Watipatsa Wishesighz Nsunza  
(I201220124)

MULTI BLUETOOTH  
LOW ENERGY SENSOR CONNECTOR  
(DBT#22)

Communications Project

Assigned by: Professor HuHai  
11/15/2015
ABSTRACT

In this project, my team’s goal was to develop a better Basketball Training System than the one that exists in the world today. We've realized that the Methods Coaches are using to condition Basketball players with right now are no longer as effective as they used to be, because currently our players are stronger, faster than any players in History that ever played the game. DBT#22 is a training system that is being developed to properly train and position Basketball Players to the next level of Basketball. With the DBT#22, coaches no longer need to focus on training Basketball players but simply monitoring their progress as the system analyzes the player at scheduled times. The system will help coaches build much better and stronger teams by knowing for the first time the ingredients needed for a strong offense and defense lineup, this system will help them analyze that.

One of the key steps in developing this system is to invest in an optimal Motion Capture System. After extensive research, we realized a lot of outdoor and fitness systems are being developed on Bluetooth low energy (Bluetooth Smart), mainly because it is a low power technology that can cover up to a distance over 150 meters with only 3volts, which is very remarkable for such small device, making it applicable to almost any design. Most Outdoor activities don't require such a long distance especially Basketball which is played inside a 28.7m x 7.2m court. For this reason we decided to develop the DBT#22 on top of the TI CC2540/41 Bluetooth low energy Chips. This hardware is very low cost and any system successfully developed on this Chip would have a High market simply because the hardware is easy to produce and purchase in bulk with lower prices, especially here in China.

The GY-Sensor is also used in the DBT#22 motion capture system. We used the GY-87 IMU to collect 3-Dimensional Spatial Data from the Accelerometer, Gyroscope, and Magnetometer/Barometer (to measure air pressure). Each of the sensors Inside the GY-87 sensor has a 3-Dimensional space and can be used to send all the Inertial Measurement independently, but to get a more precise reading we need to make the sensors work together to bring optimal results. For this test we were not able to wire the GY-Sensor to the Bluetooth and call the data from the sensors wirelessly, simply because we didn't have enough time to learn how to interface the sensor to the Bluetooth via I2C after multiple failures of trying to. We did however find that it was possible to interface this connection because TI and ByteReal managed to interface this I2C connection and the ByteReal "realtag" had our exact design but they provided no firmware or datasheet to achieve this. TIs “SensorTag” had a similar connection, we tried to follow the datasheet and instructions and managed to interface the connection but the results were not successful, the data could not be read from the I2C pins, we believe this is simply because we have not done enough reading on this part due to time to figure out how to call these I2C addresses. Using the Arduino Uno, I was able to use some sample code from the GY-Sensor developers to read data from the hex addresses of the Accelerometer, gyroscope, and magnetometer/ barometer in the device and
discovered the address of each sensors register using the I2C_scanner code for the Arduino.

The next phase was to develop a Java Application to read from these addresses and connect to the Bluetooth devices via UART through the TI CC2540 USB Dongle. The CC254x series along with other BLE Devices from TI operate using GATT commands. TI developed an application to connect their SensorTag to the PC called "Btool" using the USB dongle. I learned the basic connection commands and how the sensor should respond when certain commands are sent and then I developed a Java application using the same Hex call addresses. I started working on improving this App better but also due to time, the progress of this development has only remained at the basic level, what’s next is to call the register addresses of the IMU sensor using this same app and create a virtual Serial Port for every device connected.

Once the Java app is finalized and we can read all the data from the sensors and direct them to the right ports/ channels, the 3ds max can then read this data directly from the ports and this data can be utilized in this software using a language called Maxscript. Using Maxscript all the graphical objects can be assigned their specific port numbers to collect the data directly from the server end sensors. The score of the game can be updated systematically as the point is made, the player MOCAP data can be stored into a database and replayed over for analysis and system programming, creating a virtual environment that monitors and develops basketball skills and can be expendable and grow as it is trained by better players, and train upcoming teams with these skills. This system of training can adapt to any player quick.

