Detailed Design Specification

BehindtheCurtain Enterprises

Project AVALANCHE

Team Members:
Kyle Burgess
Kyle Crumpton
Austen Herbst
Bilal Nawaz
Jason Sprowl

Last Updated: 2/28/2013 5:00 pm
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sDocument Revision History</td>
<td>.......................................................................................................................6</td>
</tr>
<tr>
<td>List of Figures</td>
<td>............................................................................................................................7</td>
</tr>
<tr>
<td>List of Tables</td>
<td>............................................................................................................................8</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>..........................................................................................................................9</td>
</tr>
<tr>
<td>1.1 Overview</td>
<td>..........................................................................................................................9</td>
</tr>
<tr>
<td>1.2 Product Overview</td>
<td>.........................................................................................................................9</td>
</tr>
<tr>
<td>1.3 Product Scope</td>
<td>........................................................................................................................9</td>
</tr>
<tr>
<td>2. Architecture Overview</td>
<td>.....................................................................................................................12</td>
</tr>
<tr>
<td>2.1 Layer Descriptions</td>
<td>.....................................................................................................................12</td>
</tr>
<tr>
<td>2.2 Subsystems Overview</td>
<td>....................................................................................................................13</td>
</tr>
<tr>
<td>2.3 Module Descriptions</td>
<td>....................................................................................................................15</td>
</tr>
<tr>
<td>2.3 Data Flow Overview</td>
<td>....................................................................................................................20</td>
</tr>
<tr>
<td>3. GUI Layer</td>
<td>..........................................................................................................................24</td>
</tr>
<tr>
<td>3.1 Description</td>
<td>..........................................................................................................................25</td>
</tr>
<tr>
<td>3.2 User Input: Drag &amp; Drop Controller</td>
<td>.........................................................................................25</td>
</tr>
<tr>
<td>3.3 User Input: Pop-Out Menu Controller</td>
<td>.....................................................................................27</td>
</tr>
<tr>
<td>3.4 User Input: Page Slider Controller</td>
<td>.........................................................................................29</td>
</tr>
<tr>
<td>3.5 User Input: Preferences Controller</td>
<td>.........................................................................................29</td>
</tr>
<tr>
<td>3.6 Data Output: Post Process Controller</td>
<td>.........................................................................................30</td>
</tr>
<tr>
<td>3.7 Data Output: Real-Time Controller</td>
<td>.........................................................................................31</td>
</tr>
<tr>
<td>3.8 Data Output: Configuration Controller</td>
<td>.........................................................................................32</td>
</tr>
<tr>
<td>3.9 Data Output: Help View</td>
<td>.................................................................................................................33</td>
</tr>
<tr>
<td>3.10 Configure: Configuration Model</td>
<td>.................................................................................................34</td>
</tr>
<tr>
<td>3.11 Configure: Configuration Model Map</td>
<td>.............................................................................................35</td>
</tr>
<tr>
<td>3.12 Configure: User Account</td>
<td>..................................................................................................................36</td>
</tr>
<tr>
<td>4. Data Processing Layer</td>
<td>..................................................................................................................37</td>
</tr>
<tr>
<td>4.1 Description</td>
<td>......................................................................................................................38</td>
</tr>
</tbody>
</table>
4.2 Real Time Processing: RealTimeBuilder....................................................................................38
4.3 Real Time Processing: SensorSnapshotModel .................................................................................43
4.4 Real Time Processing Subsystem: SensorAggregateModel ..............................................................45
4.5 Real Time Processing Subsystem: GaugeModel ................................................................................47
4.6 Post Processing: RunListModel ....................................................................................................50
4.7 Post Processing: SensorAnalysisModel ..........................................................................................50
4.8 Post Processing: GaugeAnalysisModel ..........................................................................................51
4.9 Post Processing Subsystem: PostBuilder ........................................................................................52
4.10 Sync Subsystem: BLEAlarmService ..............................................................................................54
4.11 Save Data: GaugeSerializer ........................................................................................................55

5. Bluetooth Layer ..................................................................................................................................57
5.1 Description .....................................................................................................................................57
5.2 Sync: Wait for Sync .......................................................................................................................58
5.3 Sync: Notify MCU ..........................................................................................................................58
5.4 Transfer Data: Receive from MCU ..................................................................................................59
5.5 Transfer Data: Send Data ...............................................................................................................59

6. Data Acquisition Layer .....................................................................................................................61
6.1 Description .....................................................................................................................................61
6.2 Encode/Package Data: Set Sync Status ..........................................................................................62
6.3 Encode/Package Data: Package Data .............................................................................................62
6.4 Encode/Package Data: Send Data ..................................................................................................63
6.5 Read CAN Bus: Get Raw Data .........................................................................................................64

7. Local Storage .....................................................................................................................................67
7.1 Description .....................................................................................................................................68
7.2 Device Memory: Run Storage .........................................................................................................68
7.3 Device Memory: User Account Storage ..........................................................................................70

8. Cloud Storage .....................................................................................................................................72
## Document Revision History

<table>
<thead>
<tr>
<th>Revision Number</th>
<th>Revision Date</th>
<th>Description</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>2/21</td>
<td>Initial Draft</td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>2/24</td>
<td>DDS Review Draft</td>
<td>Revised according to peer review input.</td>
</tr>
<tr>
<td>2.0</td>
<td>2/28</td>
<td>DDS Baseline</td>
<td>Revised according to review input.</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure #</th>
<th>Title</th>
<th>Page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Product Scope Overview</td>
<td>11</td>
</tr>
<tr>
<td>2-1</td>
<td>Module Decomposition</td>
<td>12</td>
</tr>
<tr>
<td>3-1</td>
<td>GUI Layer Module Diagram</td>
<td>24</td>
</tr>
<tr>
<td>4-1</td>
<td>Data Processing Layer Module Diagram</td>
<td>37</td>
</tr>
<tr>
<td>5-1</td>
<td>Bluetooth Layer Module Diagram</td>
<td>57</td>
</tr>
<tr>
<td>6-1</td>
<td>Data Acquisition Layer Module Diagram</td>
<td>61</td>
</tr>
<tr>
<td>7-1</td>
<td>Local Storage Layer Module Diagrams</td>
<td>67</td>
</tr>
<tr>
<td>8-1</td>
<td>Cloud Storage Layer Module Diagram</td>
<td>72</td>
</tr>
<tr>
<td>9-1</td>
<td>Network Layer Module Diagram</td>
<td>77</td>
</tr>
<tr>
<td>13-1</td>
<td>MC9S12C128 Microcontroller</td>
<td>100</td>
</tr>
<tr>
<td>13-2</td>
<td>SN74LVC4245 Voltage Leveler</td>
<td>101</td>
</tr>
<tr>
<td>13-3</td>
<td>BR-LE4.0-S2A Bluetooth Module</td>
<td>102</td>
</tr>
<tr>
<td>13-4</td>
<td>Schematics Overview</td>
<td>103</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table #</th>
<th>Title</th>
<th>Page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>System Data Flows</td>
<td>23</td>
</tr>
<tr>
<td>3-1</td>
<td>GUI Layer Data Flows</td>
<td>25</td>
</tr>
<tr>
<td>4-1</td>
<td>Data Processing Layer Data Flows</td>
<td>38</td>
</tr>
<tr>
<td>5-1</td>
<td>Bluetooth Layer Data Flows</td>
<td>57</td>
</tr>
<tr>
<td>6-1</td>
<td>Data Acquisition Layer Data Flows</td>
<td>61</td>
</tr>
<tr>
<td>7-1</td>
<td>Local Storage Layer Data Flows</td>
<td>67</td>
</tr>
<tr>
<td>8-1</td>
<td>Cloud Storage Layer Data Flows</td>
<td>72</td>
</tr>
<tr>
<td>9-1</td>
<td>Network Layer Data Flows</td>
<td>78</td>
</tr>
<tr>
<td>10-1</td>
<td>Test Cases</td>
<td>92</td>
</tr>
<tr>
<td>11-1</td>
<td>Layer Requirements Traceability</td>
<td>94</td>
</tr>
<tr>
<td>11-2</td>
<td>Module Requirements Traceability</td>
<td>96</td>
</tr>
<tr>
<td>12-1</td>
<td>Acceptance Criteria</td>
<td>99</td>
</tr>
</tbody>
</table>
1. Introduction

1.1 Overview

This Detailed Design Specification will provide a low level design for Project AVALANCHE based on the Architecture laid out in the Architectural Design Specification. Project AVALANCHE was divided into seven layers and multiple subsystems per layer in the Architectural Design Specification; this Detailed Design will further define how the subsystems and their sub modules interact and function with one another. The architecture will be taken a level lower in this document detailed in both design schematics covering inputs and outputs as well as pseudo-code where appropriate. This document will include the quality assurance and acceptance plan for Project AVALANCHE.

1.2 Product Overview

The AVALANCHE is an add-on module for a pre-existing gauge by CPC/Redline Gauges, a gauge that provides a digital display for all sensors of a racing vehicle. The existing CPC/Redline Gauge is targeted towards snowmobiles, but is useable with any racing vehicle so long as the vehicle has the required sensors installed. AVALANCHE will add mobile app functionality to provide racers and maintenance specialists a tool for data logging, vehicle diagnosis, and real time stats feedback, for all existing sensors. The goal is to benefit racing enthusiasts by providing data analysis not offered with previous gauges.

1.3 Product Scope

Team BehindtheCurtain will produce a prototype of Project AVALANCHE consisting of the following components:

1.3.1 Hardware Module:

The hardware module is an add-on component for the existing Redline Gauge that will interface with the existing CAN bus. The module will consist of the following:

CAN bus – The CAN bus is responsible for providing communication from the existing gauge to the hardware module.
Microcontroller – The microcontroller is responsible for providing data acquisition and data packaging for Bluetooth transmission.

Bluetooth chip – The Bluetooth chip provides Bluetooth networking between the hardware module and mobile devices.

Hardware Enclosure – The hardware enclosure provides protection from heat and fluids getting into the microcontroller or Bluetooth chip.

1.3.2 Mobile Devices:

This consists of any iOS and/or Android device that have the AVALANCHE mobile application, which will be developed for both the iOS and Android operating systems, installed. The mobile device will communicate wirelessly through Bluetooth to the hardware module.

1.3.3 Server:

The server will provide storage for user accounts and saved runs for the user. Server storage of runs will be handled in real time if 3G networking is available.

1.3.4 Mobile Applications:

The mobile applications will provide data processing and a user interface for the user.

1.3.5 Server Software:

The server software will handle controlling the server hardware and provide account security and real-time access.

1.3.6 Microcontroller Software:

The Microcontroller will provide data acquisition and Bluetooth networking functionality.
Figure 1.1 Product Scope Overview
2. Architecture Overview

2.1 Layer Descriptions

2.1.1 GUI Layer

Figure 2-1 Module Decomposition
The GUI Layer acts as the intermediary between users and Project AVALANCHE. It displays the data from the physical gauge in a graphical form, providing both sensor readouts and charts and metrics for past runs, as well as parsing user input into the system.

2.1.2 Data Processing Layer

The Data Processing Layer processes raw data received from the physical gauge via the Bluetooth Layer and uses user inputted configuration data to format the device input into an user readable form which is outputted to the user through the GUI Layer. It will also send this data to be saved in the Storage Layer, either locally or on the server via the Network Layer. These saved runs will be able to analyzed by the Data Processing Layer to create charts, graphs, and metrics for run analysis by the user.

2.1.3 Bluetooth Layer

The Bluetooth Layer will handle the Bluetooth pairing between the hardware module and the mobile device and will be responsible for transferring gauge sensor data from the Data Acquisition Layer to the Data Processing Layer over Bluetooth.

2.1.4 Data Acquisition Layer

The Data Acquisition Layer encompasses the hardware module and will be responsible for pulling gauge data off the CAN Bus and packaging that data for transfer over the Bluetooth Layer.

2.1.5 Local Storage Layer

The Local Storage Layer will be responsible for storing saved runs on the mobile device for use in later analysis. The Local Storage will be accessed by the Data Processing Layer directly through the mobile device file system.

2.1.6 Cloud Storage Layer

The Cloud Storage Layer will be responsible for storing usernames, passwords, old runs and account settings. This layer will be accessed indirectly by the Data Processing Layer using the Network Layer as a medium.

2.1.7 Network Layer

The Network Layer is responsible for secure and reliable data transfer between the Data Processing and Storage Layer. It will use AES and RSA encryption to make sure that all communication between the layers is secure and private and TCP to ensure that the data is transferred successfully between the layers.

2.2 Subsystems Overview

2.1.1 GUI Layer Subsystems
2.1.1  Input Subsystem

Allows user to configure what data is displayed on the screen and modify AVALANCHE system settings.

2.1.1.2  Output Subsystem

Display’s a graphical representation of gauge and graph data based on configurations from the Input subsystem.

2.1.1.3  Configure Subsystem

Takes configuration data about user’s physical gauge.

2.1.2  Data Processing Layer Subsystems

2.1.2.1  Real Time Processing Subsystem

The Real Time Processing subsystem will handle real time data pulled from the physical gauge that is transferred over the Bluetooth Layer. It will use the configuration data inputted in the GUI layer to decode this data in a user readable form and then transfer the data for output in the GUI layer and the Save Data Subsystem for storage.

2.1.2.2  Save Data Subsystem

The Save Data Subsystem will receive run data from the Real Time Processing Subsystem and then save that data locally in the Storage Layer.

2.1.2.3  Post Processing Subsystem

The Post Processing Subsystem will reanalyze saved runs to create graphs, tables, and metrics for tuning purposes. The subsystem will pull the requested run to analyze from the Storage Layer, either saved locally or accessed from the server through the Network Layer, based on user configurations in the GUI Layer. Once the run is accessed the subsystem will process the data to create the requested graphs, tables, and metrics.

2.1.2.4  Sync Subsystem

The Sync subsystem will sync the mobile device with the Bluetooth Layer via Bluetooth.

2.1.3  Bluetooth Layer Subsystems

2.1.3.1  Transfer Data Subsystem

The purpose of this subsystem is to receive data from the microcontroller and send it to the mobile device

2.1.3.2  Sync Subsystem
The purpose of this sub system is to establish a connection between the hardware module and the mobile device.

2.1.4 Data Acquisition Layer Subsystems

2.1.4.1 Real CAN BUS Subsystem

The purpose of this sub system is to read the raw data off of the CAN Bus

2.1.4.2 Encode/Package Data Subsystem

The purpose of this sub system is to package end encode the raw data for wireless data transfer

2.1.5 Network Layer Subsystems

2.1.5.1 Encryption Subsystem

The Encryption Subsystem encrypts username and password data for secure transfer between the data processing and storage layers. Only username and password data is encrypted sensor data is passed through without change.

2.1.5.2 Transfer Subsystem

The transfer subsystem interfaces with both the mobile application and the server to send and receive packets based on whether a read or write is being performed

2.1.5.3 Decryption Subsystem

The decryption subsystem is responsible for decrypting packets which were encrypted and sent to the storage layer through the network layer. If the data being handled is unencrypted sensor data then the data is passed through without change.

2.1.5 Local Storage Layer Subsystems

2.1.5.1 Device Memory Subsystem

The Device Memory Subsystem is used to store saved runs locally on the mobile device.

2.1.5 Cloud Storage Layer Subsystems

2.1.5.1 Database Subsystem

The Database Subsystem is used to store saved runs in the cloud.

2.3 Module Descriptions

2.3.1 Input: Drag and Drop Controller
Handles user input for the layout setup by allowing dragging and dropping of gauge and chart widgets from the widget tray. This allows a user to add and remove them from the view.

2.3.2 Input: Pop-out Menu Controller

Handles user input for the system menu. This menu can be activated from any view by using a two finger gesture and swiping from left or right. A left swipe will activate a menu that allows swapping to: the real time grid view, post process grid view, configuration view, and help view. A right swipe will activate the widget tray that gives the user the ability to drag and drop widgets onto the grid. This widget tray will only be accessible from the real time grid view and post process grid view.

