Department of Computer Science and Engineering
The University of Texas at Arlington

Chronos
Environmentally Responsive Smart Clock
Detailed Design Specification

Team Members:
Benjamin Cahill
Shane Hanlon
Cristobal Rodriguez
Alex Salazar
Jorge Zavala

Last Updated: July 2, 2014 9:03 PM
### 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><strong>1.0</strong></td>
<td>6-23-2014</td>
<td>DDS Gate Review Draft</td>
<td>Initial offering of the DDS.</td>
</tr>
<tr>
<td><strong>2.0</strong></td>
<td>7-2-2014</td>
<td>DDS Baseline</td>
<td>Edited with gate review feedback.</td>
</tr>
</tbody>
</table>
# Table of Contents

Document Revision History ................................................................................................................. 2
Table of Contents .................................................................................................................................. 3
List of Figures ......................................................................................................................................... 6
List of Tables .......................................................................................................................................... 7
1. Introduction .................................................................................................................................... 8
   1.1 Purpose and Use ........................................................................................................................ 8
   1.2 Project Description ................................................................................................................... 8
2. Architecture Overview ..................................................................................................................... 10
   2.1 Application Input Layer .......................................................................................................... 11
   2.2 Hardware Input Layer .............................................................................................................. 11
   2.3 Data Processing Layer ............................................................................................................. 12
   2.4 File Storage Layer .................................................................................................................. 12
   2.5 Output Layer ........................................................................................................................... 13
   2.6 Module Decomposition .......................................................................................................... 14
   2.7 Module Producer-Consumer Matrix ...................................................................................... 19
3. System Hardware Description ...................................................................................................... 22
   3.1 Raspberry Pi Model B Revision 2 (512MB) ............................................................................ 22
   3.2 Tontec 7” LCD Screen ........................................................................................................... 24
   3.3 HP Flat Panel Speaker Bar (Model no. EE418AA) .............................................................. 26
   3.4 TEA5767 FM Radio Receiver Breakout Board .................................................................. 27
   3.5 GL5528 Light Dependent Resistor (Light Sensor) ............................................................. 28
   3.6 DS18B20 Digital Temperature Sensor .................................................................................. 30
   3.7 Microphone ............................................................................................................................. 31
   3.8 Generic Tactile Push Button .................................................................................................. 32
   3.9 Transcend 16GB SDHC Memory Card ................................................................................... 33
   3.10 Cirago USB Bluetooth™/Wi-Fi Combo Adapter ............................................................... 34
   3.11 Conceptual Hardware Diagram ............................................................................................ 36
4. System Software Description ...................................................................................................... 37
   4.1 ERSC Software ....................................................................................................................... 37
   4.2 Android Application ................................................................................................................ 37
Detailed Design Specification

5. Application Input Layer ......................................................................................................................... 38
   5.1 Event Handler ................................................................................................................................. 38
   5.2 Message Formatter .......................................................................................................................... 41
5. Hardware Input Layer ............................................................................................................................ 46
   6.1 Hardware Event ............................................................................................................................... 46
   6.2 Hardware Poller ............................................................................................................................... 50
   6.3 Data Packaging ............................................................................................................................... 54
6. Data Processing Layer ........................................................................................................................... 56
   7.1 Processing Controller ......................................................................................................................... 56
   7.2 Application Data Analyzer ................................................................................................................ 72
   7.3 API Request Handler ......................................................................................................................... 75
7. File Storage Layer ................................................................................................................................... 83
   8.1 File Formatter .................................................................................................................................... 83
   8.2 File System Manager ......................................................................................................................... 89
8. Output Layer ............................................................................................................................................ 93
   9.1 Output Manager ................................................................................................................................ 93
   9.2 Output Formatter ................................................................................................................................ 96
9. Requirements Mapping ............................................................................................................................ 103
   10.1 Application Input Layer .................................................................................................................. 103
   10.2 Hardware Input Layer ..................................................................................................................... 104
   10.3 Data Processing Layer .................................................................................................................... 105
   10.4 File Storage Layer .......................................................................................................................... 106
   10.5 Output Layer .................................................................................................................................... 107
10. Acceptance Plan ...................................................................................................................................... 108
   11.1 Acceptance Criteria ......................................................................................................................... 108
   11.2 Acceptance Testing ......................................................................................................................... 110
   11.3 Packaging and Installing ................................................................................................................ 110
11. Appendix ................................................................................................................................................ 111
   12.1 I2C-Tools ......................................................................................................................................... 111
   12.2 Python-SMBus ................................................................................................................................ 111
   12.3 Jasper ............................................................................................................................................... 111
   12.4 RPi.GPIO ......................................................................................................................................... 111
12.5  Interface Table Legend.................................................................................................................. 111
## List of Figures

<table>
<thead>
<tr>
<th>Figure #</th>
<th>Title</th>
<th>Page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>General System Diagram</td>
<td>9</td>
</tr>
<tr>
<td>2-1</td>
<td>Architecture Diagram</td>
<td>10</td>
</tr>
<tr>
<td>2-2</td>
<td>Detailed Architecture Design</td>
<td>14</td>
</tr>
<tr>
<td>3-1</td>
<td>Raspberry Pi Model B rev. 2</td>
<td>22</td>
</tr>
<tr>
<td>3-2</td>
<td>Tontec 7” LCD Screen</td>
<td>24</td>
</tr>
<tr>
<td>3-3</td>
<td>HP Flat Panel Speaker Bar</td>
<td>26</td>
</tr>
<tr>
<td>3-4</td>
<td>TEA5767 Breakout Board</td>
<td>27</td>
</tr>
<tr>
<td>3-5</td>
<td>GL5528 Light Dependent Resistor</td>
<td>28</td>
</tr>
<tr>
<td>3-6</td>
<td>DS18B20 Temperature Sensor</td>
<td>30</td>
</tr>
<tr>
<td>3-7</td>
<td>Microphone</td>
<td>31</td>
</tr>
<tr>
<td>3-8</td>
<td>Generic Tactile Push Button</td>
<td>32</td>
</tr>
<tr>
<td>3-9</td>
<td>Transcend 16GB SDHC Memory Card</td>
<td>33</td>
</tr>
<tr>
<td>3-10</td>
<td>Cirago USB Bluetooth™/Wi-Fi Combo Adapter</td>
<td>34</td>
</tr>
<tr>
<td>3-11</td>
<td>Conceptual Hardware Diagram</td>
<td>36</td>
</tr>
<tr>
<td>5-1</td>
<td>GUI Listener Module</td>
<td>38</td>
</tr>
<tr>
<td>5-2</td>
<td>Event Classifier Module</td>
<td>41</td>
</tr>
<tr>
<td>5-3</td>
<td>Bluetooth™ Client Module</td>
<td>44</td>
</tr>
<tr>
<td>6-1</td>
<td>PyShutdown Module</td>
<td>46</td>
</tr>
<tr>
<td>6-2</td>
<td>Jasper Module</td>
<td>48</td>
</tr>
<tr>
<td>6-3</td>
<td>Light Sensor Module</td>
<td>50</td>
</tr>
<tr>
<td>6-4</td>
<td>Temperature Sensor Module</td>
<td>52</td>
</tr>
<tr>
<td>6-5</td>
<td>Sensor Data Packager Module</td>
<td>54</td>
</tr>
<tr>
<td>7-1</td>
<td>Application Request Processor Module</td>
<td>56</td>
</tr>
<tr>
<td>7-2</td>
<td>Hardware Event Processor Module</td>
<td>59</td>
</tr>
<tr>
<td>7-3</td>
<td>Hardware Sensor Processor Module</td>
<td>62</td>
</tr>
<tr>
<td>7-4</td>
<td>API Data Processor Module</td>
<td>65</td>
</tr>
<tr>
<td>7-5</td>
<td>Request Handler Module</td>
<td>69</td>
</tr>
<tr>
<td>7-6</td>
<td>Bluetooth™ Server Module</td>
<td>72</td>
</tr>
<tr>
<td>7-7</td>
<td>Weather API Module</td>
<td>75</td>
</tr>
<tr>
<td>7-8</td>
<td>Traffic API Module</td>
<td>78</td>
</tr>
<tr>
<td>7-9</td>
<td>Calendar API Module</td>
<td>80</td>
</tr>
<tr>
<td>8-1</td>
<td>SD Card Formatter Module</td>
<td>83</td>
</tr>
<tr>
<td>8-2</td>
<td>Data Request Formatter Module</td>
<td>86</td>
</tr>
<tr>
<td>8-3</td>
<td>Data Storage Module</td>
<td>89</td>
</tr>
<tr>
<td>8-4</td>
<td>Data Retrieval Module</td>
<td>91</td>
</tr>
<tr>
<td>9-1</td>
<td>Dispatcher Module</td>
<td>93</td>
</tr>
<tr>
<td>9-2</td>
<td>Radio Formatter Module</td>
<td>96</td>
</tr>
<tr>
<td>9-3</td>
<td>Speaker Formatter Module</td>
<td>98</td>
</tr>
<tr>
<td>9-4</td>
<td>Display Formatter Module</td>
<td>100</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>Module Producer-Consumer Table</td>
<td>20</td>
</tr>
<tr>
<td>2-2</td>
<td>Module Producer-Consumer Table Cont.</td>
<td>21</td>
</tr>
<tr>
<td>3-1</td>
<td>Raspberry Pi Specifications</td>
<td>23</td>
</tr>
<tr>
<td>3-2</td>
<td>Tontec 7” LCD Screen Specifications</td>
<td>25</td>
</tr>
<tr>
<td>3-3</td>
<td>HP Flat Panel Speaker Bar Specifications</td>
<td>26</td>
</tr>
<tr>
<td>3-4</td>
<td>TEA5767 Breakout Board Specifications</td>
<td>27</td>
</tr>
<tr>
<td>3-5</td>
<td>GL5528 Light Dependent Resistor Specifications</td>
<td>28</td>
</tr>
<tr>
<td>3-6</td>
<td>DS18B20 Temperature Sensor Specifications</td>
<td>30</td>
</tr>
<tr>
<td>3-7</td>
<td>Microphone Specifications</td>
<td>31</td>
</tr>
<tr>
<td>3-8</td>
<td>Tactile Push Button Specifications</td>
<td>32</td>
</tr>
<tr>
<td>3-9</td>
<td>Transcend 16GB SDHC Memory Card Specifications</td>
<td>33</td>
</tr>
<tr>
<td>3-10</td>
<td>Cirago USB Bluetooth™/Wi-Fi Combo Adapter Specifications</td>
<td>35</td>
</tr>
<tr>
<td>5-1</td>
<td>GUI Listener Interfaces</td>
<td>38</td>
</tr>
<tr>
<td>5-2</td>
<td>Event Classifier Interfaces</td>
<td>41</td>
</tr>
<tr>
<td>5-3</td>
<td>Bluetooth™ Client Interfaces</td>
<td>44</td>
</tr>
<tr>
<td>6-1</td>
<td>Py Shutdown Interfaces</td>
<td>47</td>
</tr>
<tr>
<td>6-2</td>
<td>Jasper Interfaces</td>
<td>48</td>
</tr>
<tr>
<td>6-3</td>
<td>Light Sensor Interfaces</td>
<td>50</td>
</tr>
<tr>
<td>6-4</td>
<td>Temperature Sensor Interfaces</td>
<td>52</td>
</tr>
<tr>
<td>6-5</td>
<td>Sensor Data Packager Interfaces</td>
<td>54</td>
</tr>
<tr>
<td>7-1</td>
<td>Application Request Processor Interfaces</td>
<td>57</td>
</tr>
<tr>
<td>7-2</td>
<td>Hardware Event Processor Interfaces</td>
<td>59</td>
</tr>
<tr>
<td>7-3</td>
<td>Hardware Sensor Processor Interfaces</td>
<td>62</td>
</tr>
<tr>
<td>7-4</td>
<td>API Data Processor Interfaces</td>
<td>65</td>
</tr>
<tr>
<td>7-5</td>
<td>Request Handler Interfaces</td>
<td>70</td>
</tr>
<tr>
<td>7-6</td>
<td>Bluetooth™ Server Interfaces</td>
<td>73</td>
</tr>
<tr>
<td>7-7</td>
<td>Weather API Interfaces</td>
<td>75</td>
</tr>
<tr>
<td>7-8</td>
<td>Traffic API Interfaces</td>
<td>78</td>
</tr>
<tr>
<td>7-9</td>
<td>Calendar API Interfaces</td>
<td>80</td>
</tr>
<tr>
<td>8-1</td>
<td>SD Card Formatter Interfaces</td>
<td>83</td>
</tr>
<tr>
<td>8-2</td>
<td>Data Request Formatter Interfaces</td>
<td>86</td>
</tr>
<tr>
<td>8-3</td>
<td>Data Storage Interfaces</td>
<td>89</td>
</tr>
<tr>
<td>8-4</td>
<td>Data Retrieval Interfaces</td>
<td>91</td>
</tr>
<tr>
<td>9-1</td>
<td>Dispatcher Interfaces</td>
<td>94</td>
</tr>
<tr>
<td>9-2</td>
<td>Radio Formatter Interfaces</td>
<td>96</td>
</tr>
<tr>
<td>9-3</td>
<td>Speaker Formatter Interfaces</td>
<td>98</td>
</tr>
<tr>
<td>9-4</td>
<td>Display Formatter Interfaces</td>
<td>100</td>
</tr>
<tr>
<td>10-1</td>
<td>Requirements Mapping for Application Input Layer</td>
<td>103</td>
</tr>
<tr>
<td>10-2</td>
<td>Requirements Mapping for Hardware Input Layer</td>
<td>104</td>
</tr>
<tr>
<td>10-3</td>
<td>Requirements Mapping for Data Processing Layer</td>
<td>105</td>
</tr>
<tr>
<td>10-4</td>
<td>Requirements Mapping for File Storage Layer</td>
<td>106</td>
</tr>
<tr>
<td>10-5</td>
<td>Requirements Mapping for Output Layer</td>
<td>107</td>
</tr>
</tbody>
</table>
1. Introduction

