Developer Manual

FROGSTAR

Computer Science Department

Texas Christian University

Date: April 23, 2014

Authors: Stockton Ackermann, Nicholas Capurso, Eric Elsken, Myrella Garcia,

Casey Stephens, and David Woodworth.

Revision History

The following is a history of document revisions.

Version	Changes	Edited
Version 1.0	Initial Draft	March 17, 2014
Version 2.0	• Fixed grammatical errors • Standardized font size • Added diagrams	April 23, 2014

Revision Sign-off

By signing the following, the team member asserts that he or she has read the entire document and has, to the best of knowledge, found the information contained herein to be accurate, relevant, and free of typographical error.

Name	Signature	Date
Stockton Ackermann s.ackermann@tcu.edu		
Nicholas Capurso nick.capurso@tcu.edu		
Eric Elsken eric.elsken@tcu.edu		
Myrella Garcia m.garciaurena@tcu.edu		
Casey Stephens casey.stephens@tcu.edu		
David Woodworth d.woodworth@tcu.edu		

Table of Contents

Developer Manual

Version 2.0

1. Introduction

1.1. Purpose

This document provides a guide on how to install and setup the entire FrogStar system for future developers. Also provided will be explanations as to how to maintain and modify the system as the need may arise.

1.2. Section Overview

Section 2 – **System Overview:** Gives a brief description of the FrogStar system.

Section 3 – **Development Setup:** Gives a guide on setting up a development environment to develop for FrogStar.

Section 4 – **Android Application Development:** Shows what was necessary in order to complete specific classes inside the application.

Section 5 – **On-Board Control Unit Development:** Explains the programming done for the on-board control unit.

Section 6 - Glossary of Terms: Defines technical and project-specific terms used in this document.

2. System Overview

2.1. System Components

There are three main components in the entire FrogStar system: The smartphone application, the on-board control unit (OBCU), and the TI CC2541 SensorTags.

2.2. Smartphone Application

The smartphone application is used to store personal information about the user as well as information describing one or more vehicles. The smartphone also queries its own accelerometer and gyroscope in order to detect accidents. The user will have limited functionality while using the app, but a technician may enter Technician Mode to make sure all systems are working properly.

2.3. On-Board Control Unit (OBCU)

The OBCU is in charge of querying the SensorTags and checking them against the readings of the smartphone. In this way the phone and the OBCU may correct false positives to establish with more certainty that an accident has occurred. The OBCU must be connected to a vehicle power source so that it can be powered on when the vehicle starts. The OBCU also must have five Bluetooth LE USB adapters connected in order for communication with the SensorTags to take place.

2.4. TI SensorTags

The TI SensorTags must be placed in Bluetooth LE range of the OBCU. These SensorTags will send accelerometer and gyroscope readings to the OBCU. These readings will be used to determine if an accident has taken place. The SensorTags will run solely on battery power. If a sensor fails the OBCU will be alerted and prompt the user that a sensor must be replaced.

3. Development Setup

3.1. Smartphone Application

The FrogStar application code may be found on the DVD under trunk/code/FrogStarApp folder.

In order to develop the smartphone application, Eclipse must be installed on the development computer. The Android plugin must be downloaded and installed from the Android Development Website. The following link may be used in order to download the Android plugin: https://developer.android.com/sdk/index.html. Development takes place in Eclipse using the Android Development plugin. Testing of the application can be done on the smartphone itself or on the emulator provided by Eclipse.

3.2. On-Board Control Unit Programming

The OBCU networking programs may be found on the DVD under trunk/code/OBCU Programs folder.

To develop on the On-Board Control Unit, the Raspberry Pi, one needs an SD Card with Raspbian Linux installed – the image can be downloaded from http://www.raspbian.org/. The OS, by default, does not come with a GUI, though one is not required to develop and run the FrogStar system.

With the OS installed, a number of packages are needed to develop, modify, and compile FrogStar code. The main package used is the BlueZ Bluetooth stack for Linux. The FrogStar system is based off of a modified version of the software included in the BlueZ package. The source code can be obtained and extracted via the following commands (note: all commands are run as a root user):

```
# wget https://www.kernel.org/pub/linux/bluetooth/bluez-5.2.tar.xz
# tar xvf bluez-5.2.tar.xz
```

A number of other packages are also needed to build the BlueZ package. The required packages are enumerated in the following apt-get command:

```
# apt-get install libusb-dev libdbus-1-dev libglib2.0-dev automake libudev-dev
libical-dev libreadline-dev
```

Note: these package names are based on Debian-based repositories.
To build the package, execute the following commands:

```
# cd bluez-5.2
# ./configure --disable-systemd
# make
```

If you would like some of the software that comes with BlueZ (gatttool and hcitool, discussed in Section 5.3) to be installed on your Raspberry Pi, run the command “make install” after executing the above commands. However, this step is not necessary to modify and compile FrogStar code.

Finally, to transfer the OBCU code off of the DVD and onto the Raspberry Pi, you'll need to transfer the code onto a USB. Then, plug it into the Raspberry Pi and transfer the code to your desired location using the copy command (if the USB doesn't automatically mount, you'll have to execute the Linux commands to mount the USB).

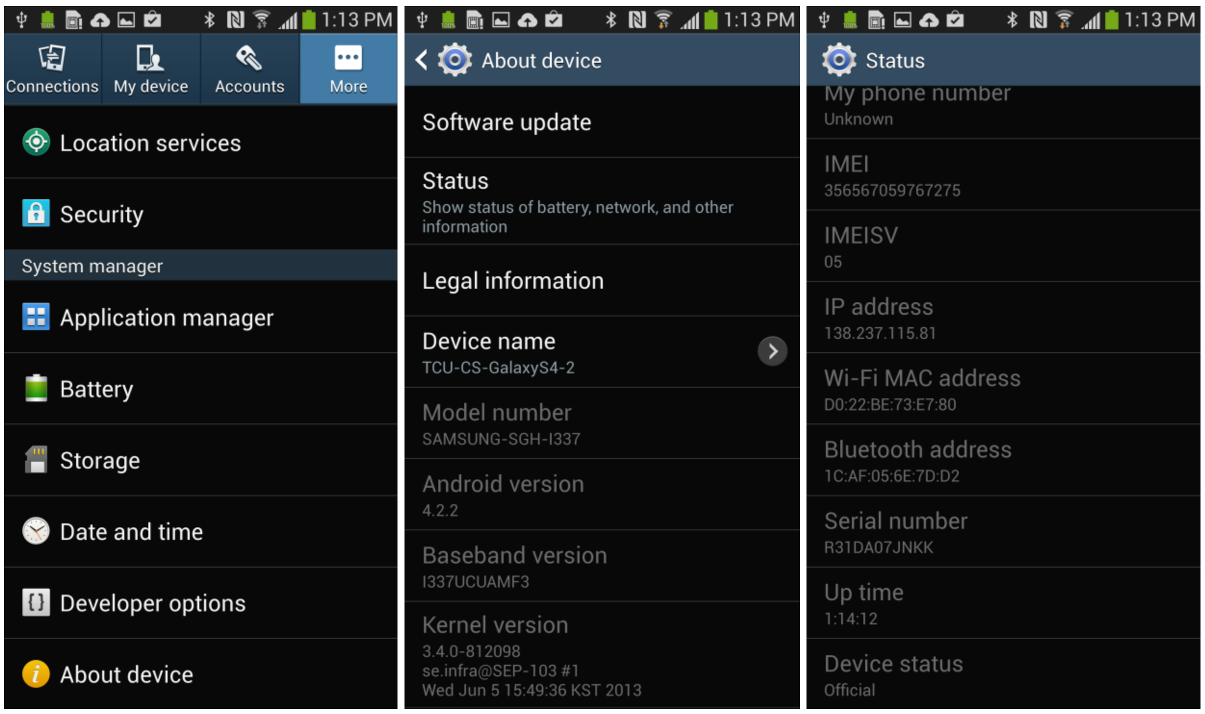
For a more in-depth explanation on modifying and compiling FrogStar code, please reference Section 5 and Section 5.3.

