[UPPER BODY EXERCISE LABLEING SYSTEM]

SOFTWARE DOCUMENTATION

PROJECT MANAGER – PAOLO NARDI, JOHAN HAKANSSON

REVISION DATE: 21/8/18

REVISION: 2

REVISION HISTORY

REVISION NUMBER	DATE	COMMENT
1.0	MAY 31,2018	First version
2.0	August 21,2018	Final version

Table of Contents

REVISION HISTORY **2**

TABLE OF CONTENTS **3**

DOCUMENT OVERVIEW **4**

SCOPE 4
AUDIENCE 4
RELATED DOCUMENTATION 4
DOCUMENT CONVENTIONS 5

SYSTEM OVERVIEW **6**

SYSTEM OVERVIEW DESCRIPTION 7
REMAINING DOCUMENT STRUCTURE 8
SOFTWARE PACKAGES 8
BLE TECHNOLOGY *8*
FIREBASE REAL TIME DATABASE *8*
BGSCRIPT *8*
AI2 MIT APP INVENTOR *9*
MATLAB *9*

SOFTWARE IMPLEMENTATION **9**

DATA ACQUISITION UNIT 9
OVERVIEW OF DATA ACQUISITION UNIT *9*
DATA ACQUISITION PROGRAMMING AND SET UP *11*
DATA LABELING APPLICATION 19
DATA LABELING APP SEQUENCE DIAGRAM AND APP OVERVIEW *19*
DATA LABELING MIT APP INVENTOR CODE IMPLEMENTATION *24*
REAL TIME APPLICATION AND MATLAB SERVER 39
REAL TIME APPLICATION A ND MATLAB SERVER OVERVIEW *39*
REAL TIME DATA MIT APP INVENTOR CODE IMPLEMENTATION *41*
MATLAB REAL TIME DATA VISUALIZATION SERVER IMPLEMENTATION *49*

DATABASE DOCUMENTATION **56**

DATA BASE DESIGN 56

SYSTEM INTERFACES **57**

BLE GAP AND GATT PROTOCOL *57*
TCP PROTOCOL *57*

SYSTEM PERFORMANCE / CONCLUSION **57**

DOCUMENT OVERVIEW

This document has been developed by Paolo Nardi and Johan Hakansson. This document was developed from Kristianstad University HKR

SCOPE

In this document we will demonstrated the code functionalities which enable the acquisition, upload and real time visualization of the data obtained from the upper body exercise labeling system. The following tools were used to be able to create the software for this project:

- **AI2 MIT APP INVENTOR**
- **BGSCRIPT (Programmed with NOTEPAD ++)**
- **MATLAB**
- **JAVA (For the creation of extensions for MIT APP INVENTOR)**
- **DATA BASE STORAGE**
- **REAL TIME DATA LOGGING**

AUDIENCE

This document is intended for students and developers. To better understand the flow of the software, one should have a basic knowledge and understanding of MATLAB, BGSCRIPT, MIT APP INVENTOR, a basic understanding of Bluetooth low energy technology and some experience programming micro-controllers. Alongside with all these skills, developers are expected to know how to read and understand sensor data sheets, since 3 different sensors where used to create the proposed system.

RELATED DOCUMENTATION

-Bluegiga BLE 113 Smart module: The following link contains documents in pdf format one can look at to better understand the BLE module used for this project alongside it's programming language; BGSCPRIT and the Bluetooth low energy technology.

https://www.silabs.com/products/wireless/bluetooth/bluetooth-low-energy-modules/ble113-bluetooth-smart-module

Some of the documents downloaded from this site to understand the basic principles of BGSCPRIT and the BLE 113 Smart module include:

1-BLE_getting_started_v1.4.pdf

2-BLE_Stack_API_reference_v2.2.pdf

3-BLEGUI_User_Guide_v1.7.pdf

4-BGScript_developer_guide_v2.4.pdf

5-BLE113-DataSheet.pdf

6-BLE112_Module.pdf

-AI2 MIT APP INVENTOR: MIT App inventor is an open source tool that developers can use to create simple android applications on the go. The interface is easy to understand, by incorporating a drop-down block diagram programing scheme, where blocks are ensembled to create the logic of the program.

The following link will take you directly to the site where one can start creating applications right away.

http://ai2.appinventor.mit.edu/

For more detail documentation please check the following link:

http://appinventor.mit.edu/explore/content/reference-documentation.html

-ALTIMU-10 V4 GYRO, ACCELEROMETER, COMPASS, AND ALTIMETER (L3GD20H, LSM303D, AND LPS25H CARRIER): The following link contains the module that was used for testing and as reference to understand the basics behind the L3GD20H (Gyroscope) and LSM303(Accelerometer), which are used in this system. The data sheets for the 2 sensors are also in the link:

https://www.pololu.com/product/2470

(Check L3GD20H data sheet and LSM303D datasheet)

-Sparkfun Flex Sensor: The following links contain information regarding the usage of the flex sensor used in this project. The flex sensor acts as a voltage divider, who's output is read by the microcontrollers ADC converter and which then can be mapped to a specific flexing angle.

https://www.sparkfun.com/products/8606

http://www.instructables.com/id/How-to-use-a-Flex-Sensor-Arduino-Tutorial/

DOCUMENT CONVENTIONS

Easy to understand flow charts alongside sequence diagrams and user diagrams are present to aid developers better understand the usage of the software implemented. The flowcharts are written based on UML using the following website:

https://www.lucidchart.com/

SYSTEM OVERVIEW

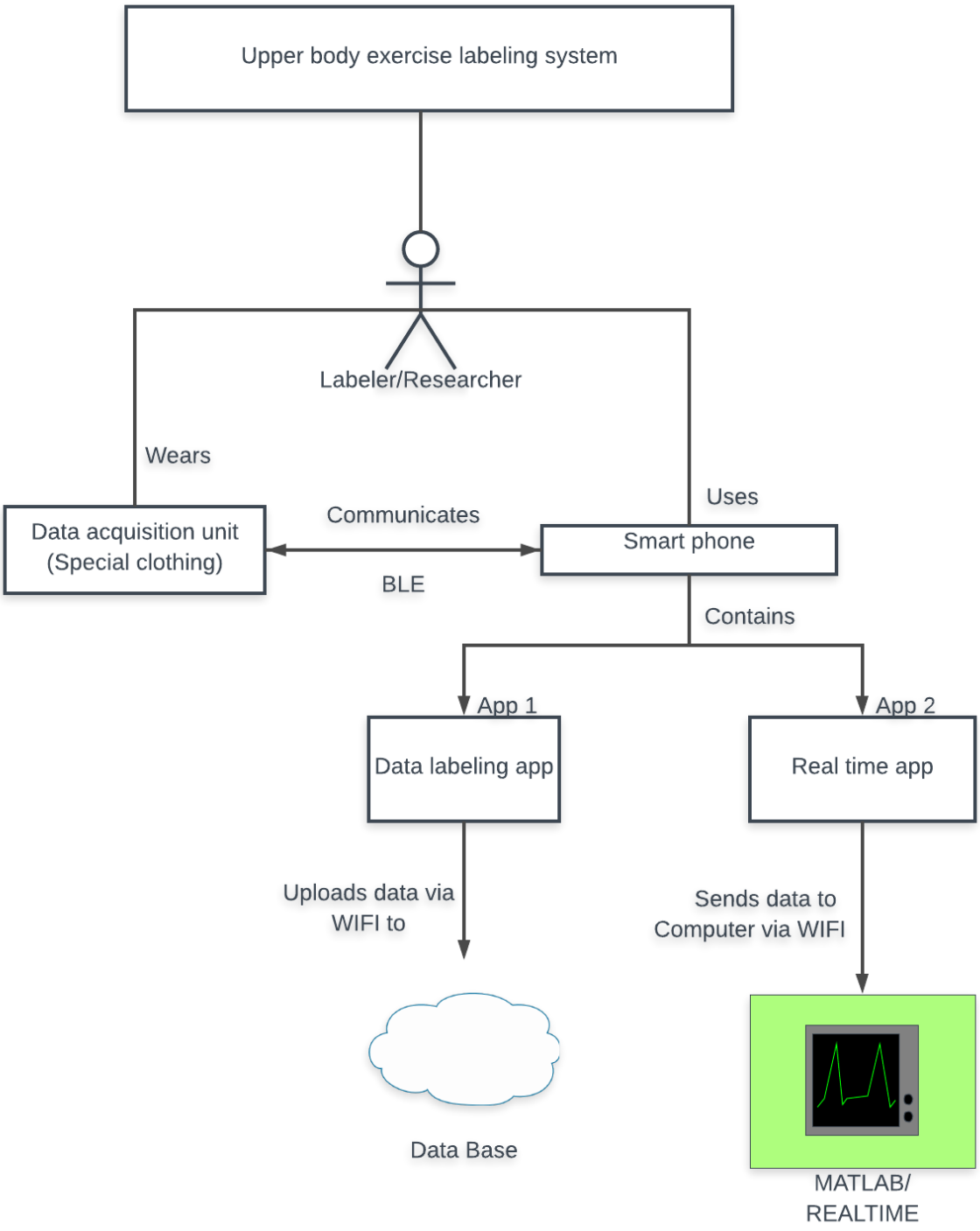


Figure 1: System overview

SYSTEM OVERVIEW DESCRIPTION

The proposed system consists of 4 main parts which work together to allow the user to either acquire data to be labeled and uploaded to the cloud from the special clothing or visualize the data in real time. The 4 parts are as followed:

1-Data acquisition unit: Special clothing, which consists of the Bluegiga 113 BLE smart module, 2 16-bit 3-axis gyroscopes(L3GD20H), 1 16-bit 3-axis accelerometer(LSM303D) and 1 flex sensor. The data acquisition unit is programmed using BGSCRIPT. The microcontroller simply sets up the sensors accordingly and waits for a connection. Once a connection is made, the device is now waiting for a command to start recording data. When this command is received, the module simply reads the correct registers from each sensor every 40ms and updates the corresponding UUID characteristic with the new value, thus achieving a data rate of 25Hz. Four unique UUID characteristics are used for this project, each characteristic can be read and notify the connected device when their value has changed. The four UUID's store the following values:

1-Wrist raw accelerometer data = 6 Bytes of data (Characteristic name = xgatt_wa)

2-Wrist raw gyroscope data = 6 Bytes of data (Characteristic name = xgatt_wg)

3-Shoulder raw gyroscope data = 6 Bytes of data (Characteristic name = xgatt_shoulder)

4-Flex sensor raw data = 1 Byte of data (Characteristic name = xgatt_flex)

2-Data labeling application: The following application allows the user to connect to the data acquisition unit, to be able to acquire and label the data accordingly to the exercise being performed. Once the agent selects the exercise to be performed, they will have 5 seconds to get into position, after the 5 seconds a beep will let the agent know that the data acquisition has begun. Once the agent is done performing the set, he/she can label if the exercise was performed correctly or incorrectly and choose to either upload the data to the database or disregard the data.

3-Real time application: The following application allows the user to connect to a TCP MATLAB server in order to visualize the system data in real time. This can be useful to see patterns emerge in real time, or to calibrate sensor values.

4-MATLAB server: The MATLAB application in the computer simply receives the raw data and post-processes the data to display sensor readings. This application can be used to enable developers to calibrate sensor readings and visualize data patterns in real time.

REMAINING DOCUMENT STRUCTURE

The rest of the document will discuss and demonstrate how the different applications work together alongside their corresponding code implementation. This will allow developers to be able to debug and modify the code according to their own needs. We will first discuss the technologies use to create the software. We will then explain how the data acquisition unit sets up and gathers the data from the different sensors and updates the corresponding GATT services and characteristics to be able to be read by the mobile applications. Furthermore, we will show how the connection between the data acquisition unit and mobile applications is established, alongside the corresponding protocol used between them to establish a reliable data sending link. Moreover, we will discuss how the data is processed by the mobile applications, how it's labeled and uploaded to the cloud or sent directly to the TCP MATLAB server. Finally, we will discuss the MATLAB server implementation for visualizing the data in real time, and the database connection between the data labeling application and the cloud.

SOFTWARE PACKAGES

BLE TECHNOLOGY

BLE is used to send data from a microcontroller to the smartphone application. BLE is a technology that allows user to send data wirelessly in a low energy matter. BLE technology supports asymmetric TX/RX Packet Sequence where one can send packets with up to 20Bytes of load data at a time, thereby achieving a maximum data transfer rate of approximately 305Kbps.

FIREBASE REAL TIME DATABASE

Firebase is a NoSQL real time database technology, which allows users to upload data real time into the cloud. Since firebase implements NoSQL technology, users have a lot of control behind the creation of the database. For the standard free version of firebase, users get a total of 1GB upload and 10GB download every month. For the moment this is enough storage, since the system is mainly build for research. If the system goes into further production, the database shall be updated to support multiple user support and upgraded for greater capacity. As of now, the database is created for 1 user in mind, but the software code is easily modifiable to link a unique database into the system. More information regarding the database implementation is given in the software documentation paper.

BGSCRIPT

BGSCRIPT is a simple scripting language that allows user to program the Bluegiga BLE 113 module. Each line of code in BGSCRIPT takes approximately 1ms to execute and the code can be written in Notepad++. Because of the long command execution time, our system is only able to achieve a consistent sample rate of 25Hz. The Bluegiga 113 module is a smart module, that goes directly into sleep mode, when no code is being executed. Thus, allowing developers to create energy efficient programs.

AI2 MIT APP INVENTOR

MIT App inventor is an open source tool that developers can use to create simple android applications on the go. The interface is easy to understand, by incorporating a drop-down block diagram programing scheme, where blocks are ensembled to create the logic of the program. MIT app inventor doesn't support BLE out of the gate, but extensions are available to be downloaded. App inventor also doesn't support TCP client connectivity, therefor a client-socket extension had to be downloaded and modified to fit our project.

MATLAB

MATLAB is a powerful tool for data logging, filtering and visualization. MATLAB is used in this project to be able to visualize the data obtained from the system in real time. This can allow developers to calibrate sensors accordingly, allowing developers to build a more reliable database.

SOFTWARE IMPLEMENTATION

DATA ACQUISITION UNIT

OVERVIEW OF DATA ACQUISITION UNIT

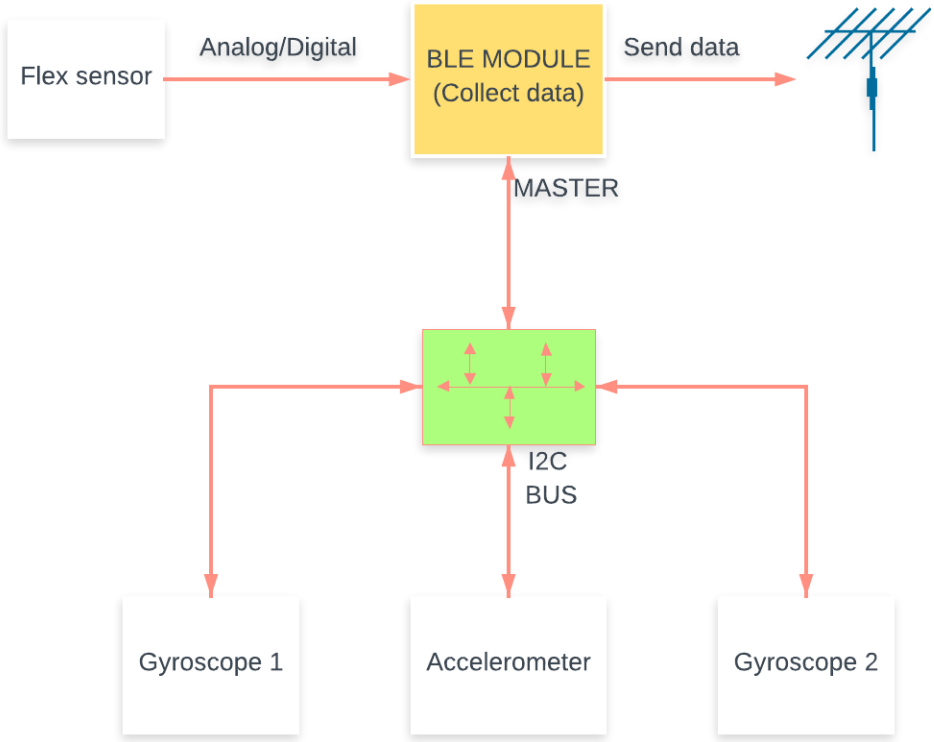


Figure 2: Overview of Data acquisition unit

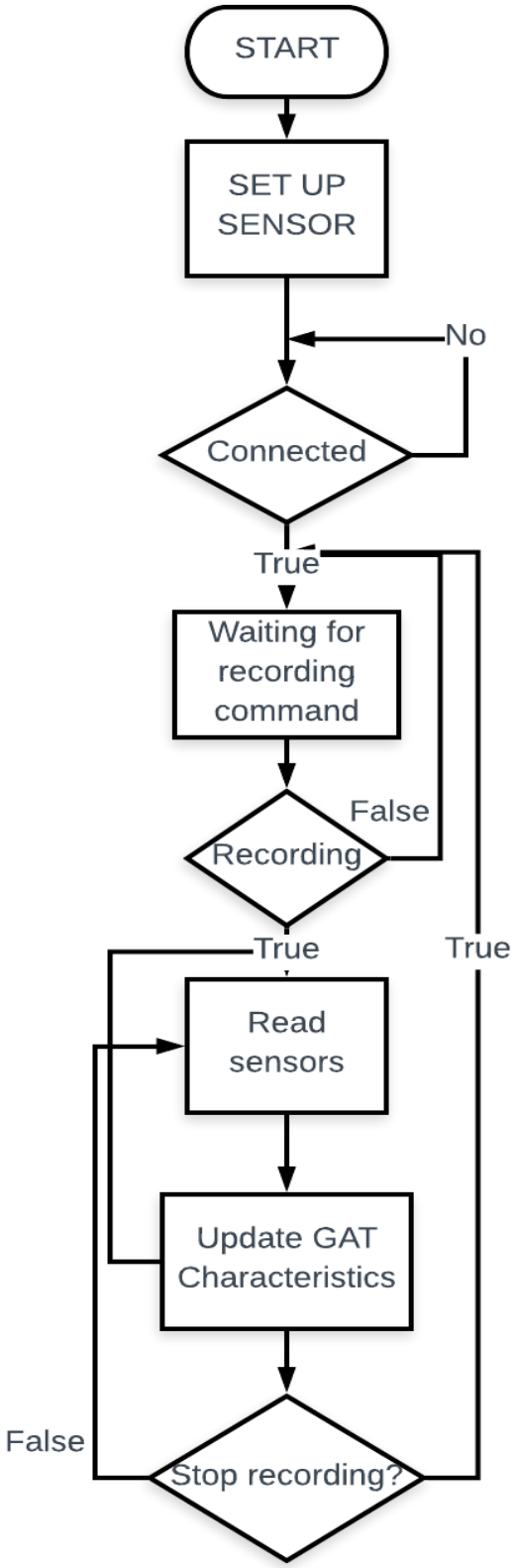


Figure 3: Program flow of Data acquisition Unit

DATA ACQUISITION PROGRAMMING AND SET UP

The data acquisition unit uses the Bluegiga 113 BLE smart module. To program this module, one needs to create 4 independent files as followed:

- An xml file which is in charge of setting up the corresponding hardware configuration for the BLE module, for the program at hand. In our program this file is called hardware.xml
- An xml file that is in in charge of configuring the different GATT Services which our application needs. In our program this file is called gatt.xml
- A BGSCRIPT file (.bgs) which programs the logic behind the data acquisition unit. For our project this file is called CO-FIT.bgs
- Finally a .bgproj file which is in charge of adding all the previous files together to create the whole program: CO-FIT.bgproj. This file is then opened using the Bluegiga BLE SWE Update tool which allows us to program the BLE Module. Once the code is run, 3 new files will appear, a .hex file which contains the binary code of our program, and 2 txt files containing information about the program.