2.3.3 Input: Page Slider Controller

Handles user input and allows the user to switch between the active grid views by using a single finger gesture and swiping left or right. The pages slider is accessible while in a grid view, either real time grid view or post process grid view.

2.3.4 Input: Preferences Controller

Handles user input and allows the user to modify configuration settings relating to the connected sensors. A user can add/remove sensors, enable/disable sensors, and adjust the calibration of each sensor.

2.3.5 Input: Post Process Controller

The Post Process Controller Module will update the gauge display with previously recorded and saved sensor data.

2.3.6 Output: Real-Time Controller

The Real-Time Controller Module will update the gauge display in real-time with active sensor data.

2.3.7 Output: Configuration Controller

The Real-Time Controller Module will update the gauge display in real-time with active sensor data.

2.3.8 Output: Help View

The Help View Module will be an immutable page displaying general information and FAQ for app users.

2.3.9 Configure: Configuration Model

Each Configuration Model Module will represent the configuration of one sensor.

2.3.10 Configure: Configuration Model Map
The Configuration Model Map Module will represent the hash map data structure storing a sensor ID as the key and a respective Configuration Model as the value.

2.3.11 Configure: User Account

The Configure User Account Module will be represented by a page for the user to enter account information (User name and password) and log into the system.

2.3.12 Real Time Processing: Real Time Builder

This module is a class that will handle building the Real Time Subsystem models; SensorSnapshotModel, SensorAggregateModel, and GaugeModel; based on the input received from the Sync Subsystem and the configurations from the Configure Subsystem in the GUI layer, received in a ConfigurationModelMap.

2.3.13 Real Time Processing: Sensor Snapshot Model

This module contains the data for a single sensor for one moment of time; as well as the sensor name, id, sensor type, and timestamp needed to identify the type of sensor the snapshot represents and help display and save the information.

2.3.14 Real Time Processing: Sensor Aggregate Model

The SensorAggregateModel class will contain an array of SensorSnapshotModels representing a run for one sensor. The class’s instance variables will also contain sensor properties including the sensor name, sensor type, sensor id, and a boolean value representing whether the sensor is being actively processed and recorded.

2.3.15 Real Time Processing Subsystem: Gauge Model

The GaugeModel class is a singleton class, where only one instance can exist at a time, which can be accessed by the RealTimeView for output of all incoming real time data to the user through the GUI. The current instance is created at the beginning of the run recording and each sensor is given a SensorAggregateModel which is stored in the sensorAggregateModelMap. This map can then be accessed by the RealTimeView to update the sensors displayed in the GUI.

2.3.16 Post Processing: Run List Model

This module will makes a list of all previous runs in device memory and the database.

2.3.17 Post Processing: Sensor Analysis Model

This module will create a line graph based on sensor data.

2.3.18 Post Processing: Gauge Analysis Model

This module is a singleton object meaning only one can be instantiated at any given time. This module holds an array of Sensor Analysis Models (or SensorAnalysisModel object).
2.3.19 Post Processing: Post Builder

PostBuilder is a builder class that uses the configuration data from a ConfigurationModelMap to load a run from the Local Storage Layer and build the corresponding SensorAnalysisModels and GaugeAnalysisModel.

2.3.20 Sync Subsystem: BLE Alarm Service

The BLEAlarmService module is a class will be responsible managing Bluetooth connections to the Bluetooth Layer and handling the passing of data from Bluetooth Layer to the Data Processing Layer.

2.3.21 Save Data: Gauge Serializer

The GaugeSerializer module is a class that will handle serializing the Real Time Processing models into files in local storage.

2.3.22 Device Memory: Run Storage

The Run Storage has the principal functionality of acting as the data storage location (temporary) for the whole system. The Run Storage of Local Storage Layer mainly stores the current race statistics and gauge data for real-time processing and presentation onto the mobile applications. The Run Storage of Local Storage Layer will not run on the cloud server. It will store present runs to be accessed by the user or anyone the user gives access to. One of the main responsibilities of this particular module is Data storage. The main purpose of the Run Storage is not just temporary accumulation of the current race statistics and gauge data for real time processing but also acting as a route for data that is to be stored into the Cloud Storage since Cloud Storage does not interact with Data Processing Layer directly but through the Network and Local Storage Layers. Some of the assumptions made for this part of the system will be that data provided by the Decryption subsystem of the Network Layer is of the right type and format so the data can be easily stored as well as that there is enough memory for the storage of all the data that is to be stored in the database.

2.3.23 Device Memory: User Account Storage

The User Account Storage has the principal functionality of acting as the data storage location for the user account information of the whole system. The User Account Storage does not store the past race statistics and gauge data. The User Account Storage will be part of the local device server. The authorized user will be allowed to access current runs from the Device Memory. Some of the assumptions made for this part of the system will be that user passwords provided by the Data Processing Layer are properly hashed and of the right format so the data can be easily stored.

2.3.24 Database: Run Storage

The Run Storage has the principal functionality of acting as the data storage location (long-term) for the whole system. The Run Storage of Cloud Storage Layer mainly stores the past race statistics and gauge data for later analysis and presentation onto the mobile applications. The Run Storage will run on the cloud server. It will store saved runs to be accessed by the user or anyone the user gives access to.
of the main responsibilities of this particular module is Data storage. Some of the assumptions made for
this part of the system will be that data provided by the Decryption subsystem of the Network Layer is
of the right type and format so the data can be easily stored as well as that there is enough memory for
the storage of all the data that is to be stored in the database.

2.3.25 Database: User Account Storage

The User Account Storage has the principal functionality of acting as the data storage location for the
user account information of the whole system. The User Account Storage does not store the past race
statistics and gauge data. The User Account Storage will be part of the cloud server. The authorized user
will be allowed to access only saved runs from the Cloud Storage but the User Accounts will only be
accessed by the system to verify the information and authenticate the user. Some of the assumptions
made for this part of the system will be that user passwords provided by the Decryption subsystem of the
Network Layer is are properly hashed and of the right format so the data can be easily stored.

2.3.26 Encrypt: User Account

This module will take two input objects: user account and gauge information. The user account will be
encrypted using AES128 encryption algorithm, and packaged with the gauge information (not
encrypted).

2.3.27 Encrypt: Gauge Data

This module will combine the non-encrypted gauge data with encrypted user account data to be sent for
packaging in the Package User Data Module.

2.3.28 Transfer Data: Package User Data

This module will package the object consisting of an encrypted user account name and password along
with the non-encrypted gauge information to send to either local storage or the cloud server.

2.3.29 Transfer Data: Send User Data

This module will take previously packaged input and use TCP client server architecture to send data
from mobile device to cloud server and vice versa.

2.3.30 Decrypt: Gauge Data

This module will take packaged data sent from the Send Data Module and separate the encrypted user
account data from the non-encrypted gauge data which will be processed and sent to the cloud storage
layer.

2.3.31 Decrypt: User Account

This module will take packaged data sent from the Send Data Module to decrypt and store on the server.

2.3.32 Sync: Wait for Sync
The Wait for Sync is a functionality that already exists on the BR-LE4.0-S2A chip. Upon chip power on, it will go into a wait mode until a device pairs with it.

2.3.33 Sync: Notify MCU

The Notify MCU will be responsible for sending a pin high output to both an LED and the MC9S12C128 Microcontroller to display it’s synced status.

2.3.34 Transfer Data: Receive from MCU

The MC9S12C128 MCU will use SCI UART to transmit the data stream 1 byte at a time to the BR-LE4.0-S2A Bluetooth Module for transfer.

2.3.35 Transfer Data: Send Data

The BR-LE4.0-S2A module will send all packed data over Bluetooth.

2.3.36 Encode/Package Data: Set Sync Status

Set Sync Status will change a flag in the Data Acquisition Layer that is checked before transmitting data, because we cannot send SCI data to Bluetooth Layer if the device is not synced.

2.3.37 Encode/Package Data: Package Data

Since we are just pulling raw data off of the CAN bus line, we need to package it into a standard form so the Data Processing Layer knows what it is reading.

2.3.38 Encode/Package Data: Send Data

Send Data will be responsible for sending the packaged byte stream from the MC9S12C122 MCU to the BR-LE4.0-S2A.

2.3.39 Read CAN Bus: Get Raw Data

Get Raw Data will be responsible for reading the sensor data off of the CAN Bus line.

2.3 Data Flow Overview

<table>
<thead>
<tr>
<th>Data Flow</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>Data is sent via SCI to BR-LE-4.0S2A byte by byte in a packaged order.</td>
</tr>
<tr>
<td>A2</td>
<td>syncStatus flag is checked before data is packaged and transmitted.</td>
</tr>
<tr>
<td>A3</td>
<td>Algorithm takes sensor data and decomposes it into two bytes for transmission.</td>
</tr>
<tr>
<td>A4</td>
<td>Raw Data is taken and stored into arrays representing its sensor.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td><strong>A5</strong></td>
<td>Raw data is read off of the CAN Bus line.</td>
</tr>
<tr>
<td><strong>B1</strong></td>
<td>Data is transmitted byte by byte in a predetermined order</td>
</tr>
<tr>
<td><strong>B2</strong></td>
<td>Bluetooth Protocol Information</td>
</tr>
<tr>
<td><strong>B3</strong></td>
<td>Data from the Data Acquisition Layer is received and sent to be transmitted</td>
</tr>
<tr>
<td><strong>B4</strong></td>
<td>BR-LE4.0-S2A pin goes high to indicate sync status</td>
</tr>
<tr>
<td><strong>B5</strong></td>
<td>BR-LE4.0-S2A high pin triggers interrupt in MCU</td>
</tr>
<tr>
<td><strong>P1</strong></td>
<td>Create GaugeModel instance based off of data from Sync Subsystem.</td>
</tr>
<tr>
<td><strong>P2</strong></td>
<td>Create new SensorAggregateModel to based off of data from Sync Subsystem.</td>
</tr>
<tr>
<td><strong>P3</strong></td>
<td>Create new SensorSnapshotModel to based off of data from Sync Subsystem.</td>
</tr>
<tr>
<td><strong>P4</strong></td>
<td>Byte array containing data of a single instance in time.</td>
</tr>
<tr>
<td><strong>P5</strong></td>
<td>SensorAggregateModels are contained in GaugeModel instance.</td>
</tr>
<tr>
<td><strong>P6</strong></td>
<td>SensorSnapshotModels are contained in a SensorAggregateModel.</td>
</tr>
<tr>
<td><strong>P7</strong></td>
<td>File descriptor of run to analyze.</td>
</tr>
<tr>
<td><strong>P8</strong></td>
<td>File descriptor of sensor to analyze.</td>
</tr>
<tr>
<td><strong>P9</strong></td>
<td>SensorAnalysisModels are contained GaugeAnalysisModel.</td>
</tr>
<tr>
<td><strong>P10</strong></td>
<td>GaugeModel instance to be serialized.</td>
</tr>
<tr>
<td><strong>P11</strong></td>
<td>Folder containing saved run.</td>
</tr>
<tr>
<td><strong>P12</strong></td>
<td>An array containing a list of saved runs.</td>
</tr>
<tr>
<td><strong>P13</strong></td>
<td>GaugeModel instance containing real time sensor data.</td>
</tr>
<tr>
<td><strong>P14</strong></td>
<td>GaugeAnalysisModel instance containing post processed run data.</td>
</tr>
<tr>
<td><strong>P15</strong></td>
<td>User Account Model containing user information.</td>
</tr>
<tr>
<td><strong>G1</strong></td>
<td>User input customizing what sensors are displayed.</td>
</tr>
<tr>
<td><strong>G2</strong></td>
<td>User input customizing which page is displayed.</td>
</tr>
<tr>
<td><strong>G3</strong></td>
<td>User input regarding account info and sensor configurations.</td>
</tr>
<tr>
<td><strong>G4</strong></td>
<td>User input regarding what menu option was chosen.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>G5</td>
<td>Page id and sensors to display.</td>
</tr>
<tr>
<td>G6</td>
<td>Page id and sensors to display.</td>
</tr>
<tr>
<td>G7</td>
<td>Sensor configurations.</td>
</tr>
<tr>
<td>G8</td>
<td>Help page to display.</td>
</tr>
<tr>
<td>G9</td>
<td>Single sensor configuration.</td>
</tr>
<tr>
<td>G10</td>
<td>What type of processing.</td>
</tr>
<tr>
<td>G11</td>
<td>User account information.</td>
</tr>
<tr>
<td>G12</td>
<td>Instances of Configuration Model are stored in Configuration Model Map.</td>
</tr>
<tr>
<td>G13</td>
<td>Configuration Model Map for real time processing.</td>
</tr>
<tr>
<td>G14</td>
<td>Configuration Model Map for post processing.</td>
</tr>
<tr>
<td>G15</td>
<td>Current configuration to display.</td>
</tr>
<tr>
<td>G16</td>
<td>An instance of User Accounts Model is stored in the configuration model map.</td>
</tr>
<tr>
<td>G17</td>
<td>Real time gauge data output to user.</td>
</tr>
<tr>
<td>G18</td>
<td>Analyzed run data output to user.</td>
</tr>
<tr>
<td>G19</td>
<td>Configuration data output to user.</td>
</tr>
<tr>
<td>G20</td>
<td>Help page output to user.</td>
</tr>
<tr>
<td>C1</td>
<td>Run data from cloud server (Object)</td>
</tr>
<tr>
<td>C2</td>
<td>User account data from cloud server (Object)</td>
</tr>
<tr>
<td>L1</td>
<td>A saved run to analyze.</td>
</tr>
<tr>
<td>L2</td>
<td>An array containing a list of saved runs.</td>
</tr>
<tr>
<td>L3</td>
<td>User account data for encryption (Object)</td>
</tr>
<tr>
<td>L4</td>
<td>Run data for encryption (Object)</td>
</tr>
<tr>
<td>N1</td>
<td>EncryptedDataPackage (CipherText and Gauge object)</td>
</tr>
<tr>
<td>N2</td>
<td>Byte stream of EncryptedDataPackage</td>
</tr>
<tr>
<td>N3</td>
<td>Decrypted byte stream to be sent for post-processing</td>
</tr>
<tr>
<td>N4</td>
<td>Decrypted byte stream to be stored on cloud server</td>
</tr>
<tr>
<td>------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>N5</td>
<td>Sent gauge information</td>
</tr>
<tr>
<td>N6</td>
<td>Byte stream of Decrypted User Account Data</td>
</tr>
<tr>
<td>N7</td>
<td>Byte stream of Decrypted Run Data</td>
</tr>
<tr>
<td>N8</td>
<td>Byte stream of Encrypted User Account Data</td>
</tr>
<tr>
<td>N9</td>
<td>Byte stream of Encrypted Run Data</td>
</tr>
</tbody>
</table>

Table 2-1 System Data Flows
3. GUI Layer

![GUI Layer Module Diagram](image)

**Figure 3-1 GUI Layer Module Diagram**

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td>User input customizing what sensors are displayed.</td>
</tr>
<tr>
<td>G2</td>
<td>User input customizing which page is displayed.</td>
</tr>
<tr>
<td>G3</td>
<td>User input regarding account info and sensor configurations.</td>
</tr>
<tr>
<td>G4</td>
<td>User input regarding what menu option was chosen.</td>
</tr>
<tr>
<td>G5</td>
<td>Page id and sensors to display.</td>
</tr>
<tr>
<td>G6</td>
<td>Page id and sensors to display.</td>
</tr>
<tr>
<td>G7</td>
<td>Sensor configurations.</td>
</tr>
<tr>
<td>G8</td>
<td>Help page to display.</td>
</tr>
<tr>
<td>G9</td>
<td>Single sensor configuration.</td>
</tr>
</tbody>
</table>
### Table 3-1 GUI Layer Data Flows

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>G10</td>
<td>What type of processing.</td>
</tr>
<tr>
<td>G11</td>
<td>User account information.</td>
</tr>
<tr>
<td>G12</td>
<td>Instances of Configuration Model are stored in Configuration Model Map.</td>
</tr>
<tr>
<td>G13</td>
<td>Configuration Model Map for real time processing.</td>
</tr>
<tr>
<td>G14</td>
<td>Configuration Model Map for post processing.</td>
</tr>
<tr>
<td>G15</td>
<td>Current configuration to display.</td>
</tr>
<tr>
<td>G16</td>
<td>An instance of User Accounts Model is stored in the configuration model map.</td>
</tr>
<tr>
<td>G17</td>
<td>Real time gauge data output to user.</td>
</tr>
<tr>
<td>G18</td>
<td>Analyzed run data output to user.</td>
</tr>
<tr>
<td>G19</td>
<td>Configuration data output to user.</td>
</tr>
<tr>
<td>G20</td>
<td>Help page output to user.</td>
</tr>
<tr>
<td>P12</td>
<td>An array containing a list of saved runs.</td>
</tr>
<tr>
<td>P13</td>
<td>GaugeModel instance containing real time sensor data.</td>
</tr>
<tr>
<td>P14</td>
<td>GaugeAnalysisModel instance containing post processed run data.</td>
</tr>
</tbody>
</table>

### 3.1 Description

The GUI Layer resides physically on the mobile device and acts as the intermediary between users and Project AVALANCHE. This layer has three responsibilities:

1.) Parses user input into the system from the mobile device.

