from both sides. This means that a custom printed circuit board (PCB) was designed and manufactured, code was written to run on this newly designed PCB and software was created to interact with this embedded system on a mobile device. Each step was designed as a template that can be expanded upon later without a requirement of understanding the entire system.

1.1 Previous Work

BLE is a popular wireless platform for new devices and research. With an estimated compound annual growth rate CAGR of nearly 20% [6], BLE is only expected to become more popular and essential to daily life. Nearly all mobile devices (phones, laptops etc..) have BLE capabilities [3]. This makes it a good choice for use in the smart home [7], and medical fields [8]. In addition to its availability, it also offers low power consumption which is attractive for many embedded applications [5]. The issue with many of these implementations is that they are only looking to improve one aspect or to provide a proof of concept for the implementation of BLE. The goal of this research is to provide a platform that can be both universal and easily expandable.

Much of the research into BLE exists to find an application to solve an existing product or find a way to use it in an emerging or existing market or field. For example, a BLE solution was proposed for diaper wetness detection [9]. There have been studies to test the efficacy of IoT or BLE in other fields or industry such as space [10], and home health [2]. Another large area of work is dedicated to benchmarking [3] or proposing new methods to improve parts of the existing platform [11].

A large portion of publicly available research into BLE implementations is in the medical field. Since BLE is aimed at being very low power and is something that is readily available to nearly all patients, it becomes a very interesting communication protocol for wearables and assisted living applications [12]. This can be seen in the immense popularity of the devices such as Fitbit or the Apple Watch. The wearable market is expected to hit over 118 billion USD in revenue by 2028 [1]. There is also work being done for new sensors that are not yet present in the market such as a blood flow meter [13], disease detection [14], and even prosthetic control [15].

2.0 BLE Introduction

2.1 Introduction

With a goal in mind of delivering universal access without an unnecessary sacrifice in performance, Bluetooth Low Energy is a great protocol to build upon [1]. BLE is available on nearly all modern smart devices meaning a user can use devices they already have to communicate with a sensor [2]. This is a large reason why we are seeing BLE being utilized in the ‘Internet-of-Things’ sensor sector [2]. With this market expected to exceed a growth rate of 30% [3], it seems obvious that BLE is a wise choice for a communication protocol for embedded sensors.

2.2 Protocol Stack

The BLE Protocol Stack is comprised of three main layers [2]. *The Application Layer* is the highest level and handles user interaction. *The Host Layer* is the next lowest layer and features most of the distinguishing features of BLE compared to other communication protocols. *The Controller Layer* is the lowest layer and handles the *Physical Layer* and *Link Layer*. Figure 2-1 shows a visualization of these layers as well as the flow of data through them. The majority of the work done in this paper focuses on the Application Layer and Host Layer. As is the nature of the Controller Layer, most of the programming is done via firmware from the manufacturer of the transceiver.

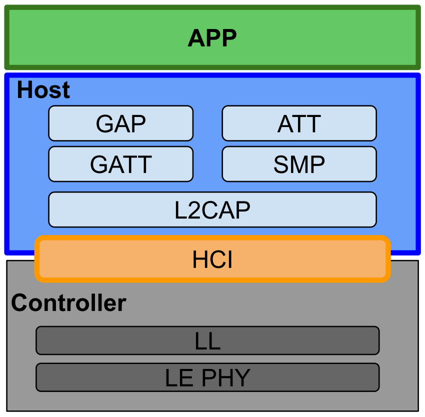


Figure 2-1 Visual Representation of the BLE protocol stack. Colors indicate which layer a feature is contained in. Adapted from [2].

2.2.1 The Application Layer

The Application Layer is the layer that is user facing. There are many application profiles that are detailed by the Bluetooth Special Interest Group (SIG) [4]. This is done to provide a uniform way to represent equivalent devices. This where the lower level is dispatched and where a developer describes how a device will operate. The application layer handles the overall flow of an application or embedded system. For example, when will the device look for new connections and when will it share information? In the board designed in this research, the app is designed to advertise as soon as the lower layers are initialized. It also handles how often sensor data is transmitted. Any commands sent from connected devices to configure the sensors are interpreted and dispatched through the application layer. From the perspective of the mobile app developed, the application layer asks to be notified about any changes and handles any configurations.

2.2.2 The Host Layer

The host layer is what separates BLE from classic Bluetooth and other wireless communication protocols. Figure 2-2 highlights this.

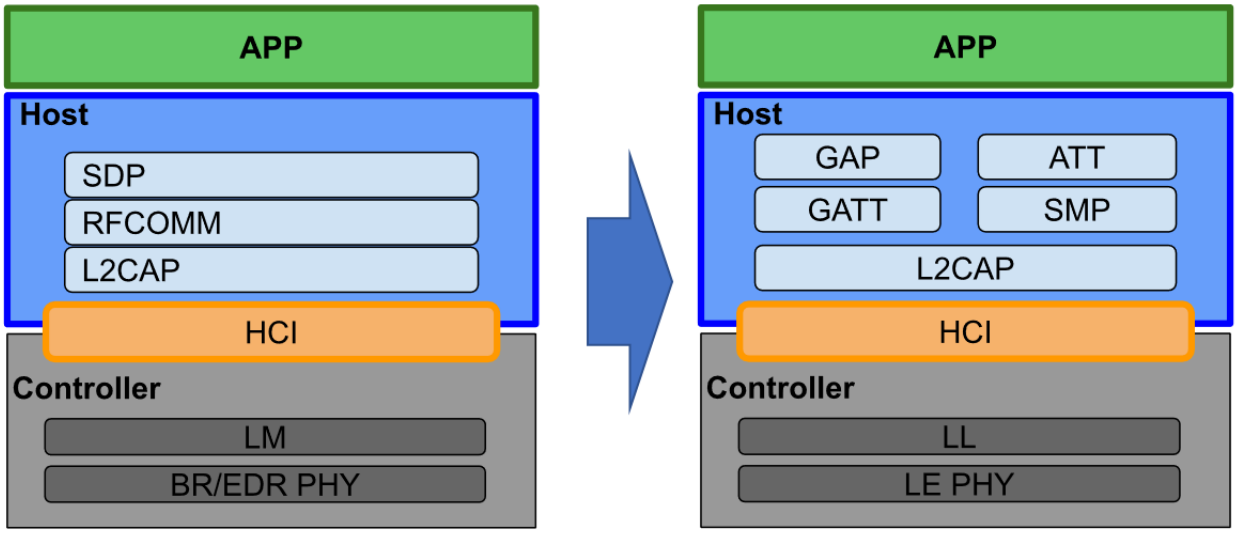


Figure 2-2 Shows the changes made from Bluetooth Classic to Bluetooth Smart (Later adapted to Bluetooth Low Energy). Adapted from [16]

It is comprised of the following components: Generic Access Profile, Generic Attribute Profile, Logical Link Control and Adaptation Protocol, Attribute Protocol, Security Manager Protocol, and the host side of the Host Controller Interface [3].