Working on DBT#22 has been a great learning experience, I've learned to manage time better, manage a team better, and other resources including money. I have also acquired better programming skills and hardware designing skills, most importantly I have learned to focus, to choose a part to play at a time and not trying to focus on everything but to work hard at my goal and accomplish it. I've also learned that success is reached even at the end of failure because from every part that I didn't succeed in during this project, there was something to learn; either that I'm not able to accomplish the part yet or that it's not my part to play. In the midst of all this focus I discovered myself more, what my focus is, what my passion is and this has guided me to choose the right major for me after I graduate, it has given me a better direction to go and i'm very proud of this achievement. I'm looking forward to completing this system in the near future.

Project Cost:
Since the project has not made very much progress, all the devices I purchased were simple for testing:

- 2 x GY-87 - ¥100
- 2 x CC2541 - ¥30
- 1x CC2540 USB Dongle - ¥30

Total: ¥160

MULTI BLUETOOTH LOW ENERGRY SENSOR CONNECTOR

Bluetooth low energy(BLE) is a new communication technology that has been growing on todays market. The technology has been applied into many fields including fitness, medical, sports, and other health monitors; Bluetooth Low Energy is used in these applications to collect data from the sensors and transmit it wirelessly to a computer or Android or iOs devices within range. Bluetooth
Low Energy is optimal for many low latency applications because it is designed to operate on a 3.3v coin cell battery for years and is capable of communicating to another BLE device within a 150m radius, this is optimal for most outdoor activities, fitness exercises, and medical applications.

HARDWARE DESIGN

My focus during the course of the project was to design a Java PC Application that can create a virtual serial port to communicate with multiple BLE devices. We are using the Texas Instrument (TI) CC2541 Bluetooth Low Energy wireless chip attached to the GY-87 IMU Sensor which has 4 sensors; Accelerometer, Gyroscope, Magnetometer. Each of these sensors transmits 3-Dimensional spatial data that can be converted to create three dimensional motion, the purpose of designing this Java application is to transmit this motion capture data into a 3D processing application like Autodesk 3ds Max and analyze motion capture data collected from sensors attached to a Basketball Player and create a virtual playing environment and bring Basketball training to the next level. The sensor can also be used to read pressure values for health and fitness applications. Since Bluetooth Low Energy can only communicate with a device enabled with Bluetooth Low Energy, the first step was to make the PC a BLE device. To do this I discovered after extensive research that the TI CC2541 BLE Chip can only communicate with the PC through the CC2540 USB dongle. This was fundamental in to enable Bluetooth Low Energy support on the PC.
JAVA PC APPLICATION

The sole focus of this paper is to demonstrate the sensor read process while gathering and collecting data from the Accelerometer and Gyroscope sensor by simply plotting the spatial data coming in from the IMU sensor. The Java Application is broken down into three main processes created into three separate classes; The “graph_main” Graphical Interface, which allows the user to select the Com Port attached to the CC2540 USB Dongle, a “BLE HCI”, which is a process to scan for a discoverable device and connect to it, and the last process is the “PeakTracking” which graphs every value read from the sensors. In the following parts, I will talk about each part in detail, there will be images illustrating the output and functions along with code snippets to further explain the process.

GRAPHICAL INTERFACE

The DBT#22 Graphical User Interface is designed to make the process of connecting to the Bluetooth sensors very easy. With help of the “jfree chart” Library, the graphical interface design was not very time consuming, jfree chart has many packages such as; ChartFactory, ChartPanel, JfreeChart, ValueAxis, XYPlot, StandardXYItemRenderer, DynamicTimeSeriesCollection, Millisecond, Minute, TimeSeries, TimeSeriesCollection, XYDataset, ApplicationFrame, all for different applications. I also used the “javax.swing” Library to create the Jbuttons, ComboBox, Jpanels, JtextArea, JTextField, and a Timer.

[DBT#22 MULTI BLE SENSOR CONNECTOR]

The packages I used in this “graph_main” are to call certain functions in in the program when a
button is clicked or plot a graph of the values read from the sensors when the function is called in the program is called. Take for example the JButton “ports” on the DBT#22 GUI, when the “ports” button is clicked, this calls for the “port” “ActionListener” in the program;

```java
if(e.getActionCommand().equals("port"))
{
    SerialPortList plist=new SerialPortList();
    String[] ports=plist.getPortNames();
    //text.setText("Ports available: ");
    list.removeAllItems();
    for(int i=0;i<ports.length;i++)
    {
        list.addItem(ports[i]);
        //text.append(ports[i]+" ");
    }
}
```

the ActionListener and ActionEvent functions are provided in the java.awt.event package and the plist.getPortNames() assignment is made available by the Java-Simple-Serial-Connector Library “JSSC”.