This Detailed Design Specification document will provide an in-depth and thorough analysis of the Environmentally-Responsive Smart Clock system. It will expand upon the system’s various layers, subsystems, interfaces, and data flows, all of which were previously introduced in the Architecture Design Specification document. In addition to this, it will provide breakdown of each subsystem into several unique modules. All of the specific details necessary to build the system from start to finish will be included in this document. The document will begin by introducing the product concept, definition, and system overview. It will then elaborate on the system’s overall architecture and proceed to outline the details of the various hardware peripherals required for the system’s construction. Following this, all of the subsystem modules residing within each layer will be thoroughly explained. The document will conclude by outlining the requirements mapping approach taken to ensure that all requirements are met accordingly.

1.1 Purpose and Use

The purpose of the Environmentally-Responsive Smart Clock (ERSC) is to provide its users with a centralized location for some of the day’s most critical pieces of information. It aims to simplify an individual’s daily routine by maintaining and providing weather, traffic, and calendar event data. The smart clock’s users will have access to several of the features found on most alarm clocks and will be given access to a plethora of information regarding the day’s events. Users of the ERSC will have the ability to synchronize their native Google™ Calendars with the device to allow for automatic calendar event alerts and reminders. The will also have the ability to select key destinations to allow for traffic monitoring. Overall, the ERSC will offer its users a more convenient and intuitive way to plan and simplify their daily routines.

1.2 Project Description

The Environmentally-Responsive Smart Clock system will consist of two major components: the physical clock itself and an Android application that will be paired with the device. Physically, the smart clock will be composed of an LCD screen and a plastic enclosure that will house several of its hardware components. The ERSC will be equipped with the necessary hardware to connect to the internet and retrieve weather, traffic, and time data. It will use this data to provide the user with relevant traffic and weather conditions. The device will also have the ability to listen for, and interpret, several voice commands (as detailed in the system requirements specification document). More importantly, the ERSC will be able to retrieve an individual’s calendar data and use that information to auto-adjust alarms and produce relevant event alerts. These and other features make the ERSC a vital asset for anyone needing more organization in their daily routine.

The accompanying Android application will serve as the main interface to the smart clock. Its main function will be to provide the ERSC’s users with an elegant and intuitive interface through which they can control the device. The application will contain menus for setting and editing alarms, creating weather-dependent alarm playlists, adding key destinations, playing the radio, and much more. The application will push all of this data to the ERSC so that it can perform its functions properly. Both the application and the ERSC will integrate seamlessly to provide users with an abundance of useful features and information, and to improve a person’s day to day efficiency.
Figure 1-1: General System Diagram
2. Architecture Overview

This section of the document provides a general overview of the ERSC’s system architecture. The system consists of a total of five layers: Application Input Layer, Hardware Input Layer, Data Processing Layer, File Storage Layer, and System Output Layer. These five layers and their corresponding subsystems will be briefly covered in the following subsections of the document. Figure 2-1 illustrates the overall system architecture that includes both the layers and their respective subsystems.

![Architecture Diagram](image_url)
2.1 Application Input Layer

The purpose of the Application Input Layer is to accept input from the user and to trigger events that correspond with the needs of the user. This will serve as the main source of input from the user into the system. The layer listens for user generated events from the application and if an action needs to be performed by the system, the layer will communicate the required task to the Data Processing Layer.

2.1.1 Event Handler Subsystem

The Event Handler will accept inputs from the user through the Android™ application. This subsystem will determine what type of message needs to be delivered to the rest of the system in order to satisfy the request or action that was received from the user.

2.1.2 Message Formatter Subsystem

The Message Formatter will accept input from the Event Handler. The formatter will then pre-process any required data and translate it into a form that is acceptable for the rest of the system. Once the data has been formatted properly, it is sent to Application Data Analyzer for the next step in creating the proper response to the user's interaction.

2.2 Hardware Input Layer

The purpose of the Hardware Input Layer is to accept data from the hardware peripherals such as the microphone, the light sensor, the temperature sensor, and the push button. Hardware input is divided into two groups, event based and polling. Event based input consists of the push button and the microphone. Event base input is sent directly to the Data Processing Layer. Polling input consists of the light sensor and the temperature sensor. Polled data is packaged into an object before it is sent out to the Data Processing Layer.

2.2.1 Hardware Event Subsystem

The Hardware Event Subsystem is designed to interface with all of the event or interrupt driven hardware peripherals. It will listen for and detect an event generated by either the microphone or the push button, both of which are event based devices. The subsystem then proceeds to decipher what specific hardware event occurred and relays the correct information down to the Data Processing Layer.

2.2.2 Hardware Polling Subsystem

The Hardware Polling Subsystem will interface directly with all of the external sensors attached to the system: These sensors include both the temperature sensor and the light sensor. The main function of this subsystem is to continuously poll for the status of each of the sensors and send the acquired sensor data to the neighboring Data Packaging Subsystem.
2.2.3 **Data Packaging Subsystem**

The Data Packaging subsystem accepts both light and temperature sensor readings from the Light Sensor and Temperature Sensor modules respectively. It encapsulates these two pieces of data in an object and sends the sensor status object down to the Data Processing Layer.

2.3 **Data Processing Layer**

The purpose of the Data Processing Layer is to accept data that has been gathered through the input layers. Data is received over the Bluetooth™ connection from the Android™ application and the resulting object is sent on to the Processing Controller. Hardware input coming from the Hardware Input Layer is routed directly to the Processing Controller. This layer also contains a component to deal with API requests to the Internet as well as their responses.

2.3.1 **Application Data Analyzer Subsystem**

The Application Data Analyzer gathers data sent from the Application Input Layer and decodes/interprets this data. The analyzer will use the gathered data and determine the next action that must be taken. This information is then sent to the Processing Controller where action will be taken to respond to the initial request from the user’s application.

2.3.2 **API Request Handler Subsystem**

The API Request Handler supplies the Processing Controller with external API data. This component serves simply as an interface between the system and any external network services.

2.3.3 **Processing Controller Subsystem**

The Processing Controller accepts input from the Application Data Analyzer subsystem and both the Hardware Event and Data Packaging subsystems. It utilizes the input data acquired from these input sources to execute an appropriate system response. This subsystem has the ability to interact with the API Request Handler to send and receive information from an external source (i.e. the Internet). It also interfaces directly with the File Storage Layer and Output Layer.