The Generic Access Profile, or GAP, acts as the link between the user facing application layer and the lower-level layers of the BLE protocol stack. It handles connection, security and general procedures. These include advertising and broadcasting. It is also responsible for communicating with the GATT on client systems.

The Generic Attribute Profile, or GATT, is the most customizable aspect of the host layer and defines what the device looks like to other connecting devices. The GATT expands upon the Attribute Protocol, or ATT, and defines two roles in any connection. Every connection contains a server and a client. The roles of each will be discussed further as the host layer is detailed further. The main goal of the GATT is to define profiles. These profiles describe what the function of the BLE device is. A profile can be thought of as a container filled with *services* which in turn is a container containing *characteristics*. A characteristic is the smallest container we will look at and contains properties such as permissions, a description, and a value. It should be noted that profiles are not defined on the hardware or in the source code. Rather, profiles are meant to be defined by the Bluetooth SIG (or proposed for inclusion if not already available) [4]. For example, someone designing a heart rate sensor would look at the defined profile for such from Bluetooth. This profile would detail the service and hence the characteristic structure. It also defines what universally unique identifier, or UUID, to use for each. The server in the BLE connection defines the GATT table and during advertising shows other devices what services and characteristics it contains. The services and characteristics that are advertised are set by the programmer. During this process the client reads this information and determines the method for connection. The client does not define a GATT table, it uses what is defined by the server.

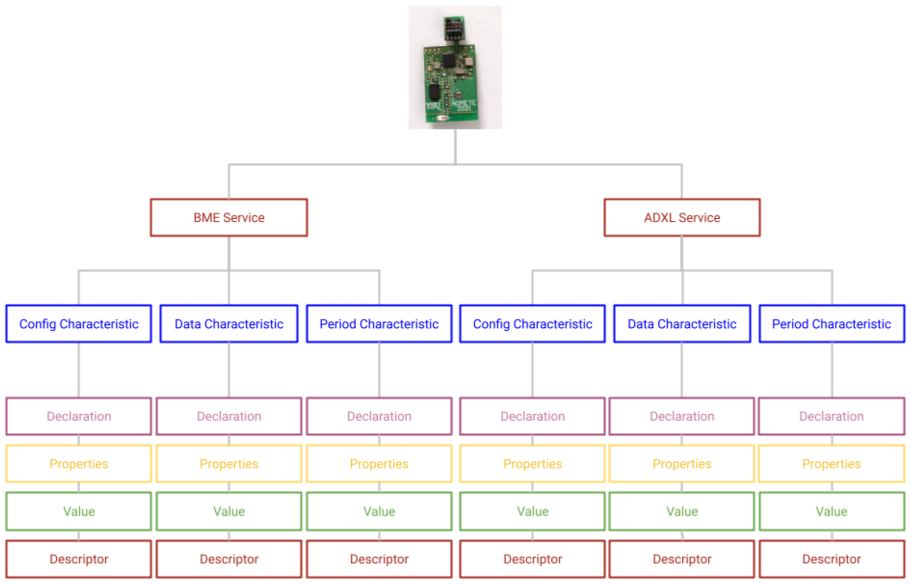


Figure 2-3 Visualization of BLE GATT

The Attribute Protocol or ATT is the layer that the BLE GATT expands upon. It defines how the client and server interaction happens. In general, the flow of data works in the following way. A client requests data from the server, the server then responds with the data requested. It should be noted that any given device can be a client, a server, or both. In the case of this research, the PCB acts as the server and the mobile device the client. The ATT is also responsible for packaging data into sections, or attributes. Each attribute has its own UUID, permissions and value.

The Logical Link Control and Adaptation Protocol, Security Manager Protocol and the Host Controller Interface are the final parts of the Host layer to be discussed. The Logical Link Control and Adaptation Protocol, or L2CAP, handles the encapsulation and fragmentation of data into and from BLE packet form. The Security Manager Protocol, or SMP, handles the encryption and decryption of raw data. The Host Controller Interface, or HCI, interfaces directly with the firmware of the specific hardware. This means it translates to commands that the Controller layer will understand.

2.2.3 The Controller Layer

The controller layer is the lowest layer of the BLE stack. It interfaces directly with the hardware and is typically developed and maintained by the manufacturer of the antenna. It contains three main parts. First is the controller side of the HCI mentioned above, there is also the Link Layer and the Physical Layer. Most of the work done in these layers is done directly on hardware. There are a few reasons for this. Firstly, much of the math done here is computationally intense and would noticeably impact the performance of a MCU. Another major reason is since it is so close to hardware there are a lot of hardware specific methods which means it is typically easier for the manufacturer or designer of the hardware to handle this.

The Link Layer, or LL, is the last layer before the physical layer and hence interfaces directly. This layer is present in nearly all communication protocols and typically performs a very similar task among all implementations. It has five main functions which will be named but their descriptions are beyond the scope of this paper. Those five functions are: error detection and correction, framing, random number generation, Advanced Encryption Standard (AES), and Cyclic Redundancy Check (CRC).

The Physical Layer, or PHY, is the actual antenna. Since this is for BLE, the antenna is designed to operate in the Industrial, Scientific and Medical (ISM). BLE is broken into 40 channels that are used as a part of the Advanced Frequency Hopping used to reduce interference. There are three channels reserved for advertising and the rest are used for data transmission. No work was done within this layer as a part of this research.

3.0 Embedded Design

In order to allow full customizability of software and firmware, a custom PCB was designed. With the new design we were also able to decrease the footprint. This allowed us to use multiple sensor peripherals and be familiar with the layout so that we may add more sensors or features later. The PCB is designed on a standard FR-4 substrate with a TI CC1350 Wireless MCU. A breakaway JTAG connecter was also added to the design that can be broken off after programming to decrease size. In Figure 3-1 we can see the physical footprint of the PCB designed compared to a comparable off-the-shelf option from Texas Instruments.

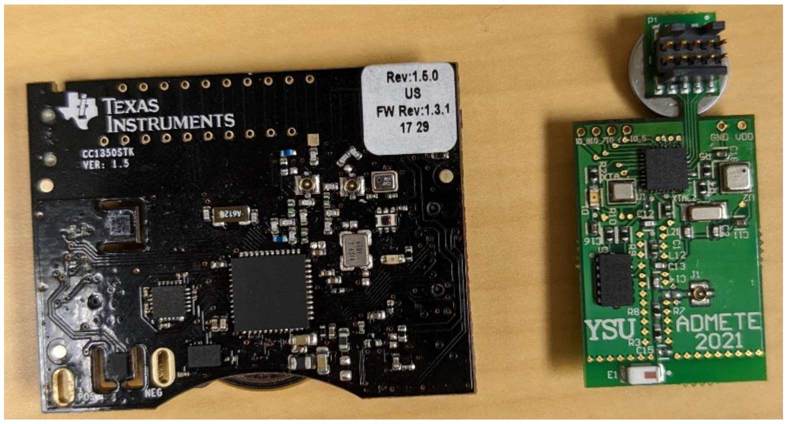


Figure 3-1 Size comparison of designed PCB vs comparable CC1350 SensorTag from TI

3.1 MCU Programming

Programming the MCU was one of the two main tasks for this project. The development of the MCU is evenly split between the Host and Application Layers mentioned above. Since the MCU acts as the central node for BLE connection, connection parameters and information must be set here. Much of this is handled through APIs but since we are using a custom PCB design, some changes must be made to match our hardware.