The “Scan” JButton on the DBT#22 GUI calls for an ActionListener. The “Scan” ActionListener gets the selected “port” from the ports list “plist” to communicate with discoverable BLE devices and calls an “ActionEvent” (ble.port(); and ble.discover(); in “BLE_HCI” class which we will discuss in further detail in the next section.

```java
if(e.getActionCommand().equals("scan"))
{
    System.out.println(list.getSelectedItem());
    new Thread(new Runnable() {
        @Override
        public void run() {
            // do stuff in this thread
            try {
                ble.init(list.getSelectedItem().toString());
                ble.port();
                ble.discover();
            } catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }).start();
}
```

The call “ble” refers to the BLE_HCI class which is assigned using the following statement.

```java
static BLE_HCI2 ble=new BLE_HCI2();
```

The “Zoom in” JButton on the interface does exactly what it states, it zooms in on the graph during the read from the sensor to give a narrower view of the data coming in from the sensors by calling the zoom_in ActionListener:
if(e.getActionCommand().equals("zoom_in"))
{
    range=range-500;
    if((3000-range)>100)
        xaxis.setFixedAutoRange(3000.0+range);
}

The Initial range on the X-Axis is 3000 and a click on the button subtracts 500 from the initial value to narrow the specteral length.

There's also the “Zoom out” Jbutton right next to eat ro reverse the view back to give a wider range of the data coming in through the sensors using the “zoom_out” ActionListener;

if(e.getActionCommand().equals("zoom_out"))
{
    range=range+500;
    xaxis.setFixedAutoRange(3000.0+range);
}

This adds 500 to the current value until 3000 max range is reached.

The “BPM” JTextArea of the GUI is used to display and record the number of BEATS PER MINUTE read from the pressure sensor int the IMU, this section is discussed in futher detain in the section “PEAK TRACKING” and the “Save” Jbutton and “file name” JTextField which calls the following ActionListener are discussed in more detail in the “BLE_HCI” section;

if(e.getActionCommand().equals("save"))
{
    if(ble.write)
    {
        ble.write=false;
        button5.setText("save");
    }
    else
    {
        ble.write=true;
        button5.setText("stop");
    }
}

The “Save” Jbutton simply checks whether write has been enabled, if it is enabled, the text on the button will change to “stop” to stop tracking and if it is disabled, the button will read save and Peak tracking can be enabled by clicking “save”. The “file name” JTextField is used to set the file name to save the tracking.

**BLE HCI**

In this section we discuss further the tasks that the two functions called inside the “scan” ActionListener inside the “graph_main” (ble.port(); and ble.discover(); class perform, in this section I explain better what an ActionEvent is.
The function ble.discover is an ActionEvent discovery; as I explained in the previous section, “ble” in this project is calling this **“BLE_HCI”** class. Inside the BLE_HCI class there are many events that call specific actions to performed to establish a bluetooth connection between a device and the PC. This is a series of communication processes that are used as the standard operations to establish a bluetooth connection. To learn these methods, it was necessary to get familiarized with TI's development tools that explain better how to establish pairing between ble devices and the PC using TI's application called Btool. Using TI's application and reference manuals, I was able to learn better how to bring two BLE devices into discovery mode, pair the devices, and also call certain “characteristics” of a sensor. The following is a sample output from TI's “Btool” application:

**EVENT INITIALIZE COM PORT(7)**

********************************

Port opened at 11/19/2015 5:06:56 PM

********************************

- Type : 0x01 (Command)
- OpCode : 0xFE00 (GAP_DeviceInit)
- Data Length : 0x26 (38) byte(s)
- ProfileRole : 0x08 (8) (Central)
- MaxScanRsps : 0x05 (5)
- IRK : 00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00
- CSRK : 00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00
- SignCounter : 0x00000001 (1)
- Dump(Tx)://****tx_init
0000:01 00 FE 26 08 05 00 00 00 00 00 00 00 00 00 ...&............
0010:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .............
0020:00 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0030:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0060:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0070:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0080:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0090:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00A0:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
---