The program folder for our application can be seen below:

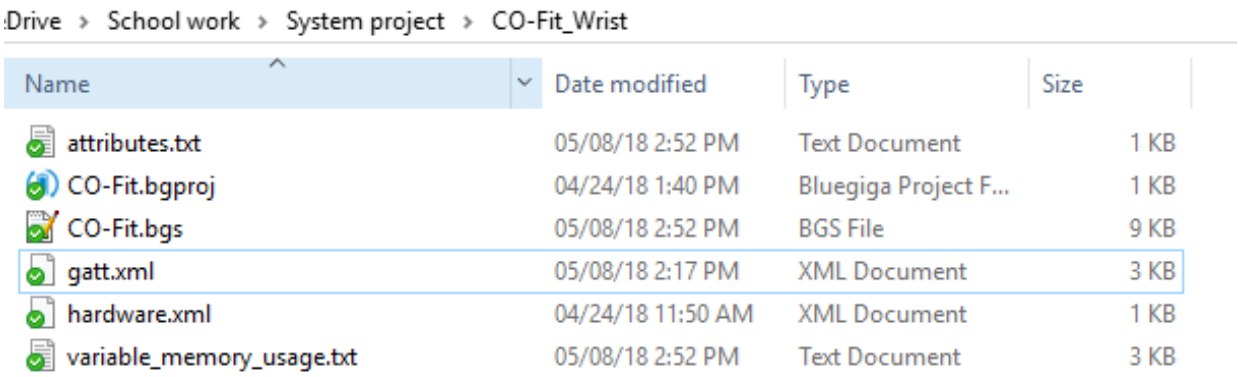


Figure 4: Folder containing required files to program BLE module

Code sample and explanation of each file is given in the following pages

-Hardware.xml:

```
<?xml version="1.0" encoding="UTF-8" ?>

<hardware>
    <sleeposc enable="true" ppm="50" />
    <usb enable="false" />
    <txpower power="15" bias="5" />
    <script enable="true" />
    <slow_clock enable="false" />
    <pmux regulator_pin="7" />
    <port index="0" tristatemask="0" pull="down" />
    <port index="1" tristatemask="0" pull="down" />
    <port index="2" tristatemask="0" pull="down" />
    <uartboot channel="1" />
</hardware>
```

Figure 5: hardware.xml file

The important things to consider in this file are the following:

-To allow the system to fall asleep when no commands are being run one must enable the sleep oscillator -> (sleeposc = true).

-To enable the device to read code from BGSCRIPT script enable must be set to true

-The slow_clock must be set to false to establish a solid I2C communication

The rest of the set up is the standard set up used by most Bluegiga code examples.

-Gatt.xml:

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>

    <service uuid="1800">
        <description>Generic Access Profile</description>

        <characteristic uuid="2a00">
            <properties read="true" const="true" />
            <value>CO-Fit</value>
        </characteristic>

        <characteristic uuid="2a01">
            <properties read="true" const="true" />
            <value type="hex">0000</value>
        </characteristic>
    </service>

   <service type="primary" uuid="180A" id="manufacturer">
        <characteristic uuid="2A29">
            <properties read="true" const="true" />
            <value>Bluegiga</value>
        </characteristic>
    </service>
```

```
<service uuid = "020e47b6-bcff-4332-a14d-ce883b858e68" advertise="true">
  <description>Check</description>

      <characteristic uuid="c91bb144-3d87-4e85-82ba-0c18826fb2a9" id="xgatt_tot_accel">
        <description>Total written in setup </description>
        <properties read = "true" notify = "true" />
        <value length="1" />
     </characteristic>

     <characteristic uuid="3adb28a9-e1f5-4a74-b990-4b1232f038ef" id="xgatt_tot_gyro">
        <description>Total written in setup </description>
        <properties read = "true" notify = "true" />
        <value length="1" />
     </characteristic>

  </service>

  <service uuid="181C" advertise="true">
     <description>Data</description>

       <characteristic uuid="e34b26a3-6c1b-497d-ae40-158f92732b69" id="xgatt_wa">
         <description>Wrist Accel Raw</description>
         <properties read = "true" notify = "true" />
         <value length="6" type ="hex" />
      </characteristic>

       <characteristic uuid="c481ad14-0f77-46dd-b4f2-9727d088e9d9" id="xgatt_wg">
         <description>Wrist Gyro Raw</description>
         <properties read = "true" notify = "true" />
         <value length="6" type ="hex" />
      </characteristic>


      <characteristic uuid="4a8d661e-069c-4f40-ad2c-fd21e1152beb" id="xgatt_shoulder">
        <description>Gyro Raw</description>
        <properties read = "true" notify = "true" />
         <value length="6" type ="hex" />
      </characteristic>

      <characteristic uuid="4e891122-821e-41ba-8526-f4d49498398c" id="xgatt_flex">
        <description>Flex Raw</description>
        <properties read = "true" notify = "true" />
         <value length="1" type ="hex" />
      </characteristic>

      <characteristic uuid="83017413-9312-4972-93de-9505bb56d2d2" id="xgatt_command">
       <description>Command</description>
      <properties write_no_response="true" />
      <value length="1" />
      </characteristic>


  </service>
```

Figure 6: Gatt.xml file

The GATT.xml file simply contains 3 services as followed:

1- General access profile service the first service with UUID 1800, enables devices to get information about the module used.

2- Check service with UUID 020e47b6-bcff-4332-a14d-ce883b858e68, contains the characteristics xgatt_tot_accel, and xgatt_tot_gyro. These characteristics contain the number of correct written set up instructions sent to our sensors, to allow developers to check if the sensors were set up appropriately. If the sensors where correctly set up, the xgatt_tot_accel should contain a value of 0x10 (16) and the xgatt_tot_gyro should contain a value of 0x14 (20).

3- Data service with UUID 181C. This service is in charge of displaying the sensor data as it is being read from the sensors. The service contains 5 characteristics as followed: Wrist Accel Raw which notifies the devices when new accelerometer data from the wrist accelerometer sensor has been read. Wrist Gyro Raw which notifies the devices when new gyroscope data from the wrist gyroscope sensor has been read. Shoulder Gyro Raw which notifies the devices when new gyroscope data from the shoulder gyroscope sensor has been read. Flex raw which notifies when a new adc reading has been taken from the flex sensor. Command characteristic which listens from commands from the mobile applications. This characteristic is used to start a recording session! If a command with character'1' arrives, the device starts recording and sending data! If a command with character'0' arrives the device stops recording data, until a new command is received.

-CO-FIT.bgs:

```
# API:
# call hardware_i2c_write(address,stop,data_len, data_data)(written)
# call hardware_i2c_read(address,stop,length)(result,data_len,data_data)
# call attributes_write(handle, offset, value_len, value_data)(result)

dim written
dim result
dim port
dim data
dim data_len
dim tot
dim timer
dim totA
dim totG
dim flex
dim check
const WHO_AM_I  = $0F #WHO_AM_I Register address
const v4Gyro    = $D6 #0x6B << 1 -> Pololu gyro slave address
const v4Accel   = $3A #0x1D << 1 -> Pololu accel slave address
const SparkL    = $D0 #0X68 << 1 -> Sparkfun gyro slave address low
```

```
 #-----------------------------------------------*****************************-----------------------------------------------------------------------------------------------------------
#For Gyroscope module configure the CTR_REG's as followed: (Slave address's: 0x68,0x69,0x6B) Slave addreses << 1 left shifted 1==(0xD0(208), 0xD2(210), 0xD6(214))
#CTR_REG1 = 0b00001111 -> Normal mode. 100Hz and activating x,y and z-axis -> 0x0F
#CTR_REG2 = 0x00 -> Default mode
#CTR_REG3 = 0x00 -> Deffault mode
#CTR_REG4 = 0x80 -> BDU = 1 (Update mode -> output registers not updated until MSB and LSB reading), 250 dps
#CTR_REG5 = 0x00 -> Default mode

#To disable gyro, write to CTR_REG_1 0x00 -> Power down mode + disable Zen-Xen

#-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
#-----------------------------------------------*****************************-----------------------------------------------------------------------------------------------------------
#For accelerometer module configure the CTR_REG's as followed: (Slave address 1D, left shifted 1 = 0x3A(58))
#CTR_REG_0 = 0x00 -> Default mode
#CTR_REG_1 = 0b01101111 -> 100HHZ, aenable AX -> AZ, BDU = 1 -> 0x6F
#CTR_REG_2 = 0x00 -> 2g mode
#CTR_REG_3 = 0x00 -> Default mode
#CTR_REG_4 = 0x00 -> Default mode
#CTR_REG_5 = 0b00011000 -> Disable temperature sensor, do not use Magnetic data -> 0x18
#CTR_REG_6 = 0x00 -> Default value
#CTR_REG_7 = 0b00000011 -> Magnetic sensor power-down mode! -> 0x03

#To power down accel write to CTR_REG_1 0x00 -> (Disables the x-z axis and puts device in power down mode)
#-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
procedure setup()
    totA = 0#Used to store total commands written properly to accelerometer sensors
    totG = 0#Used to store total commands written properly to gyroscope sensors
    #Setting up Accelerometer
      call hardware_i2c_write(v4Accel,1,2,"\x1F\x00")(written)#CTR_REG0 setup
      totA = totA + written
      call hardware_i2c_write(v4Accel,1,2,"\x20\x6F")(written)#CTR_REG1 setup
      totA = totA + written
      call hardware_i2c_write(v4Accel,1,2,"\x21\x00")(written)#CTR_REG2 setup
      totA = totA + written
      call hardware_i2c_write(v4Accel,1,2,"\x22\x00")(written)#CTR_REG3 setup
      totA = totA + written
      call hardware_i2c_write(v4Accel,1,2,"\x23\x00")(written)#CTR_REG4 setup
      totA = totA + written
      call hardware_i2c_write(v4Accel,1,2,"\x24\x18")(written)#CTR_REG5 setup
      totA = totA + written
      call hardware_i2c_write(v4Accel,1,2,"\x25\x00")(written)#CTR_REG6 setup
      totA = totA + written
      call hardware_i2c_write(v4Accel,1,2,"\x26\x03")(written)#CTR_REG7 setup
      totA = totA + written
      #Wrist gyroscope setup
      call hardware_i2c_write(v4Gyro,1,2,"\x20\x0F")(written)#CTR_REG1  setup
      totG = totG + written
      call hardware_i2c_write(v4Gyro,1,2,"\x21\x00")(written)#CTR_REG2  setup
      totG = totG + written
      call hardware_i2c_write(v4Gyro,1,2,"\x22\x00")(written)#CTR_REG3  setup
      totG = totG + written
      call hardware_i2c_write(v4Gyro,1,2,"\x23\x80")(written)#CTR_REG4  setup
      totG = totG + written
      call hardware_i2c_write(v4Gyro,1,2,"\x24\x00")(written)#CTR_REG5  setup
      totG = totG + written
```

```
    #Shoulder Gyroscope set up
    call hardware_i2c_write(SparkL,1,2,"\x20\x0F")(written)#CTR_REG1  setup
    totG = totG + written
    call hardware_i2c_write(SparkL,1,2,"\x21\x00")(written)#CTR_REG2  setup
    totG = totG + written
    call hardware_i2c_write(SparkL,1,2,"\x22\x00")(written)#CTR_REG3  setup
    totG = totG + written
    call hardware_i2c_write(SparkL,1,2,"\x23\x80")(written)#CTR_REG4  setup
    totG = totG + written
    call hardware_i2c_write(SparkL,1,2,"\x24\x00")(written)#CTR_REG5  setup
    totG = totG + written

  call attributes_write(xgatt_tot_gyro, 0, 1, totG) #Sending the number of

  call attributes_write(xgatt_tot_accel, 0, 1, totA) #Sending the number of correct set up

  call attributes_write(xgatt_tot_gyro, 0, 1, totG) #Sending the number of

end



# Buffers to hold the 6 bytes of acceleration, and rotation data from 3 sensors
dim sensor(6)
dim sensor2(6)
dim sensor3(6)

procedure send_data()
    # Read the acceleration X, Y, Z (high, then low bytes) registers
    call hardware_i2c_write(v4Accel,0,1,"\x29")(written)
    call hardware_i2c_read(v4Accel,1,1)(result,data_len,sensor(0))#Xhigh
    call hardware_i2c_write(v4Accel,0,1,"\x28")(written)
    call hardware_i2c_read(v4Accel,0,1)(result,data_len,sensor(1))#Xlow
    call hardware_i2c_write(v4Accel,0,1,"\x2B")(written)#Yhigh
    call hardware_i2c_read(v4Accel,0,1)(result,data_len,sensor(2))#Yhigh
    call hardware_i2c_write(v4Accel,0,1,"\x2A")(written)
    call hardware_i2c_read(v4Accel,0,1)(result,data_len,sensor(3))#Ylow
    call hardware_i2c_write(v4Accel,0,1,"\x2D")(written)#Zhigh
    call hardware_i2c_read(v4Accel,0,1)(result,data_len,sensor(4))#Zhigh
    call hardware_i2c_write(v4Accel,0,1,"\x2C")(written)#Zlow
    call hardware_i2c_read(v4Accel,1,1)(result,data_len,sensor(5))#Zlow
    # Read gyro X,Y,Z (high, then low bytes) registers
    call hardware_i2c_write(v4Gyro,0,1,"\x29")(written)
    call hardware_i2c_read(v4Gyro,0,1)(result,data_len,sensor2(0))#Xhigh
    call hardware_i2c_write(v4Gyro,0,1,"\x28")(written)
    call hardware_i2c_read(v4Gyro,0,1)(result,data_len,sensor2(1))#Xlow
    call hardware_i2c_write(v4Gyro,0,1,"\x2B")(written)#Yhigh
    call hardware_i2c_read(v4Gyro,0,1)(result,data_len,sensor2(2))#Yhigh
    call hardware_i2c_write(v4Gyro,0,1,"\x2A")(written)
    call hardware_i2c_read(v4Gyro,0,1)(result,data_len,sensor2(3))#Ylow
    call hardware_i2c_write(v4Gyro,0,1,"\x2D")(written)#Zhigh
    call hardware_i2c_read(v4Gyro,0,1)(result,data_len,sensor2(4))#Zhigh
    call hardware_i2c_write(v4Gyro,0,1,"\x2C")(written)#Zlow
    call hardware_i2c_read(v4Gyro,1,1)(result,data_len,sensor2(5))#Zlow
```

```
    # Read gyro X,Y,Z (high, then low bytes) registers
    call hardware_i2c_write(SparkL,0,1,"\x29")(written)
    call hardware_i2c_read(SparkL,0,1)(result,data_len,sensor3(0))#Xhigh
    call hardware_i2c_write(SparkL,0,1,"\x28")(written)
    call hardware_i2c_read(SparkL,0,1)(result,data_len,sensor3(1))#Xlow
    call hardware_i2c_write(SparkL,0,1,"\x2B")(written)#Yhigh
    call hardware_i2c_read(SparkL,0,1)(result,data_len,sensor3(2))
    call hardware_i2c_write(SparkL,0,1,"\x2A")(written)
    call hardware_i2c_read(SparkL,0,1)(result,data_len,sensor3(3))#Ylow
    call hardware_i2c_write(SparkL,0,1,"\x2D")(written)#Zhigh
    call hardware_i2c_read(SparkL,0,1)(result,data_len,sensor3(4))
    call hardware_i2c_write(SparkL,0,1,"\x2C")(written)#Zlow
    call hardware_i2c_read(SparkL,1,1)(result,data_len,sensor3(5))
    # Write Accel to GATT
    call attributes_write(xgatt_wa, 0, 6, sensor(0:6))
    # Write Gyro to GATT
    call attributes_write(xgatt_wg, 0, 6, sensor2(0:6))
    # Write Gyro to GATT
    call attributes_write(xgatt_shoulder, 0, 6, sensor3(0:6))

    call hardware_adc_read(0,3,2) #Input AIN0, 12 bits (Decimation 3), AVVD pin as reference voltage (2)

end
event hardware_adc_result(input,value)
   if input = 0 then
      flex = value >> 4 #Shifting result into 8 bits
    call attributes_write(xgatt_flex,0,1,flex)
    else
    call attributes_write(xgatt_flex,0,1,$FF)

   end if
end

# Boot event listener
event system_boot(major ,minor ,patch ,build ,ll_version ,protocol_version ,hw)
    # Initialize as 'disconnected'
    call gap_set_mode(gap_general_discoverable, gap_undirected_connectable)# Start advertisement
    call sm_set_bondable_mode(1)
    #Procedure to set up sensor parameters
    call setup()

end


 # Connection event listener
event connection_status(connection, flags, address, address_type, conn_interval, timeout, latency, bonding)
    #Connected, now waiting for recording command to start recording data
end
```