2.4 **File Storage Layer**

The purpose of the File Storage Layer is to store or retrieve data for the Data Processing Layer. The components inside this layer will facilitate saving or retrieving data as necessary. Data can flow in or out of this layer. Data flowing in from the Data Processing Layer will be converted to a more suitable format for being saved on the file system. Data flowing out will be converted from its file format to a data structure that the Data Processing Layer can use.
2.4.1 File Storage Converter Subsystem

The File Storage Converter takes data from the Data Processing Layer and converts from a data structure that is easy for the program to understand to a file format that can be passed on to the File System Manager to store on the physical disk. It can also request and convert data from a file format to a data structure should the Data Processing Layer need to access data that has been saved previously.

2.4.2 File System Manager Subsystem

The File System Manager has the job of locating space on the physical storage device and allocating a sufficient amount to save files that have been converted by the File Storage Converter. It is also responsible for locating files that have been saved on the physical storage device and retrieving them to be converted and returned for processing.

2.5 Output Layer

The purpose of the Output Layer is to present information to the user. This could be via the display or the speakers. Data to be outputted is sent from the Data Processing Layer to the Output Layer and is then routed to the correct output formatter based on the data received. After the data has been formatted to the correct medium it is presented to the user via that channel.

2.5.1 Output Manager Subsystem

The Output Manager’s job is to receive all data to be outputted from the Data Processing Layer and route each piece to the Output Data Formatter subsystem based on the expected output location.

2.5.2 Output Data Formatter Subsystem

The Output Data Formatter receives various pieces of data from the Output Manager. Each piece corresponds to a specific output so the Output Data Formatter can prepare the data to be outputted correctly. Once formatted, the data is sent on to its respective output device.
2.6 Module Decomposition

This subsection introduces the architecture module breakdown and provides a high level description of each subsystem module. A more detailed explanation of each module will be given in later sections of the document. The figure below illustrates the detailed decomposition of the system architecture.

Figure 2-2: Detailed Architecture Design
Application Input Layer

Event Handler Subsystem

2.6.1 GUI Listener

The GUI Listener module listens for button presses or other type of user interaction with the Android™ application.

Message Formatter Subsystem

2.6.2 Event Classifier

The Event Classifier analyzes the type of event that the user generated via the application and proceeds to compose a message object to send to the ERSC.

2.6.3 Bluetooth™ Client

The Bluetooth™ Client is responsible for connecting to the ERSC via Bluetooth™ and sending any required data or commands.

Hardware Input Layer

Hardware Event Subsystem

2.6.4 PyShutdown

The PyShutdown module waits for an event to be generated by the system’s push button. It serves as a direct interface to the button.

2.6.5 Jasper

The Jasper module waits for a microphone or voice input event to occur and determines what type of voice command was received.

Hardware Polling Subsystem

2.6.6 Light Sensor

The Light Sensor module polls for a light sensor reading to be used by the system.

2.6.7 Temperature Sensor

The Temperature Sensor module polls the temperature sensor for a temperature reading to be used by the system.
Data Packaging Subsystem

2.6.8 Sensor Data Packager

The Sensor Data Packager module receives the acquired temperature and light sensor readings and packages them in an object/structure before they are passed down to the Data Processing Layer.

Data Processing Layer

Application Data Analyzer Subsystem

2.6.9 Bluetooth™ Server

The Bluetooth™ Server runs on the ERSC itself and is continuously listening for data or commands coming in from the Android™ application.

API Request Handler Subsystem

2.6.10 Weather API

The Weather API module focuses on requesting and receiving weather data for system use.

2.6.11 Traffic API

The Traffic API module focuses on requesting and receiving route traffic data for system use.

2.6.12 Calendar API

The Calendar API module focuses on requesting and receiving the user’s calendar events for system use.

Processing Controller Subsystem

2.6.13 Application Request Processor

The Application Request Processor services the commands or requests received by the Android™ application.

2.6.14 API Data Processor

The API Data Processor periodically analyzes weather, traffic, and calendar data and determines what, if any, action should be taken by the system in response to these external conditions.
2.6.15 **Hardware Event Processor**

The Hardware Event Processor is notified of any events or interrupts generated by the push button or microphone and responds to these events.

2.6.16 **Hardware Sensor Processor**

The Hardware Sensor Processor receives and analyzes the external sensor readings and determines what, if any, action must be taken in response to the readings.

2.6.17 **Request Handler**

The Request Handler receives requests for data storage, data retrieval, or data output. It routes these requests to the appropriate module.

**File Storage Layer**

**File Storage Converter Subsystem**

2.6.18 **SD Card Formatter**

The SD Card Formatter formats any data that needs to be stored on the SD card.

2.6.19 **Data Request Formatter**

The Data Request Formatter focuses on encapsulating requested or stored SD card data into an object that can be used by the Data Processing Layer.

**File System Manager Subsystem**

2.6.20 **Data Storage**

The Data Storage module is responsible for taking formatted data and storing it in the SD card.

2.6.21 **Data Retrieval**

The purpose of the Data Retrieval module is to retrieve any desired data from the SD card.

**Output Layer**

**Output Manager Subsystem**

2.6.22 **Dispatcher**

The Dispatcher module will analyze data sent for output and route the data to its appropriate formatter before it is output.
Output Data Formatter Subsystem

2.6.23 Speaker Formatter

This module will receive data that must be output through the speakers and format the data accordingly.

2.6.24 Radio Formatter

The Radio Formatter will perform conversion operations regarding frequency to output to the radio receiver.

2.6.25 Display Formatter

The Display Formatter will take any data that needs to be displayed onto the ERSC’s screen and format it accordingly (i.e. appropriate font, size, etc.).
2.7 Module Producer-Consumer Matrix

The following table illustrates the data flows occurring between the producer and consumer modules. Modules with a large number of incoming or outgoing data flows may indicate that it is more complex. This may also show that the particular module is a critical component of the system or that it is overloaded. In our case, the Request Handler module, with its large number of producer and consumer data flows, gives us the impression that it is a critical component. Another module that has an abundant amount of data flows coming in and out of it is the API Data Processor. Since we request multiple types of data from the Internet, we also receive multiple responses which results in a large number of input and output data flows. Other than the two modules mentioned the rest of the system is reasonably well balanced in terms of data flows.
## Table 2-1: Module Producer-Consumer Table

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
<td>H</td>
<td></td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td></td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></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>
<td></td>
<td></td>
<td>H</td>
<td></td>
<td>H</td>
<td>1</td>
<td>H</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</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>
<td></td>
<td></td>
<td>H</td>
<td></td>
<td>3</td>
<td></td>
<td>H</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</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>
<td></td>
<td></td>
<td>H</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</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>
<td></td>
<td></td>
<td>H</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</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>
<td></td>
<td></td>
<td>H</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</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>
<td></td>
<td></td>
<td>H</td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</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>
<td></td>
<td></td>
<td>H</td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</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>
<td></td>
<td></td>
<td>H</td>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</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>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</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>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</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>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</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>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</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>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</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>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</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>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</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>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</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>
<td></td>
<td></td>
<td>F</td>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</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>
<td></td>
<td></td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

**July 2, 2014**

Chronos
Due to the amount of modules the producer-consumer table had to be broken up slightly. The following is the remaining part.

<table>
<thead>
<tr>
<th></th>
<th>Producer</th>
<th>Dispatcher</th>
<th>Radio Formatter</th>
<th>Speaker Formatter</th>
<th>Display Formatter</th>
<th>Speakers (HW)</th>
<th>Radio (HW)</th>
<th>Display (HW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispatcher</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Radio Formatter</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speaker Formatter</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Display Formatter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Speakers (HW)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radio (HW)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Display (HW)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 2-2: Module Producer-Consumer Table Cont.**
3. System Hardware Description

This section will detail a few of the core hardware components that will make up the ERSC. Information such as the quantity required, intended purpose, manufacturer specifications, and other components it will interface with will be listed below.

3.1 Raspberry Pi Model B Revision 2 (512MB)

![Raspberry Pi Model B rev. 2](image)

Figure 3-1: Raspberry Pi Model B rev. 2

3.1.1 Quantity

The ERSC will require a single Raspberry Pi microcontroller.

3.1.2 Purpose

The Raspberry Pi will be the heart of the ERSC. It is responsible for processing every part of the system. Sensors and other hardware devices will connect via the general purpose input/output pins, the Android™ application and APIs will be contacted via the USB Bluetooth™/Wi-Fi combo adapter, data will be stored on the onboard SD card, and audio will be output via the onboard 3.5mm line out jack. The Raspberry Pi is an extremely critical component of the ERSC. Without it, the system would not function.
3.1.3 Specifications

<table>
<thead>
<tr>
<th>specifications</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoC (System on a Chip)</td>
<td>Broadcom BCM 2835 (700MHz ARM Processor)</td>
</tr>
<tr>
<td>Operating Voltage</td>
<td>5V DC</td>
</tr>
<tr>
<td>Minimum Amperage</td>
<td>700mA</td>
</tr>
<tr>
<td>Digital I/O Pins</td>
<td>17</td>
</tr>
<tr>
<td>SDRAM</td>
<td>512MB</td>
</tr>
<tr>
<td>Connections</td>
<td>HDMI, Composite RCA, 3.5mm audio jack, 10/100 Ethernet, 2x USB 2.0</td>
</tr>
<tr>
<td>Onboard Storage</td>
<td>SD card slot</td>
</tr>
</tbody>
</table>

Table 3-1: Raspberry Pi Specifications

3.1.4 Interfaces

The Raspberry Pi interfaces with all hardware that the system is comprised of. It will make use of its general purpose input/output pins to connect with the light sensor, temperature sensor, FM radio receiver, and push button. The FM radio receiver will specifically use the two pins (3 and 5 with respect to the board) intended for use with the I2C protocol. The other hardware devices that will use the general purpose input/output pins are free to use any of the remaining locations, but the software may need to be updated to reflect the pins chosen. The microphone and Bluetooth™/Wi-Fi combo adapter will connect to the Raspberry Pi via USB 2.0 ports. The LCD screen will be connected via the onboard HDMI port. The speaker bar will connect to the Raspberry Pi via the 3.5mm audio jack.
3.2 Tontec 7" LCD Screen

![Tontec 7" LCD Screen](image)

**Figure 3-2: Tontec 7" LCD Screen**

3.2.1 Quantity

The ERSC will require a single Tontec LCD screen.