3.0.1 Inter-Integrated Circuit

Inter-Integrated Circuit (I2C) Protocol is a communication method that allows different components on a PCB to communicate with each other using a standard protocol [17]. It is a two-wire serial communication protocol where one wire is a universal clock, and the other is a data line. It is a serial communication protocol meaning that data can only follow one path in one direction at a time. To ensure that data gets where it needs to go, each device on the bus has a unique address. The protocol follows a controller, responder architecture. There is one device on the bus that acts as the controller and provides the universal clock signal and initiates all communications. There can be multiple responders on a bus, and each will listen for its' address and act accordingly.

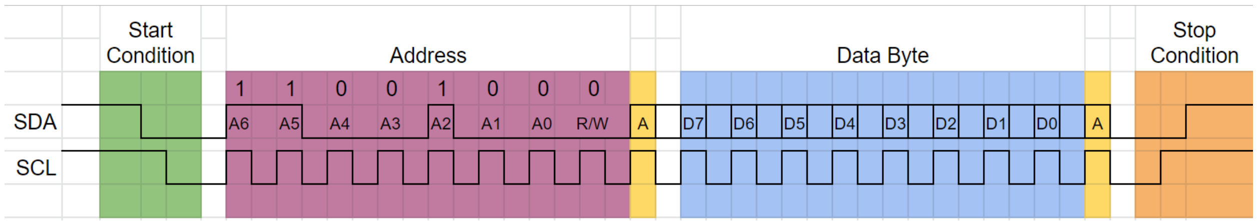


Figure 3-2 I2C Transaction Structure. Adapted From [17]

Figure 3-2 shows how an I2C transaction is structured. The controller sends out a start condition which is when the SDA goes low before the SCL goes low. This notifies all devices on the bus that a message is coming. The address of the desired responder is then sent. I2C supports addresses of 7 or 10 bits. A bit is then sent showing the intent of the controller. It specifies if it wants to read from or write to the responder. This means that if both are required, two separate transactions are needed. The responder then sends an acknowledgement bit. If the controller receives a one, it continues with the transaction. If it does not receive a reply or receives a zero, then the transaction is aborted. Assuming the acknowledgement bit was received, the transmission of data continues. If the read/write bit was set to read, then the responder sends a byte of data and waits for an acknowledgement from the controller. If the controller is attempting to write this transaction happens in reverse. Data is sent one byte at a time with an acknowledgement after each byte. Finally, when all data has been transmitted, the controller sends the stop condition which pulls the SDA line high after the SCL goes high.

3.0.2 I2C Interface

Since our design has two separate I2C busses and only one interface is available on the MCU, there must be some middleware to ensure that multiple devices are not attempting to access the same resource. Figure 3-3 shows an abbreviated version of such interface.

```
void aquireI2CInterface(I2C_Device_ID dev){
	uint8_t i;
	uint8_t sum = 0;
	if(!initialized){
		init_i2c();
	}

	for(i=0; i<total_count; i++){
		sum += devices[i];
	}

	if(sum > 0){ // If something already has the interface
		if(devices[dev]){ //Check if its the same device
			return;
		} // If it is not, sleep until it is available
	Semaphore_pend(i2c_sem, BIOS_WAIT_FOREVER);
	aquireI2CInterface(dev);
	} else {
		devices[dev] = 1; // Checkout the interface
		return;
	}
}

void releaseI2CInterface(I2C_Device_ID dev){
	devices[dev] = 0; // Make sure it doesn't block future acquisitions
	Semaphore_post(i2c_sem);
}
```

Figure 3-3 I2C Interface developed for custom PCBs to allow multiple physical busses to be used. Manages task timing.

To achieve this, a semaphore is used. This allows a task to sleep while waiting on the resource and be awoken as soon as it is available. This saves power as the processor is not "polling" or checking the status repeatedly and can help responsiveness as the task is awoken as soon as it is ready to proceed. First, if the interface has not been used yet, a semaphore is initialized. Next a simple loop that checks if any devices have claimed the interface. If the interface is in use by another device, then the caller will sleep until the bus is available. Otherwise, it claims the bus. Once a task is completed, it should remove its claim from the bus and “post” a semaphore waking up any device that is waiting. It should be noted that the *BIOS_WAIT_FOREVER* ensures that there is no timeout. This means that it must always wait until the interface is available before proceeding.

This is easily expandable to new devices. A new developer or designer would simply add their device to the list and that is all they need to change. Ideally, all the devices would be on a single bus or the MCU would support multiple busses. This does, however, have the added benefit that, if in the future, a board was made with more sensors, they could be grouped logically among multiple busses.

3.0.3 BME Interfacing

The BME688 is a multipurpose sensor developed by Bosch. It contains 4 separate sensors: a barometer and thermometer as well as a humidity and a gas sensor. The gas sensor can be trained using AI to detect the presence of specific materials. This can used for simple gasses or something more complex like coffee [18].

Bosch provides an API to interface with their device but most of it had to be rewritten to be more universal and easier to use for an inexperienced developer. The first step is to initialize the device with all the proper configurations for our use case. After the device is configured and we have an I2C interface, getting data from the sensors is relatively trivial.

We will utilize TI-RTOS and create a task for interfacing with the BME. A shortened version of this task loop is shown in Figure 3-4. It is a standard infinite loop that checks if the client wants to receive data from this sensor (done by writing not a 0 to the config byte). If the client has not enabled this sensor, then the task will sleep for a predefined period. This saves power and processor time by not constantly checking if a sensor is enabled. This also simplifies work for the main thread and hence makes future expansion easier by having it check itself again a little later. This means that no other part of the program must be aware of this device which simplifies the code for a someone looking to expand. If the sensor is enabled, the data is read from the device. It should be noted that no hardware interaction goes on in this function. That is to improve readability and promote modularity. Once that data is read, there is some interpretation that goes on. This is specific to the BME and is not done in the *get_data* function to make it clearer for a future developer what they would need to do to break the BME into multiple BLE services in the future. The new data is then ‘set’. This will be detailed later in the embedded BLE section. Finally, the task sleeps for a user set period. This value is set via the client over BLE.

```
// Task loop
  while (true)
  {
    if (sensorConfig == ST_CFG_SENSOR_ENABLE)
    {

      ///Note: You could turn sensor on before reading then off immediatly
after if power consumption is more important
      // Read data
      get_data(&bme);

      data[0] = bme.data.temperature;
      data[1] = bme.data.temperature >> 8;
      data[2] = bme.data.pressure;
      data[3] = bme.data.pressure >> 8;
      data[4] = bme.data.pressure >> 16;
      data[5] = bme.data.pressure >> 24;
      data[6] = bme.data.humidity;
      data[7] = bme.data.humidity >> 8;
      data[8] = bme.data.humidity >> 16;
      data[9] = bme.data.humidity >> 24;
      data[10] = bme.data.gas_resistance;
      data[11] = bme.data.gas_resistance >> 8;
      data[12] = bme.data.gas_resistance >> 16;
      data[13] = bme.data.gas_resistance >> 24;

      // Update GATT
      BME_setParameter(SENSOR_DATA, BME_DATA_LEN, data);

      // Next cycle
      DELAY_MS(sensorPeriod - BME_DELAY_PERIOD);
    }
    else
    {
      DELAY_MS(SENSOR_DEFAULT_PERIOD);
    }
  }
```

Figure 3-4 Demonstration of a sensor task to ensure BLE stack wants data and it is available.