2.) Displays the data from the physical gauge in a graphical form, providing both sensor readouts, charts and metrics for both real time and past runs.

3.) Allows setup and configuration for individual sensors, user accounts and layout.

### 3.2 User Input: Drag & Drop Controller

#### 3.2.1 Prologue

Handles user input for the layout setup by allowing dragging and dropping of gauge and chart widgets from the widget tray. This allows a user to add and remove them from the view.
3.2.2 Interface

This module interfaces with the Mobile Device by capturing the users touch inputs. It also interfaces with the Post Process Controller module and Real Time Controller module by communicating which gauge widgets and charts will be displayed in each respective view.

3.2.3 External Data Dependencies

Cocoa Touch UI framework to take user inputs

3.2.4 Internal Data Dependencies

GUI libraries in Xcode using Objective C

3.2.5 Process/Pseudocode

```objective-c
- (IBAction)longPress:(UIGestureRecognizer *)sender {
  if (sender.state == UIGestureRecognizerStateChanged) {
    // figure out which item in the table was selected
    NSIndexPath *indexPath = [self.tableView indexPathForRowAtPoint:[sender locationInView:self.tableView]];
    if (!indexPath) {
      inDrag = NO;
      return;
    }
    inDrag = YES;

    // get the text of the item to be dragged
    NSString *text = [NSString stringWithFormat:__[objects objectAtIndex:indexPath.row] description];

    // create item to be dragged
    UIView *splitView = self.splitViewController.view;
    CGPoint point = [sender locationInView:splitView];
    UIFont *font = [UIFont systemFontOfSize:12];
    CGSize size = [text sizeWithFont:font];
    CGRect frame = CGRectMake(point.x - (size.width / 2.0), point.y - (size.height / 2.0), size.width, size.height);
    UIView *draggedView = [[UILabel alloc] initWithFrame:frame];
    [draggedView setFont:font];
    [draggedView setText:text];
    [draggedView setBackgroundColor:[UIColor clearColor]];
    [splitView addSubview:draggedView];
  } else if (sender.state == UIGestureRecognizerStateChanged && inDrag) {
    // update the coordinates of the dragged view
    UIView *splitView = self.splitViewController.view;
    CGPoint point = [sender locationInView:splitView];
    draggedView.center = point;
  } else if (sender.state == UIGestureRecognizerStateEnded && inDrag) {
    // do something
  }
}
```
{ //remove it from the view
    [draggedView removeFromSuperview];

    //where we dropped it
    UIView *detailView = self.detailViewController.view;
    CGPoint point = [sender locationInView:detailView];

    UIAlertView *alert;
    if (CGRectContainsPoint(detailView.bounds, point))
        alert = [[[UIAlertView alloc] initWithTitle:@"dropped in details view" message:nil delegate:nil cancelButtonTitle:@"Ok" otherButtonTitles:nil] autorelease];
    else
        alert = [[[UIAlertView alloc] initWithTitle:@"dropped outside details view" message:nil delegate:nil cancelButtonTitle:@"Ok" otherButtonTitles:nil] autorelease];
    [alert show];
}

3.3 User Input: Pop-Out Menu Controller

3.3.1 Prologue

Handles user input for the system menu. This menu can be activated from any view by using a two finger gesture and swiping from left or right. A left swipe will activate a menu that allows swapping to: the real time grid view, post process grid view, configuration view, and help view. A right swipe will activate the widget tray that gives the user the ability to drag and drop widgets onto the grid. This widget tray will only be accessible from the real time grid view and post process grid view.

3.3.2 Interface

This module interfaces with the Mobile Device by capturing the users touch inputs. It also interfaces with the Help Controller module by loading the help view when the user selects it from the menu. It also requests and loads past runs by interfacing with the RunListModel module.

3.3.3 External Data Dependencies

Cocoa Touch UI framework to take user inputs

3.3.4 Internal Data Dependencies

GUI libraries in Xcode using Objective C

3.3.5 Process/Pseudocode

- (id) initWithStyle:(UITableViewStyle)style
{
    self = [super initWithStyle:style];
    if (self) {
        // Sample menu items
        _data = [NSArray arrayWithObjects:@"Test 1", @"Test 2", @"Test 3", nil];
func viewDidLoad()
{
    super.viewDidLoad();

    UIView *bgView = [[UIView alloc] init];
    bgView.backgroundColor = [UIColor scrollViewTexturedBackgroundColor];
    [self.tableView setBackgroundView:bgView];

    self.tableView.separatorStyle = UITableViewCellSeparatorStyleNone;
}
3.4 User Input: Page Slider Controller

3.4.1 Prologue

Handles user input and allows the user to switch between the active grid views by using a single finger gesture and swiping left or right. The pages slider is accessible while in a grid view, either real time grid view or post process grid view.

3.4.2 Interface

This module interfaces with the Mobile Device by capturing the users touch inputs. It also interfaces with the Post Process Controller module and Real Time Controller module by telling each respective module which active grid view to display.

3.4.3 External Data Dependencies

Cocoa Touch UI framework to take user inputs

3.4.4 Internal Data Dependencies

GUI libraries in Xcode using Objective C

3.4.5 Process/Pseudocode

```swift
-(void)viewDidLoad {
    [super viewDidLoad];

    NSArray *colors = [[NSArray arrayWithObjects:[UIColor redColor], [UIColor greenColor], [UIColor blueColor], nil];
    for (int i = 0; i < colors.count; i++) {
        CGRect frame;
        frame.origin.x = self.scrollView.frame.size.width * i;
        frame.origin.y = 0;
        frame.size = self.scrollView.frame.size;

        UIView *subview = [[UIView alloc] initWithFrame:frame];
        subview.backgroundColor = [colors objectAtIndex:i];
        [self.scrollView addSubview:subview];
        [subview release];
    }

    self.scrollView.contentSize = CGSizeMake(self.scrollView.frame.size.width * colors.count, self.scrollView.frame.size.height);
}
```

3.5 User Input: Preferences Controller

3.5.1 Prologue

February 28, 2013
Handles user input and allows the user to modify configuration settings relating to the connected sensors. A user can add/remove sensors, enable/disable sensors, and adjust the calibration of each sensor.

3.5.2 Interface

This module interfaces with the Mobile Device by capturing the users touch inputs. It also interfaces with the Configuration Controller module by passing the users selected sensor settings along for further use.

3.5.3 External Data Dependencies

Cocoa Touch UI framework to take user inputs

3.5.4 Internal Data Dependencies

GUI libraries in Xcode using Objective C

3.5.5 Process/Pseudocode

```swift
- (IBAction)insertAndDeleteRows:(id)sender {
    // original rows: Sensor1, Sensor2
    [states removeObjectAtIndex:1]; // Sensor2
    [states removeObjectAtIndex:0]; // Sensor1
    [states insertObject:@"Sensor3" atIndex:2];
    [states insertObject:@"Sensor4" atIndex:3];
    [states insertObject:@"Sensor5" atIndex:4];

    NSArray *deleteIndexPaths = [NSArray arrayWithObjects:
        [NSIndexPath indexPathForRow:2 inSection:0],
        [NSIndexPath indexPathForRow:4 inSection:0],
        nil];
    NSArray *insertIndexPaths = [NSArray arrayWithObjects:
        [NSIndexPath indexPathForRow:0 inSection:0],
        [NSIndexPath indexPathForRow:3 inSection:0],
        [NSIndexPath indexPathForRow:5 inSection:0],
        nil];

    UITableView *tv = (UITableView *)self.view;
    [tv beginUpdates];
    [tv insertRowsAtIndexPaths:insertIndexPaths
        withRowAnimation:UITableViewRowAnimationRight];
    [tv deleteRowsAtIndexPaths:deleteIndexPaths
        withRowAnimation:UITableViewRowAnimationFade];
    [tv endUpdates];
}
```

3.6 Data Output: Post Process Controller

3.6.1 Prologue

The Post Process Controller Module will update the gauge display with previously recorded and saved sensor data.
3.6.2 Interface

This module will interface the GaugeAnalysis Module in the Post-Processing Subsystem of the Data Processing Layer by taking the PostBuilder module information and forming a model out of it. This module will also interface with the two modules from the Input Subsystem; the Drag and Drop Controller and the Page Slider Controller which will allow users to select this output option.

3.6.3 External Data Dependencies

Inputs from the Drag and Drop Controller and the Page Slider Controller for selection using the Cocoa Touch UI framework

3.6.4 Internal Data Dependencies

GUI libraries in Xcode using Objective C

3.6.5 Process/Pseudocode

class PostProcessController extends Thread{
    GaugeModel gauge;
    private final long SLEEP_TIME;
    /**
     *@constructor
     *@param model
     * the model passed from the data-processing layer
     */
    public PostProcessController(GaugeModel model, long frequency){
        gauge = model;
        //post builder should offer a post-process connector to the gauge
        SLEEP_TIME = frequency;
        //update rate for post process transfer
    }
    //multi-threaded for updating the gui
    @Override
    void run(){
        try{
            display(gauge);
        } catch(Exception e){
            e.printStackTrace();
        } finally{
            Thread.sleep(SLEEP_TIME);
        }
    }
}

3.7 Data Output: Real-Time Controller

3.7.1 Prologue

The Real-Time Controller Module will update the gauge display in real-time with active sensor data.
3.7.2 Interface

This module will interface the Gauge Model Module in the Real-Time Subsystem of the Data Processing Layer by taking the Real-Time Builder information and forming a model out of it. This module will also interface with the two modules from the Input Subsystem; the Drag and Drop Controller and the Page Slider Controller which will allow users to select this output option.

3.7.3 External Data Dependencies

Inputs from the Drag and Drop Controller and the Page Slider Controller for selection

3.7.4 Internal Data Dependencies

GUI libraries

Premade Gauge Libraries for Objective C

3.7.5 Process/Pseudocode

```java
class RealTimeController extends Thread {
    GaugeModel gauge;
    private final long SLEEP_TIME;
    /**
     * @constructor
     * @param model
     *       the model passed from the data-processing layer
     */
    public RealTimeController(GaugeModel model, long frequency) {
        gauge = model;
        // gauge model should offer a real-time connector to the gauge
        SLEEP_TIME = frequency;
        // update rate for realtime transfer
    }
    // multi-threaded for updating the gui
    @Override
    void run() {
        try {
            display(gauge);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            Thread.sleep(SLEEP_TIME);
        }
    }
}
```

3.8 Data Output: Configuration Controller

3.8.1 Prologue
The Configuration Controller Module will display the configuration information about how the gauges are configured to a particular user.

### 3.8.2 Interface

This module will interface the Configuration Model Map Module, User Account Module, and Settings Controller Module.

### 3.8.3 External Data Dependencies

Settings from the User Account Module, options from the Settings Controller Module, and gauge information from the Configuration Model Map Module which stores multiple sensors.

### 3.8.4 Internal Data Dependencies

GUI libraries in Xcode using Objective C.

### 3.8.5 Process/Pseudocode

```java
class ConfigurationController{
   private SettingsController sc;
   private userAcct;
   public ConfigurationController(UserAccount user, ConfigurationModelMap cmm){
      userAcct = user;
      sc = userAcct.getSettings();
      sc.addConfigMap(cmm);
      setup();
   }

   public void setup(){
      GUI.setDisplay(sc);
   }
}
```

### 3.9 Data Output: Help View

#### 3.9.1 Prologue

The Help View Module will be an immutable page displaying general information and FAQ for app users.

#### 3.9.2 Interface

This Module will only interface with the Pop-Out Menu Controller Module in that a user will select it from the Controller.

#### 3.9.3 External Data Dependencies

User input from Pop-Out Menu Controller Module
3.9.4 Internal Data Dependencies

None

3.9.5 Process/Pseudocode

class HelpView{
    // Hard Coded Static Strings to represent the
    // help information on the page
    user.HelpSelected();
    displayHelpMenu();
}

3.10 Configure: Configuration Model

3.10.1 Prologue

Each Configuration Model Module will represent the configuration of one sensor.

3.10.2 Interface

This module will interface with the Configuration Model Maps module. The Configuration Model will represent the “value” portion of the Configuration Model Maps Module where the sensor’s ID will be the “key”.

3.10.3 External Data Dependencies

None.

3.10.4 Internal Data Dependencies

None.

3.10.5 Process/Pseudocode

class ConfigurationModel{
    private int sensorID, transformConstant;
    private String sensorName, type;
    private boolean active;
    public Configuration(int id, int transConst, String name, String type, boolean active){
        this.active = active;
        sensorID = id;
        transformConstant = transConst;
        sensorName = name;
        this.type = type;
    }
    public String getName(){
        return sensorName;
    }
    public String getType(){

3.11 Configure: Configuration Model Map

3.11.1 Prologue

The Configuration Model Map Module will represent the hash map data structure storing a sensor ID as the key and a respective Configuration Model as the value.

3.11.2 Interface

This module will interface with the Configuration Model Module and use the various ConfigurationManager objects as the value in the key/value pairs.

3.11.3 External Data Dependencies

Dependent on the ConfigurationManager object.

3.11.4 Internal Data Dependencies

Libraries for NSMutableArray array.

3.11.5 Process/Pseudocode

```java
import Foundation
class ConfigurationManager:
    private var sensorMap = 
```
3.12 Configure: User Account

3.12.1 Prologue

The Configure User Account Module will be represented by a page for the user to enter account information (User name and password) and log into the system.

3.12.2 Interface

This

3.12.3 External Data Dependencies

Dependent on the ConfigurationModel object.

3.12.4 Internal Data Dependencies

Libraries for NSMutableArray array.

3.12.5 Process/Pseudocode

```java
public ConfigurationModelMap(int initialSize)
    configMag = new NSMutableArray(initialSize);
}
public void addSensor(ConfigurationModel sensorConfig)
    configMap[sensorConfig.getID()] = sensorConfig;
}
```

```java
import NSUserDefaults library
class ConfigurationModelMap{
    private NSMutableArray configMap;
    public ConfigurationModelMap(int initialSize){
        configMag = new NSMutableArray(initialSize);
    }
    public void addSensor(ConfigurationModel sensorConfig){
        configMap[sensorConfig.getID()] = sensorConfig;
    }
}
```
4. Data Processing Layer

Figure 4-1 Data Processing Layer Module Diagram

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>Data is transmitted byte by byte in a predetermined order</td>
</tr>
<tr>
<td>B2</td>
<td>Bluetooth Protocol Information</td>
</tr>
<tr>
<td>P1</td>
<td>Create GaugeModel instance based off of data from Sync Subsystem.</td>
</tr>
<tr>
<td>P2</td>
<td>Create new SensorAggregateModel to based off of data from Sync Subsystem.</td>
</tr>
</tbody>
</table>
Create new SensorSnapshotModel to based off of data from Sync Subsystem.