3.2.2 Purpose

The Tontec LCD screen will display a large amount of information coming from the Raspberry Pi. This information includes the weather graphic, any alerts, the time and date, indoor and outdoor temperature, and any events listed on the user’s calendar for that day.
3.2.3 Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCD Size</td>
<td>7 inches</td>
</tr>
<tr>
<td>Resolution</td>
<td>800 x 480 pixels</td>
</tr>
<tr>
<td>Color</td>
<td>RGB</td>
</tr>
<tr>
<td>Aspect Ratio</td>
<td>16:9 (width: height)</td>
</tr>
<tr>
<td>LCD Panel Type</td>
<td>TN</td>
</tr>
<tr>
<td>Operating Voltage</td>
<td>5-12V DC</td>
</tr>
<tr>
<td>Minimum Amperage</td>
<td>2A</td>
</tr>
<tr>
<td>Input Connections</td>
<td>HDMI, VGA, Composite RCA</td>
</tr>
</tbody>
</table>

Table 3-2: Tontec 7” LCD Screen Specifications

3.2.4 Interfaces

The Tontec LCD will interface with the Raspberry Pi via the HDMI connection on both devices. This interface will provide data from the Raspberry Pi to the LCD for displaying.
3.3 HP Flat Panel Speaker Bar (Model no. EE418AA)

![Figure 3-3: HP Flat Panel Speaker Bar](image)

3.3.1 Quantity

The ERSC will require a single HP speaker bar.

3.3.2 Purpose

The HP speaker bar’s only purpose is to output any sounds coming from the system. Some sounds that the system will be outputting are alarms, alerts, and FM radio stations.

3.3.3 Specifications

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Power</td>
<td>2 watt RMS (1 watt RMS per channel)</td>
</tr>
<tr>
<td>Input Connections</td>
<td>3.5mm male audio connector, USB (for power)</td>
</tr>
<tr>
<td>Driver Impedance</td>
<td>16 ohms</td>
</tr>
<tr>
<td>Frequency Response</td>
<td>75 Hz to 20 kHz</td>
</tr>
</tbody>
</table>

Table 3-3: HP Flat Panel Speaker Bar Specifications

3.3.4 Interfaces

The HP speaker bar will interface with both the TEA5767 breakout board (radio receiver) and the Raspberry Pi via its 3.5mm audio input connector to provide audio output support for alarms, alerts, and FM radio stations.
3.4 TEA5767 FM Radio Receiver Breakout Board

![TEA5767 Breakout Board](image)

**Figure 3-4: TEA5767 Breakout Board**

3.4.1 Quantity

The ERSC will require a single TEA5767 FM radio receiver board.

3.4.2 Purpose

The TEA5767 is a small, cheap FM radio receiver and it will be used to tune into local FM radio stations.

3.4.3 Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating Voltage</td>
<td>5V DC</td>
</tr>
<tr>
<td>Frequency Spectrum</td>
<td>76-108MHz</td>
</tr>
<tr>
<td>FM Receiver Chipset</td>
<td>TEA5767</td>
</tr>
<tr>
<td>Input Connections</td>
<td>I2C Pins, 3.5mm jack (antenna)</td>
</tr>
<tr>
<td>Output Connections</td>
<td>I2C Pins, 3.5mm jack (L/R audio)</td>
</tr>
</tbody>
</table>

**Table 3-4: TEA5767 Breakout Board Specifications**

3.4.4 Interfaces

The TEA5767 breakout board connects to the Raspberry Pi via the I2C bus. The I2C bus allows for easy control of multiple devices with relatively few connections. The breakout board has 4 pins for connection to the Raspberry Pi: VCC, Ground, SDA, and SCL. The other interfaces provided are via 3.5mm jacks, one for the antenna lead and the other for the left and right audio output to the speaker bar.
3.5 GL5528 Light Dependent Resistor (Light Sensor)

![Image of GL5528 Light Dependent Resistor](image.jpg)

Figure 3-5: GL5528 Light Dependent Resistor

3.5.1 Quantity

The ERSC will require a single light dependent resistor.

3.5.2 Purpose

The light dependent resistor’s sole purpose is to determine the current ambient light in the room. With the data the ERSC can decide whether or not to enable its Lights Out mode, which disables all sound from the smart clock with the exception of alarms.

3.5.3 Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Voltage</td>
<td>150V DC</td>
</tr>
<tr>
<td>Maximum Wattage</td>
<td>95 mw</td>
</tr>
<tr>
<td>Operating Temperature</td>
<td>-30 to 70 degrees C</td>
</tr>
<tr>
<td>Resistance in Light</td>
<td>5-10K ohm</td>
</tr>
<tr>
<td>Resistance in Darkness</td>
<td>.5M ohm</td>
</tr>
</tbody>
</table>

Table 3-5: GL5528 Light Dependent Resistor Specifications
3.5.4 Interfaces

The light dependent resistor is an analog sensor and since the Raspberry Pi only accepts digital input on its general purpose input/output pins special steps need to be taken in order to use the sensor. Using an RC charging circuit allows us to make use of a simple equation:

\[ t = RC \]

‘t’ is the time required to charge the capacitor to 63%, R is the resistance, and C is the capacitance. At the 63% voltage mark, the input/output pin is driven HIGH and a counter is restarted. When the resistance increases due to a dark room, the time required to charge the capacitor also increases. This allows us to measure the time via a counter and have the system respond accordingly. A low counter value means the room is lit and a high counter value means it is dark.
3.6 DS18B20 Digital Temperature Sensor

![DS18B20 Temperature Sensor](image)

Figure 3-6: DS18B20 Temperature Sensor

3.6.1 Quantity

The ERSC will require a single DS18B20 temperature sensor.

3.6.2 Purpose

The purpose of the DS18B20 temperature sensor is to take readings of the indoor temperature and provide the system with the data. The data is then provided to the user via the ERSC’s display.

3.6.3 Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating Voltage</td>
<td>3-5.5V DC</td>
</tr>
<tr>
<td>Sensing Temperature</td>
<td>-55 to 125 degrees C</td>
</tr>
<tr>
<td>Accuracy of Readings</td>
<td>+-.5 degrees C</td>
</tr>
</tbody>
</table>

Table 3-6: DS18B20 Temperature Sensor Specifications

3.6.4 Interfaces

The DS18B20 temperature sensor interfaces with the Raspberry Pi via the general purpose input/output pins.
3.7 Microphone

3.7.1 Quantity

The ERSC will require a single microphone.

3.7.2 Purpose

The microphone will allow the system to provide voice recognition functionality in the form of voice commands. The commands available will be: radio on, radio off, alarm off, and read events.

3.7.3 Specifications

<table>
<thead>
<tr>
<th>Connection</th>
<th>USB 2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cable Length</td>
<td>45 inches</td>
</tr>
</tbody>
</table>

Table 3-7: Microphone Specifications

3.7.4 Interfaces

The microphone will interface with the Raspberry Pi via USB 2.0.
3.8 Generic Tactile Push Button

Figure 3-8: Tactile Push Button

3.8.1 Quantity

The ERSC will require a single push button.

3.8.2 Purpose

The push button will allow the user to power the system on or off. When the system is on and running, the user can press the button and the ERSC will perform a shutdown. Since the system halts instead of actually powering off, the system needs to be reset using the onboard reset header. Pressing and holding the button down for a few seconds will short the two reset header pins together and restart the system.

3.8.3 Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Voltage</td>
<td>24V DC</td>
</tr>
<tr>
<td>Maximum Current</td>
<td>50mA</td>
</tr>
<tr>
<td>Contact Type</td>
<td>SPST (Single Pole Single Throw)</td>
</tr>
</tbody>
</table>

Table 3-8: Tactile Push Button Specifications

3.8.4 Interfaces

The push button will be providing power from the Raspberry Pi to two different circuits. The first circuit is connected to a general purpose input/output pin and will cause the system to shut down when a change is registered. The second circuit is connected to the reset header on the Raspberry Pi and will cause the system to restart when the two header pins are shorted together.
3.9 Transcend 16GB SDHC Memory Card

![Transcend 16GB SDHC Memory Card](image)

**Figure 3-9: Transcend 16GB SDHC Memory Card**

3.9.1 Quantity

The ERSC will require a single 16GB SDHC card.

3.9.2 Purpose

The 16GB SDHC memory card is a critical component to the ERSC. It holds the OS that the Raspberry Pi will run on as well as all of the ERSC software. Without it the Raspberry Pi will not even boot.

3.9.3 Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage Capacity</td>
<td>16GB</td>
</tr>
<tr>
<td>Class</td>
<td>10</td>
</tr>
<tr>
<td>Operating Voltage</td>
<td>2.7-3.6V</td>
</tr>
<tr>
<td>Transfer Rate Max</td>
<td>45MB/s</td>
</tr>
</tbody>
</table>

**Table 3-9: Transcend 16GB SDHC Memory Card Specifications**

3.9.4 Interfaces

The SDHC memory card interfaces with the Raspberry Pi through its SD card slot.
3.10 Cirago USB Bluetooth™/Wi-Fi Combo Adapter

![Image of Cirago USB Bluetooth™/Wi-Fi Combo Adapter](image)

**Figure 3-10: Cirago USB Bluetooth™/Wi-Fi Combo Adapter**

3.10.1 Quantity

The ERSC will require a single Cirago USB Bluetooth™/Wi-Fi combo adapter.

3.10.2 Purpose

The USB Bluetooth™/Wi-Fi combo adapter is another critical component of the ERSC. The Wi-Fi portion of the adapter is used to connect to the user’s wireless network and provide Internet access. The Bluetooth™ portion of the adapter is used completely for communication with the application on the user’s smart phone. Without it, the system would be unable to pull data from the APIs over the Wi-Fi or communicate via Bluetooth™ with the Android™ application for setup and control.
### 3.10.3 Specifications

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating Voltage</td>
<td>5V DC</td>
</tr>
<tr>
<td>Power Consumption</td>
<td>&lt; 300mA</td>
</tr>
<tr>
<td>Bluetooth™ Standard</td>
<td>3.0 + HS</td>
</tr>
<tr>
<td>Wi-Fi Standards</td>
<td>802.11bgn</td>
</tr>
<tr>
<td>Operating Frequency</td>
<td>2.4-2.4835GHz ISM Band</td>
</tr>
<tr>
<td>Operating Range</td>
<td>33 feet</td>
</tr>
<tr>
<td>Transmit Power</td>
<td>+4dBm</td>
</tr>
<tr>
<td>Wi-Fi Transmit Power</td>
<td>+12dBm (b), +11dBm (g), +10dBm (n)</td>
</tr>
</tbody>
</table>

#### Table 3-10: Cirago USB Bluetooth™/Wi-Fi Combo Adapter Specifications

### 3.10.4 Interfaces

The USB Bluetooth™/Wi-Fi combo adapter interfaces with the Raspberry Pi via USB 2.0. It also connects with the user’s Android™ smart phone via Bluetooth™ and the Internet (for API data) via Wi-Fi.
3.11 Conceptual Hardware Diagram

Figure 3-11: Conceptual Hardware Diagram
4. System Software Description

This section will briefly introduce and describe the software development technologies used in both the ERSC and the Android™ application. Details on the specific programming languages and development frameworks used will be provided. These descriptions serve as a basis for understanding the underlying environment that our product will run on. More information on any third-party tools utilized in the development of our product will be included in the document’s appendix.