After studying the commands that need to written to the port to initialize a “service discover” event and also the acknowledgements from the device when it reads a recognized action call, I then went on to assign the “ActionEvent” bytes into “byte arrays” and the “acknowledgement” into other “byte arrays” in the following manner:

```java
static int[] scan_over={0x06,0x01},
    device_info={0x06,0x0D},
    event_init={0x06,0x00},
    event_link={0x06,0x07},
    event_notification={0x05,0x13},
    event_data={0x05,0x1b},
    ter={0x06,0x06};
```

```java
static int channel=0xf5f5;
static int fil[] = new int[150];
static int fpointer=0,fsize=1;
static boolean finit=false,header=false,write=false;
```

// bt device address
```java
static int[] device_address=new int[6];
//static int[] device_address={0x00,0x15,0x83,0x00,0x3C,0xF3};
```
static int[] IMUsensor1 = new int[6];
static int[] IMUsensor2 = new int[6];
// static int[] IMUsensor3 = new int[6];

// received packet
static int[] packet = new int[100];
static int[] event = {0x00, 0x00};

// packet size
static int packet_size;

// tx dump for discovery, link and enable notification
static int[] tx_init = {
    0x01, 0x00, 0xFE, 0x26, 0x08, 0x05, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
};
static int[] tx_discovery = {
    0x01, 0x04, 0xFE, 0x03, 0x03, 0x01, 0x00,
};
static int[] tx_link = {
    0x01, 0x09, 0xFE, 0x09, 0x00, 0x00, 0x00,
};
static int[] tx_enable_notification = {
    0x01, 0x92, 0xFD, 0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
};
static int[] terminate = {
    0x01, 0x0A, 0xFE, 0x03, 0x00, 0x00, 0x13,
};

To begin the write process of the byte arrays, we can first specify some connection settings, example “BAUDRATE” which is set to 115200 readable bits for BLE, we can also set the “STOPBITS”, “DATABITS”, and “PARITY”, all these settings are defined better in Tis Bluetooth Low Energy Developers Guide, for my project I used the following code before beginning the “discovery”:

```java
public void port() throws SerialPortException, InterruptedException {
    port.openPort();
    port.setParams(SerialPort.BAUDRATE_115200, SerialPort.DATABITS_8,
                   SerialPort.STOPBITS_1, SerialPort.PARITY_NONE);
}

public void discover() throws SerialPortException, InterruptedException {

    // TODO Auto-generated method stub
    /*
    port.openPort();
    port.setParams(SerialPort.BAUDRATE_115200, SerialPort.DATABITS_8,
                   SerialPort.STOPBITS_1, SerialPort.PARITY_NONE);
    */
    System.out.println("Scanning....");
    graph_main.text.append("\n Scanning....\n");
    port.writeIntArray(tx_init);
    process_event(event_init);
    port.writeIntArray(device_address);
    port.writeIntArray(tx_discovery);
    process_event(scan_over);
    if (device_found)
```

During an ActionEvent “discovery”, the following code is run to process all the actions in the event discovery of BLE devices;

```java
    public void discover() throws SerialPortException, InterruptedException {
        // TODO Auto-generated method stub
        /*
        port.openPort();
        port.setParams(SerialPort.BAUDRATE_115200, SerialPort.DATABITS_8,
        SerialPort.STOPBITS_1, SerialPort.PARITY_NONE);
        */
        System.out.println("Scanning....");
        graph_main.text.append("\n Scanning....\n");
        port.writeIntArray(tx_init);
        process_event(event_init);
        port.writeIntArray(device_address);
        port.writeIntArray(tx_discovery);
        process_event(scan_over);
        if (device_found)
```
The "discover" process is began when the "Scan" button is pressed on the "DBT#22 Multi Connector" GUI, then when event discover is called from the "graph_main" class using the line:

```java
ble.port();
ble.discover();
```

The class "BLE_HCI" refered to as "ble" is called and the event discover is executed in the java class BLE HCI, also notice that the function "port" is called first which we discussed earlier referring to the selected Com Port we are communicating on.