3.3. Bluetooth Pairing via Command Line

In order for the OBCU and a smartphone to communicate via Bluetooth, the two devices must be "paired" and "trusted." In addition to the BlueZ, the following packages are needed to performing pairing (note: all commands are run as a root user):

```
# apt-get install python-gobject python-dbus
```

Next, you will need to know the smartphone's Bluetooth MAC address. This can be found in the "About device -> Status -> Bluetooth address" screen in the Settings application – this is shown below:


1:13 PM

Connections | My device | Accounts | More

Location services
Security
System manager
Application manager
Battery
Storage
Date and time
Developer options
About device

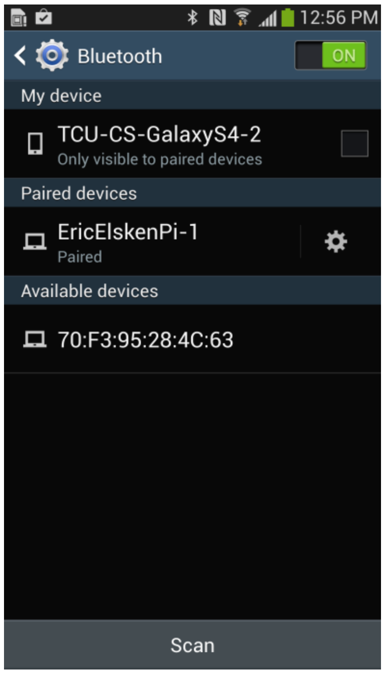
About device

Software update
Status
Show status of battery, network, and other information
Legal information
Device name
TCU-CS-GalaxyS4-2
Model number
SAMSUNG-SGH-I337
Android version
4.2.2
Baseband version
I337UCUAMF3
Kernel version
3.4.0-812098
se.infra@SEP-103 #1
Wed Jun 5 15:49:36 KST 2013

Status

My phone number
Unknown
IMEI
356567059767275
IMEISV
05
IP address
138.237.115.81
Wi-Fi MAC address
D0:22:BE:73:E7:80
Bluetooth address
1C:AF:05:6E:7D:D2
Serial number
R31DA07JNKK
Up time
1:14:12
Device status
Official

To pair the two devices, first make sure your smartphone is advertising its Bluetooth MAC address by going into the Bluetooth options in the Settings application, shown below:



TCU-CS-GalaxyS4-2
Only visible to paired devices

EricElskenPi-1
Paired

70:F3:95:28:4C:63

In this example screenshot, you would need to tap on your phone's listing (ex "TCU-CS-GalaxyS4-2") to cause it to be visible to the OBCU. The phone stays visible for a limited amount of time before hiding itself again.

Next, with the phone visible to the OBCU, run the following command on the OBCU:

```
# bluez-simple-agent hci0 XX:XX:XX:XX:XX:XX
```

The XX's represent the phone's Bluetooth MAC address. The bluez-simple-agent program will ask you to input a PIN number – input any four-digit number and hit Enter. Next, a Bluetooth pairing dialog will be displayed on the phone and you will be prompted to enter the same four-digit number you chose. If entered correctly, the two devices will now be connected.

Next, you need to make the phone trusted by the OBCU (so this process won't have to be repeated). To do this, run the following command after the two devices are connected:

```
# bluez-test-device trusted XX:XX:XX:XX:XX:XX yes
```

Finally, the Bluetooth daemon on the OBCU must be restarted:

```
# /etc/init.d/Bluetooth restart
```

3.4. Obtaining MAC Addresses

SensorTag MAC addresses are needed to be programmed into a NFC tag for the FrogStar system to startup correctly. To obtain SensorTag MAC addresses, run the following command as root:

```
# hcitool lescan
```

At this point, the hcitool program will be actively scanning for Bluetooth LE devices. Press the white button the side of each SensorTag which causes it to start Bluetooth advertising. This will cause its MAC address to be printed by hcitool.