4.1 ERSC Software

The ERSC will make use of the Raspberry Pi microcontroller which runs a Linux based operating system known as Raspbian. Utilizing a microcontroller that runs its own operating system gives us access to various useful Linux functions that we can use when coding. The Raspbian operating system also comes with Python 2.7 pre-installed, allowing us to code the majority of the ERSC’s software in the Python programming language. Python provides us with the necessary hardware interface libraries to communicate with each of the hardware peripherals attached to the system. It also gives us access to web interface libraries that allow us to send and receive weather, traffic, and calendar data requests.

4.2 Android Application

The Android™ application used as the primary interface to the ERSC will be developed to run on Android™ versions dating back to Android™ 4.0.3 (Ice Cream Sandwich™). To streamline the development process, our team has elected to utilize the Phone Gap/Cordova mobile application framework. This framework allows us to build the application’s user interface using web technologies (HTML, JavaScript, CSS) while still maintaining the ability to program the back-end using Java. The Cordova platform compiles the web development code into executable native Android code and produces an APK file that can be used to install and run the application. Our team maintains the rights to any and all code produced as a result of using this platform.
5. Application Input Layer

The Application Input Layer is one half of the ERSC’s Twin Input System. This layer is meant to handle the interactions between the Android™ application and the ERSC. This will serve as the main source of input from the user into the system. The layer listens for user generated events from the application and if an action needs to be performed by the system, the layer will communicate the required task to the Data Processing Layer. It is comprised of two subsystems: the Event Handler and the Message Formatter.

5.1 Event Handler

The Event Handler will accept inputs from the user through the Android™ application. This subsystem will determine what type of message needs to be delivered to the rest of the system in order to satisfy the request or action that was received from the user.

5.1.1 GUI Listener

![GUI Listener Module Diagram](image)

**Figure 5-1: GUI Listener Module**

**Prologue**

The GUI Listener will wait for input from the user by using listeners attached to buttons or text fields. It will then gather what button was pressed or what information was entered and then pass this information to the Event Classifier module to be further processed.

**Interfaces** (See Appendix section 12.5 for table legend)

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>GUI Listener</td>
<td>Takes input from the user via buttons or text fields.</td>
<td>N/A</td>
</tr>
<tr>
<td>GUI Listener</td>
<td>Event Classifier</td>
<td>String: Event</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Table 5-1: GUI Listener Interfaces**
External Data Dependencies

- User-Android™ application interactions

Internal Data Dependencies

- Android™ SDK API level 15 or up
- Java JDK 1.7
- Cordova library: Converts web languages to the mobile platform languages.

Object Descriptions

N/A

Pseudo-code

```javascript
// Example of setting up buttons on the GUI for the user to click.
<li id="sunnyplaylist">Sunny Playlist<button class="ui-btn" data-role="button" data-rel="popup" a href="#popupPlaylistSun" data-transition="slideup"></button></li>

$("button").on("click", function(){
  // Go to a new page based on button clicked.
});

// For loop to search through user inputs on the page that are in field elements.
var string = (document.getElementById().innerText ||
  document.getElementById().textContent);
for(i in localStorage){
  for (i in localStorage){
    if((i.search != 1) && (flag = 0)){
      document.getElementById().innerText = localStorage.getItem();
      // Search through storage and find saved items.
    }
    else if(i.search != -1){
      flag = 1;
      document.getElementById().innerText = localStorage.getItem();
      // Search through storage and find saved items.
    }
  }
}

function updateInput(value){
  // Update input fields based on values.
}

function captureForm(){
  // Saves items to storage based on the input the user puts in. Also pulls items already in storage.
}

function printList(){
  // Checks every needed item in storage and displays the useful information to the user based on input.
}

function formatData(){
  // Formats selected data, such as hours:minutes:seconds. 24 hour format or 12 hour format.
}

function fillList(){
  // Based on user input, will fill fields with data depending on request.
}
```
Quality Assurance

The development team will use white box testing to validate this module. Dummy data will be provided simulating user input. The output resulting from inputting the dummy data will be compared to a control output that the development team will create. The module will be considered valid if the test output matches the control output.
5.2 Message Formatter

The Message Formatter will accept input from the Event Handler. The formatter will then pre-process the data and translate it into a form that is acceptable for the rest of the system. Once the data has been formatted properly, it is sent via Bluetooth™ to the Application Data Analyzer subsystem for the next step to respond to the user's requested action.

5.2.1 Event Classifier

![Event Classifier Module](image)

**Figure 5-2: Event Classifier Module**

**Prologue**

The Event Classifier examines what kind of action was detected by the GUI Listener and builds the specific object that needs to be sent to the Bluetooth™ Client for transmission to the ERSC.

**Interfaces** *(See Appendix section 12.5 for table legend)*

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUI Listener</td>
<td>Event Classifier</td>
<td>String: Event</td>
<td>N/A</td>
</tr>
<tr>
<td>Event Classifier</td>
<td>Bluetooth™ Client</td>
<td>User Event Object</td>
<td>N/A</td>
</tr>
</tbody>
</table>

*Table 5-2: Event Classifier Interfaces*

**External Data Dependencies**

N/A
Detailed Design Specification

Environmentally Responsive Smart Clock

Internal Data Dependencies

- Android™ SDK API level 15 or up
- Java JDK 1.7
- Cordova library: Converts web languages to the mobile platform languages.

Object Descriptions

UserEvent
{
    String eventType; //String description of the event that occurred.
    Boolean dataIncluded; //Is there data associated with the event?
    String data; //String representation of data associated with the event.
    Byte[] mp3; //Byte data of an mp3 file if it exists, otherwise null.
}

Pseudo-code

global List<EventObject> objs
MessageComposer{
    List<Message> message
    for each obj in objs {
        message.add(new Message(obj));
    }
}

//A stand alone Java class
class Message{
    var message
    Constructor Message (EventObject obj){
        var process
        var value
        process=obj.getEvent().toString();
        value = obj.get
        message = process+","+value+";"
    }
    sendMessage(List<Message> messages, BlueTooth connection){
        connection.SendBluetoothMessage(messages);
    }
}

class EventObject{
    event payload
    Constructor EventObject(ActionEvent evt){
        event = evt.getSource();
        payload=evt.getActionCommand();
    }
    getEvent(){
        return event;
    }
    getPayload(){
        return payload;
    }
}
Quality Assurance

Events will be fired and sent to the Event Classifier then the user event object will be built and decomposed by a unit test for analysis. The event will pass the test if it matches the desired output provided by the test.
5.2.2 Bluetooth™ Client

Prologue

The Bluetooth™ module will be responsible for establishing a connection with the ERSC and sending string or mp3 data to it.

Interfaces  
(See Appendix section 12.5 for table legend)

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Classifier</td>
<td>Bluetooth™ Client</td>
<td>User Event Object</td>
<td>N/A</td>
</tr>
<tr>
<td>Bluetooth™ Client</td>
<td>Bluetooth™ Server</td>
<td>String and/or Byte data</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 5-3: Bluetooth™ Client Interfaces

External Data Dependencies

N/A

Internal Data Dependencies

- Android™ SDK API level 15 or up
- Java JDK 1.7
- Android™ Bluetooth™ libraries
- Cordova library: Converts web languages to the mobile platform languages.
Object Descriptions

UserEvent
{
    String eventType;               //String description of the event that occurred.
    Boolean dataIncluded;          //Is there data associated with the event?
    String data;                   //String representation of data associated with the event.
    Byte[] mp3;                    //Byte data of an mp3 file if it exists, otherwise null.
}

Pseudo-code

class Bluetooth{
    BluetoothAdapter adapter
    BluetoothDevice device
    BluetoothSocket socket

    Constructor Bluetooth(ERSC_Address){
        adapter = BluetoothAdapter.getDefaultAdapter()
        device = adapter.getRemoteDevice(ERSC_address).createBond()
        socket = device.createInsecureRfcommSocketToServiceRecord(UUID uuid)
    }
    SendBluetoothMessage(List<Message> messages){
        OutputStream outStream = socket.getOutputStream()
        for each message in messages {
            outStream.write(message.toString());
        }
    }
    Kill(){
        socket.close();
    }
}

Based on Google’s Android™ documentation:

Quality Assurance

A unit test will create dummy data and request the Bluetooth™ module to create a connection, transmit the dummy data, and then terminate connection. A test will also be created to test whether the ERSC is in Bluetooth™ range.
6. Hardware Input Layer

The Hardware Input Layer handles all input coming from the hardware devices connected to the system. This includes the microphone, push button, light sensor, and the temperature sensor. The inputs are divided into two sections, event driven and polled. The event driven devices are the microphone and the push button because they provide no input to the system until the user has performed some action. The polled hardware devices are the light sensor and the temperature sensor. These two devices are constantly updating and can provide input whenever the system requires it. The polled input will be packaged into a single object and sent to the Data Processing Layer for updating the displayed indoor temperature on the ERSC’s GUI or deciding whether to turn Lights Out mode on or off.

6.1 Hardware Event

The sole purpose of the Hardware Event subsystem is to handle user input that emanates from either the microphone or the push button. The types of input will either be a short button press or a voice command. Once an event has been recognized it will be processed in this subsystem and a message will be passed to the Data Processing Layer to take the required action.