After the discover process has started, the selected comport is written with the Transmission Initializing byte array "tx_init" which has number of bytes that need to be sent over the port, once the byte array is written, the BLE device responds with the bytes inside the event initialization array "event_init". Next we write the "tx_discovery" byte array to the port and the BLE device responds with the bytes inside the byte array "scan_over", this ends the discovery process and we copy the device address and send another discovery for the rest of the devices, until all devices are connected. We have to send a "service discovery" each time because at this time all the bluetooth sensors are responding and we can only connect to one at time, I'm still working on optimizing this code better to discover the sensors at once and the option select the ones to connect to if not all.
All events in BLE_HCI are called using the following “process_event” function:

```java
public void process_event(int[] ev) throws SerialPortException {
    // TODO Auto-generated method stub
    do{
        get_packet();
        if(packet_size>2){
            get_event();
            // if device found copy device address
            event_task();
        }
    }while(!event_equal(ev));
}
```

In the `process_event` function we read the bytes from the BLE device to determine the event to call every time we receive a byte packet larger than 2, `event_task` is the process of acquiring the device address while the event is not `event_equal`, the address is copied to an array “Device Address[]” using the following code:

```java
private void event_task() {
    if(event_equal(device_info)) {
        for(int i=5;i<11;i++)
            device_address[i-5]=packet[i];
        int[] DiscoverArr = Arrays.copyOfRange(packet,5,11);
        for(int num:DiscoverArr){
            String hex = Integer.toHexString(num & 0xFF);
            if(hex.length()==1){
                hex = '0'+hex;
            }
            System.out.println(hex.toUpperCase());
            device_found=true;
        }
    }
    if(event_equal(event_data)) {
```
One of the events processed during an ActionEvent in the BLE_HCI class is `device_info` which is an action event specifically for copying a devices address, once a BLE device is found, the device replies with it's address during the “scan_over” event, we can then copy the device address during this stage in the discovery process. All events of the program are under “event task”

We use the event `event_data` to obtain data from the BLE device transmitting data, all the data is stored into the `packet[]` array and is processes using the same is called using the “process_event”, and all the actions inside are performed; we plot the data on “jfree chart” XY plotter inside the “DBT#22 Multi Connector” GUI byte after byte until stopped, the “save” option to save the file is also called when “write” is enabled after the “save” button click, all the data packets are saved into a file that can be read to review this data.

```java
if(event_equal(event_data))
{

    // #############for single plot
    for(int i=8;i<16;i=i+2)
    {
        int temp=Integer.rotateLeft(packet[i+1],8)+packet[i];
        if(temp!=0xffff)
        {
            try {
                write2file(temp,graph_main.filename.getText());
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        track.process_data(temp);
        graph_main.lastValue=temp/0.07536*50/temp;
        graph_main.series.addOrUpdate(new Millisecond(), graph_main.lastValue);
    }
}
```
This is discussed further in the Peak Tracking of this paper.

The following line of code is used to write the data into the file when the ActionEvent calls for "write2file":

```java
private void write2file(int temp, String fname) throws IOException {
    // TODO Auto-generated method stub
    File f = new File("E:/logged_data/" + fname);
    if (!f.exists()) {
        f.createNewFile();
    }
    BufferedWriter wr = new BufferedWriter(new FileWriter(f, true));
    wr.append(temp + ",");
    wr.flush();
    wr.close();
}
```

This code checks if a file already exists with the file specified in the JTextField "fname" and creates a new file. The rest of the code on the BLE_HCI class is to compare 2 events and to read the packets to call any of the above events discussed:

```java
// compare 2 events
private boolean event_equal(int[] ev) {
    // TODO Auto-generated method stub
    if (event[0] == ev[0] && event[1] == ev[1])
        return true;
    else
        return false;
}
```

```java
// get event code from the packet
private void get_event() {
    // TODO Auto-generated method stub
    //System.out.println("event:" + event[0] + event[1]);
    event[0] = packet[1];
    event[1] = packet[0];
    //System.out.println("event:" + event[0] + event[1]);
}
```

```java
// get packet
private void get_packet() throws SerialPortException {
    packet = port.readIntArray(3);
    packet_size = packet[2];
    //System.out.println("Packet_size:" + packet_size);
    packet = port.readIntArray(packet_size);
}
```
The actions are performed when “event_equal”, “get_event”, and “get_packet” actions discussed above are called. And the final part of the code is called whenever the “port” command is called.