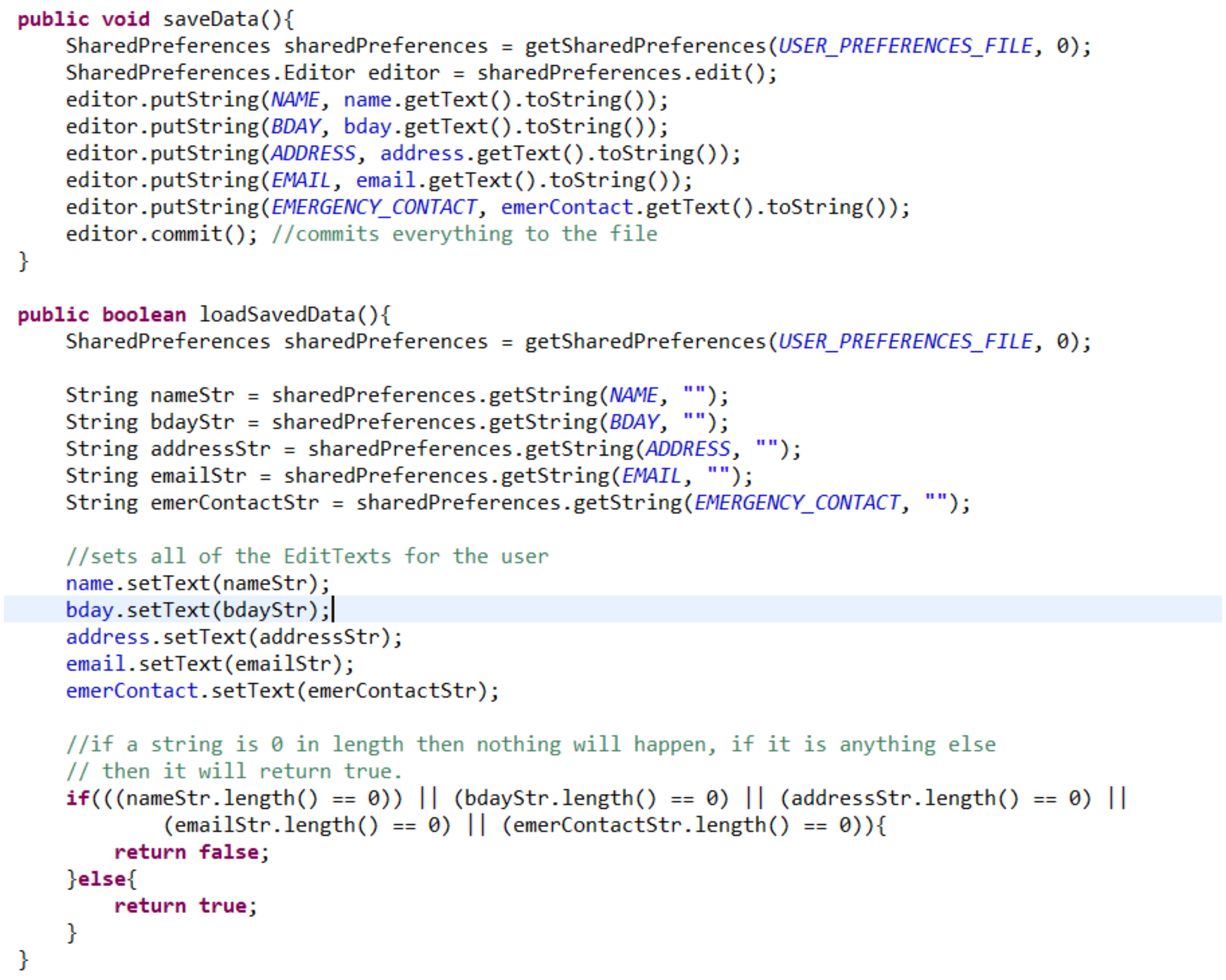
To get the MAC addresses of the Bluetooth adapters plugged in to the OBCU, run the following command:

```
# hcitool dev
```

4. Android Application Development

4.1. User Profile

The User Profile screen is made up of five different input fields and one button to move onto the next screen. The user must enter his name, birthday, address, e-mail, and an emergency contact. The name, address, and e-mail are all EditTexts. The birthday is chosen by a DatePicker and the emergency contact is chosen by using a ContactPicker. Once all fields are filled out and the user clicks on the “Submit” button, all of the data is saved to a flat file within the application. Below is the code that saves all of the data the user has input.



```
public void saveData(){
    SharedPreferences sharedPreferences = getSharedPreferences(USER_PREFERENCES_FILE, 0);
    SharedPreferences.Editor editor = sharedPreferences.edit();
    editor.putString(NAME, name.getText().toString());
    editor.putString(BDAY, bday.getText().toString());
    editor.putString(ADDRESS, address.getText().toString());
    editor.putString(EMAIL, email.getText().toString());
    editor.putString(EMERGENCY_CONTACT, emerContact.getText().toString());
    editor.commit(); //commits everything to the file
}

public boolean loadSavedData(){
    SharedPreferences sharedPreferences = getSharedPreferences(USER_PREFERENCES_FILE, 0);

    String nameStr = sharedPreferences.getString(NAME, "");
    String bdayStr = sharedPreferences.getString(BDAY, "");
    String addressStr = sharedPreferences.getString(ADDRESS, "");
    String emailStr = sharedPreferences.getString(EMAIL, "");
    String emerContactStr = sharedPreferences.getString(EMERGENCY_CONTACT, "");

    //sets all of the EditTexts for the user
    name.setText(nameStr);
    bday.setText(bdayStr);
    address.setText(addressStr);
    email.setText(emailStr);
    emerContact.setText(emerContactStr);

    //if a string is 0 in length then nothing will happen, if it is anything else
    // then it will return true.
    if(((nameStr.length() == 0)) || (bdayStr.length() == 0) || (addressStr.length() == 0) ||
            (emailStr.length() == 0) || (emerContactStr.length() == 0)){
        return false;
    }else{
        return true;
    }
}
```

Using the SharedPreferences API call to write data to a file was relatively simple. Error checking is also implemented to ensure the user does not enter anything incorrectly into the required fields. Regular expressions are used to do this sort of error checking. Below is an example of one of the regular expressions:

```
if(!name.getText().toString().matches("[a-zA-Z '.-]+")) {
     result += "Name\n";
}
```

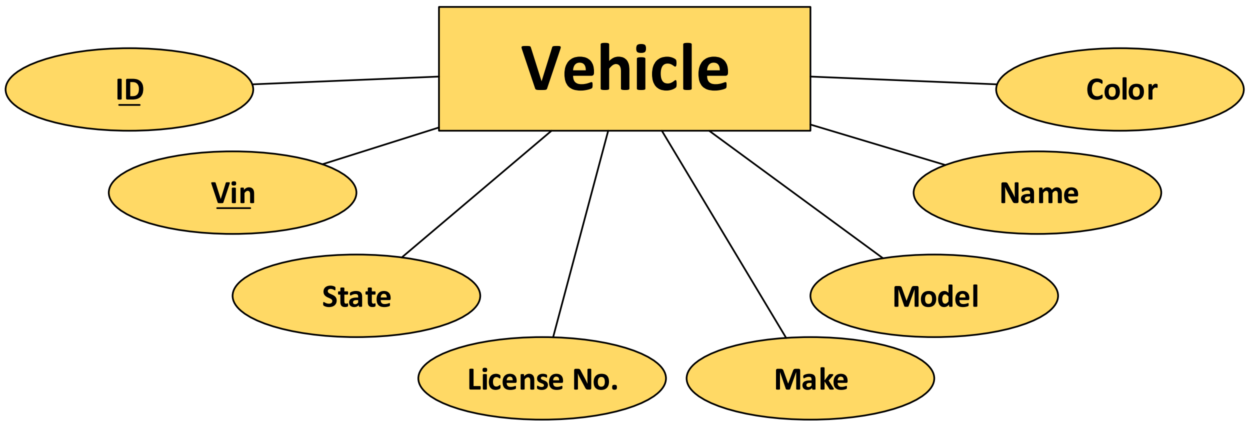
There are similar statements to ensure valid data has been input into the EditTexts. Once all data is validated the user may click the submit button and move onto the Vehicle Profile screen.

4.2. Vehicle Profile

The VehicleProfile activity is where the user will have to input information about the vehicle he will be driving. The make, model, license plate, state, V.I.N., name, and color are required to be entered. Unlike the UserProfile activity, we store all of this information in a database. The layout is very similar to the UserProfile activity, with EditTexts and dropdown menus for the user to input all of the relevant data. Again, if information has already been input into the database, the user will be taken to the HomeScreen activity. If there is no data in the database the user will stay on this activity until information has been input and validated.

4.2.1. Database Development

The database for this project is not very complex. The user may input as many vehicles he wishes. Here is the ER diagram for the database that we have used.