Byte array containing data of a single instance in time.

SensorAggregateModels are contained in GaugeModel instance.

SensorSnapshotModels are contained in a SensorAggregateModel.

File descriptor of run to analyze.

File descriptor of sensor to analyze.

SensorAnalysisModels are contained GaugeAnalysisModel.

GaugeModel instance to be serialized.

Folder containing saved run.

An array containing a list of saved runs.

GaugeModel instance containing real time sensor data.

GaugeAnalysisModel instance containing post processed run data.

User Account Model containing user information.

Configuration Model Map for real time processing.

Configuration Model Map for post processing.

A saved run to analyze.

An array containing a list of saved runs.

Table 4-1 Data Processing Layer Data Flows

4.1 Description

The Data Processing Layer resides physically on the mobile device. The Data Processing Layer decodes and processes the data received from the Bluetooth Layer for output through the GUI Layer and for storage in the Storage Layer. The Data Processing Layer will have two distinct processing modes, real time and post data processing. Real time data processing will handle parsing data received from either the gauge. Post processing will analyze a prior run saved in storage and produce graphs and metrics based on the data from the run.

4.2 Real Time Processing: RealTimeBuilder

4.2.1 Prologue
This module is a class that will handle building the Real Time Subsystem models; SensorSnapshotModel, SensorAggregateModel, and GaugeModel; based on the input received from the Sync Subsystem and the configurations from the Configure Subsystem in the GUI layer, received in a ConfigurationModelMap.

### 4.2.2 Interface

This class interfaces with both the ConfigurationModelMap module of the GUI Layer and the BLEGaugeAlarmService in the Sync Subsystem. The interface with the ConfigurationModelMap module is responsible for passing configuration data into the Real Time Processing Subsystem, and the interface with the BLEGaugeAlarmService is responsible for passing in the raw gauge data into the Real Time Processing Subsystem.

The method `setConfigurationMap` is called by the ConfigurationModelMap class when the application loads or the user changes any sensor configurations. This method is used to pass the configuration options, a ConfigurationModelMap containing ConfigurationModel, the user makes into the Real Time Processing Subsystem to define the sensor input; name, type, and id of each sensor; as well as the transformConstant, a user inputted constant that defines how to transform the gauge data into actual units.

The method `beginProcessing` is called by the ConfigurationModelMap class when the user selects to begin outputting and recording a run in progress.

The method `endProcessing` is called by the ConfigurationModelMap when the user selects to end the run. This will stop run recording and saves all recorded data.

The method `snapshotCreation` is called by BLEGaugeAlarmService periodically when connected to a gauge. This passes the raw data into the Real Time Processing Subsystem to be processed and saved.

### 4.2.3 External Data Dependencies

- ConfigurationModelMap: Map object containing all relevant configuration information from the Configuration Subsystem.
- Byte[] Data: An array of bytes containing a snapshot of all sensors on the gauge, one data point for each sensor at one moment of time,

### 4.2.3 Internal Data Dependencies

Dependent on SensorSnapshotModel, SensorAggregateModel, and GaugeModel objects.

### 4.2.4 Process/Pseudocode

```c
/*
 * RealTimeBuilder is a builder object that takes the configurations made in the GUI Layer, user defined sensor names
 * and what sensors are active, and creates the models for real time update.
 */
```
@interface RealTimeBuilder : NSObject
{
}

// Public class methods.

// Create a new GaugeModel singleton instance.
+(void) gaugeModelFactory;

// Set configuration map.
+(void) setConfigurationMap: (ConfigurationModelMap*) configurationMap;

// Turn on processing.
+(void) beginProcessing;

// Turn off processing.
+(void) endProcessing;

// Create all snapshots for all active sensors.
+(void) snapshotCreation: (Byte[]) data;

@end;

// Class variables
static ConfigurationModelMap* configurationMap;
static const int DEFAULT_NUM_SENSORS = 12;
static BOOL processing = NO;

@interface RealTimeBuilder()

// Private class methods.

// Creates SensorSnapshotModel objects.
+(SensorSnapshot) sensorSnapshotModelFactory: (long) timeStamp
   withName:    (NSString*) sensorName
   withType:   (NSString*) sensorType
   withSensorID:    (int) sensorID
   withData:    (int) sensorData;

// Creates SensorAggregateModel objects.
+(SensorAggregateModel) sensorAggregateModelFactory: (NSString*) sensorName
   withType:   (NSString*) sensorType
   withSensorID:    (int) sensorID
   isActive:    (BOOL) active;

// Transform sensor data into readable form.
+(int) transformSensorData: (int) sensorID
   ofTypeSensorType:   (NSString*) sensorType
   withID:    (int) sensorID;

@end

@implementation RealTimeBuilder

// Public class methods.

// Create a new GaugeModel singleton instance.
+(void) gaugeModelFactory
{
    // Make a map with a capacity for all sensors.
    NSMutableArray* SensorAggregateModelMap = [NSMutableArray arrayWithCapacity: DEFAULT_NUM_SENSORS];

    // Use configuration data to set up SensorAggregateModels
    for each configuration in configurationMap
    {
        NSString* sensorName = [configuration getSensorName];
        NSString* sensorType = [configuration getSensorType];
        NSString* sensorID = [configuration getSensorType];
        BOOL active = [configuration isActive];

        if(active)
        {
            SensorAggregateModel* sensorAggregateModel = [self SensorAggregateModelFactory: sensorName withType: sensorType withSensorID: isActive: active];

            [SensorAggregateModelMap insertObject: sensorAggregateModel atIndex: sensorID];
        }
    }
}

// If there is an instance reset the instance
if([GaugeModel isNotNill])
{
    [GaugeModel resetInstance];
}

[[GaugeModel getInstance] setSensorAggregateModels];

// Set configuration map.
+(void) setConfigurationMap: (ConfigurationModelMap*) configurationMap
{
    self.configurationMap = configurationMap;
}

// Called when user begins run. Tells model to start saving data being pulled in from Bluetooth connection.
+(void) beginProcessing
{
    processing = YES;
}

// Called when user ends run. Serializes the GaugeModel, saving the run in local storage, and turns off processing.
+(void) endProcessing
{
    [GaugeModel serialize];

    processing = NO;
}

// Creates a SensorSnapshotModel for each active sensor and adds each snapshot to their related SensorAggregateModel.
+(void) snapshotCreation: (Byte[]) data
{
    if(processing)
    {
        long timeStamp = [getTimeStamp];
    }
for(int i = 0; i < [data size]; i++)
{
    SensorAggregateModel* aggregate = [GaugeModel getInstance getAggregateAtIndex: i];
    NSString* sensorName = [aggregate getSensorName];
    NSString* sensorType = [aggregate getSensorType];
    BOOL active = [aggregate isActive];

    if(active)
    {
        SensorSnapshotModel* snapShotModel = [self sensorSnapshotModelFactory: timeStamp withSensorName: sensorName
            withSensoryType: sensorType withSensorID: i];
        [aggregate addObject: snapShotModel];
    }
}
//Private class methods.

// Validates SensorSnapshotModel parameters and then creates and returns that object.
+(SensorSnapshotModel*) sensorSnapshotModelFactory: (long) timeStamp
    withName: (NSString*) sensorName
    withType: (NSString*) sensorType
    withSensorID: (int) sensorID
    withData: (int) sensorData
{
    Validate parameters

    sensorData = [self transformSensorData: sensorData ofType: sensorType withID: sensorID];

    return([SensorSnapshotModel initWithTimeStamp: timeStamp withName: sensorName
        withType: sensorType
        withSensorID: sensorID
        withData: sensorData]);
}
// Transform the raw sensor data into formatted readable sensor data.
+(int) transformSensorData: (int) sensorData
    ofType: (NSString*) sensorType
    withID: (int) sensorID;
{
    int transformConstant = [[configurationMap objectAtIndex: sensorID] getTransformConstant];

    switch(sensorType)
    {
    // Transform sensorData based on formula for each data type.
    case "tachometer":
    {
        // # of ticks / ticks per revolution * 8 hertz * 60 sec per min. = revolutions per minute.
        return (sensorData / transformConstant) * 8 * 60;
    }
    case "speedometer":
    {
        // Convert tix/8th second to tix/hour. Divide that by tix per mile. gives miles/hour.
        return (sensorData * 8 * 60 * 60) / (8 * tixPer8thOfMile);
    }
    }
case "temp":
{
    return sensorData / 10; // Sensor value off of CAN BUS is degrees celcius * 10
}

case "pressure":
{
    return sensorData / 10; // Sensor value off of CAN BUS is PSI * 10
}

case "voltage":
{
    return sensorData;
}

case "oxygen":
{
    return sensorData;
}
}

// Validates SensorAggregateModel parameters and then creates and returns that object.
+(SensorAggregateModel*) sensorAggregateModelFactory: (NSString*) sensorName
    withType: (NSString*) sensorType
    withSensorID: (int) sensorID
    isActive: (BOOL) active;
{
    Validate parameters

    return ([SensorAggregateModel initWithName: sensorName withType: sensorType withSensorID: sensorID isActive: active]);
}

@end

4.3 Real Time Processing: SensorSnapshotModel

4.3.1 Prologue

This module contains the data for a single sensor for one moment of time; as well as the sensor name, id, sensor type, and timestamp needed to identify the type of sensor the snapshot represents and help display and save the information.

4.3.2 Interface

SensorSnapshotModels will be created by RealTimeBuilder and put into a SensorAggregateModel. From there the properties of this class can be accessed using the setter methods for each property.

4.3.3 External Data Dependencies

None
4.3.4 Internal Data Dependencies

None

4.3.5 Process/Pseudocode

/ *
  * SensorSnapshotModel encapsulates a single sensor data point.
  */
@interface SensorSnapshotModel : NSObject
{
}

// SensorSnapshotModel properties.
@property long timeStamp;
@propertyNSString* sensorName;
@propertyNSString* sensorType;
@propertyint sensorID;
@propertyint sensorData;

// Public instance methods.
-(id)initWithTimeStamp: (long)timeStamp
    withName: (NSString*)sensorName
    withType: (NSString*)sensorType
    withSensorID: (int)sensorID
    withData: (int)sensorData

-(NSString*) serialize;

-(id) deserialize: (NSString*) serializedData;
@end

@implementation SensorSnapshotModel
{

    // Generate setter and getter methods.
    @synthesize timeStamp;
    @synthesize sensorName;
    @synthesize sensorType;
    @synthesize sensorID;
    @synthesize sensorData;

    // Constructor method that sets all of the snapshots data.
    -(id)initWithTimeStamp: (long)timeStamp
        withName: (NSString*)sensorName
        withType: (NSString*)sensorType
        withSensorID: (int)sensorID
        withData: (int)sensorData;
    {
        self = [super init];

        self.timeStamp = timeStamp;
        self.sensorName = sensorName;
        self.sensorType = sensorType;
        self.sensorID = sensorID;
        self.sensorData = sensorData;
    }

    // Constructor method that sets all of the snapshots data.
    -(id)initWithTimeStamp: (long)timeStamp
        withName: (NSString*)sensorName
        withType: (NSString*)sensorType
        withSensorID: (int)sensorID
        withData: (int)sensorData;
    {
        self = [super init];

        self.timeStamp = timeStamp;
        self.sensorName = sensorName;
        self.sensorType = sensorType;
        self.sensorID = sensorID;
        self.sensorData = sensorData;
    }

@end
4.4 Real Time Processing Subsystem: SensorAggregateModel

4.4.1 Prologue

The SensorAggregateModel class will contain an array of SensorSnapshotModels representing a run for one sensor. The class’s instance variables will also contain sensor properties including the sensor name, sensor type, sensor id, and a boolean value representing whether the sensor is being actively processed and recorded.

4.4.2 Interface

Instances of this class are created in the RealTimeBuilder, where each instance of the SensorAggregateModel is added to an array inside the GaugeModel instance. Each SensorAggregateModel contains an array of SensorSnapshotModels.

4.4.3 External Data Dependencies

None

4.4.4 Internal Data Dependencies

Dependent on SensorSnapshotModel objects.

4.4.5 Process/Pseudocode

/*@ SensorAggregateModel contains an array of SensorSnapshotModels as well as the associated metadata; name of sensor, type of sensor, sensorID, and whether the sensor is actively being processed; for that sensor. */
@interface SensorAggregateModel: NSObject
{ return self;
}
// Will serialize the object to a NSString.
-(NSString*) serialize
{
    return A serialization string;
}
// Create a deserialized object from a NSString.
-(id) deserialize: (NSString*) serializedData
{
    Create object based on serializedData.
    return self;
}
@end
// Instance properties.
@property NSMutableArray* snapshots;

// Instance properties.
@property NSString* sensorName;
@property NSString* sensorType;
@property int sensorId;
@property BOOL active;

//Initialize SensorAggregateModel with inputted metadata.
-(id)initWithName: (NSString*)sensorName
   withType: (NSString*)sensorType
   withSensorID: (int)sensorID
   isActive: (BOOL)active;

-(void)serialize: (File*) file;

@end

static const int DEFAULT_NUM_SENSORS = 12;

@implementation SensorAggregateModel

// Generates getter and setter methods for all properties.
@synthesize snapshots;
@synthesize sensorName;
@synthesize sensorType;
@synthesize sensorId;
@synthesize active;

// Init with metadata.
-(id)initWithName: (NSString*)sensorName
   withType: (NSString*)sensorType
   withSensorID: (int)sensorID
   isActive: (BOOL)active;
{
    self = [super init];

    self.sensorName = sensorName;
    self.sensorType = sensorType;
    self.sensorID = sensorID;
    self.isActive = active;

    return self;
}

// Deserialize the object from a file.
-(id) deserialize: (File*) file
{
    self = [super init];
    Read metadata from file

    for each line in file
{  
    int count++;  
    SensorSnapshotModel* snapshot = [[SensorSnapshotModel alloc] deserialize: line];  
    [snapshots addObject: snapshot atIndex: count];  
}

return self;

// Serialize the object to a file.
-(void) serialize: (File*) file  
{
    Write metadata to file

    for each sensorSnapshotModel in snapshots  
    {
        [sensorSnapshotModel serialize: file];
    }
}

-(id) deserialize: (File*) file
@end

4.5  Real Time Processing Subsystem: GaugeModel

4.5.1 Prologue

The GaugeModel class is a singleton class, where only one instance can exist at a time, which can be accessed by the RealTimeView for output of all incoming real time data to the user through the GUI. The current instance is created at the beginning of the run recording and each sensor is given a SensorAggregateModel which is stored in the sensorAggregateMdoelMap. This map can then be accessed by the RealTimeView to update the sensors displayed in the GUI.

4.5.2 Interface

The GaugeModel singleton object is managed through the RealTimeBuilder module; the singleton is created and the snapshots are added through this class. The singleton contains a map of SensorAggregateModel objects. The data in the singleton is accessed the Real Time View module.

4.5.3 External Data Dependencies

None

4.5.4 Internal Data Dependencies

Is Dependent on SensorAggregateModel objects.