// create link with specified parameter
public void init(String p) throws SerialPortException {

   //Define COM PORT
   port=new SerialPort(p);
   final Marker start = new ValueMarker(3400000.0);
   start.setPaint(Color.red);
   start.setLabel("Current Value");
   start.setLabelAnchor(RectangleAnchor.BOTTOM_LEFT);
   start.setLabelTextAnchor(TextAnchor.TOP_LEFT);
   // plot.addRangeMarker(start);
   graph_main.plot.addRangeMarker(start);
   //open port and configure port
}

This opens a port to communicate with the BLE device. This marks the end of this section and in the final section we will discuss further about the plotting of the data packets coming through the “port”.

PEAK TRACKING

In order to “save” the peaks of the graph, we use four main packages;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

These java input and output packages have the functions that we can call to write to a file, the following code shows how the data packets which are collected in the “BLE_HCI” class are interpreted to plot the data unto a read graph;

```java
static int max=0, time=0, index=0, count=0, min=0, inc=0, m1;
static boolean start=false, mnt=true, s=false;
public void process_data(int temp) {
    // TODO Auto-generated method stub
    time=time+1;
    if(mnt){
        if(max<temp){
            max=temp;
            index=time;
            start=true;
        }
        else if(start){
            count=count+1;
        }
        if(count==150){
            //System.out.println("max:"+max+" at pos. "+index);
            graph_main.BPM.setBackground(Color.GREEN);
            count=0;
            min=(int)Math.pow(2, 12);
            start=false;
            mnt=false;
        }
    }
    else {
        if(min>temp){
            min=temp;
            start=true;
            index=time;
        }
        else if(start){
            count=count+1;
        }
        if(count==30){
            //System.out.println("min:"+min+" at pos. "+index);
            graph_main.BPM.setBackground(Color.YELLOW);
            count=0;
            max=0;
            start=false;
            mnt=true;
            if(!s){
                s=true;
                m1=index;
            }
            inc=(inc+1)%10;
        }
    }
}
```
```java
if (inc == 0)
{
    int dx = index - m1;
    double bpm = (double) (400 * 9) / dx * 60;
    System.out.println("BPM:" + bpm);
    graph_main.BPM.setText("BPM:" + (int)bpm);
    try {
        write2file(bpm,"bpm.csv");
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    s = false;
    time = 0;
}
}

private void write2file(double temp, String fname) throws IOException {
    // TODO Auto-generated method stub
    File f = new File("E:/logged_data/" + fname);
    if (!f.exists())
    {
        f.createNewFile();
    }
    BufferedWriter wr = new BufferedWriter(new FileWriter(f, true));
    wr.append(temp + ",");
    wr.flush();
    wr.close();
}
}

When the event “process_data” during the “event_data”, the graph is plotted byte after byte. Notice that when the “process_data is called in the class BLE_HCI “ble”, the class “PeakTracking” is referred to as “track” now;

    track.process_data(temp);
    graph_main.lastValue = temp; //0.07536*50/temp;
    graph_main.series.addOrUpdate(new Millisecond(),
    graph_main.lastValue);
    header = false;
    //} //
}

this is due to the following class link specified at the start of the class “BLE_HCI”

PeakTracking track = new PeakTracking();

The data received from the byte packets can also be logged byte after byte into the “file” that is created using a buffered writer, file, and file writer packages. The BPM “beats per minute” read from the pressure sensor are recorded byte after byte into the file “bpm.csv” when “save” file is enabled and are plotted on the jfree chart plotter.
ACKNOWLEDGEMENT

This marks the end of the demonstration of designing a Java Application to collect sensor readings from a Bluetooth Low Energy “BLE” sensor. I hope by this paper many more uses for Bluetooth Low Energy can be discovered to easily connect and transmit data through multiple devices. The progress on the final project for Digital Basketball Trainer “DBT #22” is still in the first stages, but this marks the beginning of our system in development.

I thank my God helping progress and the knowledge I was receiving to implement this Idea. I thank my parents for always working hard for us and proving me the opportunity to learn at Huazhong University of Science & Technology and the many sacrifices they make for my learning. I also thank my fiancé Tonthozo for her inspiration, support, and encouragement. I thank professor HuHai for assigning us this project early enough for us to improve our learning and prepare well for the thesis work ahead of us; I would not have been able to do it without this support.