6.1.1 PyShutdown

![PyShutdown Module](image)

**Prologue**

This module waits for a button press from the user. Once pressed, the PyShutdown module will then send a message to the Data Processing Layer to inform the system that all data needs to be saved and the system needs to be shut down.
### Interfaces

*(See Appendix section 12.5 for table legend)*

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push Button</td>
<td>PyShutdown</td>
<td>Digital Pulse</td>
<td>N/A</td>
</tr>
<tr>
<td>PyShutdown</td>
<td>Hardware Event Processor</td>
<td>String: Shutdown command</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Table 6-1: PyShutdown Interfaces**

### External Data Dependencies

- User-Push button interaction

### Internal Data Dependencies

- RPi.GPIO library: Python library to enable access to the Raspberry Pi’s GPIO pins.

### Object Descriptions

N/A

### Pseudo-code

```python
pin = 13
GPIO.setmode(GPIO.BOARD)
def shutdown():
    # send message to Hardware Event Processor to begin shutdown
def waitForInput():
    GPIO.setup(pin, GPIO.IN)
    while(True):
        # Check for digital pulse on GPIO pin
```

### Quality Assurance

The development team will use white box testing to validate this module. Dummy data will be sent to the module that represents the data returned by the push button. The data will be analyzed by the module and the module will send appropriate commands for the ERSC to execute. These commands will be compared to the control output that the development team will create. The module will be considered valid if the commands that are created for execution match the tester’s control commands.
6.1.2 Jasper

Figure 6-2: Jasper Module

Prologue

The Jasper module is built using a third party, open source voice recognition system. It will actively listen via the microphone for one of the four supported voice commands: radio on, radio off, alarm off, read events. After recognizing one of the voice commands, Jasper will send a voice command string to the Hardware Event Processor module to process the action.

Interfaces (See Appendix section 12.5 for table legend)

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microphone</td>
<td>Jasper</td>
<td>Voice signal</td>
<td>N/A</td>
</tr>
<tr>
<td>Jasper</td>
<td>Hardware Event Processor</td>
<td>String: Voice command</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 6-2: Jasper Interfaces

External Data Dependencies

- User voice input via the microphone.
Internal Data Dependencies

- Jasper: Open source voice recognition system.

Object Descriptions

N/A

Pseudo-code

WORDS = ["RADIO", "ON", "OFF", "READ", "EVENTS", "ALARM"]

def handle(text, mic, profile):
    #Sends a message on to complete the user’s requested action.

def isValid(text):
    return bool(re.search(r'\bsearchcriteria\b', text, re.IGNORECASE))

Quality Assurance

The development team will use white box testing to validate this module. Dummy data will be sent to the module that represents the data returned by the microphone. The data will be analyzed by the module and the module will send appropriate commands for the ERSC to execute. These commands will be compared to the control output that the development team will create. The module will be considered valid if the commands that are created for execution match the tester’s control commands.
6.2 Hardware Poller

The Hardware Poller deals with the hardware devices that can be updated on demand. The two devices handled by this subsystem are the light sensor and the temperature sensor. When a request for updated information is received, most likely due to a timer running out, new information will be polled from each hardware device. Each module in the Hardware Poller will handle converting and formatting the data received from its respective hardware device into a system friendly format. The converted data is then sent to the Data Packaging subsystem to be combined with the rest of the polled hardware data into a single object.

6.2.1 Light Sensor

![Figure 6-3: Light Sensor Module](image)

Prologue

The Light Sensor module will interface directly with the light sensor circuit to measure the ambient light in the room. Since the sensor is analog we will measure how long it takes for the general purpose input/output pin to be driven HIGH. The length of time it takes is inversely proportional to the amount of light available in the room.

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Light Sensor (Light Dependent Resistor)</td>
<td>Light Sensor</td>
<td>Digital Pulse</td>
<td>N/A</td>
</tr>
<tr>
<td>Light Sensor</td>
<td>Sensor Data Packager</td>
<td>Int: Light reading</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 6-3: Light Sensor Interfaces
External Data Dependencies

- Light sensor readings

Internal Data Dependencies

- RPi.GPIO library: Python library to enable access to the Raspberry Pi’s GPIO pins.

Object Descriptions

N/A

Pseudo-code

class LightSensor:
    def __init__(self):
        self.ldrPin = 11

    def readLight(self):
        reading = 0
        gpio.setmode(gpio.BORDER)
        gpio.setup(self.ldrPin, gpio.OUT)
        gpio.output(self.ldrPin, gpio.LOW)
        time.sleep(.1)

        gpio.setup(self.ldrPin, gpio.IN)
        while(gpio.input(self.ldrPin) == gpio.LOW):
            reading = reading + 1
        return reading

    def getAverage(self):
        average = 0
        for i in range(0,15):
            average = average + self.readLight()
        return average/15

Quality Assurance

The development team will use white box testing to validate this module. Dummy data will be sent to the module that represents the data returned by the light sensor. The data will be analyzed by the module and the module will send appropriate reading for the ERSC to use. This data will be compared to the control output that the development team will create. The module will be considered valid if the dummy output matches the tester’s control output.
6.2.2 Temperature Sensor

Prologue

The Temperature Sensor module is responsible for interfacing directly with the temperature sensor connected to the system. The sensor’s stream of data is read and converted from a floating point value to an integer for easy display on the ERSC’s LCD. The module also provides the ability to choose between degrees Celsius and degrees Fahrenheit.

Interfaces (See Appendix section 12.5 for table legend)

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temperature Sensor (DS18B20)</td>
<td>Temperature Sensor</td>
<td>Float: Temperature reading</td>
<td>N/A</td>
</tr>
<tr>
<td>Temperature Sensor</td>
<td>Sensor Data Packager</td>
<td>Int: Temperature reading</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 6-4: Temperature Sensor Interfaces

External Data Dependencies

- Temperature sensor reading

Internal Data Dependencies

- Raspbian libraries: The module is dependent on OS file system libraries
Object Descriptions

N/A

Pseudo-code

class TempSensor:
    def __init__(self, tempUnit):
        self.sensorLocation = “String of sensor location”
        self.tempUnit = tempUnit

    def readTemp(self):
        rawTemp = self.readRawTemp()
        if self.tempUnit == "F":
            return self.convertToF(rawTemp)
        else:
            return round(rawTemp,0)

    def readRawTemp(self):
        tempFile = open(self.sensorLocation)
        text = tempFile.read()
        tempFile.close()
        rawTemp = float(text[-6:-1])/1000
        return rawTemp

    def convertToF(self, rawTemp):
        fah = int(round(((9*rawTemp)/5 + 32),0))
        return fah

Quality Assurance

The development team will use white box testing to validate this module. Dummy data will be read by the module that represents the data the temperature sensor provides. The data will be analyzed by the module and the module will send appropriate reading for the ERSC to use. This data will be compared to the control output that the development team will create. The module will be considered valid if the dummy output matches the tester’s control output.
6.3 Data Packaging

The Data Packaging subsystem is very simple in design. Its job is to provide updated information from the polled hardware devices. Once the new information has been acquired, the Data Packaging subsystem combines all polled data into a single object and returns it to the Data Processing Layer to be used.

6.3.1 Sensor Data Packager

![Diagram of Sensor Data Packager Module]

**Figure 6-5: Sensor Data Packager Module**

**Prologue**

The Sensor Data Packager module is responsible for creating Temperature Sensor and Light Sensor instances. When the Hardware Sensor Processor requires updated data from the sensors, the Sensor Data Packager uses both of the objects to get their respective values. Once the values have been acquired a SensorState object is created and returned to the Hardware Sensor Processor module.

**Interfaces**  (See Appendix section 12.5 for table legend)

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Light Sensor</td>
<td>Sensor Data Packager</td>
<td>Int: Light reading</td>
<td>N/A</td>
</tr>
<tr>
<td>Temperature Sensor</td>
<td>Sensor Data Packager</td>
<td>Int: Temperature reading</td>
<td>N/A</td>
</tr>
<tr>
<td>Sensor Data Packager</td>
<td>Hardware Sensor Processor</td>
<td>SensorState Object</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Table 6-5: Sensor Data Packager Interfaces**

**External Data Dependencies**

N/A
Internal Data Dependencies

N/A

Object Descriptions

SensorState
{
    int light_reading; //Current light sensor reading.
    int temp_reading; //Current temperature sensor reading.
}

Pseudo-code

class SensorDataPackager:
    def __init__(self, tempUnit):
        self.tempSensor = TempSensor(tempUnit)
        self.lightSensor = LightSensor()

    def getSensorData():
        sensorState = []
        sensorState.append(self.lightSensor.getAverage())
        sensorState.append(self.tempSensor.readTemp())
        return sensorState

Quality Assurance

The development team will use white box testing to validate this module. Dummy data will be sent to the module that represents the data returned by the Light Sensor module and the Temperature Sensor module. The data will be analyzed by the module and will package the data into an object for the ERSC to use. This object will be compared to the control output that the development team will create. The module will be considered valid if the objects that are created with the dummy data match the control objects.
7. Data Processing Layer

This section introduces the various modules embedded within the Data Processing Layer’s subsystems. As mentioned previously, the Data Processing Layer receives data from both the Application and Hardware Input layers and uses the data to execute and perform crucial system functions. This section will progress by moving from one subsystem to the next and describing the specific modules within each subsystem. Several of the Data Processing Layer modules revolve around the analysis and execution of both system and user invoked actions.

7.1 Processing Controller

The Processing Controller subsystem is perhaps the most crucial component of the system architecture. It will serve as the central location for the execution of both user and system requested tasks. All external input data, including data acquired through hardware peripherals, data received by the Android™ application, and data received externally via the network, will be routed through this subsystem to be processed. The modules in the subsystem will be implemented as a series of threads that will each handle the execution of a set of specified tasks.

7.1.2 Application Request Processor

The Application Request Processor obtains data from the Bluetooth™ Server module. The module processes the data to determine what type of action must be taken to fulfill the request. Requests can be sent to the Request Handler module to gather information from the storage system. The Application Request Processor may also contact the Request Handler to output information to the system.