The ID and the VIN are used as primary keys for vehicles; all the other fields are simply attributes. The ID field is actually named "_id" in the application. This is a field required by Android to be in its tables. There are many classes that have been created in order for this database to function correctly: DatabaseActivity, DatabaseLoader, MySQLiteHelper, VehicleDataSource, VehicleProfile, and Vehicle.

DatabaseActivity is a ListActivity that implements the standard Loader callbacks provided by Android. The Loader being used is a custom Loader implementation (DatabaseLoader) that loads all Vehicles currently stored in the database. Reading the documentation on a ListActivity and Loaders should be enough to understand how this activity works.
http://developer.android.com/reference/android/app/ListActivity.html
http://developer.android.com/reference/android/content/Loader.html
http://developer.android.com/guide/components/loaders.html

DatabaseLoader is an implementation of a Loader that loads a Cursor containing all the Vehicles in the database and copies them to a List. This implementation is a common, one-time Loader implementation. Documentation can be found online as to how this works. The important part of the class is the following:

```
public List<Vehicle> loadInBackground() {
	Log.d("FrogStar", "loading cursor");
	Cursor cur = datasource.getAllVehicles();
	List<Vehicle> temp = new ArrayList<Vehicle>();
	while(cur.moveToNext()){
		temp.add(VehicleDataSource.vehicleFromCursor(cur));
	}
	return temp;
}
```

This is the method that is called whenever data needs to be loaded by the Loader. The datasource object is an instance of our VehicleDataSource class described later. A Cursor of all the Vehicles is loaded and copied into a list.

The MySQLiteHelper and VehicleDataSource classes are the two drivers for data storage and retrieval. In MySQLiteHelper, the DATABASE_CREATE String is the definition of our Vehicle table schema. The VehicleDataSource and MySQLiteHelper classes are implemented based on the following site: http://www.vogella.com/tutorials/AndroidSQLite/article.html

Of important note are the static public methods defined in VehicleProfile: setDefaultVehicle() and getDefaultVehicleId(). These both store and retrieve the default, or currently selected, Vehicle ID to and from the Preferences of the application.

Finally, Vehicle is a holder class for all the fields in the Vehicle table. All fields have typical getters and setters. toString() is also overridden to allow simple formatting of Vehicles in different TextViews throughout the application.

4.3. Home Screen

4.3.1. NfcActivity

The NfcActivity class serves as the main screen, or hub, of the FrogStar application - it is from here that the user can go to any other activity in the application. This activity is responsible for initiating system startup or shutdown via an NFC tag swipe, launching and interfacing with the Bluetooth and SensorServices, prompting the user to enable Bluetooth and NFC functionality, and alerting the user of errors and confirmed crashes via dialog popups.

The operation sequence of the activity is as follows: if Bluetooth or NFC functionality is disabled, the user will first be prompted via dialogs to enable the required functions – if this is not done, the application will exit. Next, the activity starts the Bluetooth and SensorServices and waits for an NFC tag to be swiped. If a FrogStar-compatible (MIFARE Ultralight) NFC tag is swiped with the smartphone, five MAC addresses are read from the tag and are passed to the BluetoothService to begin connecting to the OBCU. When the OBCU has been connected to and is ready to begin accident detection, the user is told that the system is ready and the calibration dialog will be displayed. Calibration prompts the user to place their smartphone on a flat surface and hit the ‘Okay’ button – at this point, the SensorService is given the signal to perform calibration and both the smartphone and the OBCU begin accident detection.

While performing accident detection, the user is shown the uptime of the system, the user’s name, and their selected vehicle. If any errors occur, such as a disconnected SensorTag or network timeout, the user will be alerted via a dialog and the system will continue operating if the error isn’t fatal. If an accident is detected and confirmed by both devices, the user is shown the emergency response dialog where they have 30 seconds to either choose to notify emergency services or decline aid – the default action if the timeout expires is to notify emergency services.

To shut the system down when no longer needed, the user simply swipes the initial FrogStar-compatible NFC tag again. After receiving confirmation that the OBCU is preparing to shutdown, the NfcActivity stops both the Bluetooth and SensorServices and the application exits.

4.3.2. Helper Classes – BluetoothManager and NfcManager

The NfcActivity utilizes two helper classes to function properly: the BluetoothManager class and the NfcManager class. These classes aid in modularizing the code and diving responsibilities appropriately between all three classes: the NfcActivity is responsible for user interactions, the BluetoothManager is responsible for interfacing with the BluetoothService, and the NfcManager is responsible for NFC-related actions.

The BluetoothManager class acts as an interface between the NfcActivity and the BluetoothService. Mainly, it is responsible to binding to the service, invoking the service’s public methods, sending messages to the service, and unbinding as well as stopping the service. Messages that the BluetoothManager takes care of are as follows: sending MAC addresses, the shutdown command, and the confirm accident command. The BluetoothManager also provides the BluetoothService with a reference to the NfcActivity's Handler, so that the service may send messages directly back to the activity.

The NfcManager class handles reading from and writing to NFC tags. In addition, it is also responsible for setting up the foreground dispatch so that NFC-related Intents may be delivered to the NfcActivity. Its key role in the system is to extract the five MAC addresses that have been written to an NFC tag. In Technician Mode, the NfcManager is also used to write MAC addresses to a new or existing tag. To conserve space on the NFC tag, MAC addresses are written without colons, so they must be placed back in after being read in order for them to be used by the OBCU. Finally, the NfcManager is responsible for analyzing a blank NFC tag to make sure it can be written to and that it has enough space to fit all of the required data.

4.4. Technician Mode

Technician Mode provides a FrogStar technician the ability to read the contents (MAC addresses) on a FrogStar-formatted NFC tag, modify an NFC tag's content, or format a blank NFC tag to work with the FrogStar system. Technician Mode also allows a FrogStar technician to view real-time graphs of a device's accelerometer or gyroscope or to open saved accident data. It can be accessed via the NfcActivity menu.

Technician Mode makes use of the aforementioned NfcManager to read from and write to NFC tags.

4.4.1. Read and Write NFC Tags

To read and write from NFC tags place an NFC tag against the back of the phone while in the technician mode main screen. The five text boxes on this screen are used to display NFC-stored MAC addresses which have either been entered by a user or read from an NFC tag. The first four belong to the SensorTags configured to be used while the last one belong to the Bluetooth interface of the OBCU. The app automatically detects if the tag has previously been formatted to store FrogStar information and displays a prompt accordingly. If the tag is blank the prompt will ask if the user would like to write to it. If the tag already has FrogStar data on it the user will be prompted either to read the data or to overwrite it. If the user opts to write data to the tag then the contents of the five text boxes will be written. If the user opts to read then the tag will be read and the data automatically entered into the five text boxes.

4.4.2. View Real-time Sensors

The SensorGraphsActivity class serves as the screen where a technician may view real-time sensor graphs or accident data. It starts and binds to the SensorService and allows the service to deliver sensor readings to the activity so that they may be graphed. Accident detection is also performed on this screen so that a technician may test and optimize threshold values.