```
 # Connection event listener
event connection_status(connection, flags, address, address_type, conn_interval, timeout, latency, bonding)
   #Connected, now waiting for commmand to start recording dat
end

#Command listener
event attributes_value(connection, reason, handle, offset, value_len, value_data)
    data = value_data(0:value_len)
    if data>48 then #If data received > char'0' (48), start timer that repeats every 40 ms(This timer then calls the send data procedure)
     call hardware_set_soft_timer(1310, 0, 0)#Start timer, repeat every 40ms (32768/1000) * 40ms = 1310->Timer val
    else
     call hardware_set_soft_timer(0, 0, 0)#Stop timer
    end if
end

# Disconnection event listener
event connection_disconnected(connection, reason)
    call gap_set_mode(gap_general_discoverable, gap_undirected_connectable) # Start advertisement
    call hardware_set_soft_timer(0, 0, 0)#Stop timer
end

#Timer event listener
event hardware_soft_timer(handle)
  if handle = 0 then
     call send_data()
    end if
end
```

Figure7: CO-FIT.bgs

As seen from the code implementation above, detailed comments have been given to allow developers to read and understand the main functionality of the program. When the module first boosts, it starts advertising and setting up the sensors appropriately. After the sensor configuration is done, we simply wait for a connection. Once connected the device is now listening for a recording command, which will activate a timer every 40ms for a sample frequency of 25Hz.

-CO-FIT.bgproj:

```
<?xml version="1.0" encoding="UTF-8" ?>
<project>
    <gatt in="gatt.xml" />
    <hardware in="hardware.xml" />
    <script in="CO-Fit.bgs" />
    <image out="CO-Fit_out.hex" />
    <device type="ble113" memory = "256" />
    <boot fw="bootuart" />
</project>
```

Figure 8: CO-FIT.bgproj file

This file simply specifies the location of the hardware configuration, script, and GATT service configuration file, as well as the device type.

DATA LABELING APPLICATION

DATA LABELING APP SEQUENCE DIAGRAM AND APP OVERVIEW

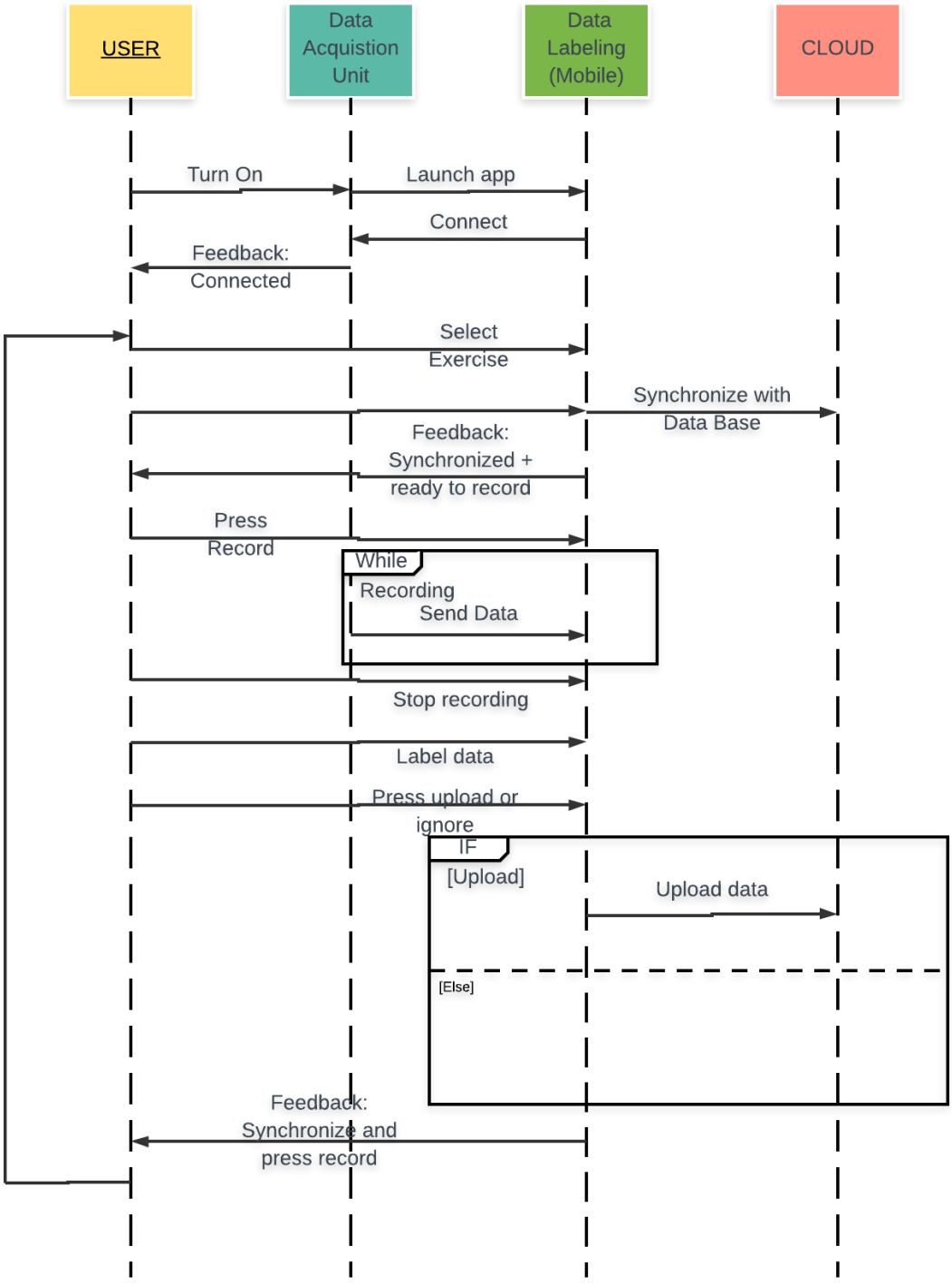


Figure 9: Data acquisition sequence diagram

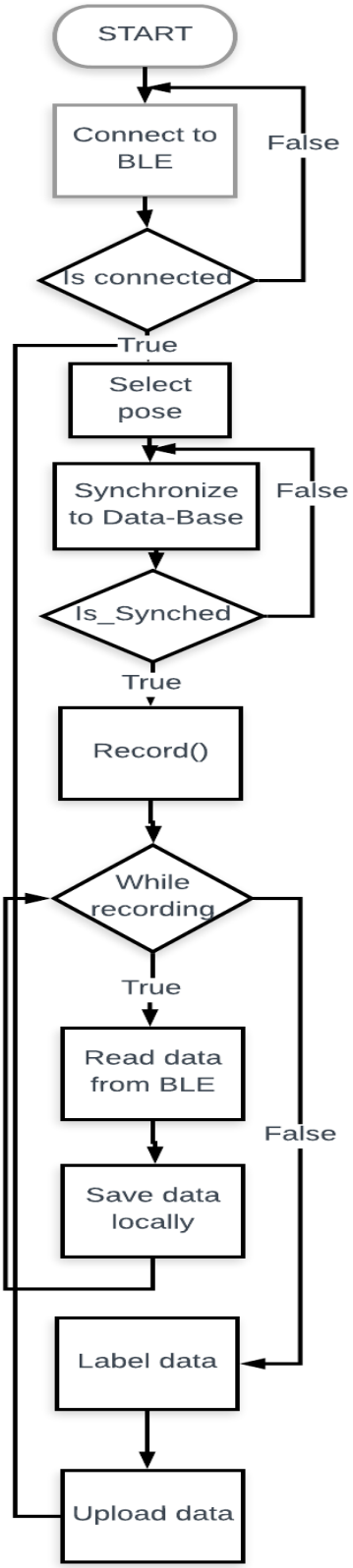
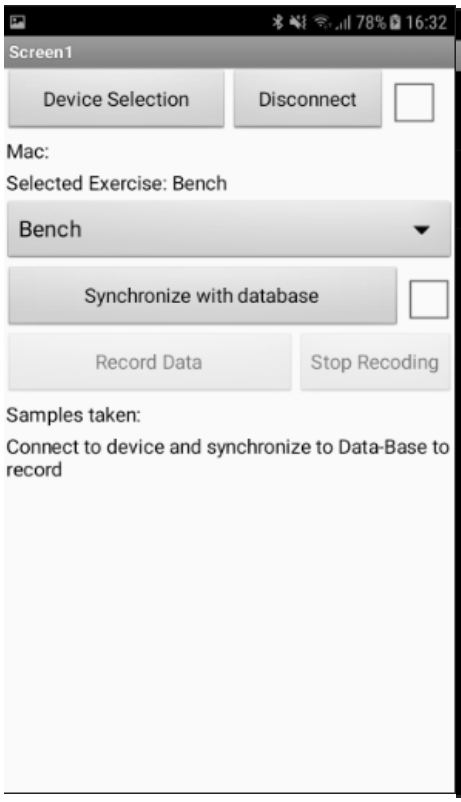


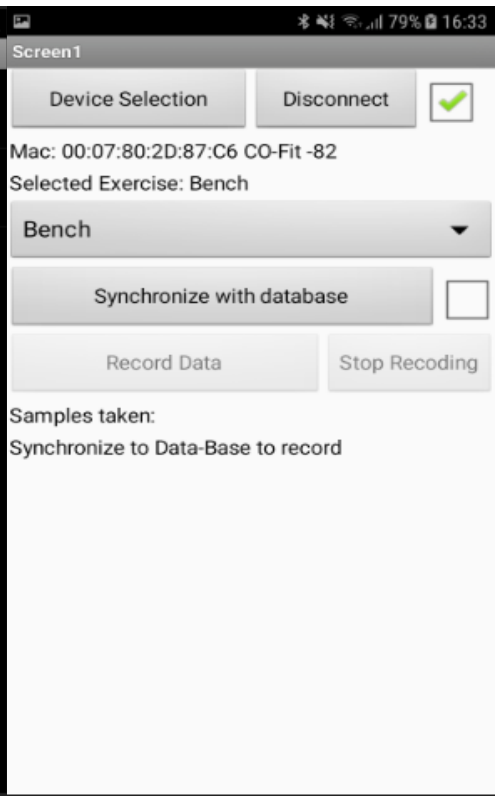
Figure 10: Program flow of data labeling app

1-Main launch screen

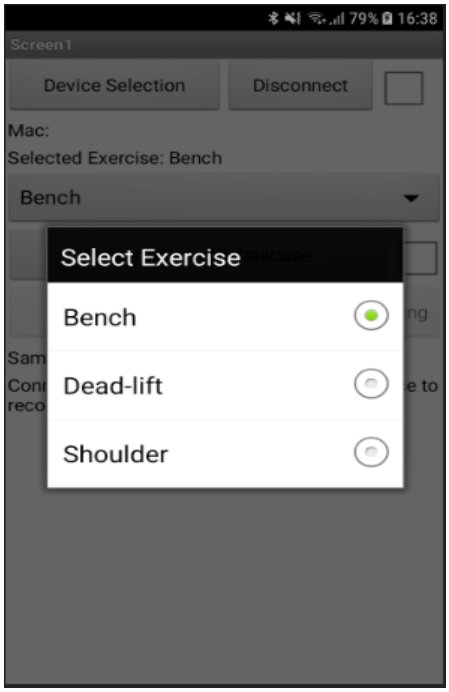


2-Device Selection

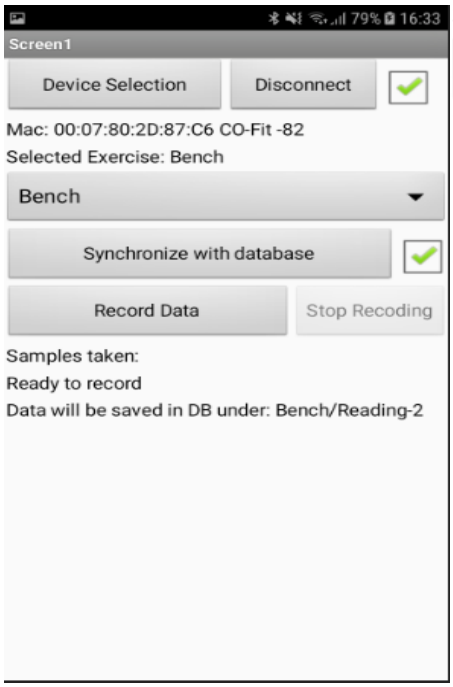
3-Connected



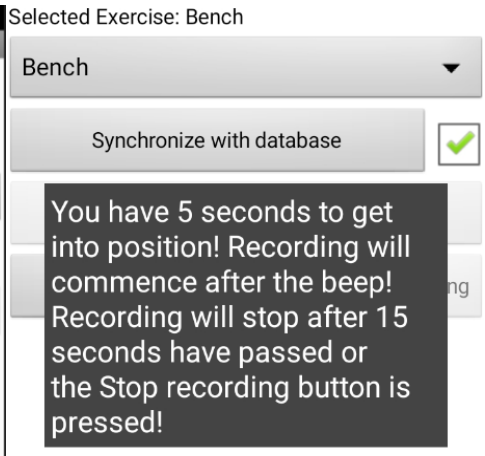
4-Exercise Selection



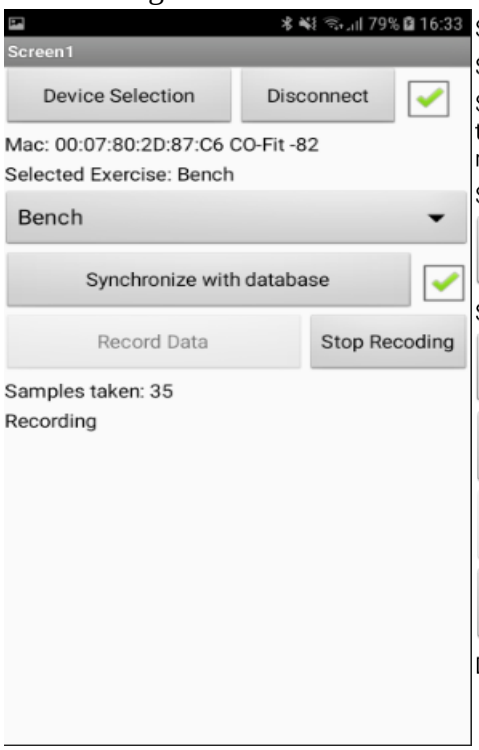
5-Synchronized with database



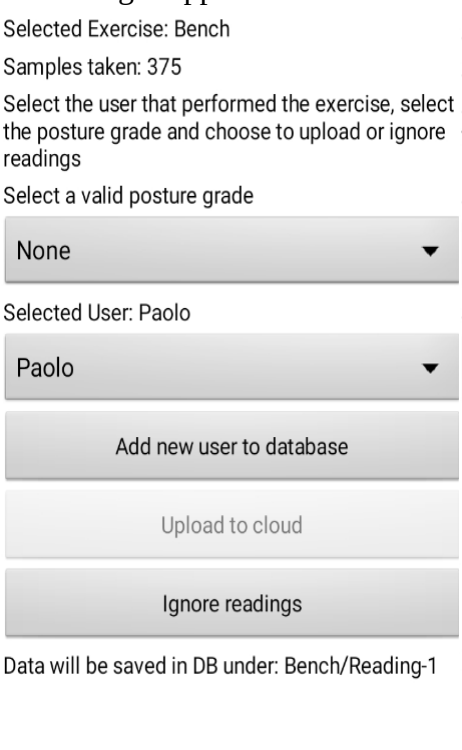
6-Record button pressed



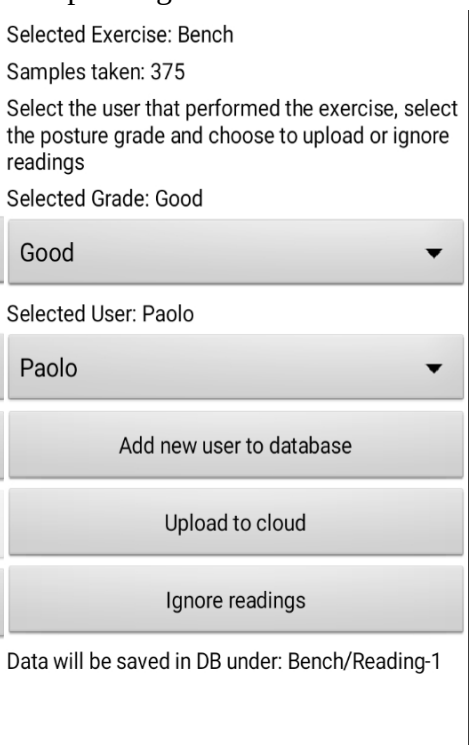
7-Recording



8-Recording Stopped menu



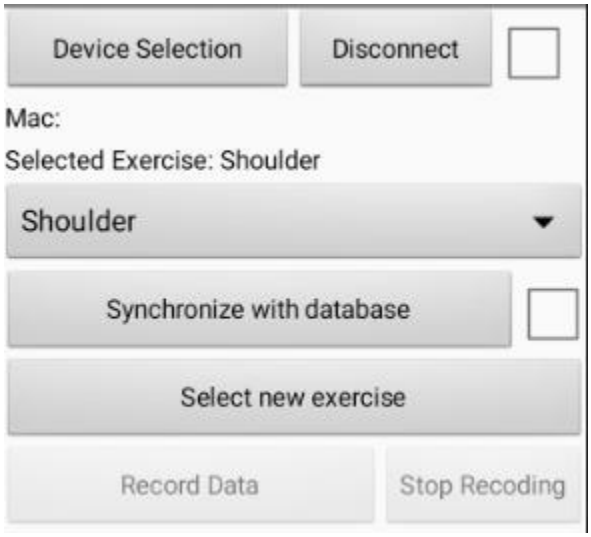
9-Upload/ignore



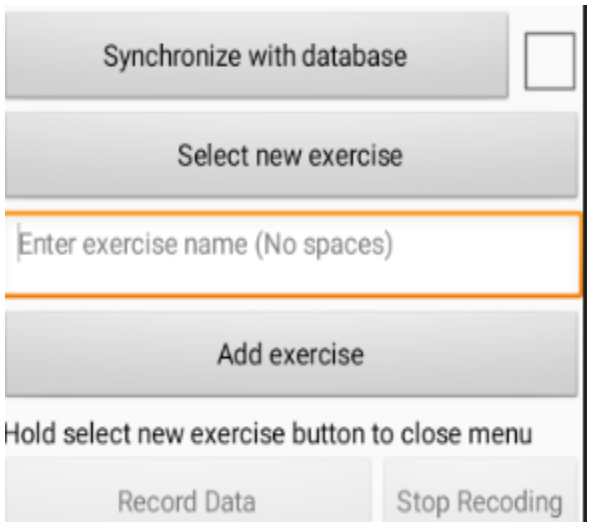
Figures11-19: Show the Sequence diagram in the app.

Functionality to enter new exercise pose to add to the database is available as seen in the pictures below:

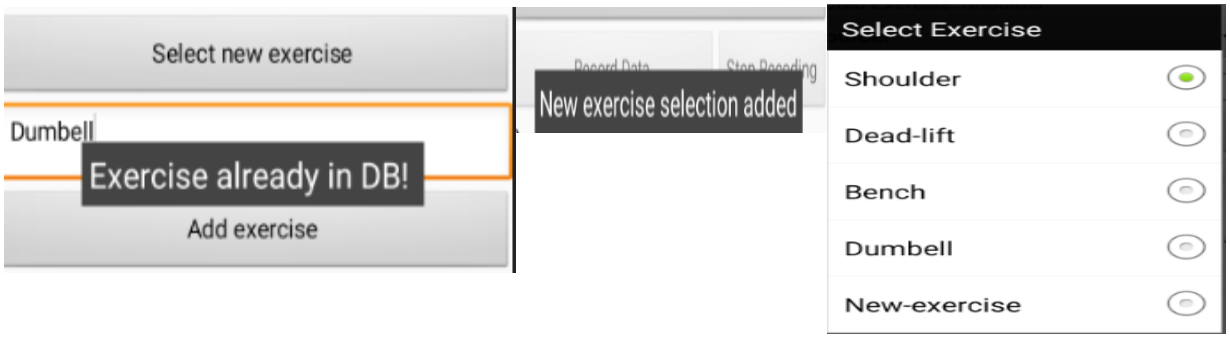
1-Before clicking new exercise Button



2-On click put new exercise to DB Menu



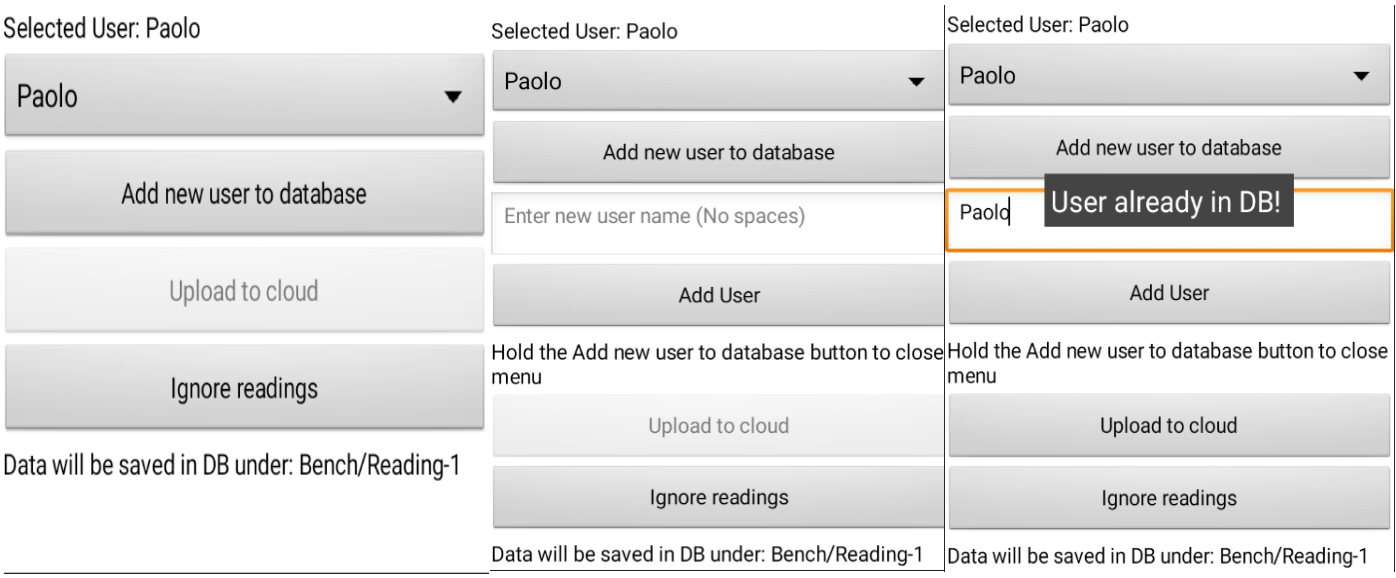
3-If exercise already in DB notification 4-New exercise added 5-New exercise available



Figures 20-24: Adding new Exercise functionality

Functionality to enter new user the database is available as seen in the pictures below:

1-Before clicking Add new user button 2-Add new user menu 3-Invalid user



4-New user added 5-New user available

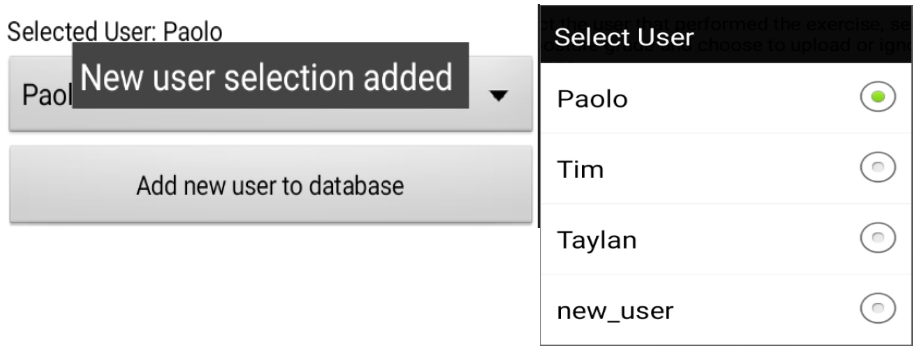


Figure 24-28: Add new user to database functionality

DATA LABELING MIT APP INVENTOR CODE IMPLEMENTATION

As explained earlier, MIT APP INVENTOR is a simple to use code block drop down programming language which allow developers to easily develop android applications. The code will be presented as code blocks and each functionality will be explained in detail. App inventor contains 2 parts: The design page as seen below, and a block diagram code page which allows users to program the components in the application.

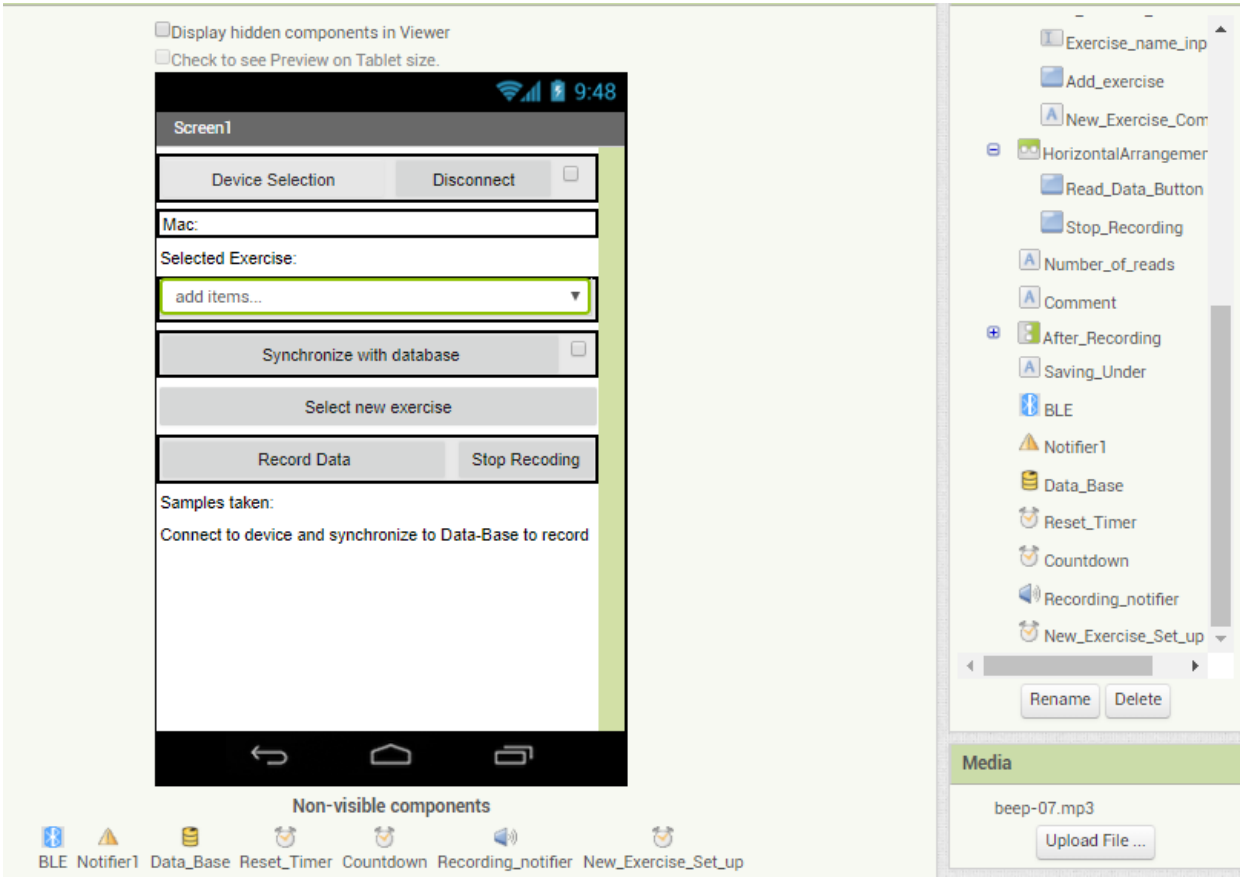


Figure 29: MIT APP INVENTOR project designer page

The project designer page simply allows user to customize the GUI of the application, as well as drop the require components needed for the application. Each component then has a set of block diagrams with their unique functions. To better understand the code, I will first explain the non-visible components and how they are used. For this application we have a total of 7 non-visible components, 3 of them are timers which simply run a specific line of code when they are triggered, these are named Reset_Timer, Countdown and New_Exercise_Set_up as seen above. The remaining non-visible components are as followed:

-Notifier1: This component simply allows us to show the user a notification alert when something has happened in the application

-BLE: The BLE component is an extension of MIT APP INVENTOR, which doesn't come by default! This extension allows us to connect to BLE devices and write to a specific service and characteristic, as well as set up a notifier of when a characteristic's value has been updated. To download this extension please head over to the following link:

http://appinventor.mit.edu/extensions/

-Data_Base: This is a firebase extension which is used to connect and upload data to our database.

-Recording_notifier: This is a sound playing component which simply allows us to play a sound file, to indicate the user that a recording session has started.

When coding in MIT APP INVENTOR, the code will look something like this:

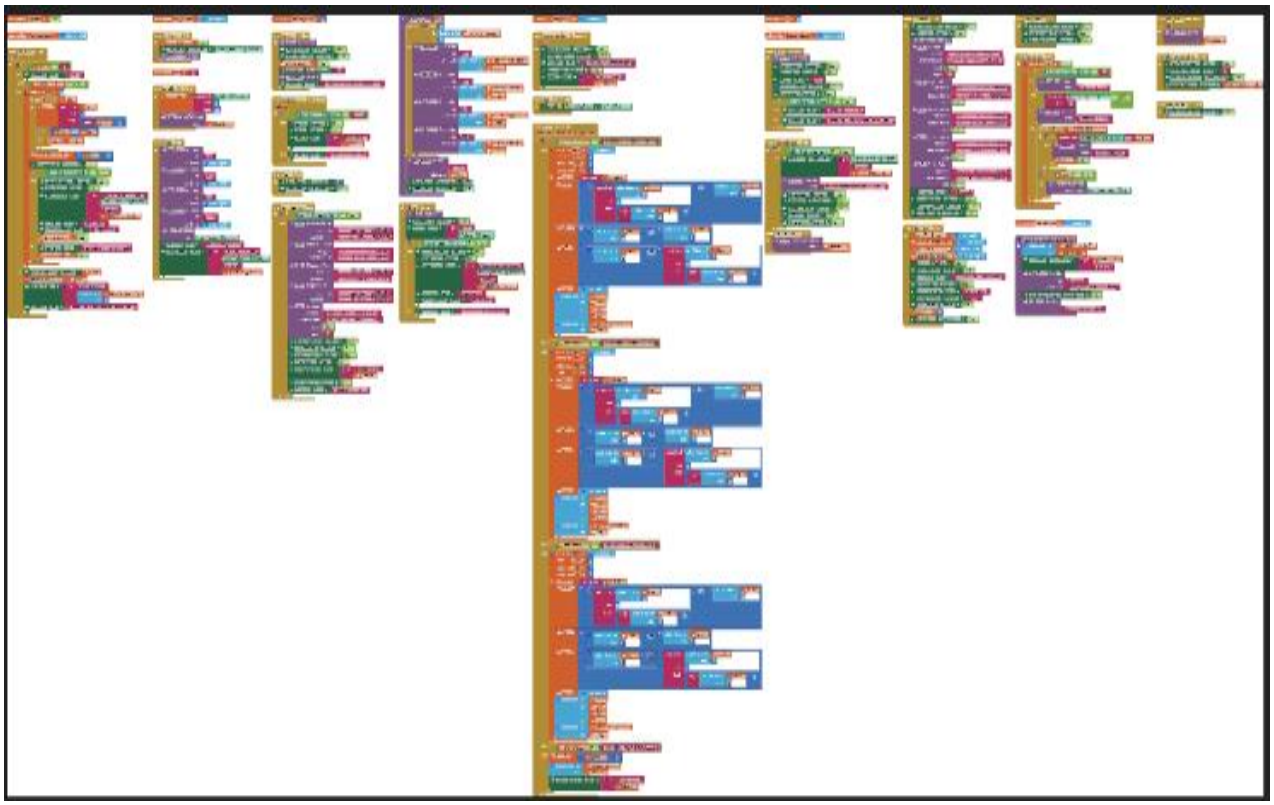


Figure 30: Block code overview for data labeling application

Here is the list of all the components our application currently has:

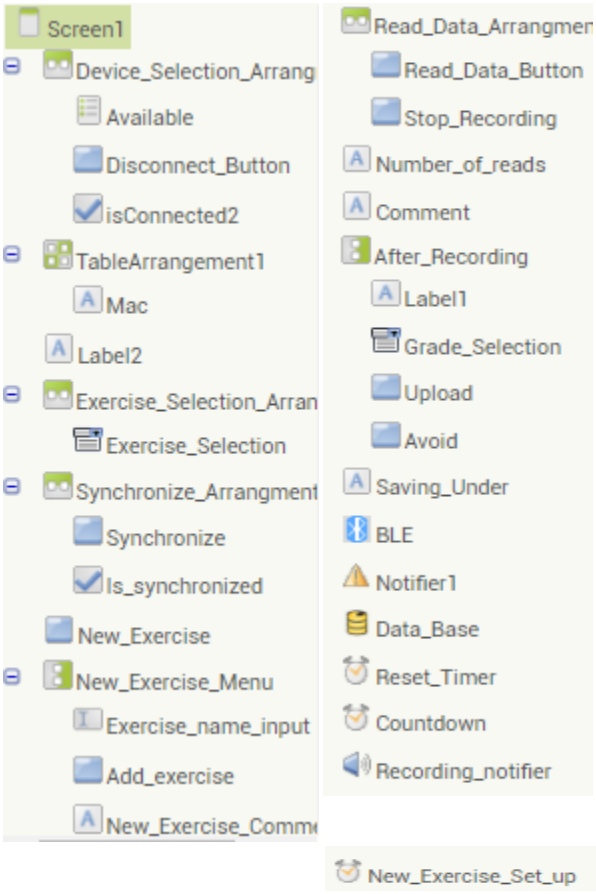


Figure 31: Data labeling component list

For a more detail explanation behind each component usability, check the following link:

http://appinventor.mit.edu/explore/content/reference-documentation.html

Now we will discuss the code blocks and how they work together to create the program logic. We will start of by the explaining the code initializing screen and global variables:

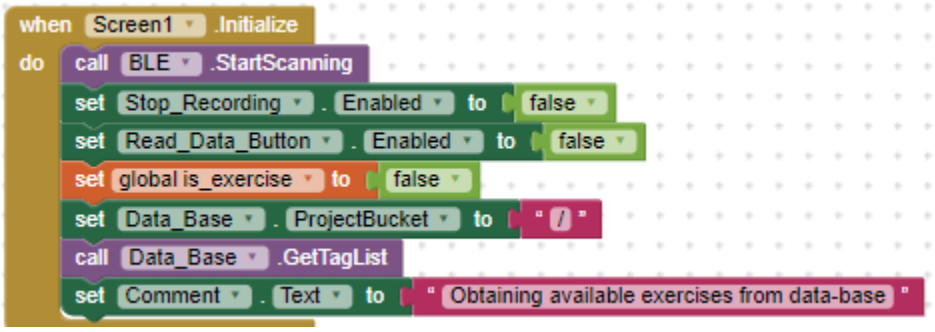


Figure 32: App screen initialization setup

Figure 33: Global Variables

-Wrist Rotations: List which will contain the rotational gyroscope readings from the wrist gyroscope sensor. This list is cleared after every recording session.

-Wrist Accelerations: List which will contain the acceleration readings from the wrist accelerometer sensor. This list is cleared after every recording session.

-Shoulder Rotations: List which will contain the rotational gyroscope readings from the shoulder gyroscope sensor. This list is cleared after every recording session.

-Device: This will simply store the mac address of the device we choose to connect to.

-ctr: Simple counter to know how many sensor samples we have taken.

-Flex Readings: List which will contain the flex readings from the flex sensor. This list is cleared after every recording session.

-Reading Number: This variable is used to obtain the latest database reading number entry, to allow users to upload data into the database, without overriding existing data

-Exercises in DB: List which downloads the existing exercises in the database when the app is launched.

-Is exercise: Boolean used for the database tag list implementation. If this variable is set to false, we expect that the tags we will obtain from the database are not exercise number readings, but the type of exercises available.

When the app first launches it starts to scan for BLE devices and disables the read data and stop recording buttons until a BLE device has been connected and we have synchronized to our database. As seen in line 5 in the initializing screen, we set up the database root directory to obtain the current exercises available to gather data from in the database. The function Data_Base.GetTagList will simply return us the nodes from our selected directory in the Database as seen below:

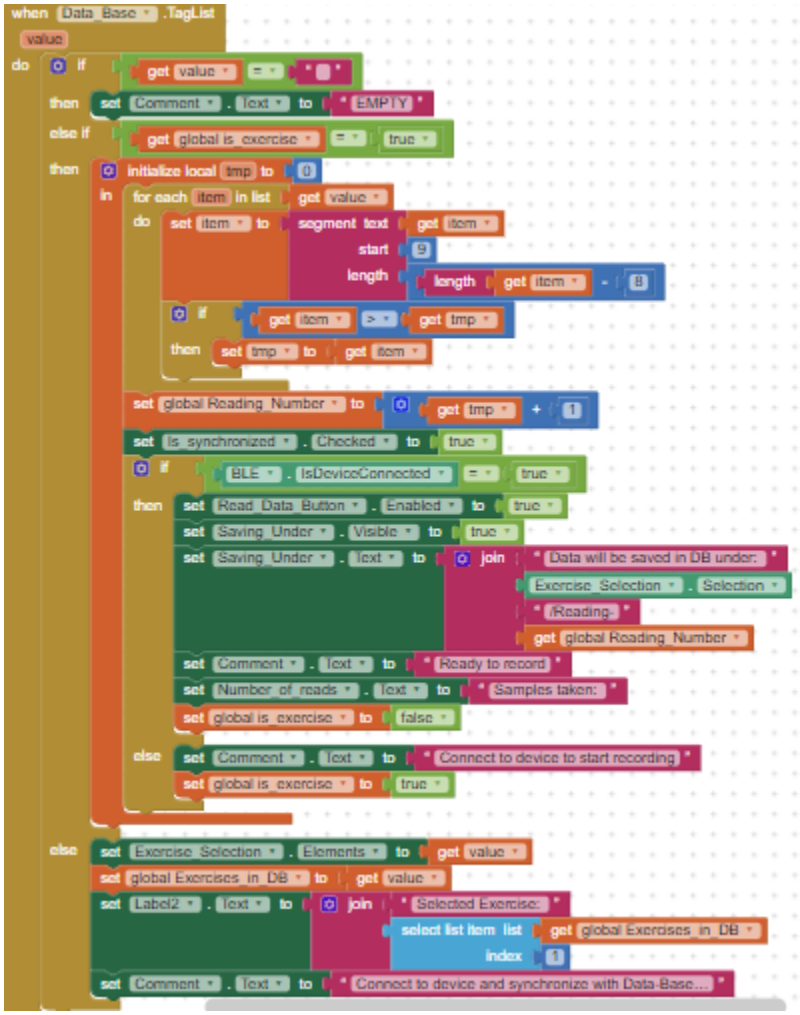
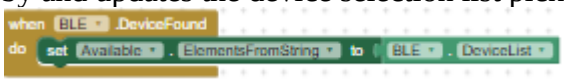