For the user facing portion to be so simple, a little extra work is done behind the scenes. Typically, in an embedded application, this should be avoided as performance is a top priority and it is assumed that all developers working on a project are experienced in both the hardware being used and general programming practices. In this use case however, our goal is to provide a universal platform that is easy to use for developers of all experience. In addition, advanced sensors have long reading times meaning that there is extra CPU time to work with before becoming the bottleneck.

Getting the data from the BME688 has some minor nuances. For example, the temperature, pressure, and humidity can be read more often than any gas measurements. This is because, the BME688 heats the gas to a standard temperature before measuring.

```
uint8_t get_data(struct bme *bme){
    uint8_t rslt; // Temp variable for status codes
    uint32_t del_period; // Temp variable for device delay
    uint8_t n_fields; // Temp variable for number of data fields retrieved
    bool gotData = false; // Flag to see if we got all the data

    while(!gotData){ // While we have not gotten enough data

        /* Set the operation mode of BME device */
        rslt = bme68x_set_op_mode(BME68X_FORCED_MODE, &(bme->bme_dev));
        _check_rslt("bme68x_set_op_mode", rslt);

        /* Calculate delay period in microseconds */
        del_period = bme68x_get_meas_dur(BME68X_FORCED_MODE, &(bme->conf),
&(bme->bme_dev)) + (bme->heatr_conf.heatr_dur * 1000);
        bme->bme_dev.delay_us(del_period, bme->bme_dev.intf_ptr);

        /* Check if rslt == BME68X_OK, report or handle if otherwise */
        rslt = bme68x_get_data(BME68X_FORCED_MODE, &(bme->data), &n_fields,
&(bme->bme_dev));
        _check_rslt("bme68x_get_data", rslt);

        if(n_fields){ // If we got any data we will exit
            gotData = true;
        }

    }
    return n_fields; // return how many data fields we got
}
```

Figure 3-5 Demonstration of getting data from the Bosch BME688 sensor

The *get_data* function (shown in Figure 3-5) starts by setting up a few temporary variables. The *rslt* is used during debugging. It indicates which part of the process is causing an error. When compiling for release this is removed to make up for some performance sacrificed elsewhere. The *del_period* variable is used to configure how long to wait for the heater. The final two variables are used internally for status checking. They ensure that meaningful data is collected before returning. A loop is then used to get data while not enough has been collected. In the version above, any data is considered enough but both variables are kept if a future developer wants to tweak those settings. The data loop starts by setting the operating mode of the BME device. Then a delay period is calculated based on provided power, current temperature, goal temp etc. The task will then sleep for that long so that the plate is properly heated and accurate measurements are taken. Sleeping instead of looping allows other parts of the program to utilize that time. Finally, the function that reads data from the BME is called. The *bme68x_get_data* function will not be shown here as it is yet another level of abstraction down and that is beyond the scope of this paper. Its basic function is to write to the appropriate registers, wait the specified amount of time and then read from the data registers and perform some error checking.

3.0.4 ADXL Interfacing

The ADXL343 is a three-axis accelerometer developed by Analog Devices. It has an output data rate (ODR) of 800Hz and has multiple settings for range. There is a setting for all, +-2g, +-4g, +-8g, +-16g with ADC resolution ranges from 10-bit to 13-bit. It can withstand 10,000g force and works in the -40 to 85 C temperature range [19] making it a great solution for a multiuse sensor.

Once again, the RTOS is utilized for easier scheduling independent of other devices. There is no API for the ADXL. There is a small quirk with the ADXL where we have to reconfigure the sensor each time we use it since we are closing the I2C bus and reopening between measurements. The main RTOS task for the ADXL will not be shown as it is relatively redundant after seeing the task for the BME. It can be seen in the official documentation on the YSU SensorTag GitHub.

The interface for the ADXL however, will be discussed. A much-abbreviated version can be seen in Figure 3-6. Initially a flag is set so that a user reading from the client app can see where an error with the ADXL is coming from. First, the function ensures the I2C interface is available.

```
void ADXL_read(uint8_t *d){
    /* 0xFF Will mean an error */
    d[0] = 0xFF;
    /* Create I2CParam for device */
    I2C_Params_init(&(dev.i2cParams));
    dev.i2cParams.bitRate = I2C_100kHz;
    aquireI2CInterface(adxl);  // Checkout the I2C Inte

#ifdef Board_I2C_ADXL
    /* Open the I2C Interface */
    dev.i2c = I2C_open(Board_I2C_ADXL, &(dev.i2cParams)
    /* If we cannot open it, print the error and quit t
    if(dev.i2c == NULL){
        releaseI2CInterface(adxl);
        return;
    }
    /* ================ BEGIN CONFIGURATION ===========
    txBuffer[0] =  0x0; // Empty message
    while(!I2C_transfer(dev.i2c, &(dev.i2cTransaction))
    device_id = rxBuffer[0];

    txBuffer[0] = BW_RATE;
    while(!I2C_transfer(dev.i2c, &(dev.i2cTransaction))
    bw_rate = rxBuffer[0];

    txBuffer[0] = DATA_FORMAT;
    txBuffer[1] = RANGE_16G;
    while(!I2C_transfer(dev.i2c, &(dev.i2cTransaction))
    data_format = rxBuffer[0];

    txBuffer[0] = POWER_CTL;
    txBuffer[1] = ADXL_MEASURE;
    while(!I2C_transfer(dev.i2c, &(dev.i2cTransaction))
    power_control = rxBuffer[0];
    /* ================ END CONFIGURATION =============
    /* Set up packet to receive data */
    txBuffer[0] = DATA_X0;
    dev.i2cTransaction.readBuf = d;
    if(!I2C_transfer(dev.i2c, &(dev.i2cTransaction))){
        /* 0x00 [5] Will mean a data read error */
        d[5] = 0x00;
    }
     I2C_close(dev.i2c);
     releaseI2CInterface(adxl);
#endif
}
```

Figure 3-6 Demonstration of getting data from the Analog Devices ADXL343 sensor

Assuming it is, the I2C bus the ADXL is on is then opened. If this fails, the function releases the bus and returns (as will be the case with any error). Next, an empty transaction is sent to wake the ADXL. The Bandwidth, format, mode and power are then set. After the device is fully configured, the data can be retrieved. If the transaction fails, a flag is set, otherwise the data is placed in the buffer. The I2C bus is then closed and released and the function returns.