When an accident is detected, the technician is notified via a dialog and it told which sensor detected the accident.

A technician has the ability to view graphs for either the accelerometer or gyroscope, pause/resume graphs, clear the graphs, and modify threshold values for accident detection. The technician is also shown minimum and maximum values for each of the graphs.

The class makes use of a helper class, GraphViewManager, to handle setting up the graphs, adding new values, and creating the “real-time” effect. The class is explained in detail in Section 4.4.2.

The key method in this class is the Handler’s handleMessage method. Sensor values from the SensorService are delivered to this method and it is here that they are sent to the GraphViewManager to be added to the graphs and are also tested against our accident detection thresholds. The following code snippet comes from the handleMessage method:

```
//Get sensor readings from the Bundle
float x = data.getFloat(prefix + "X");
float y = data.getFloat(prefix + "Y");
float z = data.getFloat(prefix + "Z");

//Graph the new sensor reading
mGraphViewManager.addNewValue(GraphViewManager.TYPE_X_VALUE, x);
mGraphViewManager.addNewValue(GraphViewManager.TYPE_Y_VALUE, y);
mGraphViewManager.addNewValue(GraphViewManager.TYPE_Z_VALUE, z);

//Update min/max values
tvXMaxY.setText(""+round4Decimals
      (mGraphViewManager.getMaxValue(GraphViewManager.TYPE_X_VALUE)));
tvXMinY.setText(""+round4Decimals
      (mGraphViewManager.getMinValue(GraphViewManager.TYPE_X_VALUE)));
tvYMaxY.setText(""+round4Decimals
      (mGraphViewManager.getMaxValue(GraphViewManager.TYPE_Y_VALUE)));
tvYMinY.setText(""+round4Decimals
      (mGraphViewManager.getMinValue(GraphViewManager.TYPE_Y_VALUE)));
tvZMaxY.setText(""+round4Decimals
      (mGraphViewManager.getMaxValue(GraphViewManager.TYPE_Z_VALUE)));
tvZMinY.setText(""+round4Decimals
      (mGraphViewManager.getMinValue(GraphViewManager.TYPE_Z_VALUE)));

//Check the new readings against our thresholds to detect an accident
if(!mAccidentDialogShowing && checkThreshold(x, y, z, sensor1) && mBound){
      mAccidentDialogShowing = true;

      //Pause graphs
      ((ToggleButton)findViewById(R.id.btnPauseStart)).performClick();
      //Display accident dialog
      if(prefix.equals("accel"))
            showAccidentDialog("Accel");
      else
            showAccidentDialog("Gyro");
}
```

Note: The “prefix” variable refers to either the String “accel” or “gyro” according to what sensor the activity is interested in graphing.

First, the x, y, and z components of the sensor readings are taken from the Bundle and given to the GraphViewManager to graph on their respective graphs (denoted by a constant (ex: TYPE_X_VALUE)). Each of the minimum and maximum value TextViews are updated accordingly (with their values rounded to up to 4 decimal places). Finally, the sensor reading is checked against our predetermined thresholds and if an accident is detected the graphs are paused and the user is shown the accident dialog.

Not shown is the snippet of code that checks the sensor readings of the sensor not being graphed against the predetermined thresholds.

4.4.3. View Accident File

A technician also has the ability to view information about the last accident that was recorded. The SensorService keeps track of sensor readings that occurred in the last minute of operation, thus a technician can review sensor readings up to one minute before an accident was confirmed.

The SensorGraphActivity and GraphViewManager classes are also used for this purpose, though they operate differently from real-time sensors mode. In this case, the GraphViewManager is instructed to read sensor readings from the accident data file and graph them accordingly. Because of the amount of sensor readings that may exist in the accident file, I/O and graphing is done in the background while the technician is shown a “loading” dialog and asked to wait. The technician has the ability to switch between the accelerometer and gyroscope graphs and can also view the minimum and maximum values of each of the graphs being displayed.

4.4.4. GraphViewManager

The GraphViewManager class is responsible for setting up sensor graphs for the SensorGraphActivity. We use an external graphing library called AChartEngine for this functionality. The library can be found here: https://code.google.com/p/achartengine/

When in real-time graphing mode, the graphs are created and initialized with a horizontal line at y=0. To create a “real-time” effect, the graphs remove the oldest sensor reading when a new one comes in, thus causing the graph to appear to move towards the left and “disappear.” The following code performs this action:

```
public void addNewValue(byte type, float value){
      XYSeries current = mCurrentSeries.get(type);

      if(current.getItemCount() < maxPoints){
            current.add(current.getMaxX() + 1, value);
```

```
        }else{
                //"Cycle" graph to produce realtime effect
                current.remove(0);
                current.add(current.getMaxX() + 1, value);
        }

        mGraphViews.get(type).repaint();
}
```

When graphing from file, a thread is launched to read from the accident file and fill the graphs. When done graphing, the thread notifies the SensorGraphActivity to refresh its Views via its Handler. The following is the code for the thread:

```
private class DisplayReadingsThread extends Thread {
        public DisplayReadingsThread(){}

        @Override
        public void run(){
                mQueue = readFromFile();
                Log.d(TAG, "Number of readings in file: " + mQueue.size());

                SensorReading r = new SensorReading(0l, 0.0f, 0.0f, 0.0f,(byte)0);
                Iterator<SensorReading> iterator = mQueue.iterator();
                //For each sensor reading, place it in the appropriate graph
                while(iterator.hasNext()){
                        r = iterator.next();

                        if(r.type == mSensorType){
                                        Log.d(TAG, "YES");
                                        mCurrentSeries.get(0).add(TYPE_X_VALUE, r.x);
                                        mCurrentSeries.get(1).add(TYPE_Y_VALUE, r.y);
                                        mCurrentSeries.get(2).add(TYPE_Z_VALUE, r.z);
                        }
                }

                //Tell the activity that it can refresh the graphs now
                 mActivityHandler.sendMessage(mActivityHandler.obtainMessage
                        (SensorGraphsActivity.GRAPHS_READY));
        }
}
```

First, sensor readings are read from the accident file into a Queue. Then, every reading that matches the type of sensor being graphed is graphed. Finally, when all points are graphed, the SensorGraphsActivity is told, via a Handler message, to refresh the UI.