4.5.5 Process/Pseudocode
/**
* GaugeModel is a singleton model object that will be used to display gauge data through the UI while in real time
* processing mode. This model is a container model that holds all the SensorAggregateModels that are active, which
* in turn will hold the SensorSnapshotModels which contain one sensor datapoint.
*/
@interface GaugeModel : NSObject
{
}
// Instance properties.
@property long startTimeStamp;
@property long endTimeStamp;
@property NSMutableArray* sensorAggregateModelMap;

// Public class methods.
+(GaugeModel*) getInstance;
+(void) resetInstance;

// Public instance methods.
-(SensorAggregateModel*) getAggregateAtIndex: (int) index;
-(void) serialize: (File*) file;
-(void) deserialize: (File*) file;
@end

@interface GaugeModel()

// Private instance methods
-(id) init;
@end

// Class variables.
static GaugeModel gaugeModelInstance;
static const int DEFAULT_NUM_SENSORS = 12;

@implementation GaugeModel

// Create getters and setters for instance properties.
@synthesize startTimeStamp;
@synthesize endTimeStamp;
@synthesize sensorAggregateModelMap;

// Get the instance. If their is not an instance init one.
+(GaugeModel*) getInstance
{
    if(gaugeModelInstance == nil)
    {
        gaugeModelInstance = [GaugeModel alloc] init];
    }

    return(gaugeModelInstance);
}

// Reset the singleton object.
+(void) resetInstance
{
gaugeModelInstance = nil;

// Serialize the object.
-(void) serialize: (File*) filePath
{
    [gaugeModelInstance setEndTimeStamp: [getTimeStamp]];

    Save properties to filePath
    for each sensorAggregateModel in sensorAggregateModelMap
    {
        (File*) aggregateModel = filepath + sensorID + [getTimeStamp];
        [sensorAggregateModel serialize: file];
    }

    [self resetInstance];
}

// Deserialize the object.
-(void) deserialize: (File*) filePath
{
    Get properties from filePath
    for each sensorAggregateModel file in the folder
    {
        SensorAggregateModel* sensorAggregate = [SensorAggregateModel deserialize: file];

        [[self getInstance] getSensorAggregateModelMap] addObject sensorAggregate];
    }
}

// Get the requested SensorAggregateModel.
-(SensorAggregateModel*) getAggregateAtIndex: (int) index
{
    if sensorAggregateModelMap == nil
    {
        return nil;
    }
    else
    {
        return([sensorAggregateModelMap objectAtIndex: index]);
    }
}

// Private initializer methods.
-(id) init
{
    self = [super init];
    self.startTimeStamp = [getTimeStamp];

    return self;
}

@end
4.6 Post Processing: RunListModel

4.6.1 Prologue

This module will make a list of all previous runs in device memory and the database.

4.6.2 Interface

This module interfaces with the Local Storage Layer which will query the Cloud Server to get a list of previous runs from both sources.

4.6.3 External Data Dependencies

Run data from Local Storage Layer

4.6.4 Internal Data Dependencies

None

4.6.5 Process/Pseudocode

class RunListModel{
    private NSMutable[] runArray;
    public RunListModel(){
        NSMutable[] runArray = new NSMutable(runs.size());
    }
    public void queryRuns(){
        int i = 0;
        while(runs.hasNext)){
            runArray[i] = runs.get(i);
            i++
        }
    }
}

4.7 Post Processing: SensorAnalysisModel

4.7.1 Prologue

This module will create a line graph based on sensor data.

4.7.2 Interface

This module will interface with the Post Builder Module which it will accept saved data from (from a single sensor).

4.7.3 External Data Dependencies

Requires data from a single sensor built in Post Builder Module
4.7.4 Internal Data Dependencies

Built in libraries for creating line graphs

4.7.5 Process/Pseudocode

```java
class SensorAnalysisModel {
    private SensorAggregateModel sensorAggregate;
    private int range;
    private int mean;
    private int mode;
    private int median;

    public analyze(File filePath) {
        sensorAggregate.deserialize(filePath);

        for each SensorSnapshotModel in sensorAggregate {
            Calculate range, mena, mode, and median.
        }
    }

    public SensorAggregateModel getSensorAggregate() {
        return sensorAggregate;
    }

    public int getRange() {
        return range;
    }

    public int getMean() {
        return mean;
    }

    public int getMode() {
        return mode;
    }

    public int getMedian() {
        return median;
    }
}
```

4.8 Post Processing: GaugeAnalysisModel

4.8.1 Prologue
This module is a singleton object meaning only one can be instantiated at any given time. This module holds an array of Sensor Analysis Models (or SensorAnalysisModel object).

### 4.8.2 Interface

This module will interface with the Sensor Analysis Model Module and the Post Builder Module. It will accept control variables from the Post Builder Module and sensor data from the Sensor Analysis Model module.

### 4.8.3 External Data Dependencies

Dependent on control variables from PostBuilder for setup. Dependent on various SensorAnalysisModel from the Sensor Analysis Model Module.

### 4.8.4 Internal Data Dependencies

None

### 4.8.5 Process/Pseudocode

```java
class GaugeAnalysisModel{
    private NSMutableArray sensorAnalysisModel;

    private int duration;

    public void analyze(File filePath)
    {
        GaugeModel gaugeModel = GaugeModel.deserialize(filePath);

        duration = gaugeModel.getEndTimeStamp() - gaugeModel.getStartTimeStamp();

        for each SensorAggregateModel in gaugeModel
        {
            SensorAnalysisModel sensorAnalysis = new SensorAnalysisModel();

            sensorAnalysis.analyze(SensorAggregateModel);
        }
    }

    public NSMutableArray getSensorAnalysisModel()
    {
        return sensorAnalysisModel;
    }
}
```

### 4.9 Post Processing Subsystem: PostBuilder

### 4.9.1 Prologue
PostBuilder is a builder class that uses the configuration data from a ConfigurationModelMap to load a run from the Local Storage Layer and build the corresponding SensorAnalysisModels and GaugeAnalysisModel.

4.9.2 Interface

The class interfaces with the ConfigurationModelMap class to load the

4.9.3 External Data Dependencies

Is dependent ConfigurationModelMap that contains the file path of the run folder to load. The PostBuilder class is also dependent on the saved run folder stored in the Local Storage Layer.

4.9.4 Internal Data Dependencies

PostBuilder is dependent on the SensorAnalysisModel and the GaugeAnalysisModel classes.

4.9.5 Process/Pseudocode

```java
/**
 * PostBuilder is builder class that uses configuration data from a ConfigurationModelMap to load a run from the
 * Local Storage Layer and build the corresponding SensorAnalysisModels and GaugeAnalysisModel.
 */
public class PostBuilder
{
    // The ConfigurationModelMap containing user configuration.
    static ConfigurationModelMap configurationMap;

    // Set the configuration map.
    public static void setConfigurationModelMap(ConfigurationModelMap configurationMap)
    {
        this.configurationMap = configurationMap;
    }

    // A SensorAnalysisModel factory that creates SensorAnalysisModel objects based on saved runs in local memory.
    public static SensorAnalysisModel SensorAnalysisModelFactory(File filepath)
    {
        SensorAggregateModel sensorAggregate = new SensorAggregateModel();
        sensorAggregate.deserialize(filepath);

        SensorAnalysisModel sensorAnalysis = new SensorAnalysisModel();
        sensorAnalysis.analyze(sensorAggregate);

        return sensorAnalysis;
    }

    // A factory method that creates GaugeAnalysis.
    public static void GaugeAnalysisModelFactory(File filepath)
    {
        GaugeModel gauge = GaugeModel.getInstance();
        gauge.deserialize(filePath);

        GaugeAnalysisModel gaugeAnalysis = new GaugeAnalysisModel();
        gaugeAnalysis.analyze(GaugeModel);
```
// Analyze a file path to create the GaugeAnalysisModel and add the SensorAnalysisModel.
public static GaugeAnalysisModel analyze(File filepath)
{
    GaugeAnalysis gagueAnalysis = GaugeAnalysisModelFactory(filepath);

    for each SensorAggregateModel file in the filepath
    {
        boolean active = configurationMap.isActive(index);

        if(active)
        {
            SensorAnalysisModel sensorAnalysis = SensorAnalysisModelFactory(file);

            gagueAnalysis.add(sensorAnalysis);
        }
    }

    return gagueAnalysis;
}

4.10 Sync Subsystem: BLEAlarmService

4.10.1 Prologue

The BLEAlarmService module is a class will be responsible managing Bluetooth connections to the Bluetooth Layer and handling the passing of data from Bluetooth Layer to the Data Processing Layer.

4.10.2 Interface

The BLEAlarmService module will interface with the sync module of the Bluetooth Layer to sync to the hardware module and the Transfer Data module from which it will receive a Byte Stream containing the gauge’s sensor information. This byte Stream will then be passed onto the RealTimeBuilder subsystem in the Data Processing Layer.

4.10.2 External Data Dependencies

Apple Core Bluetooth Framework.

4.10.3 Internal Data Dependencies

A Byte stream from the Bluetooth Layer.

4.10.4 Process/Pseudocode

/*
 * RealTimeBuilder is a builder object that takes the configurations made in the GUI Layer, user defined sensor names
 * and what sensors are active, and creates the models for post analysis.
 */
public Class BLEAlarmService
private static CBCentralManager manager;

private static BPeripheral peripheral;

public void startScan()
{
    peripheral = manager.scanForPeripherals();
}

public void updateValueForCharacteristic()
{
    Characteristic characteristic = peripheral.getCharacteristic();
    
    Byte[] data = new Byte[numberSensors];
    
    while bluetooth transmission
    {
        data[index] = characteristic.value;
        
        if number of sensors parsed
        {
            RealTimeBuilder.snapshotCreation(data);
            
            data = new Byte[numberSensors];
        }
    }
}

4.11 Save Data: GaugeSerializer

4.11.1 Prologue

The GaugeSerializer module is a class that will handle serializing the Real Time Processing models into files in local storage.

4.11.2 Interface

The GaugeSerializer module interfaces with the GaugeModel module in the Data Processing Layer which will be serialized into a folder in the Run Storage module in the Local Storage Layer

4.11.3 External Data Dependencies

iOS app file storage

4.11.4 Internal Data Dependencies
A GaugeModel to serialize and a file path to serialize the GaugeModel to.

### 4.11.5 Process/Pseudocode

```java
import java.io.File;
import java.io.Folder;

public class GaugeSerializer {
    // The default place where the runs will be saved in local memory.
    File defaultPath = "...";

    public void serialize(GaugeModel gaugeModel) {
        // Create a folder using the current time stamp as the name.
        Folder folder = new folder(getTimeStamp);

        // Create file path to serialize in based on new folder.
        Filepath savePath = defaultPath + folder;

        // The gaugeModel with create one file with basic information and a file for each sensor aggregate.
        gaugeModel.serialize(savePath);
    }

    public void setDefaultPath(File path) {
        this.defaultPath = path;
    }
}
```
5. Bluetooth Layer

![Bluetooth Layer Module Diagram]

**Table 5-1 Bluetooth Layer Data Flows**

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>Data is transmitted byte by byte in a predetermined order</td>
</tr>
<tr>
<td>B2</td>
<td>Bluetooth Protocol Information</td>
</tr>
<tr>
<td>B3</td>
<td>Data from the Data Acquisition Layer is received and sent to be transmitted</td>
</tr>
<tr>
<td>B4</td>
<td>BR-LE4.0-S2A pin goes high to indicate sync status</td>
</tr>
<tr>
<td>B5</td>
<td>BR-LE4.0-S2A high pin triggers interrupt in MCU</td>
</tr>
<tr>
<td>A1</td>
<td>Data is sent via SCI to BR-LE-4.0S2A byte by byte in a packaged order</td>
</tr>
</tbody>
</table>

**5.1 Description**

The Bluetooth Layer resides physically on the hardware module. The Bluetooth Layer is responsible for transmitting data 1 byte at a time to the Data Processing Layer. It will be composed of a BR-LE4.0-S2A...
Bluetooth Module that receives data pre-packed from a MC9S12C128 Microcontroller in the Data Acquisition Layer.

5.2  Sync: Wait for Sync

5.2.1 Prologue
The Wait for Sync is a functionality that already exists on the BR-LE4.0-S2A chip. Upon chip power on, it will go into a wait mode until a device pairs with it.

5.2.2 Interface
This module will interface with the Notify MCU Module

5.2.3 External Data Dependencies
None

5.2.4 Internal Data Dependencies
Bluetooth Protocol

5.2.5 Process/Pseudocode
Since this Module is something that already exists on the BR-LE4.0-S2A chip, there is no pseudocode required.

5.3  Sync: Notify MCU

5.3.1 Prologue
The Notify MCU will be responsible for sending a pin high output to both an LED and the MC9S12C128 Microcontroller to display it’s synced status.

5.3.2 Interface
This module will interface with the Wait for Sync Module

5.3.3 External Data Dependencies
None

5.3.4 Internal Data Dependencies
None

5.3.5 Process/Pseudocode
The BR-LE-4.0-S2A device will drive a pin high when the device is synced. As a result, we simply need to split the line in parallel. One will go to the MCU, who will generate an interrupt on receipt, and one will go to drive an LED.

5.4 Transfer Data: Receive from MCU

5.4.1 Prologue

The MC9S12C128 MCU will use SCI UART to transmit the data stream 1 byte at a time to the BR-LE4.0-S2A Bluetooth Module for transfer.

5.4.2 Interface

This module will interface with the Send Data module in the Bluetooth Layer, and the Send Data module in the Data Acquisition Layer

5.4.3 External Data Dependencies

None

5.4.4 Internal Data Dependencies

Packaged byte stream for all sensors.

5.4.5 Process/Pseudocode

The BR-LE-4.0-S2A device will receive the packaged data stream from the MC9S12C128 via SCI UART communication to the TX port on the Bluetooth module.

5.5 Transfer Data: Send Data

5.5.1 Prologue

The BR-LE4.0-S2A module will send all packed data over Bluetooth

5.5.2 Interface

This module will interface with the Receive from MCU module, and the Sync module in the Data Processing Layer.

5.5.3 External Data Dependencies

None

5.5.4 Internal Data Dependencies

Packaged byte stream for all sensors.
5.5.5 Process/Pseudocode

The BR-LE-4.0-S2A device will receive the packaged data stream from the MC9S12C128 via SCI UART communication to the TX port on the Bluetooth module, and is pre-configured to automatically transfer this data.
6. Data Acquisition Layer

Figure 6-1 Data Acquisition Layer Module Diagram

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>Data is sent via SCI to BR-LE-4.0S2A byte by byte in a packaged order</td>
</tr>
<tr>
<td>A2</td>
<td>syncStatus flag is checked before data is packaged and transmitted.</td>
</tr>
<tr>
<td>A3</td>
<td>Algorithm takes sensor data and decomposes it into two bytes for transmission</td>
</tr>
<tr>
<td>A4</td>
<td>Raw Data is taken and stored into arrays representing its sensor</td>
</tr>
<tr>
<td>A5</td>
<td>Raw data is read off of the CAN Bus line</td>
</tr>
<tr>
<td>B5</td>
<td>BR-LE-4.0S2A pin goes high on sync to trigger interrupt on MC9S12C128</td>
</tr>
</tbody>
</table>

Table 6-1 Data Acquisition Layer Data Flows

6.1 Description
The Data Acquisition Layer will run on the hardware module, and is mainly composed of a MC9S12C128 Freescale Microcontroller. The Data Acquisition Layer will act as a median between the existing gauge and the Bluetooth layer to acquire the data from the CAN Bus, and to encode/package the data for Bluetooth transfer.

### 6.2 Encode/Package Data: Set Sync Status

#### 6.2.1 Prologue

Set Sync Status will change a flag in the Data Acquisition Layer that is checked before transmitting data, because we cannot send SCI data to Bluetooth Layer if the device is not synced.

#### 6.2.2 Interface

This module will interface with the Notify MCU Module in the Bluetooth Layer and the Package Data Module in the Data Acquisition Layer.

#### 6.2.3 External Data Dependencies

None

#### 6.2.4 Internal Data Dependencies

Depends on high output from BR-4.0LE-2A to IRQ pin. IRQ interrupt will be active low, so we will need a not gate in between the Bluetooth module and the IRQ pin.

#### 6.2.5 Process/Pseudocode

```c
#pragma interrupt_handler enableSyncStatus

void enableSyncStatus()
{
    syncStatus = 1;
}
```

### 6.3 Encode/Package Data: Package Data

#### 6.3.1 Prologue

Since we are just pulling raw data off of the CAN bus line, we need to package it into a standard form so the Data Processing Layer knows what it is reading.

#### 6.3.2 Interface

This module will interface with the Set Sync Status module and the send Data module

#### 6.3.3 External Data Dependencies
6.3.4 Internal Data Dependencies

Raw byte stream representing all sensors is expected.