This is abstracted so that it can be called with a simple *adxl_read(data).* If this device was to be used for a new PCB design, a good feature to add would be to connect to hardware interrupt (HWI) pins on the MCU. Another possible feature to add would be to transmit the result from each of the configurations and modes over BLE so that the user could view or edit them.

3.0.5 BLE Interfacing

While the previous two sections detailing the BME688 and ADXL343 were contained at the application level of the BLE stack, this section will focus on the host layer. More specifically, the GAP and GATT. The Generic Access Profile, or GAP, controls how the BLE device interacts with other devices. This includes how it appears to other devices through advertising. The Generic Attribute Profile, or GATT, allows us to maintain multiple streams of data between devices.

Firstly, a service is defined for each subsystem. It should be noted that for this device all of the BME sensors are grouped into one service but, ideally each of these would have their own service with separate configurations and data transmission periods. That means that there are three total services: a device information service, the ADXL service and the BME service. The ADXL and BME services follow a simple structure that allows us to make an app that can easily add support for new devices. This structure is a service with three characteristics, a data characteristic that can be read or set to notify, a configure characteristic that enables the sensor and can written to or read from, and finally a period characteristic that also has read and write permissions that sets how often a given sensor transmits its data.

```
for (;;)
  {
    ICall_Errno errno = ICall_wait(ICALL_TIMEOUT_FOREVER);

    if (errno == ICALL_ERRNO_SUCCESS)
    {
      ICall_EntityID dest;
      ICall_ServiceEnum src;
      ICall_HciExtEvt *pMsg = NULL;

      if (ICall_fetchServiceMsg(&src, &dest,
                                (void **)&pMsg) == ICALL_ERRNO_SUCCESS)
      {
        if ((src == ICALL_SERVICE_CLASS_BLE) && (dest == selfEntityMain))
        {
          // Process inter-task message
          SensorTag_processStackMsg((ICall_Hdr *)pMsg);
        }

        if (pMsg)
        {
          ICall_freeMsg(pMsg);
        }
      }

      // If RTOS queue is not empty, process app message.
      while (!Queue_empty(appMsgQueue))
      {
        stEvt_t *pMsg = (stEvt_t *)Util_dequeueMsg(appMsgQueue);
        if (pMsg)
        {
          // Process message.
          SensorTag_processAppMsg(pMsg);

          // Free the space from the message.
          ICall_free(pMsg);
        }
      }
    }

    if (!!(events & ST_PERIODIC_EVT))
    {
      events &= ~ST_PERIODIC_EVT;

      if (gapProfileState == GAPROLE_CONNECTED || gapProfileState ==
GAPROLE_ADVERTISING)
      {
        Util_startClock(&periodicClock);
      }
    }
```

Figure 3-7 BLE Task code. Checks for messages, triggers any periodic tasks.

The code shown in Figure 3-7 is inside of a section marked as "DO NOT EDIT". This is done so that an inexperienced programmer does not edit this section and unknowingly destroy BLE communications. The BLE task is broken into three sections. The first is message retrieval. Assuming a message is successfully received, we will first ensure it is directed at the BLE stack of this thread. Once we ensure it is we will send it to a message router to be further processed. This typically means adding it to the RTOS Queue. Once that is completed, the RTOS Queue is then evaluated. A Queue is used so that all messages are processed in the order they are received. Finally, all periodic events are processed and the periodic clock is reset.

4.0 Mobile Programming

Mobile development was done using the Flutter framework. Flutter was developed at Google with the goal of bringing near native performance to multiple platforms while only needing to maintain one code base. The goal of the *YSU Bluetooth Suite* (listed as such on Apple App Store and Google Play Store) is to provide an easy way for future students to interface with either existing sensors or ones they develop themselves.

4.1 BLoC Pattern and Streams

The Business Logic Components, or BLoC, is a state management method and a software design pattern [5]. Its goal is to keep the user interface sperate from important business logic. In the case of this project, this means that such processes as BLE connection and data acquisition and interpretation are not handled by the part of the app the user sees. Instead, BLoC is based on a concept known as Streams [20]. A stream consists of two parties: publishers, or providers, and listeners, or subscribers. There can be many subscribers to the same stream, but a given stream should typically have only one provider. There are some exceptions to this but that is not in the scope of this paper. Relating back to the application being discussed here that means that there are providers that subscribe to the state of the BLE antenna. When the antenna state is updated, the information is interpreted, and a data stream is sent out with this new data. The UI components only listen to the streams that pertain to them, and their update cycle coincides with the streams they are listening to.

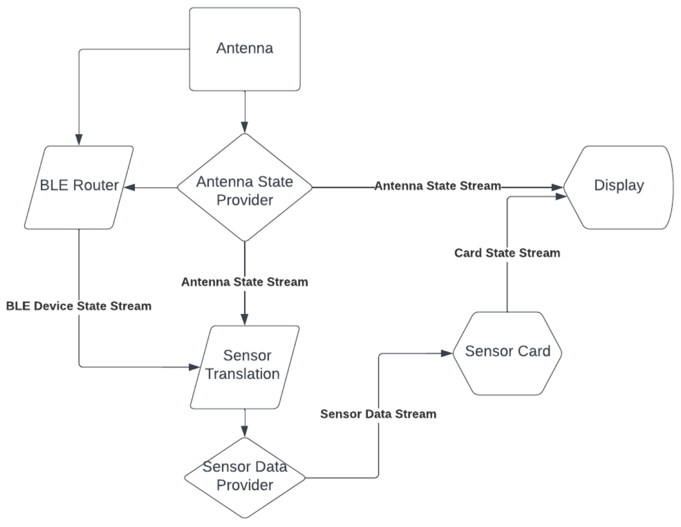
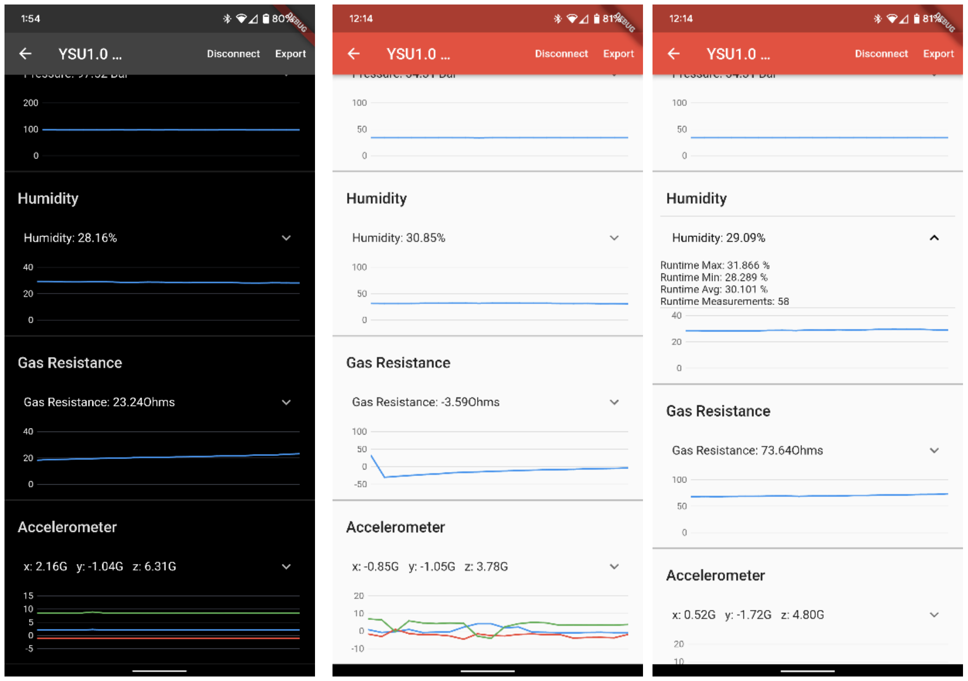