4.5. SensorService

The SensorService runs in the background of the smartphone application and will read the smartphone's accelerometer (measured in meters/second^2) and gyroscope (measured in radians/second) sensors for accident detection. In the case of the accelerometer, we actually use the Linear Accelerometer software sensor which automatically removes gravity and keeps the coordinate system relate to the phone, regardless of its orientation. Gyroscope readings are treated to use an absolute coordinate system rather than the position of the phone. The code used to calibrate the gyroscope is as follows:

```
//Calculate the Rotation Matrix according to the initial orientation of the
//smartphones
if(mAccelSampled && mMagnetSampled){
	Log.d(tag, "Getting R");
	mCalculatedR = SensorManager.getRotationMatrix(R, null, mAccelerationValues,
		mMagnetometerValues);
	if(mCalculatedR){
			Log.d(tag, "R Calculated");
			//Unregister unneeded magnetometer and accelerometer after the
			rotation matrix has been calculated
			mSensorManager.unregisterListener(this, mMagnetometer);
			mSensorManager.unregisterListener(this, mAccelerometer);
			Matrix.invertM(RInv, 0, R, 0);
			Matrix.multiplyMV(inputNormal, 0, RInv, 0, input, 0);
			return;
	}else{
			return;
	}
}
```

Essentially, the idea is to rotate the gyroscope vector to the world (absolute) coordinate system (see http://developer.android.com/reference/android/hardware/SensorEvent.html, about halfway down, for the axes description in the world coordinate system). To do this, we use Android's built-method to get the required rotation matrix which requires accelerometer (standard, non-linear) and magnetometer readings. We obtain accelerometer and magnetometer readings, calculate the rotation matrix, stop poling these sensors, and then we finally inverse the matrix. After this point, every time gyroscope readings are received, they are multiplied by this rotation matrix to get them into the world coordinate system.

The service saves readings into a linked list queue for a set amount of time (one minute), using the sensor reading's timestamp to define the amount of time passed. When the set time is passed, the oldest readings will be popped off the queue's stack in favor of new readings. This service also handles accident detection on the phone. When an accident is confirmed, all readings on the queue are dumped to file and the sensors are stopped.

Here are the main methods that execute in the service:

- `public int onStartCommand(Intent intent,int flags,int startId)`

The onStartCommand method is used to initialize the RandomAccessFile (the object we use to read and write to file). This method is also used to setup the sensors when the service is created.

- `public void onSensorChanged(SensorEvent event)`

The onSensorChanged method is called whenever new sensor readings are available. If the application is in Technician Mode, the sensors will read at GUI update speeds and bundles holding the values are sent to the SensorGraphsActivity. Here is some example code showing what happens when Technician Mode is on:

```
if (mTechnicianMode){             // used to only send bundles if Technician Mode on
(for graph)
       sensorvalues = new Bundle();
       sensorvalues.putBoolean(prefix+"Values",true);
       sensorvalues.putLong(prefix+"Time", event.timestamp);


if(source == mGyroscope){
       sensorvalues.putFloat(prefix+"X", inputNormal[X]); // inputNormal = Gyro
       sensorvalues.putFloat(prefix+"Y", inputNormal[Y]);
       sensorvalues.putFloat(prefix+"Z", inputNormal[Z]);
}else{
       sensorvalues.putFloat(prefix+"X", input[X]); //input = Accel
       sensorvalues.putFloat(prefix+"Y", input[Y]);
       sensorvalues.putFloat(prefix+"Z", input[Z]);
}

       send.setData(sensorvalues);
```

If the method is not in Technician Mode, the phone will query its sensors at a normal rate and will push the new sensor values onto the linked list queue:

```
if(!mTechnicianMode){//adds values to bundle and puts them in queue(linked list)
       temp = new SensorReading(event.timestamp, event.values[0], event.values[1],
event.values[2], sensorType);

       mQueue.add(temp);
       queueTimeCheck();
}
```

- `class ServiceHandler extends Handler`

The ServiceHandler class is used to setup the service's Handler, allowing the service to send and receive messages from other activities and respond accordingly. The following messages are handled: registering the sensor's reading speeds, recalibrating the sensors, and exiting the SensorService.

- `class  SensorReading`

The SensorReading class is used as a data structure to hold all pertinent information received from the sensors. These SensorReading objects are what are added to the Queue.

- `public void queueTimeCheck()`

The queueTimeCheck method is used to set the length of the Queue to a certain number of objects (sensor readings) based on time passed between the oldest sensor reading and the newest reading. Each time a new object is added to the queue, this method is called to see if the oldest element needs to be popped. The constant SECONDS_LIMIT is used to decide the time length of the queue.

- `private boolean checkThreshold(float x, float y, float z, byte sensor)`

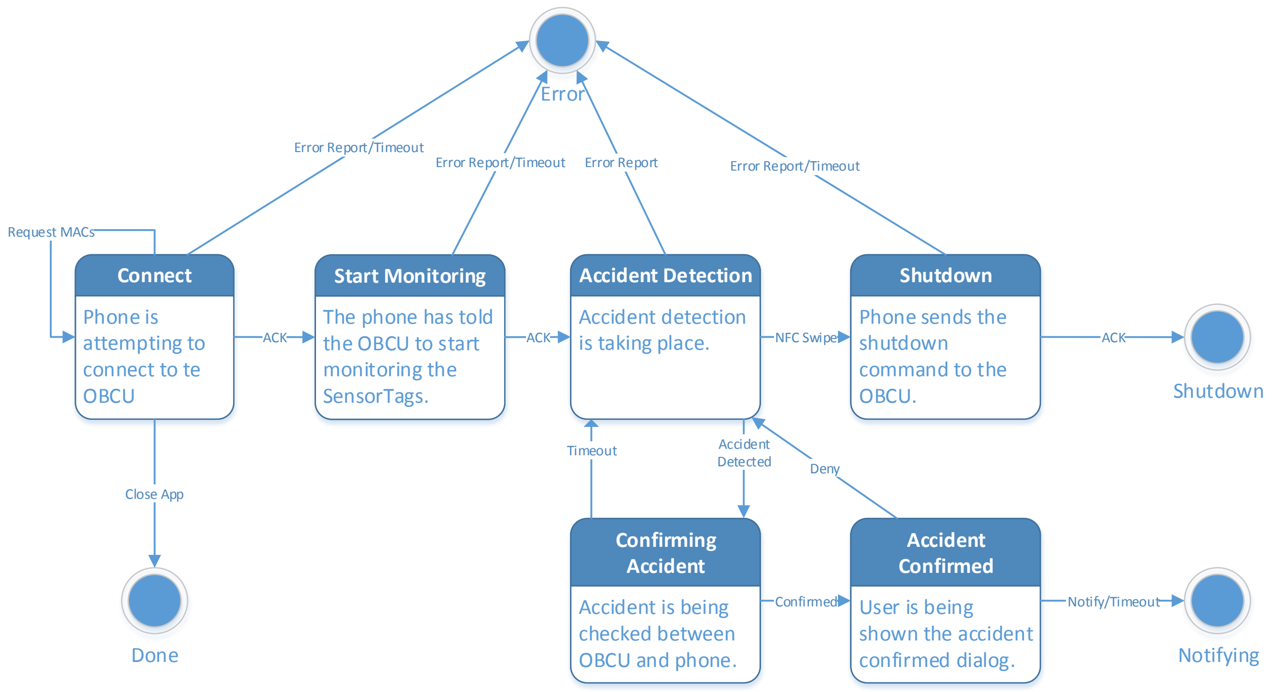
The checkThreshold method is used to determine if any of the sensor readings exceed the minimum or maximum threshold value that were predetermined to be the equivalent of an accident. If a value does go over the threshold then the method returns true, indicating that the phone has detected an accident. Here is an example of how the method operates for the accelerometer:

```
if(sensor == GraphViewManager.SENSOR_ACCELEROMETER)
            return  ((x*x) + (y*y) + (z*z)) > THRESHOLD_ACCEL_MAGNITUDE
            || (x < THRESHOLD_ACCEL_X_MIN) || (x > THRESHOLD_ACCEL_X_MAX)
            || (y < THRESHOLD_ACCEL_Y_MIN) || (y > THRESHOLD_ACCEL_Y_MAX)
            || (z < THRESHOLD_ACCEL_Z_MIN) || (z > THRESHOLD_ACCEL_Z_MAX);
```