6.3.5 Process/Pseudocode

```c
void transmitData()
{
    int index = 0;

    // If the bluetooth module isn't synced, don't transmit any data.
    if (syncStatus)
    {
        // Write the 0xFFFF Start Pattern to SCI
        SCI_Write_Hex(START_SEQUENCE);

        // Write the number of sensors to SCI
        SCI_Write_Hex(NUMBER_SENSORS);

        // Loop through all active sensors and write their values to SCI
        for (index = 0; index < NUMBER_SENSORS; index++)
        {
            SCI_Write_Hex(sensors[index]);
        }
    }
}
```

6.4 Encode/Package Data: Send Data

6.4.1 Prologue

Send Data will be responsible for sending the packaged byte stream from the MC9S12C122 MCU to the BR-LE4.0-S2A.

6.4.2 Interface

This module will interface with the Package Data Module in the Data Acquisition Layer, and the Receive from MCU module in the Bluetooth Layer.

6.4.3 External Data Dependencies

None

6.4.4 Internal Data Dependencies
Packaged byte stream representing all sensors is expected.

6.4.5 Process/Pseudocode

```c
void SCI_Write_Hex(unsigned short value)
{
    int i = 0;
    int2U bit = 0;
    int2U transmit = 0; //create a byte of data for transfer
    //Pull up the 2 most bits first
    transmit = (value >> 2);
    //store the transmit byte into the register that SCI reads from.
    SCI0DRL = transmit;

    //Wait for Bluetooth to pull sent data
    while((SCI0SR1 & RDRF) == 0){};

    for(i = 0; i < 2; i++) //Go through lower eight bits
    {
        //Pull the ith bit off the data string
        bit = (data & (1 << i)) >> i;
        transmit += bit * raisePower(2, i); //restore the byte of data
    }

    //store the transmit byte into the register that SCI reads from.
    SCI0DRL = transmit;

    //Wait for Bluetooth to pull sent data
    while((SCI0SR1 & RDRF) == 0){};
}
```

6.5 Read CAN Bus: Get Raw Data

6.5.1 Prologue

Get Raw Data will be responsible for reading the sensor data off of the CAN Bus line.

6.5.2 Interface

This module will interface with the Package Data Module in the Data Acquisition Layer, and the Receive from MCU module in the Bluetooth Layer.

6.5.3 External Data Dependencies

None

6.5.4 Internal Data Dependencies

None
6.5.5 Process/Pseudocode

```c
void CAN_Transmit(CanMessageBufferStruct2U* pBuffer)
{
    INT2U x;
    INT2U reg;
    BOOL done;

    // BSB
    if (aCAN_writeFlag != 0)
    {
        aCAN_writeFlag = 0;
        *aCAN_writeAddress = aCAN_writeData;
    }
    if (aCAN_readFlag != 0)
    {
        aCAN_readFlag = 0;
        aCAN_readData = *aCAN_readAddress;
    }
    // BSB End

    done = FALSE;
    for (x = 0; (x < 40) && (done == FALSE); x++)
    {
        /* Read which buffers are available */
        reg = CANTFLG;
        /* if any TX buffer is available */
        if (reg != 0)
        {
            /* Select the available buffer - note, CANTBSEL only sets
             * the least significant bit. I.e. if buffers 1 and 2
             * are available, then reading CANTFLG returns 0x06.
             * Setting CANTBSEL = 0x06 selects buffer 1. A following read
             * of CANTBSEL would return 0x02. */
            CANTBSEL = reg;

            /* Read back to get lsbit - the actual buffer used
             * so that we can set it for transmission */
            reg = CANTBSEL;

            /* Copy message to transmit buffer */
            memcpy((void*) &CANTXIDR0, pBuffer, 
                    sizeof(CanMessageBufferStruct2U));

            /* Initiate transmission of this buffer */
            CANTFLG = reg;

            // BSB Debug
            aCAN_totalTransmissions0++;
            done = TRUE;
        }
    }
}
```
if (done == FALSE)
{
    aCAN_txOverflowCount++;
}
}
7. Local Storage

Figure 7-1 Local Storage Layer Module Diagrams

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>A saved run to analyze.</td>
</tr>
<tr>
<td>L2</td>
<td>An array containing a list of saved runs.</td>
</tr>
<tr>
<td>L3</td>
<td>User account data for encryption (Object)</td>
</tr>
<tr>
<td>L4</td>
<td>Run data for encryption (Object)</td>
</tr>
<tr>
<td>P11</td>
<td>Folder containing saved run.</td>
</tr>
<tr>
<td>P15</td>
<td>User Account Model containing user information.</td>
</tr>
<tr>
<td>N8</td>
<td>Byte stream of Encrypted User Account Data</td>
</tr>
</tbody>
</table>
7.1 Description

The Local Storage will be responsible for storing real time runs for real time processing. The Local Storage will encompass the temporary storage on the mobile device. The Local Storage will be accessed by the Data Processing Layer directly through the mobile device file system through the Save Data Subsystem. The Local Storage will also be storing user account information on the database along with Cloud Storage which will be responsible for that as well. The main purpose of the Local Storage is not just temporary accumulation of the current race statistics and gauge data for real time processing but also acting as a route for data that is to be stored into the Cloud Storage since Cloud Storage does not interact with Data Processing Layer directly but through the Network and Local Storage Layers. Local Storage is also responsible for communicating with the Data Processing Layer as well as the Network Layer to store stats that might be retrieved at a later time or real-time.

7.2 Device Memory: Run Storage

7.2.1 Prologue:

The Run Storage has the principal functionality of acting as the data storage location (temporary) for the whole system. The Run Storage of Local Storage Layer mainly stores the current race statistics and gauge data for real-time processing and presentation onto the mobile applications. The Run Storage of Local Storage Layer will not run on the cloud server. It will store present runs to be accessed by the user or anyone the user gives access to. One of the main responsibilities of this particular module is Data storage. The main purpose of the Run Storage is not just temporary accumulation of the current race statistics and gauge data for real time processing but also acting as a route for data that is to be stored into the Cloud Storage since Cloud Storage does not interact with Data Processing Layer directly but through the Network and Local Storage Layers. Some of the assumptions made for this part of the system will be that data provided by the Decryption subsystem of the Network Layer is of the right type and format so the data can be easily stored as well as that there is enough memory for the storage of all the data that is to be stored in the database.

7.2.2 Interfaces

The Run Storage of Local Storage Layer will receive data from the decryption Subsystem of the Network Layer as well as the Post Processing and Save Data Subsystems of Data Processing Layer and will store the run on the local storage. The Run Storage of Local Storage Layer sends and receives byte streams comprising statistics from the gauge input or the database to use for either real-time processing.
or storing. Run Storage of Local Storage Layer returns data to Encryption subsystem of Network Layer that was fetched by the Data Processing Layer.

7.2.3 External data dependencies

Data provided by the decryption Subsystem of the Network Layer as well as the Post Processing and Save Data Subsystems of Data Processing Layer is of the right type.

7.2.4 Internal data descriptors

This module receives the particular Byte Stream from Decryption subsystem of Network Layer. The data is then stored into the Device Memory of Local Storage Layer. Also waits for a request by the Data Processing Layer and provides the required information about the runs in the form of byte streams. While the Post Processing Subsystem of the of Data Processing Layer fetches information from the Local Storage, the Save Data Subsystem actually stores data into it in the form of byte streams.

7.2.5 Process/Pseudo code

This module is used mainly for storage. So there are not many additional software operations that are needed but some that are required are listed below.

The method inside Runs Storage module of Local Storage called sendToEncryption() sends data to Encryption subsystem of Network Layer. In the process, the required byte streams are transferred between the two modules.

```java
public void sendToEncryption()
{
// Sends the user account information to Encryption Subsystem of
// Network Layer so it can be authenticated by the Data
// Processing Layer.

// Inserts data into the RunsData Table

INSERT INTO RunsData (RunID, Temp1, Temp2, Temp3, Temp4, Pres1, Pres2, Pres3, Tach1, Tach2, Tach3, Voltage, Oxygen)

// Selects data from RunsData Table

SELECT * FROM RunsData;
```
7.3 Device Memory: User Account Storage

7.3.1 Prologue:

The User Account Storage has the principal functionality of acting as the data storage location for the user account information of the whole system. The User Account Storage does not store the past race statistics and gauge data. The User Account Storage will be part of the local device server. The authorized user will be allowed to access current runs from the Device Memory. Some of the assumptions made for this part of the system will be that user passwords provided by the Data Processing Layer are properly hashed and of the right format so the data can be easily stored.

7.3.2 Interfaces

The User Account Storage will receive data from the Decryption Subsystem of Network Layer and will store the user account information on the local database storage. The User Account Storage sends and receives byte streams comprising statistics from the Data Processing Layer about user accounts to use for authentication.

7.3.3 External data dependencies

Data in the form of user accounts information provided by the Decryption subsystem of the Network Layer is properly hashed. This basically means that it should be of the right format so the data can be easily stored.

7.3.4 Internal data descriptors

The method inside Runs Storage module of Local Storage called sendToEncryption() sends data to Encryption subsystem of Network Layer. In the process, the required byte streams are transferred between the two modules.

7.3.5 Process/Pseudocode

This module is used mainly for storage. So there are not many additional software operations that are needed but some that are required are listed below.

servlet's processRequest function:

protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

response.setContentType("text/html;charset=UTF-8");

try {

// Instantiate login Command
String cmdPath = "DomainLogic.Commands.login";

February 28, 2013

70

BehindtheCurtain Enterprises
The method inside User Accounts Storage module of Local Storage called sendToEncryption() sends data to Encryption subsystem of Network Layer. In the process, the required byte streams are transferred between the two modules.

```java
public void sendToEncryption()
{
    // Sends the user account information to Encryption Subsystem of
    // Network Layer so it can be authenticated by the Data
    // Processing Layer.
}
```
8. Cloud Storage

![Diagram of Cloud Storage Layer Module]

**Figure 8-1 Cloud Storage Layer Module Diagram**

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Run data from cloud server (Object)</td>
</tr>
<tr>
<td>C2</td>
<td>User account data from cloud server (Object)</td>
</tr>
<tr>
<td>N6</td>
<td>Byte stream of Decrypted User Account Data</td>
</tr>
<tr>
<td>N7</td>
<td>Byte stream of Decrypted Run Data</td>
</tr>
</tbody>
</table>

*Table 8-1 Cloud Storage Layer Data Flows*
8.1 Description

The Cloud Storage Layer will be responsible for storing saved runs for later analysis. The storage layer will encompass the database storage on the server. The Cloud Storage Layer will be accessed by the Data Processing Layer through the mobile device file system and through the Network Layer. The Cloud Storage Layer will also store user account information on the database for access by any mobile device with the required applications. The main purpose of the Cloud Storage Layer is accumulation of the previous race statistics and gauge data for future analysis. Cloud Storage Layer is also responsible for communicating with the Network Layer to store records that might be retrieved at a later time or by a different consumer.

8.2 Database: Run Storage

8.2.1 Prologue:

The Run Storage has the principal functionality of acting as the data storage location (long-term) for the whole system. The Run Storage of Cloud Storage Layer mainly stores the past race statistics and gauge data for later analysis and presentation onto the mobile applications. The Run Storage will run on the cloud server. It will store saved runs to be accessed by the user or anyone the user gives access to. One of the main responsibilities of this particular module is Data storage. Some of the assumptions made for this part of the system will be that data provided by the Decryption subsystem of the Network Layer is of the right type and format so the data can be easily stored as well as that there is enough memory for the storage of all the data that is to be stored in the database.

8.2.2 Interfaces

The Run Storage will receive data from the Decryption Subsystem of Network Layer and will store the run on the server database storage. The Run Storage sends and receives byte streams comprising statistics from the gauge input or the database to use for either analytics or storing. Run Storage returns data to Encryption subsystem of Network Layer that was fetched by the Data Processing Layer.

8.2.3 External data dependencies

Data provided by the Decryption subsystem of the Network Layer is of the right type.

8.2.4 Internal data descriptors

Receives the particular Byte Stream from Decryption subsystem of Network Layer through the receiveFromDecryption() method. The data is then stored into the Runs Storage Module of Database of Cloud Storage Layer.

8.2.4 Process/Pseudocode
This module is used mainly for storage. So there are not many additional software operations that are needed but some that are required are listed below.

The method inside Runs Storage module of Cloud Storage called sendToEncryption() sends data to Encryption subsystem of Network Layer. In the process, the required byte streams are transferred between the two modules.

```java
public void sendToEncryption()
{
    // Sends the user account information to Encryption Subsystem of
    // Network Layer so it can be authenticated by the Data
    // Processing Layer.
}
```

------ Inserts data into the RunsData Table

```sql
INSERT INTO RunsData (RunID, Temp1, Temp2, Temp3, Temp4, Pres1, Pres2, Pres3, Tach1, Tach2, Tach3, Voltage, Oxygen)
```

------ Selects data from RunsData Table

```sql
SELECT * FROM RunsData;
```

### 8.3 Database: User Account Storage

#### 8.3.1 Prologue:

The User Account Storage has the principal functionality of acting as the data storage location for the user account information of the whole system. The User Account Storage does not store the past race statistics and gauge data. The User Account Storage will be part of the cloud server. The authorized user will be allowed to access only saved runs from the Cloud Storage but the User Accounts will only be accessed by the system to verify the information and authenticate the user. Some of the assumptions made for this part of the system will be that user passwords provided by the Decryption subsystem of the Network Layer is are properly hashed and of the right format so the data can be easily stored.

#### 8.3.2 Interfaces

The User Account Storage will receive data from the Decryption Subsystem of Network Layer and will store the user account information on the server database storage. The User Account Storage sends and receives byte streams comprising statistics from the Data Processing Layer about user accounts to use for authentication.
8.3.3 External data dependencies

Data in the form of user accounts information provided by the Decryption subsystem of the Network Layer is properly hashed. This basically means that it should be of the right format so the data can be easily stored.

8.3.4 Internal data descriptors

The method inside Runs Storage module of Cloud Storage called sendToEncryption() sends data to Encryption subsystem of Network Layer. In the process, the required byte streams are transferred between the two modules.

8.3.5 Process

This module is used mainly for storage. So there are not many additional software operations that are needed but some that are required are listed below.

servlet's processRequest function:

```java
protected void processRequest(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    try {
        //Instantiate login Command
        String cmdPath = "DomainLogic.Commands.login";

        user usr = (user) cmdFactory.getInstance().getCommand(cmdPath).execute();

        if(usr!=null && usr.isLoggedIn())
        {
            request.getSession(false).setAttribute("ok", usr);
            getServletConfig().getServletContext().getRequestDispatcher("/WEB-INF/main.jsp").forward(request, response);
        }
        else
            response.sendRedirect("/index.jsp");
    } finally {}}"
```

February 28, 2013
servlet in web.xml:

```xml
<servlet>
  <servlet-name>auth</servlet-name>
  <servlet-class>bridge.auth</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>auth</servlet-name>
  <url-pattern>/auth</url-pattern>
</servlet-mapping>
```

The method inside User Accounts Storage module of Cloud Storage called sendToEncryption() sends data to Encryption subsystem of Network Layer. In the process, the required byte streams are transferred between the two modules.

```java
public void sendToEncryption()
{
    // Sends the user account information to Encryption Subsystem of
    // Network Layer so it can be authenticated by the Data
    // Processing Layer.
}
```
9. Network Layer

Figure 9-1 Network Layer Module Diagram
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L3</td>
<td>User account data for encryption (Object)</td>
</tr>
<tr>
<td>L4</td>
<td>Run data for encryption (Object)</td>
</tr>
<tr>
<td>C1</td>
<td>Run data from cloud server (Object)</td>
</tr>
<tr>
<td>C2</td>
<td>User account data from cloud server (Object)</td>
</tr>
<tr>
<td>N1</td>
<td>EncryptedDataPackage (CipherText and Gauge object)</td>
</tr>
<tr>
<td>N2</td>
<td>Byte stream of EncryptedDataPackage</td>
</tr>
<tr>
<td>N3</td>
<td>Decrypted byte stream to be sent for post-processing</td>
</tr>
<tr>
<td>N4</td>
<td>Decrypted byte stream to be stored on cloud server</td>
</tr>
<tr>
<td>N5</td>
<td>Sent gauge information</td>
</tr>
<tr>
<td>N6</td>
<td>Byte stream of Decrypted User Account Data</td>
</tr>
<tr>
<td>N7</td>
<td>Byte stream of Decrypted Run Data</td>
</tr>
<tr>
<td>N8</td>
<td>Byte stream of Encrypted User Account Data</td>
</tr>
<tr>
<td>N9</td>
<td>Byte stream of Encrypted Run Data</td>
</tr>
</tbody>
</table>

Table 9-1 Network Layer Data Flows

9.1 Description

The Network Layer’s main functionality is to move data from the device to the cloud or vice versa depending on what the user would like to do. The Network Layer has four responsibilities:

1.) Transfer data from the device memory to a cloud server.
2.) Transfer data from a cloud server to device memory to send to the Data Processing Layer.
3.) Encrypt user information (account name and password) in preparation for transfer.
4.) Decrypt user information (account name and password) on the server side to be stored on the cloud.