Figure 4-1 Simplified visualization of the streams for a single sensor card

There are a few benefits of this. The first is that the render time for each widget is drastically lowered since it does not have to do any math or interpretation so it can wait until it has all of the information to render. It also cleans up the code from a readability stand point as all logic is now in one place and all the UI design is in another. Finally, it is more secure as elements of the UI make no decisions and do not see the data they are interacting with. Figure 4-1 shows a very simplified version of the streams leading to sensor data being displayed in the UI.

4.2 User Interface

If the app intends to be a useful and powerful tool, it starts with the user interface. Making data easy to see and interpret is the most important characteristic of an interface with the goal of providing real-time sensor data. There is a balancing act however, the UI must be responsive or the data being received would no longer be ‘real-time’ or easy to interpret. To do this, a template was made to display all types of sensors and each of the displayed sensors updates independently. This means that only portions of the screen need to be rebuilt each time new data is received. Another technique was a dynamic display method. This means that if a ‘widget’ is not currently on the screen then the sensor data is collected but does not waste time updating the ‘widget'. With the added performance gained from these methods, there is more information that can be added to the screen.



Humidity: 28.16%

Gas Resistance: 23.24Ohms

x: 2.16G y: -1.04G z: 6.31G

Humidity: 30.85%

Gas Resistance: -3.59Ohms

x: -0.85G y: -1.05G z: 3.78G

Humidity: 29.09%

Runtime Max: 31.866 %
Runtime Min: 28.289 %
Runtime Avg: 30.101 %
Runtime Measurements: 58

Gas Resistance: 73.64Ohms

x: 0.52G y: -1.72G z: 4.80G

Figure 4-2 Screenshots from the YSU Bluetooth Suite application showing live data acquisition. From left to right: Live data acquisition (dark mode), Live data acquisition (light mode), demonstrating runtime statistics for humidity sensor (light mode).

This includes keeping runtime statistics such as maximum, minimum and average values. There is also a live graph so that changes over time can easily be seen. If more in-depth analysis is needed, data can be exported to a csv file which can easily be imported into more powerful tools like Python, MATLAB, Excel, etc.

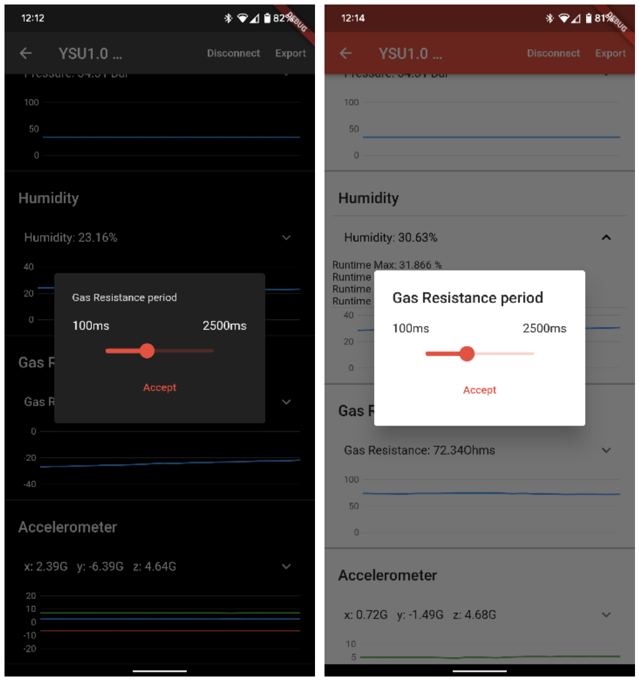


Figure 4-3 Screenshots from the YSU Bluetooth Suite application showing sensor period configuration

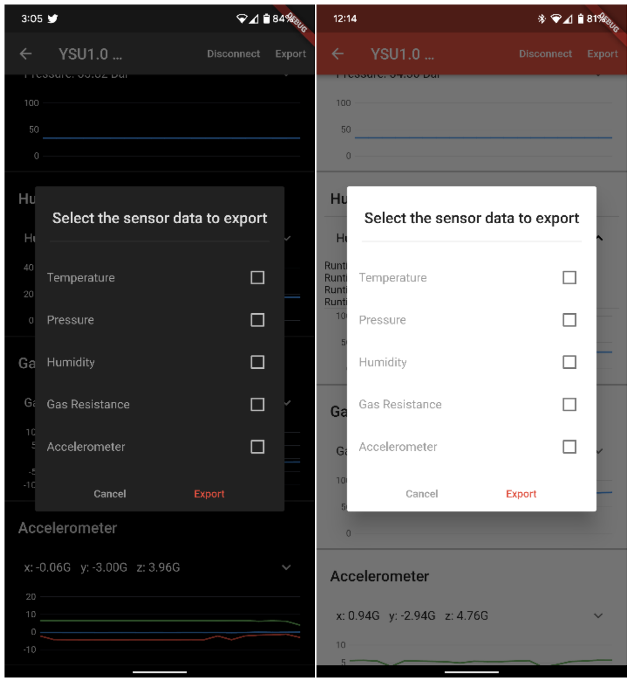


Figure 4-4 Screenshots from the YSU Bluetooth Suite application showing a sensor select for data exportation