4.6. BluetoothService

The BluetoothService class is responsible for maintaining a connection with the Bluetooth Server running on the OBCU. A single thread is used to connect to the OBCU and perform I/O according to a number of predefined commands. The NfcActivity's (home screen) Handler is registered with the BluetoothService so that it can be notified about connection information, confirmed accidents, or network errors.

Following is the state diagram for the BluetoothService:


Error
Error Report/Timeout
Error Report/Timeout
Error Report
Error Report/Timeout
Request MACs
Connect
Phone is attempting to connect to te OBCU
ACK
Start Monitoring
The phone has told the OBCU to start monitoring the SensorTags.
ACK
Accident Detection
Accident detection is taking place.
NFC Swipe
Shutdown
Phone sends the shutdown command to the OBCU.
ACK
Shutdown
Close App
Done
Timeout
Accident Detected
Deny
Confirming Accident
Accident is being checked between OBCU and phone.
Confirmed
Accident Confirmed
User is being shown the accident confirmed dialog.
Notify/Timeout
Notifying

While in the Connect state, the BluetoothService attempts to connect to the OBCU's Bluetooth Server. Upon an established connection, the OBCU requests MAC addresses to be sent from the smartphone and uses these MAC addresses to connect to and configure the SensorTags. Once all SensorTags have been configured successfully, the OBCU sends an ACK command to notify the Bluetooth Server that it is ready to begin monitoring.

In the Start Monitoring state, the BluetoothService sends the Start Monitoring command and waits for the OBCU to send an ACK back to signify that it will begin accident detection.

During the Accident Detection state, the BluetoothService waits for an accident to be detected by the SensorService. When a potential accident is detected by either the smartphone or the OBCU, the BluetoothService moves into the Confirming Accident state where it waits a small amount of time for both devices to deliver "accident detected" messages to the service. If the timeout is reached, an accident is not confirmed and the service moves back into the Accident Detection state. If both devices agree, the BluetoothService tells the NfcActivity to display the emergency response dialog to the user and the service moves into the Accident Confirmed State.

During the Accident Confirmed state, the user is dealing with the emergency response dialog. If they select the "I'm Alright" option, the system resumes accident detection. If the user selects the "Notify" option or the dialog timeout expires, the system is shut down while emergency response is notified.

When the user swipes an NFC tag to shut down the system, the BluetoothService is told to send the shutdown command to the OBCU. Upon receiving an ACK from the OBCU, the smartphone application is allowed to quit.

During any of these stages, errors may occur. Error reports may be sent from the OBCU following the ERR command. These reports are displayed to the user, but if it is not a fatal error, the system continues operation. The BluetoothService also uses timeout threads to detect possible connection drops between the smartphone and the OBCU.

The key method in the BluetoothService is the processState method. This method is called when a message is received from the OBCU and the appropriate action is taken by the BluetoothService depending on what state the service is currently in (filtered by case statements). For example, an ACK has one meaning while in the Connecting state (OBCU has configured all SensorTags) and another in the Shutdown state (OBCU is shutting down). The following is an example of the Connecting state:

```
switch(result){
      case ResultCodes.COMMAND_REQUEST_MACS:
            mTimeoutThread.interrupt();

            Log.d(TAG, "OBCU is requesting MACs...");

            ClientHandler.sendMessage(mClientHandler.obtainMessage
                  (ResultCodes.COMMAND_REQUEST_MACS));

            mTimeoutThread = new TimeoutThread(2 * TIMEOUT_PERIOD);
            mTimeoutRunning = true;
            mTimeoutThread.start();
            break;

      case ResultCodes.COMMAND_ERR:
            mTimeoutThread.interrupt();
            sendErrorToActivity(new String(buffer, 2, (byte)buffer[1]));
            break;

      case ResultCodes.COMMAND_ACK:
            mTimeoutThread.interrupt();

            Log.d(TAG, "OBCU has connected and configured all SensorTags.");
            mState = STATE_SYSTEM_STARTUP;
            mThread.writeMessage(
                  (char)ResultCodes.COMMAND_START_MONITORING + "");

            mTimeoutThread = new TimeoutThread(TIMEOUT_PERIOD);
            mTimeoutThread.start();
            break;
}
```

While in the Connecting state, receiving the REQUEST MACs command sends a message to the main activity to pass MAC addresses down to be sent to the OBCU. Receiving the ERR command causes the service to send the error report to the main activity to display to the user. Receiving an ACK allows the service to move to the System Startup state.

5. On-Board Control Unit Development

To compile the FrogStar .c files that are not included in the BlueZ package (Master.c, NetworkSetup.c, NetworkSetup.h, and BluetoothServer.c), run the following command:

```
# gcc –o BluetoothServer BluetoothServer.c NetworkSetup.c $(pkg-config --cflags --
libs glib-2.0 ) –l Bluetooth; gcc –o Master Master.c NetworkSetup.c $(pkg-config --
cflags --libs glib-2.0 ) –l Bluetooth
```

To compile the FrogStar file that is modified from BlueZ software (interactive.c), see Section 5.3.

5.1. Master Program

The Master program (Master.c) serves as the parent of all FrogStar programs on the OBCU. It is responsible for getting MAC addresses from the smartphone and forking to run the Bluetooth Server program and the Bluetooth LE Client Program.