9.2 Encrypt: User Account
9.2.1 Prologue

This module will take a user account. The user account will be encrypted using AES128 encryption algorithm, and packaged with the gauge information (not encrypted).

9.2.2 Interface

This module provides an encrypted user account name and password as well as non-encrypted gauge data to be sent to the Transfer Module for packaging.

9.2.3 External Data Dependencies

None

9.2.4 Internal Data Dependencies

AES128 Encryption Library

9.2.5 Process/Pseudocode

```java
import javax.crypto.Cipher;

public class NetworkInterface{
    private class EncryptedDataPackage{
        private UserAccount userAct;
        private GaugeData gaugeInfo;
        private Cipher cipherText = Cipher.getInstance("AES");
        private byte[] key = key to be used for encryption;
        private SecretKeySpec sks = new SecretKeySpec(key, "AES");
        private byte[] encryptedAccount;
        public EncryptedDataPackage(UserAccount user, GaugeData gauge){
            userAct = user;
            gaugeInfo = gauge;
        }
        public void EncryptData(){
            byte[] dataToSend = userAct.getUsername().getBytes() +
                               userAct.getPassword().getBytes();
            cipherText.init(Cipher.ENCRYPT_MODE, sks);
            encryptedAccount = (byte[]) cipherText.doFinal(dataToSend);
        }
    }
}
```

9.3 Encrypt: Gauge Data

9.3.1 Prologue

This module will combine the non-encrypted gauge data with encrypted user account data to be sent for packaging in the Package User Data Module.

9.3.2 Interface
This module non-encrypted gauge data to the Package User Data module.

9.3.3 External Data Dependencies
None

9.3.4 Internal Data Dependencies
None

9.3.5 Process/Pseudocode
Gauge data is not encrypted but included in this subsystem as it is packaged with the data.

9.4 Transfer Data: Package User Data

9.4.1 Prologue
This module will package the object consisting of an encrypted user account name and password along with the non-encrypted gauge information to send to either local storage or the cloud server.

9.4.2 Interface
This module accepts an object consisting of encrypted user credentials along with gauge information to package together. The packaged object will be sent to the Send Packaged Data module.

9.4.3 External Data Dependencies
EncryptedDataPackage to prepare data for packaging

9.4.4 Internal Data Dependencies
None.

9.4.5 Process/Pseudocode
EncryptedDataPackage edp = new EncryptedDataPackage(UserAccount user, GaugeData gauge);
edp.getByteStreamSize();
edp.packageData(); (convert the data to a readable byte stream to be sent over the Send User Data module)

9.5 Transfer Data: Send User Data

9.5.1 Prologue
This module will take previously packaged input and use TCP client server architecture to send data from mobile device to cloud server and vice versa.

9.5.2 Interface
This module accepts a package from the Package User Data module and pushes it through the network onto the server for decryption.

9.5.3 External Data Dependencies

Wifi, 3G or LTE connection

9.5.4 Internal Data Dependencies

java.net framework

java.lang.Thread

9.5.5 Process/Pseudocode

There will be multiple classes that interface with one another in this module including:

**Server:**

```java
import java.*;
import java.lang.*;

class JavaServer{
    //Constructor should set up the server's IP and port
    private ServerSocket srvSkt;
    //server's ip and port
    private String ipAddr, srvPort;
    public JavaServer(String ip, String port){
        ipAddr=ip;
        srvPort=port;
    }
    //class to constantly check for new connections
    //utilizes java threads incase multiple connections
    public void connect(){
        //set a new server socket to the port to listen for new connections
        srvSkt = new ServerSocket(srvPort);
        //infinite loop waiting on new connections
        while(true){
            //connecting device socket
            Socket incomingSocket = srvSkt.accept();
            //accept the connection in a new thread
            Thread newConnection = new Thread(new connection(incomingSocket));
        }
    }
}
```

**Client:**

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

class Client implements Runnable{
    private Socket skt;
    private String hostIP, port;
    private byte[] data;
    public Client(String host, String hostPort, byte[] dataSent){
        hostIP = host;
    }
    ``
port = hostPort;
data = dataSent;
}

@Override
public void run() {  // abstract method for threads
    // thread will run these methods
    connect();
    receive();
    send();
}

public void connect() {
    // connect to server socket
    skt = new Socket(hostIP, port);
}

public void receive() {
    // read data off of server
    InputStreamReader isr = new InputStreamReader(skt.getInputStream());
    BufferedReader br = new BufferedReader(isr);
    String input = null;
    while ((input = br.readLine()) != null) {
        // take input from buffered reader sent through server here.
    }
    // close connection after pulling what you need
    skt.close();
}

public void send() {
    // will send a byte stream to server to be decrypted and stored
    send dataSent to the server
}

9.6 Decrypt Data: User Account

9.6.1 Prologue

This module will take packaged data sent from the Send Data Module to decrypt and store on the server.

9.6.2 Interface

This module accepts a package from the Send Data Module and decrypts it to send to the Cloud Storage Layer.

9.6.3 External Data Dependencies

Data package from the Send Data Module

9.6.4 Internal Data Dependencies

CipherText key from Encryption: User Data Module

9.6.5 Process/Pseudocode

```java
public byte[] decryptWithmanagedIV(byte[] cipherText, byte[] key) {
    byte[] initialVector = Arrays.copyOfRange(cipherText, 0, 16);
```
byte[] trimmedCipherText = Arrays.copyOfRange(cipherText, 16, cipherText.length);
        return decrypt(trimmedCipherText, key, initialVector);
    }

    public static byte[] decrypt(byte[] cipherText, byte[] key, byte[] initialVector) throws Exception{
        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        SecretKeySpec secretKeySpecy = new SecretKeySpec(key, "AES");
        IvParameterSpec ivParameterSpec = new IvParameterSpec(initialVector);
        cipher.init(Cipher.DECRYPT_MODE, secretKeySpecy, ivParameterSpec);
        cipherText = cipher.doFinal(cipherText);
        return cipherText;
    }

9.6 Decrypt Data: Gauge Data

9.6.1 Prologue

This module will take packaged data sent from the Send Data Module and separate the encrypted user account data from the non-encrypted gauge data which will be processed and sent to the storage layer.

9.6.2 Interface

This module accepts a package consisting of an encrypted user account and a non-encrypted set of gauge data to be sent to the cloud storage layer.

9.6.3 External Data Dependencies

None

9.6.4 Internal Data Dependencies

None

9.6.5 Process/Pseudocode

This module will return the gauge data from the package received from the Send Data Module to be stored on the cloud server.
10. Quality Assurance

10.1 Test Plans and Procedures

The system detailed design shall be tested by Team BehindTheCurtain to verify each requirement specified in the System Requirements Specification Document. Each component, module, subsystem and layer will be designed around a test plan and be tested both independently and as a whole to validate satisfaction of all specifications.

10.2 Module/Unit Tests

10.2.1 GUI Layer

10.2.1.1 Drag & Drop Controller

The Drag & Drop Controller shall allow the user to add and remove widgets from the grid view by drag and dropping them to and from the widget tray. When the user holds their finger on a widget for a couple seconds it shall become editable and moveable by dragging. When the user releases their finger from a widget being dragged it will be placed in it’s current position, only if that grid space is large enough to hold that widget.

10.2.1.2 Page Slider Controller

When the Real-Time or Post Process view is active, the Page Slider Controller shall allow the user to swipe left and right to swap between multiple grid views. The user shall be able to modify each grid view separately. Each grid view will be checked to verify that it’s layout configuration is being saved.

10.2.1.3 Preferences Controller

The Preferences Controller module shall allow the user to open the sensor configuration menu. Within the menu, the user shall be able to add/remove, enable/disable and adjust the calibration of each sensor. Each sensor’s range of calibration shall be within the bounds of that physical sensor.

10.2.1.4 Pop-Out Menu Controller

The Pop-Out Menu Controller module shall be appropriately triggered with specific user input. A user shall swipe left or right with two fingers to verify that the appropriate menu is displayed.
10.2.1.5  Post Process Controller

The Post Process Controller Module shall accurately represent and output the information from the user’s saved and selected run. The rendered charts will be compared with data points in the Data Processing layer to check for consistency.

10.2.1.6  Real-Time Controller

The Real-Time Controller Module shall accurately represent and output the information displayed on the actual racing gauge. The gauge and the mobile device will be paired and checked for consistency between one another.

10.2.1.7  Configuration Controller

The Configuration Controller Module shall accurately represent the configuration chosen by a specific user. The Configuration Controller shall be tested for proper interfacing with the Configuration Model Map Module, User Account Module, and Settings Controller Module. BehindTheCurtain shall verify that the configuration controller accurately represents the input settings received from the other modules.

10.2.1.8  Help View

The Help View Module shall display a static FAQ and instructions about how to run the device. Team BehindTheCurtain shall verify that the information is properly displayed in the correct format.

10.2.1.9  Configuration Model

The Configuration Model Module shall represent the configuration of a single sensor. For quality assurance BehindtheCurtain will write a test case for this particular module to assert that data is stored in the proper and accurately depicts the sensor configuration for various sensors.

10.2.1.10  Configuration Model map

The Configuration Model Map shall be similar to a hash map in data structure and store a map of sensors to their IDs (represented as the key).

10.2.2  Data Processing Layer

10.2.2.1  RealTimeBuilder

The RealTimeBuilder first needs to receive a ConfigurationModelMap to set the configurations needed to set what sensors are active. The sensor data being received will be transformed and then will be verified that tachometer, pressure, temperature, speedometer, oxygen, and voltage are calculated correctly. The ability to start and stop processing mode will also be verified.
Only one instance of GaugeModel should exist at a time. This instance will be tested to verify that the SensorAggregateModels are stored correctly in the sensorAggregateModelMap. Also the object’s ability to be serialized to a file and deserialized from a file will be verified.

10.2.2.3 SensorAggregateModel

SensorAggregateModel will be tested to verify that the properties for the sensor are set correctly. Also the object’s ability to be serialized to a file and deserialized from a file will be verified.

10.2.2.4 SensorSnapshotModel

SensorSnapshotModel will be tested to verify that the properties for the snapshot are set correctly. Also the object’s ability to be serialized to a file and deserialized from a file will be verified.

10.2.2.5 PostBuilder

The ConfigurationModelMap should be set correctly before the PostBuilder can function. Afterwards the GaugeAnalysisModel should be made from the saved runs from file and verify that this data is analyzed correctly.

10.2.2.6 GaugeAnalysisModel

Verify that each SensorAggregateModel from the serialized file is analyzed.

10.2.2.7 SensorAnalysisModel

The ability to calculate the range, mean, mode, and median for the data set will be verified.

10.2.2.8 RunListModel

Verify that correct runs are listed.

10.2.2.9 BLEGaugeAlarmService

Verify that connections are made correctly. Also verify that the data is being transmitted correctly and reliably from the hardware module.

10.2.2.10 GaugeSerializer

Verify that the data is serialized in the correct location.

10.2.3 Bluetooth Layer

10.2.3.1 Sync: Wait for Sync

The BR-LE4.0-S2A shall sync with the Sync Subsystem on the Data Processing Layer to allow wireless data transfer. There will be a sync status flag in the microcontroller which will be high
when sync is active and low when not. Team BehindtheCurtain will verify that the sync status pin is high to pass this stage.

### 10.2.3.2 Sync: Notify MCU

The Notify MCU module will send a high signal to a pin which will output to both an LED and the MC9S12C128 Microcontroller verifying synced status. The LED will verify that the proper signal is being sent and that notifications are working properly.

### 10.2.3.3 Transfer Data: Receive from MCU

The Receive from MCU module will use UART to transmit the data stream. Team BehindtheCurtain will verify data consistency through the UART.

### 10.2.3.3 Transfer Data: Send Data

The Send Data Module will send all packaged data from the BR-LE4.0-S2A over Bluetooth. Team BehindtheCurtain will verify proper data transfer and check for packet loss on the receiving end. Data consistency will also be verified between the MCU and BR-LE4.0-S2A.

### 10.2.4 Data Acquisition Layer

#### 10.2.4.1 Encode/Package Data: Set Sync Status

The Set Sync Status Module will change the sync flag in the Data Acquisition layer to check before attempting to transmit data. Transfers will not proceed if the flag is not set. To verify that the flag is set high there will be an LED indicator.

#### 10.2.4.2 Encode/Package Data: Package Data

The Package Data module will package raw data from the CAN bus line into a standard form. Verification will include checking readability of the sent data packets in the Data Processing Layer.

#### 10.2.4.3 Encode/Package Data: Send Data

The Send Data Module will be responsible for sending a packaged byte stream from the MC9S12C128 to the BR-LE4.0-S2A. Verification will include checking consistency between both hardware components and verifying that sent data is received on each end.

### 10.2.5 Local Storage Layer

#### 10.2.5.1 Device Memory: Runs Storage

The primary function of the Runs Storage module is to interconnect with Network Layer, and accordingly, it is important for the system to ensure that a query is correctly entered and the particular data are outputted as they should be.
Testing technique will be used to make sure what is anticipated to be retrieved from the Runs Storage module is what is actually received. Database entry will also be authenticated to make sure that correct exceptions are prepared when an effort is made to put unacceptable information into the system’s Cloud Storage. Testing will also be done to make sure that the data intended to be retrieved follows the anticipated path and is efficiently acknowledged by the appropriate module.

10.2.5.2 Device Memory: User Accounts Storage

The main purpose of the User Accounts module is to communicate with Network Layer, and accordingly, it is important for the system to ensure that a query is correctly entered and the particular data are outputted as they should be. Testing will also be done for this module to make sure that the data intended to be retrieved follows the anticipated path and is efficiently acknowledged by the appropriate module.

Some methods for testing will be used to make sure what is anticipated to be retrieved from User Accounts module is what is actually received. Database entry will also be authenticated to make sure that correct exceptions are prepared when an effort is made to put unacceptable information into the system’s Cloud Storage.

10.2.6 Cloud Storage Layer

10.2.6.1 Database: Runs Storage

Testing will be done to make sure that the data intended to be retrieved follows the anticipated path and is efficiently acknowledged by the appropriate module. The primary function of the Runs Storage module is to interconnect with Network Layer as well as the Data Processing Layer, and accordingly, it is important for the system to ensure that a query is correctly entered and the particular data are outputted as they should be.

Testing procedures will be used to make sure what is anticipated to be retrieved from the Runs Storage module is what is actually received. Database entry will also be authenticated to make sure that correct exceptions are prepared when an effort is made to put unacceptable information into the system’s Local Storage.

10.2.6.1 Database: User Accounts Storage

Database entry will be authenticated to make sure that correct exceptions are prepared when an effort is made to put unacceptable information into the system’s Local Storage. Testing will be done for this module to make sure that the data intended to be retrieved follows the anticipated path and is efficiently acknowledged by the appropriate module.