Figure 34: Database Get Tag list response

When the mobile application launches, and the Data Base get tag list function is called, Boolean is_exercise is setup to false, hence the tags we obtain from this function are the exercises available in our database (value = list of exercises in database at this time). We then use this value to set up the Exercise_Selection elements to our database exercise's. From now the user must select a BLE device to connect to and synchronize to the database to start obtaining data from the data acquisition unit. As seen from figure 28 (Initializing screen) a function named BLE.Start scanning is called: this function simply finds BLE devices nearby and updates the device selection list picker with the devices available as seen here:



Once the user selects an BLE device to connect to, the following event is invoked:

Figure 35: BLE device selection picker function

When a connection has been stablished to the BLE device, the following event is invoked:

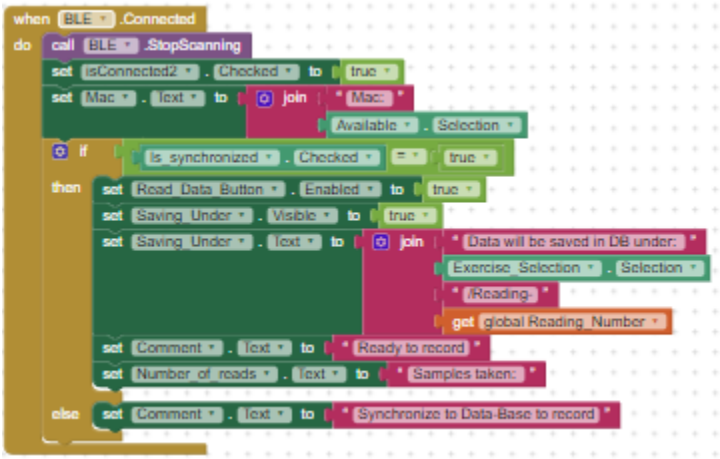


Figure 36: BLE connection established

This function simply marks the is connected check bock and sets the mac address label to the device we are currently connected to. Then it check's if we are synchronized to the database or not. This is to prevent users to click the read data button by accident, since a synchronization to the database must be made before hand. If we have both a BLE connection and we are synchronized, then the read data button is enabled, and an appropriate comment is given to the user to let them know we are ready to record. Else we put a comment telling the users to synchronize to the database to record. When the users click the disconnect button in the app, the following code block is run:

Figure 37: Disconnect button event

This disconnect button simply disconnects our phone from our BLE device and invokes the following event:

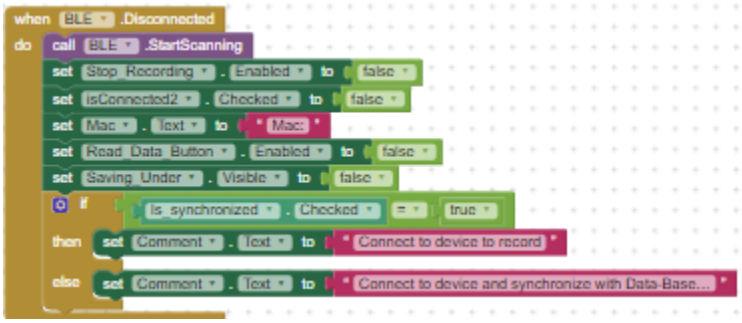


Figure 38: BLE disconnected event

This event simply starts a new scan for BLE devices and disables the read and stop recording buttons, as well as providing users with comments on what to do next to start recording. As we mentioned previously, a user must be connected to a BLE device and synchronize to the database to start a recording session. Before synchronizing to the database, the user should select an exercise to record. To select a new exercise, the user uses the Exercise_selection spinner, which simply shows the user the different type of exercises available from the database. When the user selects an exercise to record, the following event is invoked:

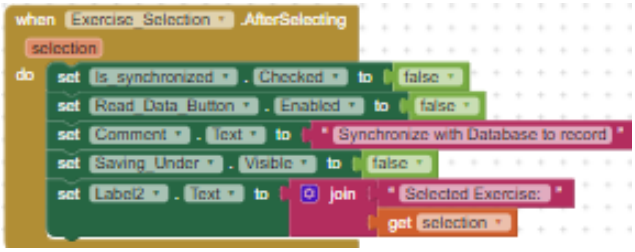


Figure 39: Exercise selection event

As seen from figure 35, this event simply sets the is synchronized box to false and tells the user to synchronize to the database to start the recording.

When the user clicks the synchronize button the following event is invoked:

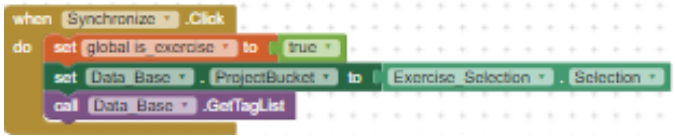


Figure 40: Synchronize button event

This event simply, sets the is_exercise Boolean to true, and sets the database directory to the selected exercise, and calls a GetTagList from the database. This function will now return the recordings tags from the selected exercise in the database in the following format: Reading-N. From this tag list we can now obtain the last entry N and specify where we are going to store the new data once the recording session has been finalized

and the upload button pressed. More information regarding the Data Base implementation is give under the section Data Base Design of this document.

To obtain the last recording number N, the following code is run when the tags from our database are obtained:

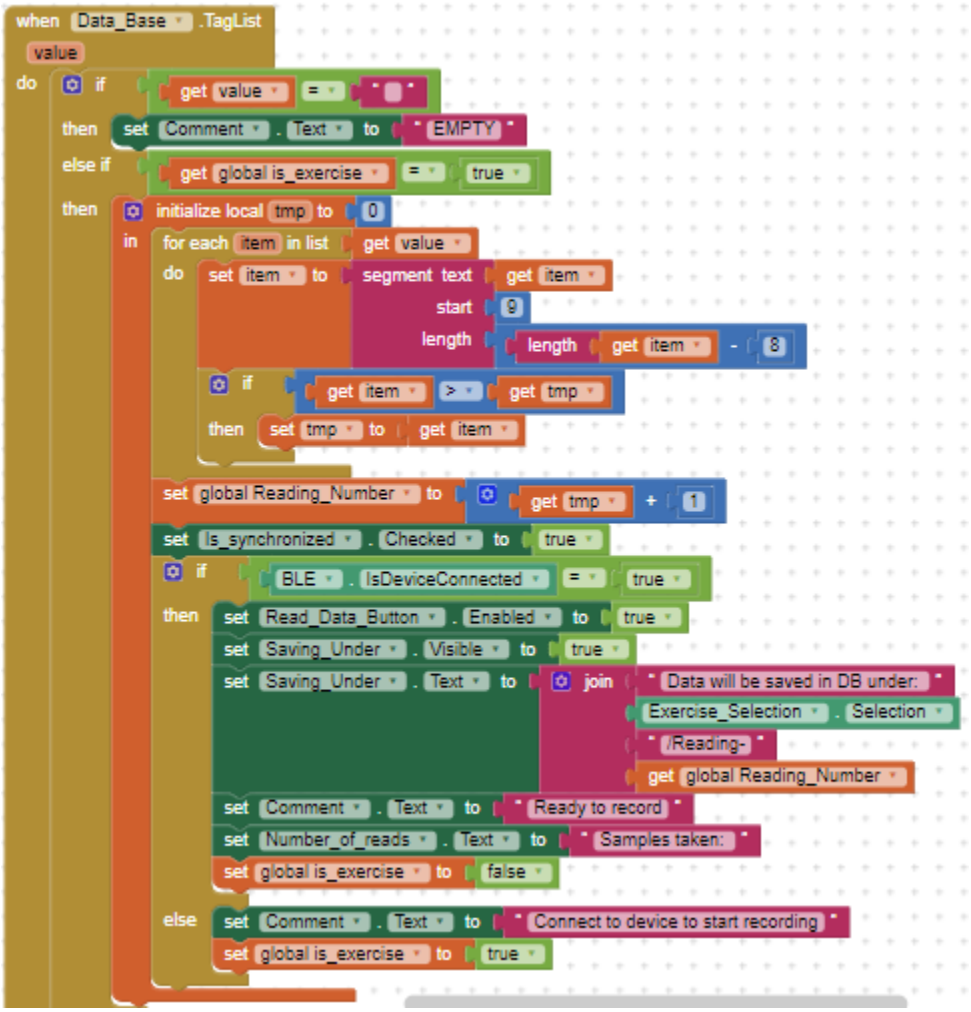


Figure 41: Obtaining last recording session number N, from database

As seen from figure 38, Boolean is_exercise is now true, therefor we now iterate over each item in our list of recordings and obtain the numerical digits by setting the item to a substring containing the latest number, and then comparing that number to our tmp variable, which simply increments as new entries are found. Then we set the global Reading_Number to our latest entry N + 1, to avoid overwriting data. Then we proceed to check if a connection with a BLE device has been made, and if this is the case, we then enable the read button and tell the users the directory under which the new data will be stored in our database. Now the user can simply click the read data button and start recording data appropriately! Next we will discuss the recording and uploading of data from our application.

When the user presses the read data button, a timer will be enabled which will fire in 5 seconds after the button is pressed. When the 5 seconds have passed and the timer triggers, a beeping sound will be played, to inform the user that the recording session has started. The reason behind this timer is to allow users to get into their exercise position and remove noise in the data. The sequence can be seen below:

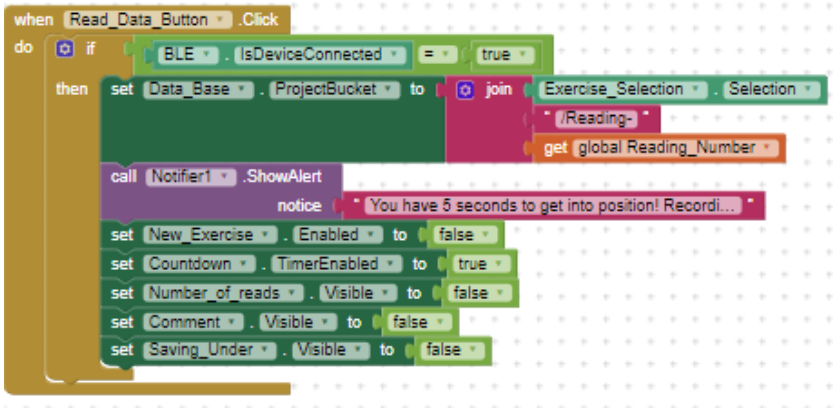


Figure 42: Read_Data_Button click event

As seen from figure 38, when we click the read data button, we check if we still have a connection to our BLE device, if this is the case, we then set up the proper Data Base recording directory, notify the user to get into position and enable our timer countdown.

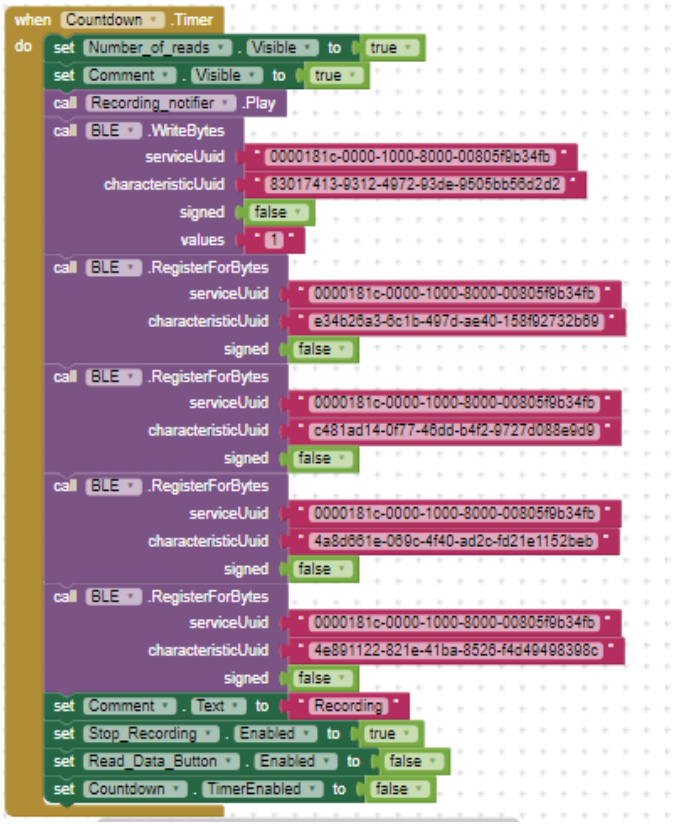
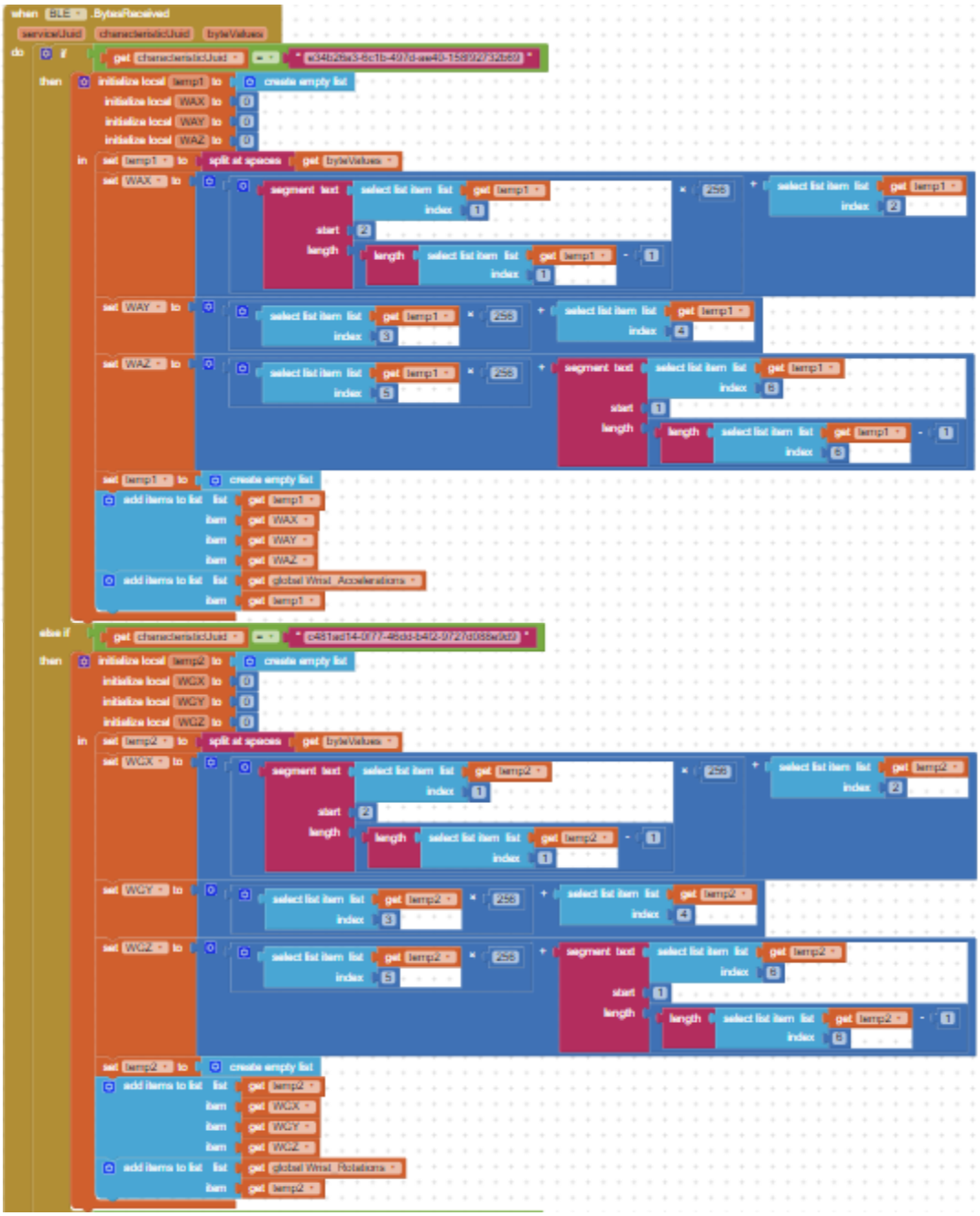


Figure 43: Countdown timer event

As seen in figure 39, when the countdown timer event is invoked, we first play a sound to indicate the user that the recording session has started. Then we write to our data acquisition unit the character “1”, so that it starts the recording and sending of the sensor data. We then register to be notified when our data service characteristics values are updated to be able to save data as it’s being recorded. This is done by the function BLE.RegisterForBytes. When new data is received from the BLE device an event is invoked, in this event, we check from which characteristic UUID the data is coming from and we save the data in the correct list: Wrist_Acclerations, Wrist_Rotations, Shoulder_Reading or Flex_Readings, as seen below:



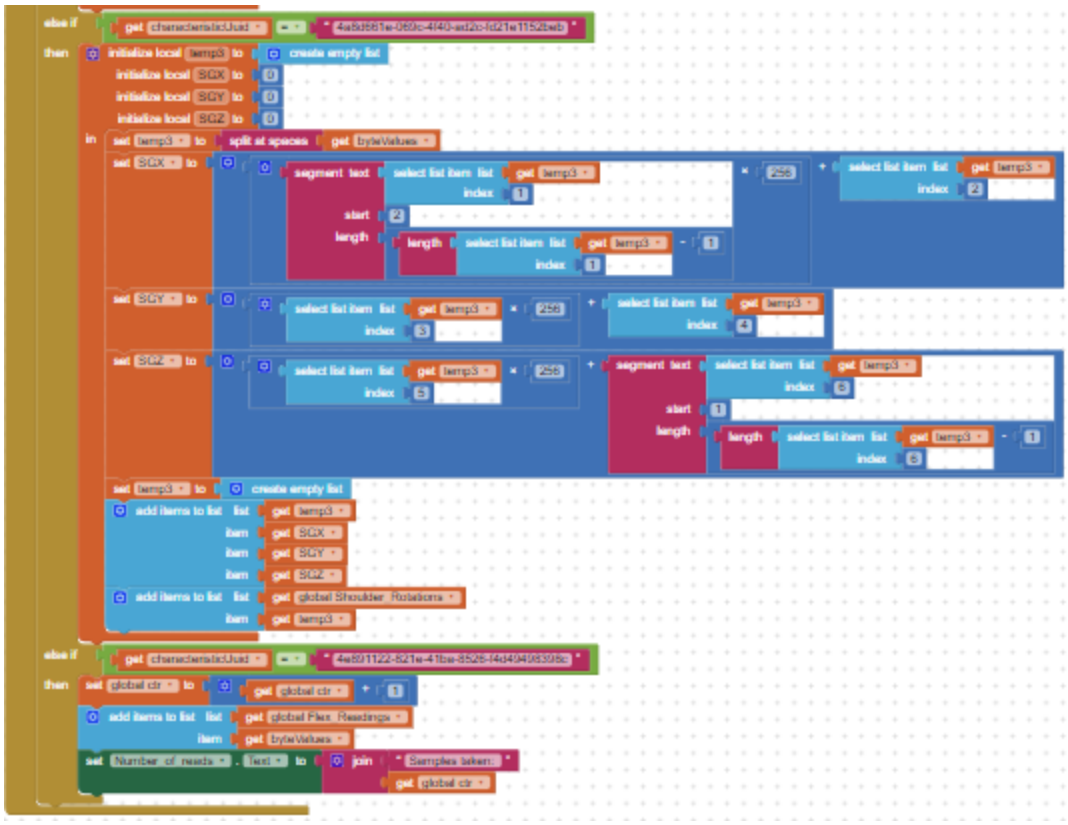


Figure 44: Data received event