![Application Request Processor Module](image_url)

**Figure 7-1: Application Request Processor Module**

**Prologue**

The Application Request Processor obtains data from the Bluetooth™ Server module. The module processes the data to determine what type of action must be taken to fulfill the request. Requests can be sent to the Request Handler module to gather information from the storage system. The Application Request Processor may also contact the Request Handler to output information to the system.
**Interfaces**  (See Appendix section 12.5 for table legend)

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bluetooth™ Server</td>
<td>Application Request Processor</td>
<td>AppMessage Object</td>
<td>N/A</td>
</tr>
<tr>
<td>Application Request Processor</td>
<td>Request Handler</td>
<td>InterfaceRequest Object</td>
<td>RequestStatus Object</td>
</tr>
</tbody>
</table>

**Table 7-1: Application Request Processor Interfaces**

**External Data Dependencies**

N/A

**Internal Data Dependencies**

N/A

**Object Descriptions**

**InterfaceRequest**

```java
{
    String source_module; //Name of the module making the request.
    String destination_module; //Name of destination module.
    String request_type; //Either data retrieval, data storage, or output.
    Boolean response_expected; //Is return data expected?
    Byte[] data; //Data to be sent.
}
```

**RequestStatus**

```java
{
    String destination_module; //Module that requested the data.
    Boolean successful; //Indicates whether request was serviced.
    Byte[] return_data; //Houses any data returned to the request initiator.
}
```

**AppMessage**

```java
{
    String MessageType; //i.e. “command”, “data”, etc.
    Byte[] MessageData; //The command or data received by the application.
}
```
Pseudo-code

```python
input_action = bluetooth_action_flag
if input_action == changeRadioStation:
    # send InterfaceRequest Object with FM radio data
elif input_action == setRadioPreset:
    # send InterfaceRequest Object with current radio frequency
elif input_action == setAlarm:
    # send InterfaceRequest Object with alarm data
```

Quality Assurance

The development team will use white box testing to validate this module. Dummy data will be sent to the module mimicking the message object that the Bluetooth™ Server module sends. The module will parse the message object and call the appropriate methods that will prepare the parsed data for transfer to storage or execution. This module will be considered valid if an acknowledgement and/or data is received from the Request Handler module.
7.1.2 Hardware Event Processor

**Prologue**

The Hardware Event Processor is an active module that waits to receive information from the Jasper module and/or the PyShutdown module. The module will accept input and process the data according to what was received. For example, voice commands recognized by the Jasper module will be executed here. The module will communicate with the Request Handler module in order to obtain other information or to send output to the system.

**Interfaces**  
(See Appendix section 12.5 for table legend)

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyShutdown</td>
<td>Hardware Event Processor</td>
<td>String: Shutdown command</td>
<td>N/A</td>
</tr>
<tr>
<td>Jasper</td>
<td>Hardware Event Processor</td>
<td>String: Voice command</td>
<td>N/A</td>
</tr>
<tr>
<td>Hardware Event Processor</td>
<td>Request Handler</td>
<td>InterfaceRequest Object</td>
<td>RequestStatus Object</td>
</tr>
</tbody>
</table>

Table 7-2: Hardware Event Processor Interfaces
External Data Dependencies
N/A

Internal Data Dependencies
N/A

Object Descriptions

InterfaceRequest
{
    String source_module; //Name of the module making the request.
    String destination_module; //Name of destination module.
    String request_type; //Either data retrieval, data storage, or output.
    Boolean response_expected; //Is return data expected?
    Byte[] data; //Data to be sent.
}

RequestStatus
{
    String destination_module; //Module that requested the data.
    Boolean successful; //Indicates whether request was serviced.
    Byte[] return_data; //Houses any data returned to the request initiator.
}

Pseudo-code

```
... This module is actively accepting input from the power button and
the voice command interface. It processes the data for valid commands
and relays the command throughout the system.
...

while TRUE:
    shutdown_input = shutdown_string
    jasper_input = jasper_string

    if shutdown_input == "shutdown":
        #send shutdown command

    if jasper_input in valid_commands:
        #if input is a valid command create and sent InterfaceRequest Object with
        Jasper command
```
Quality Assurance

The development team will use white box testing to validate this module. Dummy data will be sent to the Hardware Event Processor that represents the data returned by the Jasper and PyShutdown modules. The dummy data will fire methods that mute and un-mute the radio, turn off alarms, and read events from the user’s calendar. The module will pass the test if it calls the appropriate method successfully and is able to request the correct data from the Request Handler. If the module receives an acknowledgement and/or data from the Request Handler the module will be considered valid.
7.1.3 Hardware Sensor Processor

![Diagram of Hardware Sensor Processor Module]

Figure 7-3: Hardware Sensor Processor Module

**Prologue**

The Sensor Processor module receives a sensor state object that contains the current values coming from both the light and temperature sensors. The Sensor Processor uses the light sensor data to determine if Lights Out mode needs to be enabled or disabled. It also takes the temperature sensor reading and sends it to the Output Layer to be displayed on the ERSC’s LCD screen.

**Interfaces** *(See Appendix section 12.5 for table legend)*

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensor Data Packager</td>
<td>Hardware Sensor Processor</td>
<td>SensorState Object</td>
<td>N/A</td>
</tr>
<tr>
<td>Hardware Sensor Processor</td>
<td>Request Handler</td>
<td>InterfaceRequest Object</td>
<td>RequestStatus Object</td>
</tr>
</tbody>
</table>

Table 7-3: Hardware Sensor Processor Interfaces

**External Data Dependencies**

N/A

**Internal Data Dependencies**

N/A
Object Descriptions

InterfaceRequest
{
    String source_module; //Name of the module making the request.
    String destination_module; //Name of destination module.
    String request_type; //Either data retrieval, data storage, or output.
    Boolean response_expected; //Is return data expected?
    Byte[] data; //Data to be sent.
}

RequestStatus
{
    String destination_module; //Module that requested the data.
    Boolean successful; //Indicates whether request was serviced.
    Byte[] return_data; //Houses any data returned to the request initiator.
}

SensorState
{
    int light_reading; //Current light sensor reading.
    int temp_reading; //Current temperature sensor reading.
}

Pseudo-code

''' The following module will be tasked with receiving the sensor state information and using it to configure system settings '''

SensorState sensor_state_obj; #created in an earlier module

def process_light_sensor:
    light_sensor_val = sensor_state_obj.getLightReading();
    if light_sensor_reading < 60000:
        #set lights out mode

def process_temp_sensor:
    temp_sensor_val = sensor_state_obj.getTempReading();
    InterfaceRequest(type = Output, data = tempsensor_val, false); #output to screen
Quality Assurance

The development team will use white box testing to validate this module. Dummy data will be provided that represents the SensorState object that is expected from the Sensor Data Packager. The module will parse the dummy data and either put the system in Lights Out mode or take it out of Lights Out mode as determined by the data provided by the test. It will also route the indoor temperature reading from the object to the Request Handler for transfer to the Dispatcher module. The Request Handler will respond with an acknowledgement stating that the request was received.
7.1.4 API Data Processor

![Diagram of API Data Processor Module]

Figure 7-4: API Data Processor Module

**Prologue**

The API Data Processor is responsible for taking the external weather, traffic, and calendar data acquired by the API modules and analyzing it to determine if the user needs to be alerted of any significant changes. It also will determine if the clock’s graphical user interface needs to be updated to reflect weather, traffic, or event changes. This module was created to isolate all of the processing actions taken in response to the external internet data acquired via API services. The module will be implemented as a series of functions that are periodically called.

**Interfaces**  (See Appendix section 12.5 for table legend)

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Data Processor</td>
<td>Weather API</td>
<td>Int: Zip code</td>
<td>Forecast Object</td>
</tr>
<tr>
<td>API Data Processor</td>
<td>Weather API</td>
<td>Int: Zip code</td>
<td>CurrentConditions Object</td>
</tr>
<tr>
<td>API Data Processor</td>
<td>Traffic API</td>
<td>Tuple: Beginning</td>
<td>RouteData Object</td>
</tr>
<tr>
<td></td>
<td></td>
<td>coordinates</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tuple: Destination</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>coordinates</td>
<td></td>
</tr>
<tr>
<td>API Data Processor</td>
<td>Calendar API</td>
<td>String: CalendarID</td>
<td>String[]: CalendarEvents</td>
</tr>
<tr>
<td>API Data Processor</td>
<td>Request Handler</td>
<td>InterfaceRequest Object</td>
<td>RequestStatus Object</td>
</tr>
</tbody>
</table>

Table 7-4: API Data Processor Interfaces
External Data Dependencies

N/A

Internal Data Dependencies

N/A

Object Descriptions

InterfaceRequest
{
    String source_module; //Name of the module making the request.
    String destination_module; //Name of destination module.
    String request_type; // Either data retrieval, data storage, or output.
    Boolean response_expected; // Is return data expected?
    Byte[] data; // Data to be sent.
}

RequestStatus
{
    String destination_module; // Module that requested the data.
    Boolean successful; // Indicates whether request was serviced.
    Byte[] return_data; // Houses any data returned to the request initiator.
}

RouteData
{
    Int estimated_duration; // Estimated trip duration by car.
    String route_warnings; // String representing any route warnings
    Boolean Congestion; // Indicates congestion or not.
    String congestion_severity; // String representing congestion severity level.
}

Forecast
{
    float high_tomorrow; // High temp for tomorrow.
    float low_tomorrow; // Low temp for tomorrow.
    float upcoming_humidity; // Humidity level for tomorrow.
    Int chance_of_rain; // Percentage chance of rain for tomorrow.
    Int snow_inches; // Inches of snow expected tomorrow.
    String weather_description; // General weather description of tomorrow.
}
CurrentConditions
{
    float temp_f;  //Current temperature in degrees Fahrenheit.
    float feels_like_temp;  //What it feels like outside.
    float humidity_level;  //Current humidity.
    string description;  //Description of weather conditions.
}