The main purpose of the User Accounts module is to communicate with Network Layer as well as the Data Processing Layer, and accordingly, it is important for the system to ensure that a query is correctly entered and the particular data are outputted as they should be.
Some methods for testing will also be used to make sure what is anticipated to be retrieved from User Accounts module is what is actually received.

### 10.2.7 Network Layer

#### 10.2.7.1 Encrypt User Account

The user account shall be properly encrypted using AES128 Encryption Algorithm and packaged with non-encrypted gauge data to be prepared for transfer over the network. We shall develop a test to check whether or not user credentials were successfully converted to Ciphertext.

#### 10.2.7.2 Package User Data

The packaged user data should contain encrypted user credentials along with any non-encrypted gauge data they wish to inject into the cloud. Will develop a test to assert that the packaged values are correct and fail on any exceptions.

#### 10.2.7.3 Send User Data

The Send User Data Module shall be able to received packets from the Package User Data Module which contains both encrypted and non-encrypted data. The Send User Data Module will be tested using a Mock Instance of a server and JUnit testing. Successful transfer must be verified to pass the test.

#### 10.2.7.1 Decrypt User Account

The Ciphertext representation of the user account shall be properly decrypted into a readable format using a specified decryption algorithm. Decrypted data shall be displayed to verify readability.

### 10.3 Integration Testing

#### 10.3.1 GUI Layer

- Verify that user input from the mobile device is be recognized by the right controller.
- Verify that saved and real-time data points from the Data Processing layer are being accurately rendered as charts and graphs.
- Verify that graphical gauges and charts are being updated in real time with no noticeable lag.
- Verify that the user is able to sign into their account and access past runs over both WiFi and Cellular connections.
- Verify that users are able to add and remove sensors from the configuration menu.
- Verify that users are able to modify each individual sensors settings.
o Verify that the user is able to access the Help menu.

### 10.3.2 Data Processing Layer

- Verify the Bluetooth connection sync.
- Verify Bluetooth data transfer reliability.
- Verify that data is sent from the BLEGaugeAlarmService to the RealTimeBuilder in a timely manner.
- Verify that correct configurations for Real Time Processing are received from GUI Layer.
- Verify that the data sent contained by the GaugeModel singleton accessed by the GUI Layer is correct.
- Verify that Real Time Processing models are generated in a timely manner.
- Verify that correct saved run list is able to be generated from Local Storage Layer.
- Verify that correct RunListModel is sent to the GUI Layer.
- Verify that GaugeModel data is serialized correctly in the Local Storage Layer.
- Verify that the saved runs are able to be loaded from the Local Storage Layer.
- Verify that the correct configurations for Post Processing are received from the GUI Layer.
- Verify that the post processed data is calculated correctly before being outputted by GUI Layer.

### 10.3.3 Bluetooth Layer

- Verify the Bluetooth connection sync.
- Verify MCU is properly notified.
- Verify data is received from the MCU
- Verify that packaged data is sent over Bluetooth

### 10.3.4 Data Acquisition Layer

- Verify that sync status is set correctly.
- Verify data is packaged into a readable format.
- Verify that byte stream is reaching from the MCU to the BR-LE4.0-S2A.
- Verify that raw data is being pulled from the CAN Bus
10.3.5 Local Storage Layer

- Verify write operations received are in an accurate format
- Verify reads return the information actually required and to the correct module
- Verify user accounts do not mix with runs data
- Verify decrypted data received from Network Layer
- Verify encrypted data sent to Network Layer for transfer to Cloud Storage
- Verify data received from Data Processing Layer is in right format.

10.3.6 Cloud Storage Layer

- Verify decrypted data received from Network Layer
- Verify encrypted data sent to Network Layer for transfer to Cloud Storage
- Verify write operations received are in an accurate format
- Verify reads return the information actually required and to the correct module
- Verify user accounts do not mix with runs data

10.3.7 Network Layer

- Verify connection to database.
- Verify reliable data transfer.
- Verify encrypted data is represented in ciphertext.
- Verify encrypted data is properly decrypted on the server side.
- Verify data is packaged correctly.

10.4 Test Cases

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Expected Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>User connects Hardware Module to mobile device.</td>
<td>The hardware module LED connection light is lit and the mobile device shows connection to the Hardware Module.</td>
</tr>
<tr>
<td>User sets configuration data for the mobile application.</td>
<td>The changes are made to the GUI.</td>
</tr>
<tr>
<td>User loads mobile application</td>
<td>User configurations are loaded.</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>User enters account information</td>
<td>Account information is saved.</td>
</tr>
<tr>
<td>User begins processing a run.</td>
<td>Gauge sensor data is displayed on the mobile device in real time.</td>
</tr>
<tr>
<td>User stops processing a run.</td>
<td>A saved run folder is generated from the run data.</td>
</tr>
<tr>
<td>User requests to load a run.</td>
<td>A list of available runs is displayed to the user.</td>
</tr>
<tr>
<td>User selects to process a run.</td>
<td>Line graphs and metrics are generated from saved run.</td>
</tr>
<tr>
<td>User pushes run to database.</td>
<td>Run folder is saved on server.</td>
</tr>
</tbody>
</table>

**Table 10-1 Test Cases**
11. Requirement Mapping

The purpose of the Requirement Mapping is to give a general overview of the requirements that our team specified in our System Requirements Specification document, and to demonstrate that our System Architecture fully covers and accounts for all of our requirements.

The following table contains all of our architecture related requirements, and for each specific requirement will dictate the layers that are affected by this requirement. If we have designed our system correctly, we should be able to observe that our system is flexible, and fully covers all of our requirements.
<table>
<thead>
<tr>
<th>Req #</th>
<th>Requirement</th>
<th>Data Acquisition</th>
<th>Bluetooth</th>
<th>Data Processing</th>
<th>GUI Layer</th>
<th>Storage</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Read Data from CAN BUS</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>Profile Each Run</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>Bluetooth Capability</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>Mobile iPhone App</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.6</td>
<td>Microcontroller Data Acquisition</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.7</td>
<td>Microcontroller Packaged Output</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>Interface to CAN BUS</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.9</td>
<td>Configuration Support Page</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>Multi-Gauge Graphical User Interface</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Real-Time Output</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.2</td>
<td>Reliable Data Transfer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>6.4</td>
<td>Multi-threading</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.1</td>
<td>Statistics Database</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>9.2</td>
<td>User Accounts</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>9.3</td>
<td>Encryption of Web Traffic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>9.4</td>
<td>Salt and Hash Passwords</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>9.5</td>
<td>Accurate Gauge Display</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 11-1 Layer Requirements Traceability
## Detailed Design Specification – v2.0

### 4.1 Read Data From CAN BUS
- 4.1.1 Profile Each Run
- 4.1.2 Microcontroller Data Acquisition
- 4.1.3 Bluetooth Capability
- 4.1.4 Mobile iPhone App

### 4.2 Interface to CAN Bus
- 4.2.1 Microcontroller Packed Output
- 4.2.2 Configuration Support Page

### 4.3 Bluetooth Capability
- 4.3.1 Multi-Gauge Graphical User

### 4.4 Mobile iPhone App
- 4.4.1 Multi-Gauge Data Transfer

### 4.5 Microcontroller Packed Output
- 4.5.1 Reliable Data Transfer
- 4.5.2 Multi-threading

### 6.1 Real-Time Output
- 6.1.1 Reliable Data Transfer
- 6.1.2 Multi-threading

### 6.2 Reliable Data Transfer
- 6.2.1 Multi-threading

### 6.4 Multi-threading
- 6.4.1 Reliable Data Transfer

### 9.1 Statistics Database
- 9.1.1 Multi-threading
- 9.1.2 Reliable Data Transfer

### 9.2 User Accounts
- 9.2.1 Multi-threading
- 9.2.2 Reliable Data Transfer

### 9.3 Encryption of Web Traffic
- 9.3.1 Multi-threading
- 9.3.2 Reliable Data Transfer

### 9.4 Salt and Hash Passwords
- 9.4.1 Multi-threading
- 9.4.2 Reliable Data Transfer

### 9.5 Accurate Gauge Display
- 9.5.1 Multi-threading
- 9.5.2 Reliable Data Transfer

### Table

<table>
<thead>
<tr>
<th>Function</th>
<th>4.1</th>
<th>4.2</th>
<th>4.3</th>
<th>4.4</th>
<th>4.5</th>
<th>6.1</th>
<th>6.2</th>
<th>6.4</th>
<th>9.1</th>
<th>9.2</th>
<th>9.3</th>
<th>9.4</th>
<th>9.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get Raw Data</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Set Sync Status</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Package Data</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send Data</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Notify MCU</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wait For Sync</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive from MCU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Send Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLEGaugeAlarmService</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GaugeSerializer</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SensorSnapshotModel</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>RealTimeBuilder</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SensorAggregateModel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GaugeModel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PostBuilder</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RunListModel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>GaugeAnalysisModel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SensorAnalysisModel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Run Storage</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>User Account Storage</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>User Information</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Gauge Information</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------</td>
<td>--------------------</td>
<td>------------------</td>
<td>-------------------</td>
<td>-------------</td>
<td>----------------------</td>
<td>-------------------------</td>
<td>-------------------------</td>
<td>------------------------</td>
<td>-------------------------</td>
<td>-----------------------</td>
<td>----------------------</td>
<td>------------------------</td>
</tr>
</tbody>
</table>
12. Acceptance Plan

12.1 Overview

This section describes the conditions under which all of the stakeholders intend to consider the final product acceptable. The following are the minimal requirements to be satisfied in the actual implementation of the system. These acceptance criteria will be verified in the verification stage of AVALANCHE project. These items are critical to the success to the AVALANCHE project.

12.2 Packaging and Installation

This section describes the packaging and installation requirements for AVALANCHE. These include a user manual, secured box to hold the system, server software, printed circuit board, and application store submissions.

12.2.1 User Manual

The system shall be packaged with a user manual explaining how to use AVALANCHE.

12.2.2 Secured Enclosure

The system shall be packaged with a secured enclosure which shall connect to the CAN Bus and hold the microcontroller which interfaces with the current standing system. Our product will follow the same standards of enclosure as specified by Redline Gauges: RG-ICD001 rev A

12.2.3 Printed Circuit Board

System will be delivered with a printed circuit board loaded with the embedded code to run AVALANCHE.

12.2.4 Server Software

AVALANCHE will include server software to host documented races, users, and possibly statistics.

12.2.4 Mobile Application

AVALANCHE should be made available to both Android market and the Apple iOS App Store.

12.3 Acceptance Testing
The AVALANCHE system shall undergo module, subsystem, layer, integration, and system tests to verify that the system satisfies the acceptance criteria. Details surrounding the specific testing methods and acceptance criteria for each test shall be provided in the System Test Plan document.

### 12.4 Acceptance Criteria

<table>
<thead>
<tr>
<th>Req #</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Read Data from CAN BUS</td>
<td>The Data Acquisition Layer will read data from the gauge through the CAN Bus</td>
</tr>
<tr>
<td>4.2</td>
<td>Profile Each Run</td>
<td>The AVALANCHE mobile application will be able to profile save runs.</td>
</tr>
<tr>
<td>4.3</td>
<td>Bluetooth Capability</td>
<td>The hardware module and mobile application shall be able to communicate via Bluetooth.</td>
</tr>
<tr>
<td>4.4</td>
<td>Mobile iPhone App</td>
<td>There shall be a mobile AVALANCHE iOS application.</td>
</tr>
<tr>
<td>4.6</td>
<td>Microcontroller Data Acquisition</td>
<td>The hardware module microcontroller shall acquire gauge data from the CAN Bus.</td>
</tr>
<tr>
<td>4.7</td>
<td>Microcontroller Packaged Output</td>
<td>The Data Acquisition Layer will package the sensor data for Bluetooth transmittal.</td>
</tr>
<tr>
<td>4.8</td>
<td>Interface to CAN BUS</td>
<td>The hardware module will have an interface to the CAN BUS.</td>
</tr>
<tr>
<td>4.9</td>
<td>Configuration Support Page</td>
<td>There will be a page in the mobile application where the user will be able to configure the application.</td>
</tr>
<tr>
<td>4.10</td>
<td>Multi-Gauge Graphical User Interface</td>
<td>The GUI Layer will be able to display multiple sensors at once.</td>
</tr>
<tr>
<td>6.1</td>
<td>Real-Time Output</td>
<td>The mobile application will be capable of displaying gauge data in real time.</td>
</tr>
<tr>
<td>6.2</td>
<td>Reliable Data Transfer</td>
<td>Data transfer over the internet will be reliable with minimal data loss.</td>
</tr>
<tr>
<td>9.1</td>
<td>Statistics Database</td>
<td>A database will be used to store runs in the cloud.</td>
</tr>
<tr>
<td>9.2</td>
<td>User Accounts</td>
<td>Users shall be able to create user accounts.</td>
</tr>
<tr>
<td></td>
<td>Encryption of Web Traffic</td>
<td>User information should be encrypted for transmittal over the internet.</td>
</tr>
<tr>
<td>---</td>
<td>--------------------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>9.4</td>
<td>Salt and Hash Passwords</td>
<td>User passwords shall be salted and hashed.</td>
</tr>
<tr>
<td>9.5</td>
<td>Accurate Gauge Display</td>
<td>The data being displayed on the AVALANCHE application should be as accurate as the existing gauge.</td>
</tr>
</tbody>
</table>

Table 12-1 Acceptance Criteria
13. Hardware

13.1 Overview

The purpose of this section is to describe all of the hardware parts that will be used in the Avalanche project. For each component, we will describe its purpose, specifications, and interfaces.

13.2 MC9S12C128 Microcontroller

![MC9S12C128 Microcontroller Diagram]

**Figure 13-1 MC9S12C128 Microcontroller**

13.2.1 Purpose

The purpose of the MC9S12C128 is to interface with the CAN Bus line coming from the existing gauges RSM module. It will take this data, package it, and send it to the BR-LE4.0-S2A Bluetooth Chip.

13.2.1 Specifications
13.2.3 Interfaces

The MC9S12C128 will interface with two different voltage levelers, so as to transfer and receive messages from the BR-LE4.0-S2A Bluetooth module.

13.3 SN74LVC4245 Voltage Leveler

13.3.1 Purpose

The MC9S12C128 will be fed a 5 volt signal input, but the BR-LE4.0-S2A will require a 3.3 input voltage. The SN74LVC4245 will either take a 5V signal down to a 3.3V signal, or a 3.3V signal up to a 5V signal.

13.3.2 Specifications

- 1 5V input
- 2 3.3V input
- 8 voltage leveler inputs and outputs

13.3.3 Interfaces
The two SN74LVC4245 Voltage levelers will sit in between the BR-LE4.0-S2A Bluetooth chip, and the MC9S12C128 Microcontroller, and will adjust all signals fed to the devices.

13.4 BR-LE4.0-S2A Bluetooth Module

13.4.1 Purpose

The purpose of the BR-LE4.0-S2A Bluetooth chip will be to send all of the data that is collected on the MC9S12C128 Microcontroller to the iPhone device.

13.4.2 Specifications

- UART(2 wire or 4 wire with CTS/RTS, 9600 to 460.8K Baud Rate)
- Integrated AT.s command stack for external control via UART
- Low Power consumption: 27mA TX, RX 19.6mA, .9uA sleep
- Free Apple iPhone and Android Libraries and applications
13.5 Schematic Overview

Figure 13-4 Schematic Overview
Appendix

List of Acronyms

BLE- Bluetooth Low Energy 4.0
CAN Bus- Controller Area Network bus
iOS- IPhone Operating System
MCU- Microcontroller Unit (MC9S12C128)
RSM- Redline Gauges Remote Sensor Module
SCI- Serial Communications Interface
  TX- Transfer
  RX- Receive
  RTS- Request to Send
  CTS- Clear to Send
UART- Universal Asynchronous Receiver/Transmitter