When the data arrives from the accelerometer and gyroscope characteristics, it arrives in a format that looks something like this (255,53,255,43,255,56) where it represents the following (XMSB, XLSB, YMSB, YLSB, ZMSB, ZLSB). Next what we do is add the MSB and LSB bits of the data to create a single 16-bit long unsigned value, which will then be uploaded to the database. Since the flex sensor readings are only 1 byte long, the reading is not modified. As seen above, we update the number of reads (Samples taken) number only when we obtain new Flex readings, the reason behind this is for data synchronization! When uploading the data, we want to have an equal number of samples between all the sensors, and from the data acquisition unit, we know that the Flex sensor is always the last sensor being read. At first, we uploaded the data to the database as soon as it arrived, but after some testing, we realized that the flex readings always lacked behind all of the other readings by around 2-6 readings. For this reason, once the user stops the recording, we then upload all the values in our lists, starting from index 1 until the last index in the flex sensor. The same functionality is implemented in the real time application, meaning we only send the TCP packet containing the sample reading once the flex sensor data arrives. Once the user clicks the stop recording button we simply send the command ‘0’ to our data acquisition unit, to stop recording and sending data. We also unregister from the characteristics UUID to clear any data left in the BLE receiving buffers and we set the after recording vertical arrangement visibility to true to allow users to label the data and choose whether they want to upload or ignore the readings. The database synchronization is then set to false, so that users have to synchronize again for a new reading. All of this can be seen in the picture below:

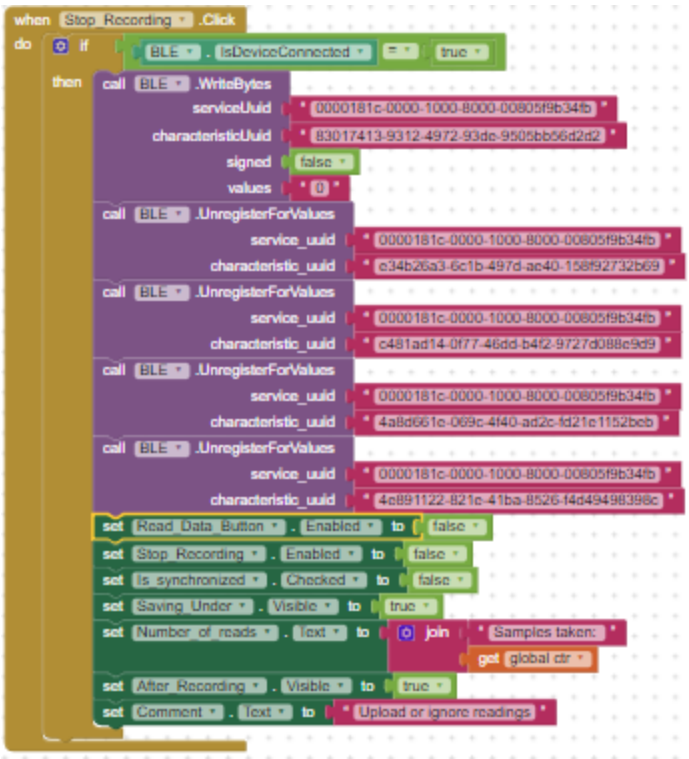


Figure 45: Stop recording button click event

If the user chooses to ignore the readings by pressing the Avoid button (Ignore reading button) the following event is invoked:

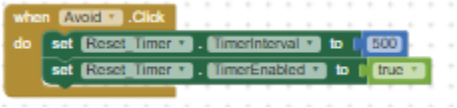


Figure 42: Avoid reading button

This button simply enables the Reset_Timer and sets a trigger time of 500ms or 0.5s. When the Reset_timer is invoked the lists of readings are cleared and the main menu is reset so that users have to synchronize with the database and select a new exercise to start a new recording seen below:

Figure 46: Reset timer event

If the user decides to upload the data, they must first select a grade for the posture being performed, either good or bad. Once this grade is selected the upload button will now be enabled.

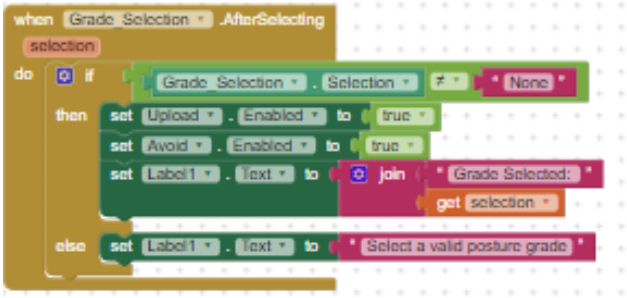


Figure 47: Grade selection event

Once a valid grade is selected and the user click the upload button, the following event is invoked:

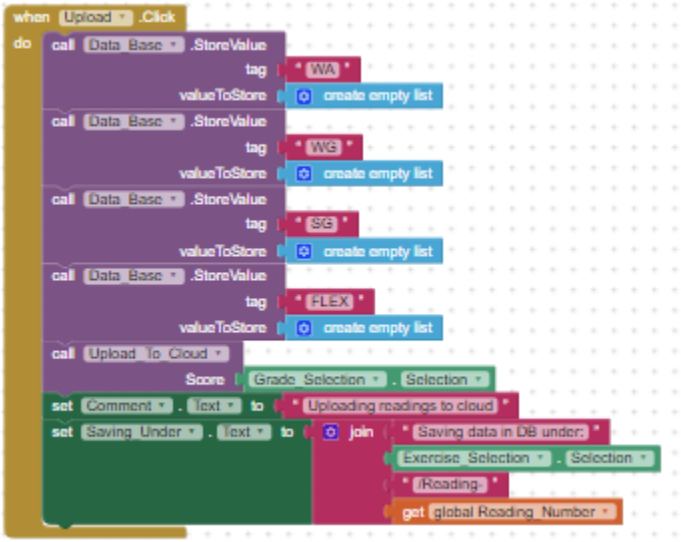


Figure 48: Upload data button event

When this button is pressed, we simply set up the tags for the new reading sub-directory and we call the function Upload_To_Cloud. This function simply takes as a parameter the score and uploads every item in our lists to their corresponding tags, starting from index 1 until the last index in the flex list as mentioned previously. Then after this, it also uploads the grade of the session and calls the reset timer with a trigger time of 4 seconds to allow the data to be fully uploaded before a new session begins. The functionality of this function can be seen in the figure below

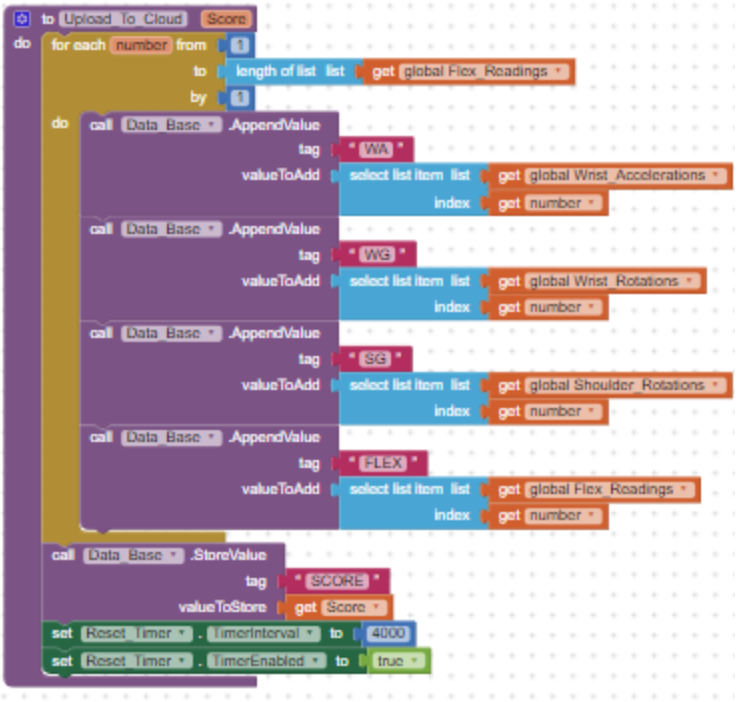


Figure 49: Upload to cloud function

Alongside uploading and recording data, the users also have the possibility to add new exercise poses to their system. To do this, the users must press the button New_Exercise (Seen in the app with text Select new exercise), once this button is pressed, the following event is invoked:

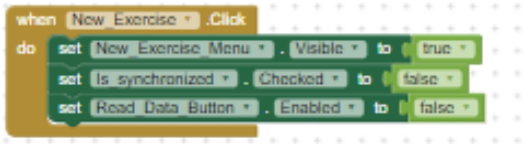


Figure 50: New exercise button click event

This button simply makes the new exercise menu visible so that the user can now enter the new exercise name (with no spaces allowed) and chose to add the new exercise pose to the cloud for future recordings. When the user clicks the Add_exercise button, we first check that the exercise is not already in the database, and if that's the case we then call the function Upload_New_Exercise. All of this can be seen in figures 48 and 49 below

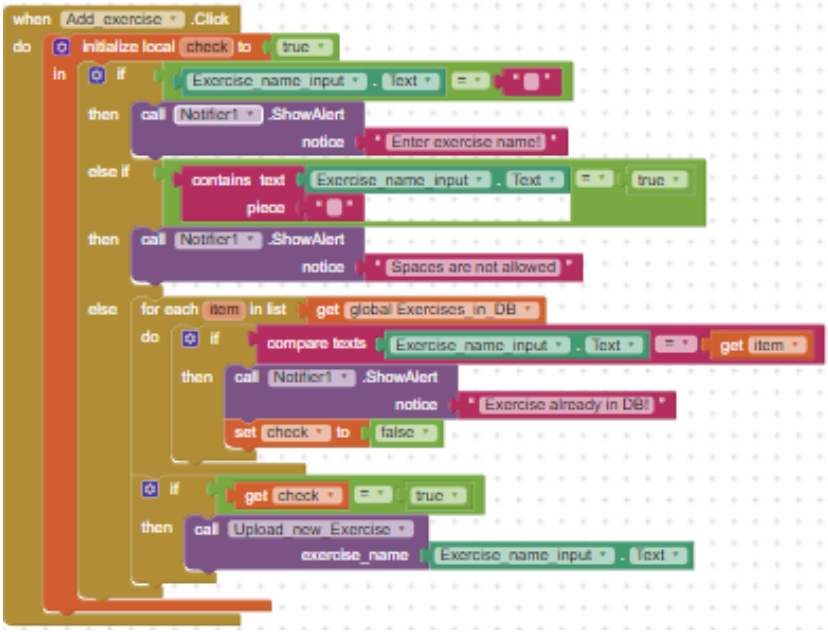


Figure 51: Add exercise button event

Figure 52: Upload_new_Exercise function

When adding a new exercise to the cloud, the exercise will be added under the project bucket exercise_name/Reading-0, which will only contain a description tags, telling the database users that the readings of each exercise will start from Reading-1 and on. The new exercise is also added to our list Exercises_in_DB, so that we don't have to invoke the database to obtain the new tag. After the exercise is added, a timer called New_Exercise_Set_up is enable, which triggers after 1 second. When the timer is triggered it simply makes the Exercise menu invisible and the exercise selection list gets updated as seen below:

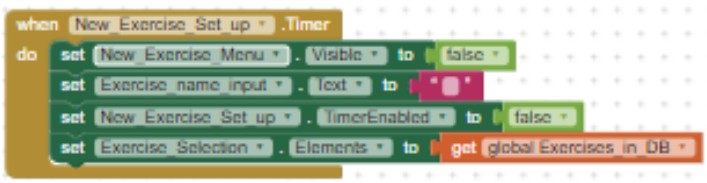


Figure 53: New_Exercise_Set_up Timer event

REAL TIME APPLICATION AND MATLAB SERVER

REAL TIME APPLICATION A ND MATLAB SERVER OVERVIEW

Note: The real time and data labeling app share similar characteristics.

When using the real time application, one will be prompted with the following screen:

Figure 54: Real time app initial screen

From this screen one can insert the IP address and Port number of the MATLAB server and establish a TCP connection with the server, as well as start starting reading data from the data acquisition unit, which will be sent to the MATLAB application

When a connection has been established with the server one should see the following:

Figure 55: Connection Established with TCP Server

Now one can simply press the read data button and start sending TCP Packets to MATLAB.

When MATLAB receives the data, it will display the following graphs as real time data is obtained in its TCP input stream:

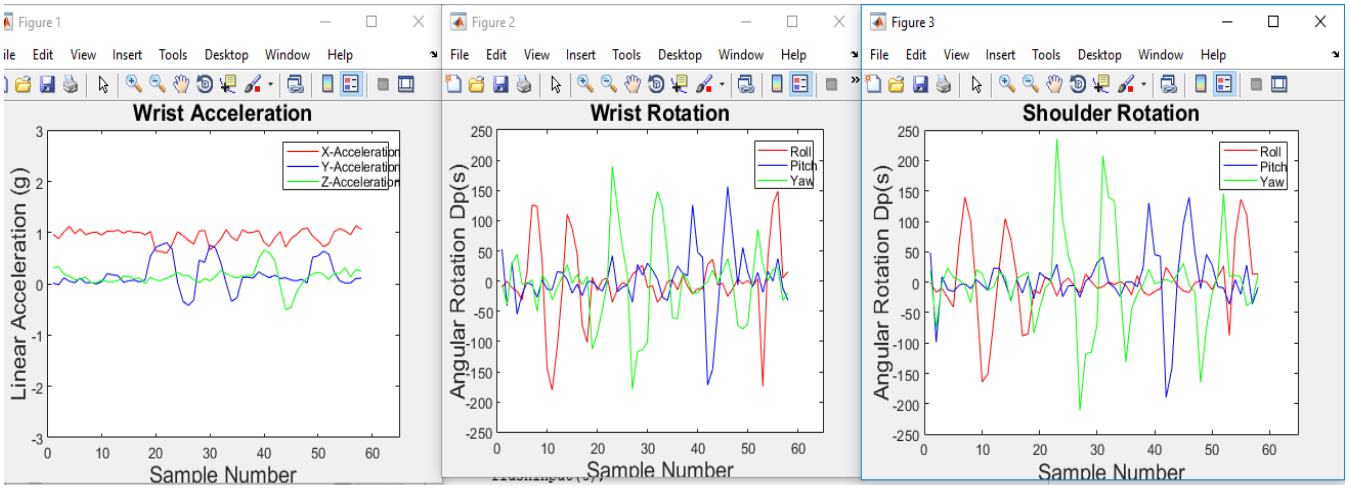


Figure 56: Real time data graphs from MATLAB

REAL TIME DATA MIT APP INVENTOR CODE IMPLEMENTATION

As mentioned previously the real time application is very similar to the Data labeling application. Meaning that they both share the same code for choosing a BLE device to connect to, connect to the device, as well as acquiring the data from the device. The main difference being that the real time application allows the user to connect to a TCP server and send TCP packets as new data arrives.

IMPORTANT NOTE: MIT APP INVENTOR DOES NOT COME WITH A TCP CLIENT EXTENSION. AN EXTENSION HAD TO BE DOWNLOADED AND MODIFIED FOR OUR APPLICATION IN THE FOLLOWING LINK:

https://groups.google.com/forum/#!msg/mitappinventortest/OCzEZC4FpEU/sEVrCeXaCQAJ

The code for the extension is given in the pictures below:

```
// -*- mode: java; c-basic-offset: 2; -*-
// Copyright 2009-2011 Google, All Rights reserved
// Copyright 2011-2012 MIT, All rights reserved
// Released under the Apache License, Version 2.0
// http://www.apache.org/licenses/LICENSE-2.0

package com.gmail.at.moicjarod;

import com.google.appinventor.components.runtime.*;
import com.google.appinventor.components.runtime.util.RuntimeErrorAlert;
import com.google.appinventor.components.annotations.DesignerComponent;
import com.google.appinventor.components.annotations.DesignerProperty;
import com.google.appinventor.components.annotations.PropertyCategory;
import com.google.appinventor.components.annotations.SimpleEvent;
import com.google.appinventor.components.annotations.SimpleFunction;
import com.google.appinventor.components.annotations.SimpleObject;
import com.google.appinventor.components.annotations.SimpleProperty;
import com.google.appinventor.components.annotations.UsesLibraries;
import com.google.appinventor.components.annotations.UsesPermissions;
import com.google.appinventor.components.common.ComponentCategory;
import com.google.appinventor.components.common.PropertyTypeConstants;
import com.google.appinventor.components.runtime.util.AsynchUtil;
import com.google.appinventor.components.runtime.util.ErrorMessages;
import com.google.appinventor.components.runtime.util.YailList;
import com.google.appinventor.components.runtime.util.SdkLevel;

import com.google.appinventor.components.runtime.errors.YailRuntimeError;

import android.app.Activity;
import android.text.TextUtils;
import android.util.Log;
import android.os.StrictMode;

import java.io.*;
import java.net.*;
```

```
/**
 * Simple Client Socket
 * @author moicjarod@gmail.com (Jean-Rodolphe Letertre)
 * with the help of the work of lizlooney @ google.com (Liz Looney) and josmasflores @ gmail.com (Jose Dominguez)
 * the help of Alexey Brylevskiy for debugging
 * and the help of Hossein Amerkashi from AppyBuilder for compatibility with AppyBuilder
 */
 // The original code of this application was modified by Paolo Nardi to fit his App inventor project

@DesignerComponent(version = 4,
  description = "Non-visible component that provides client socket connectivity.",
  category = ComponentCategory.EXTENSION,
  nonVisible = true,
  iconName = "http://jr.letertre.free.fr/Projets/AIClientSocket/clientsocket.png")
  @SimpleObject(external = true)
  @UsesPermissions(permissionNames = "android.permission.INTERNET")


public class ClientSocketAI2Ext extends AndroidNonvisibleComponent implements Component
{
  private static final String LOG_TAG = "ClientSocketAI2Ext";

  private final Activity activity;

  // the socket object
  public Socket clientSocket = null;
  // the address to connect to
  private String serverAddress = "";
  // the port to connect to
  private String serverPort = "";
  // boolean that indicates the state of the connection, true = connected, false = not connected
  private boolean connectionState = false;

  InputStream inputStream = null; //Input stream object

  OutputStream outputStream = null;//Output stream object

  PrintStream pout = null; //Prinstream object



  /**
   * Creates a new Client Socket component.
   *
   * @param container the Form that this component is contained in.
   */
  public ClientSocketAI2Ext(ComponentContainer container)
  {
    super(container.$form());
    activity = container.$context();
    // compatibility with AppyBuilder (thx Hossein Amerkashi <kkashi01 [at] gmail [dot] com>)
    StrictMode.ThreadPolicy policy = new StrictMode.ThreadPolicy.Builder().permitAll().build();
    StrictMode.setThreadPolicy(policy);
  }

  /**
   * Method that returns the server's address.
   */
  @SimpleProperty(category = PropertyCategory.BEHAVIOR, description = "The address of the server the client will connect to.")
  public String ServerAddress()
  {
    return this.serverAddress;
  }

  /**
```

```
/**
 * Method that returns the server's address.
 */
@SimpleProperty(category = PropertyCategory.BEHAVIOR, description = "The address of the server the client will connect to.")
public String ServerAddress()
{
  return this.serverAddress;
}

/**
 * Method to specify the server's address
 */
@DesignerProperty(editorType = PropertyTypeConstants.PROPERTY_TYPE_STRING)
@SimpleProperty
public void ServerAddress(String address)
{
  this.serverAddress = address;
}

/**
 * Method that returns the server's port.
 */
@SimpleProperty(category = PropertyCategory.BEHAVIOR, description = "The port of the server the client will connect to.")
public String ServerPort()
{
  return this.serverPort;
}

/**
 * Method to specify the server's port
 */
@DesignerProperty(editorType = PropertyTypeConstants.PROPERTY_TYPE_STRING)
@SimpleProperty
public void ServerPort(String port)
{
  this.serverPort = port;
}

/**
 * Method that returns the connection state
 */
@SimpleProperty(category = PropertyCategory.BEHAVIOR, description = "The state of the connection - true = connected, false = disconnected")
public boolean ConnectionState()
{
  return this.connectionState;
}
```

```
/**
 * Creates the socket, connect to the server and launches the thread to receive data from server
 */
@SimpleFunction(description = "Tries to connect to the server and launches the thread for receiving data (blocking until connected or failed)")
public void Connect()
{
  if (connectionState == true)
  {
    throw new YailRuntimeError("Connect error, socket connected yet, please disconnect before reconnect !", "Error");
  }
  try
  {
    // connecting the socket
    clientSocket = new Socket();
    clientSocket.connect(new InetSocketAddress(serverAddress, Integer.parseInt(serverPort)), 5000);
    connectionState = true;
    outputStream = new BufferedOutputStream(clientSocket.getOutputStream());
    pout = new PrintStream(outputStream);

  }
  catch (SocketException e)
  {
    Log.e(LOG_TAG, "ERROR_CONNECT", e);
    throw new YailRuntimeError("Connect error" + e.getMessage(), "Error");
  }
  catch (Exception e)
  {
    connectionState = false;
    Log.e(LOG_TAG, "ERROR_CONNECT", e);
    throw new YailRuntimeError("Connect error (Socket Creation)" + e.getMessage(), "Error");
  }
}


/**
 * Send data through the socket to the server
 */
@SimpleFunction(description = "Send data to the server")
public void SendData(final String data)
{

  if (connectionState == false)
  {
    throw new YailRuntimeError("Send error, socket not connected.", "Error");
  }

  // we then send asynchonously the data
  AsynchUtil.runAsynchronously(new Runnable()
  {
    @Override
    public void run()
    {
      try
      {
        pout.print(data + "\r\n");
        pout.flush();
      }
      catch (Exception e)
      {
        Log.e(LOG_TAG, "ERROR_UNABLE_TO_SEND_DATA", e);
        throw new YailRuntimeError("Send Data", "Error");
      }
    }
  } );
}
```

```
/**
 * Close the socket
 */
@SimpleFunction(description = "Disconnect to the server")
public void Disconnect()
{
  try {
  if (connectionState == true)
  {
    pout.println("end");
    pout.flush();
    outputStream.flush();
    outputStream.flush();
    pout.close();
    outputStream.close();
    clientSocket.close();
    connectionState = false;
    RemoteConnectionClosed();
    return;
  } else {
    throw new YailRuntimeError("Socket not connected, can't disconnect.", "Error");
}
  }catch (SocketException e)
    {
      // modifications by axeley too :-)
      if(e.getMessage().indexOf("ENOTCONN") == -1)
      {
        Log.e(LOG_TAG, "ERROR_CONNECT", e);
        throw new YailRuntimeError("Disconnect" + e.getMessage(), "Error");
       }
       // if not connected, then just ignore the exception
    }
    catch (IOException e)
    {
      Log.e(LOG_TAG, "ERROR_CONNECT", e);
      throw new YailRuntimeError("Disconnect" + e.getMessage(), "Error");
    }
    catch (Exception e)
    {
      Log.e(LOG_TAG, "ERROR_CONNECT", e);
      throw new YailRuntimeError("Disconnect" + e.getMessage(), "Error");
    }
}
 /**
  * Event indicating that the remote socket closed the connection
  *
  */
 @SimpleEvent
 public void RemoteConnectionClosed()
 {
   // invoke the application's "RemoteConnectionClosed" event handler.
   EventDispatcher.dispatchEvent(this, "RemoteConnectionClosed");
 }
}
```

Figure 57: ClientSocketAI2Ext.java

As seen above the extension takes care of the TCP socket connection, the sending of the data as well as closing the socket when the user chooses to disconnect from the server, or a connection has been lost.

The extension when implemented in APP inventor will allow the user to use the following code blocks as seen below.

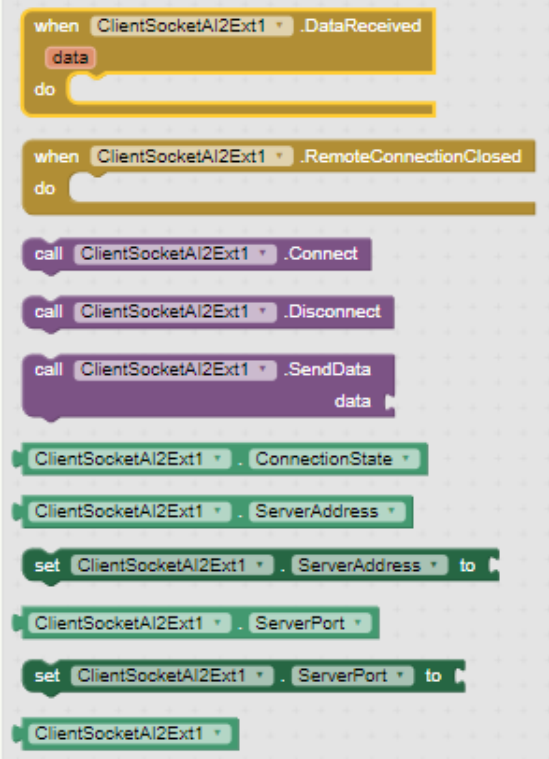


Figure 58: Client TCP extension blocks in app inventor

With this extension the user can now set up which server address and port number to connect to by using the set ClienSocketAI2Ext1.ServerAdrress/ServerPort. From here the user can then choose to connect to a server and send TCP packets with the function SendData. Once the user enters the IP address of the computer and port number used to establish the TCP connection, the user can then click the TCP_Connection button (Establish TCP connection) which will invoke the following code:

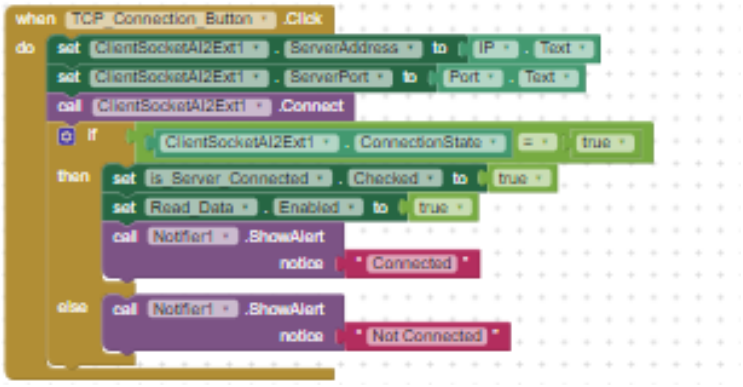


Figure 59: TCP connection button event

The TCP connection button event sets the server address and port number to the IP and Port number text box inputs and calls the connect function. If a connection is established the read data button is now enabled. Like mentioned previously the code to connect and read data from our data acquisition unit is similar from our data labeling unit, as seen below:

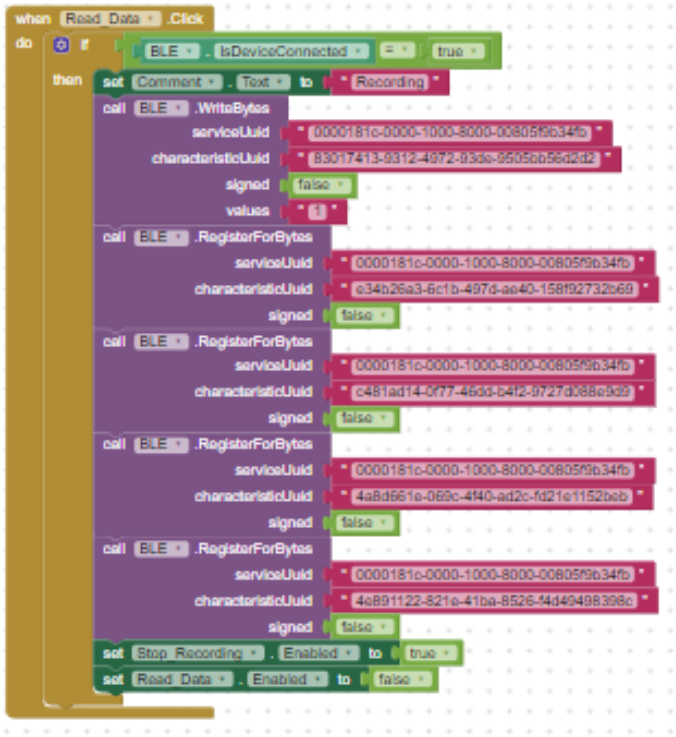


Figure 60: Read data button event

When a flex sensor reading is obtained, a TCP packet with the following format is created and sent to the server application:

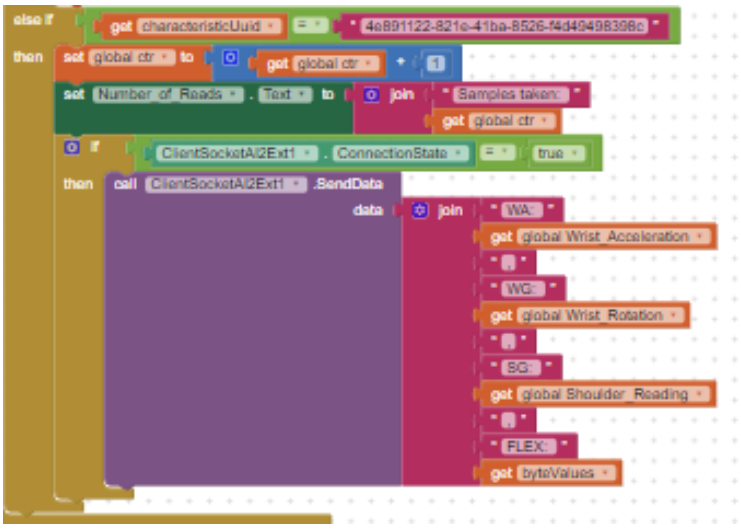


Figure 61: Sending data to TCP server (Inside Bytes Received event listener)

If the user presses stop recording button we simply sent the server application a TCP packet with the message “pause” and send the command ‘0’ to our data acquisition unit to stop the recording, as well as clearing the input buffers as seen below:

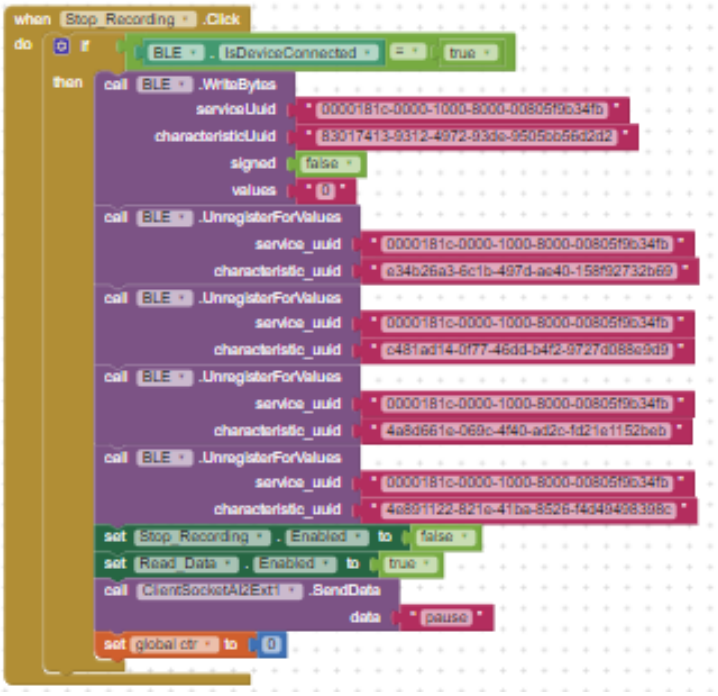


Figure 62: Stop recording button event

MATLAB REAL TIME DATA VISUALIZATION SERVER IMPLEMENTATION

The MATLAB server simply creates different graphs from all our inputs and post-processes the data with the standard calibration values given in the datasheet of the sensors. This application can be useful for calibrating sensor values, as well as detecting anomalies in the data. The code is well commented and explained as seen from the figures below:

```
clear, clc
%Setting up universal variables
check = 1;
xmax = 65;                          %X Maximum Value
xmin = 0;                           %X Minimum Value
yminA = -3;                         % set Y-min acceleration value
ymaxA = 3;                          % set Y-max acceleration value
yminG = -250;                       % set Y-min gyroscope rotation value
ymaxG = 250;                        % set Y-max gyroscope rotation value
yminFlex = -255;                    %set Y max for flex
ymaxFlex =  255;                    %set Y min for flex
delay = .05;                        % delay between readings
time = 0;                           %Variable to hold current time
count = 1;                          %Counts the number of samples taken
dX1 = 0;                            %X-axis value 1
dY1 = 0;                            %Y-axis value 1
dZ1 = 0;                            %Z-axis value 1
dX2 = 0;                            %X-axis value 2
dY2 = 0;                            %Y-axis value 2
dZ2 = 0;                            %Z-axis value 2
dX3 = 0;                            %X-axis value 3
dY3 = 0;                            %Y-axis value 3
dZ3 = 0;                            %Z-axis value 3
dFlex = 0;                          %Flex reading value

xLabel = 'Sample Number';               % Universal x-label for all figures
yLabelA = 'Linear Acceleration (g)';    % y-axis label for accelerometer
yLabelG = 'Angular Rotation Dp(s)';     % y-axis label for gyroscopes
yLabelFlex = 'Flex Raw values';
plotGrid = 'off';
```

```
%-----Variables for first figure-----------------------------------------
plotTitle1 = 'Wrist Acceleration';  % plot title
legendwax = 'X-Acceleration';
legendway = 'Y-Acceleration';
legendwaz = 'Z-Acceleration';
%----Variables for second figure-----------------------------------------
plotTitle2 = 'Wrist Rotation';  % plot title
legendwgx = 'Roll';
legendwgy = 'Pitch';
legendwgz = 'Yaw';
%----Variables for third figure-----------------------------------------
plotTitle3 = 'Shoulder Rotation';  % plot title
legendsgx = 'Roll';
legendsgy = 'Pitch';
legendsgz = 'Yaw';
%----Variables for forth figure
plotTitle4 = 'Flex Reading';
legendFlex = 'Flex raw value';
```

```
%----Setting up first figure---------------------------------------------
f1 = figure; %Figure 1 is used to display wrist accelerometer values
%Set up Plot
plotGraphwax = plot(time,dX1,'-r' );%every AnalogRead needs to be on its own Plotgraph
hold on                             %hold on makes sure all of the channels are plotted
plotGraphway = plot(time,dY1,'-b');
plotGraphwaz = plot(time,dZ1,'-g' );
title(plotTitle1,'FontSize',15);
xlabel(xLabel,'FontSize',15);
ylabel(yLabelA,'FontSize',15);
legend(legendwax,legendway,legendwaz);
xlim([xmin xmax]);
ylim([yminA ymaxA]);
grid(plotGrid);
```

```
%----Setting up Second figure--------------------------------------------
f2 = figure; %Figure 2 is used to display wrist gyroscope values
%Set up Plot
plotGraphwgx = plot(time,dX2,'-r' ); %every AnalogRead needs to be on its own Plotgraph
hold on                              %hold on makes sure all of the channels are plotted
plotGraphwgy = plot(time,dY2,'-b');
plotGraphwgz = plot(time,dZ2,'-g' );
title(plotTitle2,'FontSize',15);
xlabel(xLabel,'FontSize',15);
ylabel(yLabelG,'FontSize',15);
legend(legendwgx,legendwgy,legendwgz);
xlim([xmin xmax]);
ylim([yminG ymaxG]);
grid(plotGrid);
```

```
%----Setting up Third figure-----------------------------------------------
f3 = figure; %Figure 3 is used to display Shoulder gyrscope values
plotGraphsgx = plot(time,dX3,'-r' );%every AnalogRead needs to be on its own Plotgraph
hold on                             %hold on makes sure all of the channels are plotted
plotGraphsgy = plot(time,dY3,'-b');
plotGraphsgz = plot(time,dZ3,'-g' );
title(plotTitle3,'FontSize',15);
xlabel(xLabel,'FontSize',15);
ylabel(yLabelG,'FontSize',15);
legend(legendsgx,legendsgy,legendsgz);
xlim([xmin xmax]);
ylim([yminG ymaxG]);
grid(plotGrid);

%NOTE: NEW GRAPH CAN BE IMPLEMENTED  FOR FLEX VALUES!
% f4 = figure; %Figure 4 is used to display FLEX
% plotGraphFlex = plot(time,dFlex,'-r' );
% title(plotTitle4,'FontSize',15);
% xlabel(xLabel,'FontSize',15);
% ylabel(yLabelFlex,'FontSize',15);
% legend(legendFlex);
% xlim([xmin xmax]);
% ylim([yminFlex ymaxFlex]);
% grid(plotGrid);
%Starting TCP CLIENT
t = tcpip('0.0.0.0',8000,'NetworkRole', 'server'); %Tcp object port 8000
fopen(t);%Wait for connection
record = 1;
pause(0.5);%Wait for half a second

while record == 1 %While recording
   flushinput(t);%Flush buffer input
   while (count<xmax)%While our graph is within x-limits
    data = fscanf(t);
    disp(data);
    if contains(data,'end') == 1 %Close the application!
       count = 1000;
       check = 2;
       record = 2;
    elseif contains(data,'pause') == 1 %Wait for new command
       pause(0.01);
       flushinput(t);
```

```
elseif contains(data,'WA') == 1 %If we have data
%Split the data accordingly and then post-process it!
wristA = extractBetween(data,'WA: (',')');%Wrist accel X,Y,Z readings
wristG = extractBetween(data,'WG: (',')');%Wrist gyro X,Y,Z readings
shoulderG = extractBetween(data,'SG: (',')');%Shoulder gyro X,Y,Z readings
Flex = extractBetween(data,'FLEX: (',')');%Flex reading
WA = split(wristA);%Split readings by spaces into array WA
RawWAX = str2double(WA{1});
RawWAY = str2double(WA{2});
RawWAZ = str2double(WA{3});
if RawWAX >= 32768
  RawWAX = RawWAX - 65536;
end
if RawWAY >= 32768
   RawWAY = RawWAY - 65536;
end
if RawWAZ >= 32768
   RawWAZ = RawWAZ - 65536;
end
%Applying standard datasheet calibration
WAX = ((RawWAX*0.061)/1000);
WAY = ((RawWAY*0.061)/1000);
WAZ = ((RawWAZ*0.061)/1000);

WG = split(wristG);
RawWGX = str2double(WG{1});
RawWGY = str2double(WG{2});
RawWGZ = str2double(WG{3});
if RawWGX >= 32768
      RawWGX = RawWGX - 65536;
end
if RawWGY >= 32768
   RawWGY = RawWGY - 65536;
end
if RawWGZ >= 32768
    RawWGZ = RawWGZ - 65536;
end
WGX = ((RawWGX*8.75)/1000);
WGY = ((RawWGY*8.75)/1000);
WGZ = ((RawWGZ*8.75)/1000);
```

```
    SG = split(shoulderG);
    RawSGX = str2double(SG{1});
    RawSGY = str2double(SG{2});
    RawSGZ = str2double(SG{3});
    if RawSGX >= 32768
        RawSGX = RawSGX - 65536;
    end
    if RawSGY >= 32768
       RawSGY = RawSGY - 65536;
    end
    if RawSGZ >= 32768
       RawSGZ = RawSGZ - 65536;
    end
    SGX = ((RawSGX*8.75)/1000);
    SGY = ((RawSGY*8.75)/1000);
    SGZ = ((RawSGZ*8.75)/1000);
```

```
%----------GRAPHING-----------------------------------------------------------
    time(count) = count;
    dX1(count) = WAX;
    dY1(count) = WAY;
    dZ1(count) = WAZ;
    dX2(count) = WGX;
    dY2(count) = WGY;
    dZ2(count) = WGZ;
    dX3(count) = SGX;
    dY3(count) = SGY;
    dZ3(count) = SGZ;
    %dFlex(count) = Flex;
    %This is the magic code
    set(plotGraphwax,'XData',time,'YData',dX1);
    set(plotGraphway,'XData',time,'YData',dY1);
    set(plotGraphwaz,'XData',time,'YData',dZ1);
    set(plotGraphwgx,'XData',time,'YData',dX2);
    set(plotGraphwgy,'XData',time,'YData',dY2);
    set(plotGraphwgz,'XData',time,'YData',dZ2);
    set(plotGraphsgx,'XData',time,'YData',dX3);
    set(plotGraphsgy,'XData',time,'YData',dY3);
    set(plotGraphsgz,'XData',time,'YData',dZ3);
   %set plotGraphFlex,'XData',time,'YData',dFlex);
    pause(delay);
    count = count + 1;
    flushinput(t);%Flush input obtained while graphing to smoothen graphing
    else %If nothing is arrived wait
       pause(0.01);
       flushinput(t);
    end %End if statements
-  end %While count<max ending
```

```
  %If still connected but we have reached our maximum x-axis value
  %Reset all graphs to keep visualizing data!
if (check == 1)
clear time count dX1 dX2 dX3 dY1 dY2 dY3 dZ1 dZ2 dZ3 dFlex
count = 1;
time = 0;
dX1 = 0;
dY1 = 0;
dZ1 = 0;
dX2 = 0;
dY2 = 0;
dZ2 = 0;
dX3 = 0;
dY3 = 0;
dZ3 = 0;
dFlex = 0;
figure(f1);
cla reset %Reset all axis of the current figure
%Set up Plot
plotGraphwax = plot(time,dX1,'-r' );%every AnalogRead needs to be on its own Plotgraph
hold on                              %hold on makes sure all of the channels are plotte
plotGraphway = plot(time,dY1,'-b');
plotGraphwaz = plot(time,dZ1,'-g' );
title(plotTitle1,'FontSize',15);
xlabel(xLabel,'FontSize',15);
ylabel(yLabelA,'FontSize',15);
legend(legendwax,legendway,legendwaz);
xlim([xmin xmax]);
ylim([yminA ymaxA]);
grid(plotGrid);


figure(f2);
cla reset %Reset all axis of the current figure
%Set up Plot
plotGraphwgx = plot(time,dX2,'-r' );%every AnalogRead needs to be on its own Plotgraph
hold on                              %hold on makes sure all of the channels are plotte
plotGraphwgy = plot(time,dY2,'-b');
plotGraphwgz = plot(time,dZ2,'-g' );
title(plotTitle2,'FontSize',15);
xlabel(xLabel,'FontSize',15);
ylabel(yLabelG,'FontSize',15);
legend(legendwgx,legendwgy,legendwgz);
xlim([xmin xmax]);
ylim([yminG ymaxG]);
grid(plotGrid);
```

```
figure(f3);
cla reset %Reset all axis of the current figure
plotGraphsgx = plot(time,dX3,'-r' );%every AnalogRead needs to be on its own Plotgraph
hold on                             %hold on makes sure all of the channels are plotte
plotGraphsgy = plot(time,dY3,'-b');
plotGraphsgz = plot(time,dZ3,'-g' );
title(plotTitle3,'FontSize',15);
xlabel(xLabel,'FontSize',15);
ylabel(yLabelG,'FontSize',15);
legend(legendsgx,legendsgy,legendsgz);
xlim([xmin xmax]);
ylim([yminG ymaxG]);
grid(plotGrid);
% figure(f4);
% cla reset %Reset all axis of the current figure
% plotGraphFlex = plot(time,dFlex,'-r' );
% title(plotTitle4,'FontSize',15);
% xlabel(xLabel,'FontSize',15);
% ylabel(yLabelFlex,'FontSize',15);
% legend(legendFlex);
% xlim([xmin xmax]);
% ylim([yminFlex ymaxFlex]);
% grid(plotGrid);


pause(0.06);%Wait
flushinput(t); %Flush input
else
   fprintf('Program closed');
   flushinput(t);
end

end
fclose(t);
```

Figure 63: MATLAB server code implementation

The code above allows user to visualize the data from the data acquisition unit as it arrives. It sets a maximum x-axis limit of 65 samples, before clearing the graphs values. The reason behind this, is because as the number of samples taken increases, the memory starts to fill quite quickly, therefor the data visualization slows down. This number can be tweaked to allow users to display more samples. The delay can also be modified to allow users to only obtain data at a specific time interval.

DATABASE DOCUMENTATION

DATA BASE DESIGN

As mentioned in the section: Software Design/Software Packages, of this document, the data base of choice for this project is Firebase and it has been set up with multiple users in mind. The database is structured in a JSON tree format, where each exercise pose contains a reading number child. This reading child node contains all the information regarding the sensor readings obtained from the BLE module as seen below:

1-Main Data-Base 2-Exercises reading child nodes expanded

Figure 64-65: Data base set up format

Note: When the user synchs with the database, a list of the child reading nodes for the selected exercise is returned, from this list, the application now creates a new node with the new reading being Reading-N, where N is equal to the latest reading node + 1

SYSTEM INTERFACES

BLE GAP AND GATT PROTOCOL

Standard BLE profile access and data exchange protocol. These protocols set the standard for communication across BLE devices. GAP and GATT were used to establish the medium of communication between the data acquisition unit and the mobile applications. For more information regarding the usage of these protocols please check the link below:

https://punchthrough.com/bean/docs/guides/everything-else/how-gap-and-gatt-work/

TCP PROTOCOL

TCP is the standard transmission control protocol which is used between the real time application and MATLAB, to be able to visualize data in real time.

SYSTEM PERFORMANCE / CONCLUSION

The proposed system can acquire data from the sensors at a sampling frequency of 25HZ. For the user to be able to upload and visualize data in real time, the smartphone must always be connected to the internet! If the application is not connected to the internet, the data acquired will not be uploaded. The database can withhold a total of 1GB of uploads and 10GB of downloads per month. The data acquisition sends approximately 475 Bytes of data per second (19 Bytes per sample * 25). With this information, we can expect to record data for a total of approximately 2105263 seconds per month (1GB/475 Bytes per second), which equals 35086 minutes, which is approximately 580 hours of data per month. For research purposes we believe this is more than enough. If the product goes commercially, the database will have to be updated to be able to handle multiple users and the mobile applications will have to be updated as well. The main idea for this product was to create a simple data acquisition unit, which allows developers and researchers to acquire data for different upper body exercises and label it on the go, to be able to classify it further on with machine learning. This has been achieved by using tools that are easy to debug and understand such as AI2 MIT APP inventor and firebase database. The system is also energy efficient and one can reference to the hardware design documentation for a more detail explanation of power consumption and the overall hardware. If one wishes to further develop the mobile application and data acquisition unit, head over to the software design document for software code and implementation.