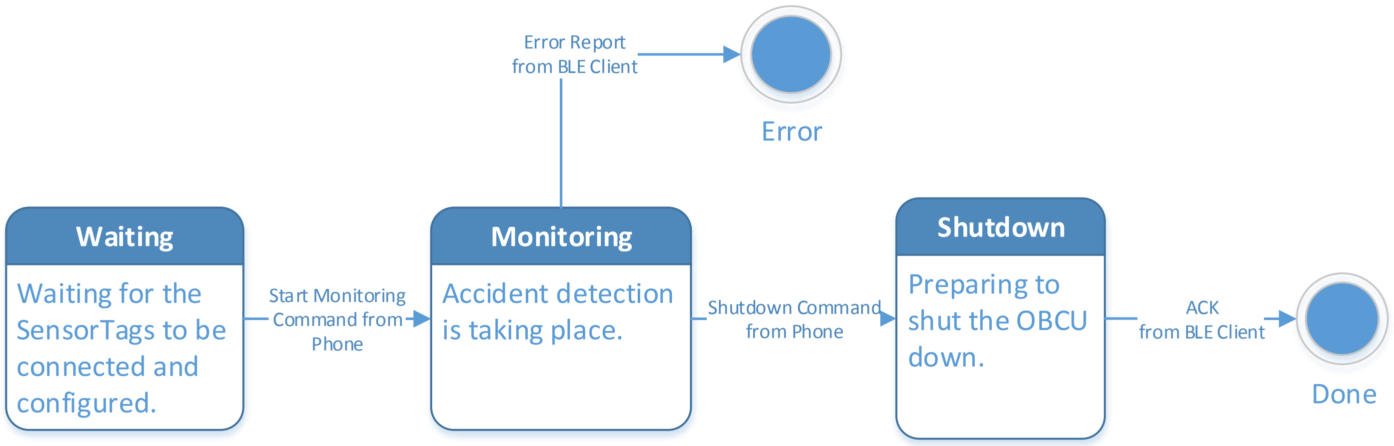
5.1.1. Network Setup

Code to setup an advertising Bluetooth server is in the NetworkSetup.c file. A Bluetooth server is created and advertises on a UUID known ahead of time (00001101-0000-1000-8000-00805F9B34FB is a default UUID for such services). The NetworkSetup class also includes functions to create non-blocking sockets so that execution will not block if there is nothing that has been sent from the smartphone.

5.2. Bluetooth Server

The Bluetooth Server program (BluetoothServer.c) has the sole purpose of managing communication with the smartphone. It reads commands that have been sent over from the phone and, if needed, forwards the commands to the Bluetooth LE Client program so that it may respond. Generally speaking, it acts as a middle-man between the Bluetooth LE Client program and the smartphone’s Bluetooth Server – example commands include accident confirmation, errors, and shutdown.

The following is state diagram for the Bluetooth Server:


Error Report
from BLE Client
Error
Waiting
Waiting for the SensorTags to be connected and configured.
Start Monitoring Command from Phone
Monitoring
Accident detection is taking place.
Shutdown Command from Phone
Shutdown
Preparing to shut the OBCU down.
ACK
from BLE Client
Done

During the Waiting state, the Bluetooth Server waits for an ACK from the Bluetooth LE Client signaling that all SensorTags have been configured. It then forwards this command to the BluetoothService on the smartphone. The smartphone next sends the Start Monitoring command to the BluetoothServer which moves the server into the Monitoring state.

In the Monitoring state, the Bluetooth Server waits for accident detected commands from the Bluetooth LE Client and forwards these commands to the BluetoothService.

Finally, upon receiving the shutdown command from the phone, the Bluetooth Server sends this command to the LE Client and waits for an ACK sent by the LE Client after it has disconnected from the SensorTags and is ready to shut down. The server sends this ACK to the BluetoothService so that the smartphone application can exit.

Errors can occur in any of these states (for example, a SensorTag being offline or lost connection). The Bluetooth Server is responsible for forwarding any error reports to the BluetoothService so that the user can be notified.

5.3. Bluetooth LE Client

The SensorTags are queried by a modified version of the gatttool program. Gatttool is a utility program provided in the BlueZ Bluetooth software stack for Linux. BlueZ is the standard package used for Bluetooth communication on Linux. BlueZ version 5.2 is used since this was the easiest to compile and get running on a Debian-based operating system. The BlueZ package includes a number of programs that are used to work with Bluetooth devices. Hcitool is one such program that is used to scan for LE devices and connect them to the Raspberry Pi.

To start development on reading from the SensorTags, look at the following sites: http://processors.wiki.ti.com/index.php/SensorTag_User_Guide, http://mike.saunby.net/2013/04/raspberry-pi-and-ti-cc2541-sensortag.html, http://stackoverflow.com/questions/17835469/using-bluetooth-low-energy-in-linux-command-line.

The first site is a reference of all handles and required data values/configurations to make the SensorTags sensors work. This is where you will go to find out what exactly needs to be sent to and queried from the SensorTags. The second is a good guide on how to setup the Raspberry Pi to communicate with SensorTags via a terminal. Read the comments and run through the command line gatttool example. The accepted answer on the third site has a lot of good links for Bluetooth on Linux and using BlueZ.

To implement SensorTag querying, modify the code of the BlueZ 5.2 package; start by downloading and extracting the package tar, run the config file, then finally run the make command to build the package. The gatttool program source resides in the attrib directory of the BlueZ package. gatttool.* are the files for the gatttool program itself. interactive.* are used by gatttool in interactive mode (command line example). Modify the actual contents of the attrib directory files in order to make compilation easy. Simply modify the source in the package, and run the make command in the parent directory of the package to build it. A my_compile script is provided and should be present in the package's directory. This script should be run to compile the gatttool program with the math library linked in. The modified gatttool program is "execed" from the Master program to query the SensorTags and perform all necessary calculations. To make this easy, the Master program is sourced, compiled, and run from the attrib directory.

The main modified file is interactive.c. The program runs in a glib main event loop. Functions are called in the event loop and callbacks are called when the request finishes. Rather than developing a mechanism to emulate this procedure, keep the current structure of the program and modify what is necessary. A child process is driving calls to the parent process, and waits for an acknowledgment from the parent signaling that the parent is ready for the next command. The order of events is system startup, connecting to all the SensorTags, configuring the SensorTags, and finally querying the sensors. Of note are the cmd_read_hnd and char_read_cb functions. These functions are the main functions used for querying sensor values. When the child process drives the parent to read a value, cmd_read_hnd is called. The appropriate call is made with the right handle (for example, to retrieve accelerometer values), then when values return from the SensorTag, char_read_cb is called and passed the result. In the char_read_cb function we have logic to move to the next sensor and/or SensorTag and make calls to our accident detection logic. The SensorTags are queried in a round robin fashion. Specifically, the line in the cmd_read_hnd function:

```
gatt_read_char(sensorTags[tagIndex].attrib, handle, char_read_cb,
sensorTags[tagIndex].attrib);
```

is where a SensorTag is queried by the gatttool package and the callback function (char_read_cb) is denoted as the function to be called when a result is ready. Changing these functions should give the most control in how the SensorTags are queried and how the results are parsed and/or used.

6. Glossary of Terms

Accelerometer – A device that measures acceleration, or the rate at which speed changes.

Android – An open-source operating system developed for mobile devices by Google.

Bluetooth – A short-ranged, peer-to-peer, wireless communication protocol. Bluetooth LE refers to a low-energy Bluetooth standard.

GATT – General Attribute, protocol used by Bluetooth LE communication.

Gyroscope – A device that measures orientation in terms of yaw, roll, and pitch.

MAC – Media Access Control address - A hardcoded physical address for a networking interface.

NFC – Near-field communication – A set of standards that allow devices to communicate in very close proximity.

TI CC2541 – A Bluetooth-capable SensorTag offered by Texas Instruments that houses various sensors including an accelerometer and a gyroscope.