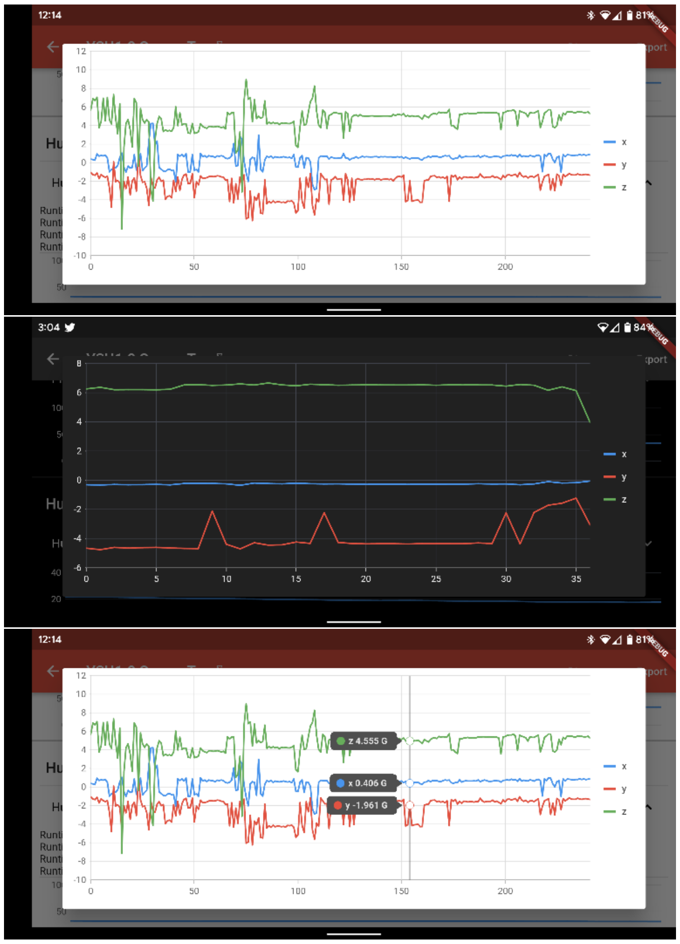


Figure 4-5 Screenshots from the YSU Bluetooth Suite application showing full screen plot viewing. From top to bottom: Plot viewing (light mode), plot viewing (dark mode), point selection (light mode).

4.3 Templated Design

Each BLE device in the app is designed as a template so that adding new devices or support for other existing devices is easy. This also means that appearance is uniform, and each sensor supports the same features.

```
Sensor(
   device: this,
   serviceUUID: UUID_BME_SERV,
   dataUUID: UUID_BME_DATA,
   configUUID: UUID_BME_CONF,
   periodUUID: UUID_BME_PERI,
   enableCode: [0x01],
   title: "Temperature",
   dimensions: 1,
   translation: translateTemperatur
   unit: "\u2109",
   labels: ["Temp"],
   axisLabels: ["\u2109"]),
```

Figure 4-6 Creation of a Sensor object demonstrating the templated design.

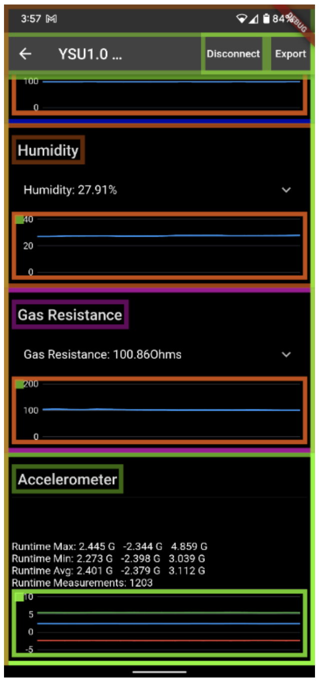
The goal of this design is to create an easy to extend app so that sensor design and development is attainable for anyone, including people with little to no software development experience. For example, to add support for a sensor, only about 300 lines of code is needed. The UUIDs for each GATT profile, how to translate bytes from the sensor into readable data and labeling information such as units etc.

```
/// Translates Temperature sensor [bytes] into
///
/// List[0] x axis
/// List[1] y axis
/// List[2] z axis
/// Translation from Bosch
List<double> translateTemperature(List<int> dat
 var value = Int8List.fromList(data);

 int x = (value[1] << 8) + value[0];

 return [x/100.0];
}
```

Figure 4-7 Example function showing a translation from bytes to human readable form to display on sensor screen

4.4 Performance

Several methods were used to increase performance and responsiveness without the user noticing any changes in functionality. The most fundamental technique is to only update ‘widgets’, or parts of the user interface, when they have new information. To do this, each widget will keep its current state until changed. This allows it to quickly give an update when the periodic screen refreshes are processed.



Humidity: 27.91%

Gas Resistance: 100.86Ohms

```
Runtime Max: 2.445 G   -2.344 G   4.859 G
Runtime Min: 2.273 G   -2.398 G   3.039 G
Runtime Avg: 2.401 G   -2.379 G   3.112 G
Runtime Measurements: 1203
```

Figure 4-8 Screenshot from the YSU Bluetooth Suite running in debug mode. Each color outline corresponds to a redraw cycle showing that each widget is not redrawn every refresh.

Flutter in Android Studio has set of developer tools known as DevTools. These tools help a developer profile their application in different ways. Figure 4-5 shows a visualization showing when each widget is ‘redrawn’ on the screen. Each color corresponds to a different cycle so it can be seen that if a widget does not have new data, it is left alone to save battery and performance. Another way performance is maximized is by only updating what is on the screen. This means that if there are enough sensors present on a device to scroll through, sensor cards that are off the screen are not processed. The sensor data is still tracked as the data acquisition is separate from the UI as discussed above.

5.0 Expansion

A major part of this research was to make this platform as universal, simple and expandable as possible. This can be broken into three phases. The first is accessibility. The project must be easy and simple to access. The next is readability. If the code written is too complex or sloppy then there can be no hope that someone looking to expand on the platform will be successful. Finally, proper explanation should be given on where to begin further development

5.1 Availability

All of the code is available on GitHub. The source code for the embedded system is a public repository meaning that anyone can view and clone to their local machine. There is a well written 'README' that breaks down how to properly and easily copy the code onto any computer for use. It should be noted that while the code is available to download, edits cannot be made to the publicly available code without permission. This is done as an added layer of security so that one rogue developer will not affect the entire project. If proper work is done to add support for a new device, it can be added to the public repository. This would most likely be done using a 'Pull Request'. A pull request is a tool that allows a repository's maintainer to approve changes before they are made public [21]. The source code for the mobile application is in a private repository. There are two main reasons for this. Firstly, the application is published to multiple app stores using my name so as a protection to myself, the repository is kept private. Secondly, as a layer of security to users, an app store requires a signature from the developer and with that signature the developer is promising to not provide malicious code. The best way for me to ensure this condition is met is to keep the repository private. When a user wants to add support for a new sensor, there are two main options. They can request permission to view the repository and do a pull request, or they can provide a maintainer with the needed information, and they can add support.

5.2 Readability

As mentioned above, if code cannot be easily interpreted, it is very difficult to expand upon. A lot of professional code is written with the assumption that another developer fundamentally understands the system they are working on which can make it difficult to read as a novice. This code was written with the assumption that a future developer has little to no experience in both embedded system programming and BLE. This sometimes means sacrificing some performance in the name of readability. There are a lot of inline comments to help describe non-trivial procedures or instructions. Most functions are designed to be as simple to use as possible which sometimes means adding multiple stack frames so that the program is easier to trace and/or step through.

5.3 Documentation

The GitHub repository is also the host to a website that acts as a wiki. This contains all of the documentation as well as some 'how-to' guides. These can guide a new developer through setting up the repository, creating or adding to a board, and developing support for a new sensor. This documentation is created using Doxygen which parses source code and looks for specially formatted comments. These can then be translated into HTML or LaTeX. This documentation can be used to learn about every part of the development process but also features a 'to-do' list that allows an inexperienced developer to jump to important portions of the source code and add their work. This is very simple to use as it is very obvious what should be edited.

```
/********************** BEGIN USER CODE 1 ***************************/
/// \todo USER CODE 1: This is where you add the includes to where your
sensor RTOS Task is.
#include "ADXL343/adxl343.h"
#include "BME688/bme688.h"
/*********************** END USER CODE ******************************/
```

Figure 5-1 Sample of a user code section which highlights where a new developer should work and what information should be added

Conclusion

We were able to develop a sensor platform using BLE. This platform utilizes a templated design which will allow future developers to easily expand upon the platform and not have to develop their own. The designed PCB is smaller than comparable offerings and still provides a steady connection and multiple sensor modules. The mobile application is publicly available and offers a smooth user interface and is powerful enough to be useful. All source code is available for distribution and is ready to use and expand upon. A full website is available detailing documentation and how to expand for each the mobile application and the embedded system. Work will be carried on by the next round of students and as of now are working to add support for a Cortisol sensor.

References

[1] Business Wire, "Global Wearable Technology Market Trends & Analysis Report 2021-2028: Adoption of Fitness Trackers and Health-based Wearables is Anticipated to Propel Growth," 4 Jan 2022. [Online]. Available: https://www.businesswire.com/news/home/20220104005806/en/Global-Wearable-Technology-Market-Trends-Analysis-Report-2021-2028-Adoption-of-Fitness-Trackers-and-Health-based-Wearables-is-Anticipated-to-Propel-Growth---ResearchAndMarkets.com#:~:text=Wearable%2.

[2] A. G, S. M, I. MF, S. MA, F. NL and R. J, "A Personalized Healthcare Monitoring System for Diabetic Patients by Utilizing BLE-Based Sensors and Real-Time Data Processing," *Sensors,* vol. 18, no. 7, p. 2183, 2018.

[3] J. Tosi, F. Taffoni, M. Santacatterina, R. Sannino and D. Formica, "Performance Evaluation of Bluetooth Low Energy: A Systematic Review," *Sensors,* vol. 17, no. 12, p. 2898, 2017.

[4] Bluetooth Technology, "Bluetooth Technology Overview," Bluetooth, [Online]. Available: www.bluetooth.com/learn-about-bluetooth/tech-overview. [Accessed 5 March 2022].

[5] C. Gomez, J. Oller and P. J, "Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology," *Sensors,* no. 12, pp. 11734-11753, 2012.

[6] MarketWatch, "Bluetooth Low Energy Devices Market Report Covers Future Trends with Research 2022-2029," 25 Jan 2022. [Online]. Available: https://www.marketwatch.com/press-release/bluetooth-low-energy-devices-market-report-covers-future-trends-with-research-2022-2029-intel-corporation-lenovo-group-ltd-microsoft-corp-2022-01-25?tesla=y#:~:text=The%20Global%20Bluetooth%20Low%20Energy,on%20the.

[7] A. A.Z and U. Buhur, "An Internet based wireless home automation system for multifunctional devices," *IEEE Trans. Consum. Electron.,* vol. 51, pp. 1169-1174, 2005.

[8] O. A.H, "Keeping S. Bluetooth Low Energy: Wireless Connectivity for Medical Monitoring," *J. Diabetes Sci. Technol.,* vol. 4, pp. 457-463, 2010.

[9] F. C. a. C. L. W. M. Y. E. Simik, "Design and Implementation of a Bluetooth-Based MCU and GSM for Wetness Detection," *IEEE Access,* vol. 7, pp. 21851-21856, 2019.

[10] H. Hihara and e. al, "Applications of Reconfigurable Processors as Embedded Automatons in the IoT Sensor Networks in Space," *Asai, S. (eds) VLSI Design and Test for Systems Dependability. Springer, Tokyo,* 2019.

[11] I. A. M. G. Atzori L., " The Internet of Things: A survey.," *Comput. Netw.,* vol. 54, pp. 2787-2805, 2010.

[12] V. A. J. B. S. R. P. J. E. A. M. E. H. G. O. G. P. R. e. a. Fafoutis X., "Designing Wearable Sensing Platforms for Healthcare in a Residential Environment," *EAI Endorsed Trans. Pervasive Health Technol. ,* vol. 3, 2017.

[13] H. Y. O. T. K. H. H. T. Kuwabara K., "Wearable blood flowmeter appcessory with low-power laser Doppler signal processing for daily-life healthcare monitoring," in *36th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, Chicago , 2014.

[14] P. S. M. F. R. L. Amaro J.P., "Bluetooth low energy profile for MPU9150 IMU data transfers," in *IEEE 5th Portuguese Meeting on Bioengineering (ENBENG)*, Coimbra, Portugal, 2017.

[15] F. E. G. D. M. B. M. I. runelli D., "Design Considerations for Wireless Acquisition of Multichannel sEMG Signals in Prosthetic Hand Control," *IEEE Sensors,* vol. 16, p. 8338–8347, 2016.

[16] C. C. W. C. D. R. Townsend K., Introduction. In: Sawyer B., Loukides M., editors. Getting Started with Bluetooth Low Energy: Tools and Techniques for Low-Power Networking., Sebastopol, CA, USA: O'Reilly Media, Inc, 2014.

[17] S. Campbell, "Basics of the I2C communication protocol," Circuit Basics, 2016. [Online]. Available: https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/.

[18] Bosch Sensortec, "Gas Sensor BME688," Bosch, [Online]. Available: https://www.bosch-sensortec.com/products/environmental-sensors/gas-sensors/bme688/.

[19] Analog Devices, *ADXL343 Documentation.*

[20] A. N. Purwandaru, "Getting Started With Flutter Bloc Pattern," Mitrais, 22 Oct 2021. [Online]. Available: https://www.mitrais.com/news-updates/getting-started-with-flutter-bloc-pattern/#:~:text=Bloc%20is%20a%20design%20pattern,and%20maintained%20by%20Felix%20Angelo..

[21] M. Johnson, "What Is A Pull Request?," OSS Watch, 8 Nov 2013. [Online]. Available: http://oss-watch.ac.uk/resources/pullrequest#:~:text=Push%20the%20branch%20to%20your,and%20how%20it%20is%20implemented.

[22] A. Theodorus, "How To Write Reactive Apps in Flutter Using Flutter Bloc," Better Programming, 22 Jun 2021. [Online]. Available: https://betterprogramming.pub/create-reactive-apps-in-flutter-using-multiblocproviders-45e6f2c8598e.

[23] Renesas Electronics Corporation, "DA145XX Tutorial Create a Custom GATT Profile," 17 Jan 2020. [Online]. Available: http://lpccs-docs.dialog-semiconductor.com/tutorial-custom-profile-DA145xx/gatt.html.