Pseudo-code

'''This module processes the external api data retrieved by the traffic and weather modules and determines if there are any actions that the system needs to take in response to external traffic and weather conditions. '"

from Weather_Module import *
from Traffic_Module import *

weather_api = WeatherAPI("75227", "709e255949333ed9")  #create weather module object
traffic_api = TrafficAPI("api_key")  #create traffic module object

def update_outdoorTemp():
    conditions = weather_api.GetCurrentConditions(zipcode)
    temp_f = conditions.CurrentTemp  #get outdoor temp
    Output_Request(temp_f)  #output temp to clock's gui
    #change digit colors on clock gui to reflect temperature
    if temp_f > 90 and temp_f < 100:
        digit_color = "Orange"
    elif temp_f > 80 and temp_f <=90:
        digit_color = "yellow"
    elif temp_f >70 and temp_f <=80:
        digit_color = "green"
    elif temp_f >=100:
        digit_color = "red"
    elif temp_f <= 70:
        digit_color = "blue"

def check_weather_alerts():  #check forecast and determine any alerts
    forecast_response = weather_api.GetCurrentForecast(zipcode)  #get forecast
    if rain_expected:
        Output_Request("Rain expected tomorrow")
    elif large_temp_drop:
        Output_Request("Extreme Temp Drop tomorrow")
    elif ice_expected:
        Output_Request("Ice expected tomorrow")

def Auto_Adjust_Alarm():  #check weather conditions, adjust waketime alarm
    wake_time = Storage_Retrieval_Request("morning_alarm")
    if(within_hour_of_waketime):
        conditions = weather_api.GetCurrentConditions(zipcode)
        if conditions == unfavorable:
            #set wake time earlier, store new wake time

def Modify_Background_Graphic():
    conditions = weather_api.GetCurrentConditions(zipcode)
    if conditions_have_changed:
        #Change gui background weather graphic to reflect the change
def Check_Route_Traffic():
    destination_list = Storage_Retrieval_Request("destinations_list")
    startCoord = Storage_Retrieval_Request("my_location")
    for i in range(0, destination_list):
        endCoord = destination_list[i].coordinates #get destination coordinates
        traffic_level = traffic_api.GetTrafficConditions(startCoord, endCoord)
        #get route data

        if traffic_level == high: #route is congested, send alert to clock gui
            Output_Request("Route to " + destination_list[i] + "is congested")

def Check_Calendar_Events():
    event_list = calendar_api.getEvents(CalendarID)
    for i in range(0, event_list):  #go through event list and send to be output
        Output_Request(event_list[i])

**Quality Assurance**

The development team will use white box testing to validate this module. A request for
data will be sent from the API Data Processor module to the respective API modules in the
form of function calls. We will compare what is returned from the Weather and Traffic API
modules using a control request that will return a response that is known by the
development team. The temperature dependent methods of this module will be tested by
providing multiple ranges of temperature values. Those values should correspond to
different display colors. The auto adjusting alarm will be tested by providing dummy
traffic, weather, and event data with an alarm set. The severe weather/traffic alerts and the
event alerts will be tested given dummy events, weather, and traffic data. This module will
be considered valid if the smart clock’s response to the dummy data matches the expected
response.
7.1.5 Request Handler

Figure 7-5: Request Handler Module

Prologue

The Request Handler module accepts input from the four processors that exist within the Processing Controller subsystem. This module serves as an interface between the processors, file storage system, and the output system. It accepts various types of data objects which are then sorted and routed to the correct destination.
Interfaces (See Appendix section 12.5 for table legend)

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Request Processor</td>
<td>Request Handler</td>
<td>InterfaceRequest Object</td>
<td>RequestStatus Object</td>
</tr>
<tr>
<td>Hardware Event Processor</td>
<td>Request Handler</td>
<td>InterfaceRequest Object</td>
<td>RequestStatus Object</td>
</tr>
<tr>
<td>Hardware Sensor Processor</td>
<td>Request Handler</td>
<td>InterfaceRequest Object</td>
<td>RequestStatus Object</td>
</tr>
<tr>
<td>API Data Processor</td>
<td>Request Handler</td>
<td>InterfaceRequest Object</td>
<td>RequestStatus Object</td>
</tr>
<tr>
<td>Request Handler</td>
<td>SD Card Formatter</td>
<td>InterfaceRequest Object</td>
<td>N/A</td>
</tr>
<tr>
<td>Request Handler</td>
<td>Data Request Formatter</td>
<td>InterfaceRequest Object</td>
<td>RequestStatus Object</td>
</tr>
<tr>
<td>Request Handler</td>
<td>Dispatcher</td>
<td>InterfaceRequest Object</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 7-5: Request Handler Interfaces

External Data Dependencies

N/A

Internal Data Dependencies

N/A

Object Descriptions

InterfaceRequest

{  
    String source_module; //Name of the module making the request.
    String destination_module; //Name of destination module.
    String request_type; //Either data retrieval, data storage, or output.
    Boolean response_expected; //Is return data expected?
    Byte[] data; //Data to be sent.
}

RequestStatus

{  
    String destination_module; //Module that requested the data.
    Boolean successful; //Indicates whether request was serviced.
    Byte[] return_data; //Houses any data returned to the request initiator.
}
Pseudo-code

```
This module will accept input and relay it to the corresponding
destination as well as return a response object.

```def Request_Handler (interfaceRequest_Object):
    if interfaceRequest_Object. destination_module == "SD_Card_Formatter":
        #forward the request to the SD_Card_Formatter
    elif interfaceRequest_Object. destination_module == "Data_Request_Formatter":
        #forward the request to the Data_Request_Formatter
    elif interfaceRequest_Object. destination_module == "Dispatcher":
        #forward the request to the Dispatcher
    else:
        #invalid destination

    return response_object;
```

Quality Assurance

The development team will use white box testing to validate this module. Dummy data will be sent to the module mimicking data being sent from the Application Request Processor, Hardware Event Processor, Hardware Sensor Processor, and API Data Processor modules. The data will be compared and sorted. The sorted data will be sent to either the File Storage Layer or the Output Layer as appropriate. If no errors occur during the execution, the Request Handler will send an acknowledgement back to the module that initiated the request. The module will be considered valid if the requested data returned matches the expected data and the output matches the expected output.
7.2 Application Data Analyzer

The Application Data Analyzer is a crucial subsystem that aids in bridging the gap between the Android™ application and the ERSC itself. As stated before, the application serves as the main source of user input to the smart clock. The Application Data Analyzer ensures that the messages or data conveyed by the application are received correctly. The subsystem then performs any necessary pre-processing on the received application data and proceeds to send the data to the Processing Controller subsystem.

7.2.1 Bluetooth™ Server

![Figure 7-6: Bluetooth™ Server Module](image)

Prologue

The Bluetooth™ Server module will be responsible for receiving commands or data sent by the Bluetooth™ Client module. The module will be a thread running continuously on the ERSC itself with the sole purpose being to receive application data via the established Bluetooth™ connection. The module will also perform any necessary pre-processing and classify the received message as either a command or configuration data. It then sends the newly classified and analyzed message to the Application Request Processor module to begin executing any user-desired action.
**Interfaces**  
(See Appendix section 12.5 for table legend)

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bluetooth™ Client</td>
<td>Bluetooth™ Server</td>
<td>String and/or Byte data</td>
<td>N/A</td>
</tr>
<tr>
<td>Bluetooth™ Server</td>
<td>Application Request Processor</td>
<td>AppMessage Object</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Table 7-6: Bluetooth™ Server Interfaces**

**External Data Dependencies**

N/A

**Internal Data Dependencies**

- Python 2.7: Version of Python being used in development of the ERSC
- BluezUtils Bluetooth™ library: Linux Bluetooth™ utilities necessary for communicating with the Bluetooth™ dongle.
- PyBluez Python library: A Bluetooth™ Python library that is downloadable and installable from the developer website.

**Object Descriptions**

```
AppMessage
{
    String MessageType; //i.e. “command”, “data”, etc.
    Byte[] MessageData; //The command or data received by the application.
}
```
Pseudo-code

```python
# desc: simple demonstration of a Bluetooth™ server application that uses RFCOMM sockets

from bluetooth import *

server_sock = BluetoothSocket(RFCOMM)
server_sock.bind(('', PORT_ANY))
server_sock.listen(1)

port = server_sock.getsockname()[1]

uuid = "00001101-0000-1000-8000-00805F9B34FB"

advertise_service(server_sock, "SampleServer",
                   service_id = uuid,
                   service_classes = [ uuid, SERIAL_PORT_CLASS ],
                   profiles = [ SERIAL_PORT_PROFILE ],
                   protocols = [ OBEX_UUID ]
                   )

print("Waiting for connection on RFCOMM channel %d" % port)

client_sock, client_info = server_sock.accept()
print("Accepted connection from ", client_info)

try:
    while True:
        data = client_sock.recv(1024)
        if len(data) == 0: break
        print("received [%s]" % data)

        if (no_preprocessing):
            Send_to_execution_module(data)
        else:
            data = pre_process(data)
            Send_to_execution_module(data)

except IOError:
    pass

print("disconnected")

client_sock.close()
server_sock.close()

Adapted From:
https://code.google.com/p/pybluez/source/browse/trunk/examples/simple/rfcomm-server.py
```

Quality Assurance

The development team will use white box testing to verify the output object provided by the Bluetooth™ Server module matches the input object that the module receives from the Bluetooth™ Client. There will also be testing conducted in conjunction with the Bluetooth™ Client module to verify the client and server are connected. To test, a connection request will be sent to the Bluetooth™ Server by the Bluetooth™ Client and a dummy string will be sent by the application input system to verify the connection and tunnel.
7.3 API Request Handler

The purpose of the API Request Handler subsystem is to both isolate and facilitate the retrieving of data from an external network source. It will serve as a bridge between the ERSC system and the Internet, which will provide several crucial services (i.e. weather, traffic, and calendar data). The external data retrieved by this subsystem will be sent to the Processing Controller to be used in execution of system or user-driven actions.

7.3.1 Weather API

![Weather API Module Diagram]

**Prologue**

The Weather API module will be tasked with requesting and accepting weather related data including, but not limited to, local area temperatures, National Weather Service alerts, and a projected 3-day forecast. The module will utilize the services provided by Weather Underground, a commercial weather service that provides real-time weather information via the internet. The module will be implemented as a class with several methods built for requesting specific weather data.

**Interfaces** (See Appendix section 12.5 for table legend)

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weather API</td>
<td>Weather Underground Server</td>
<td>URL</td>
<td>JSON Response</td>
</tr>
<tr>
<td>API Data Processor</td>
<td>Weather API</td>
<td>Int: Zip code</td>
<td>Forecast Object</td>
</tr>
<tr>
<td>API Data Processor</td>
<td>Weather API</td>
<td>Int: Zip code</td>
<td>CurrentConditions Object</td>
</tr>
</tbody>
</table>

**Table 7-7: Weather API Interfaces**
External Data Dependencies

- Weather Underground: Commercial Weather Service that supplies the module with external weather data.

Internal Data Dependencies

- Python JSON libraries

Object Descriptions

CurrentConditions
{
  float temp_f; //Current temperature in degrees Fahrenheit.
  float feels_like_temp; //What it feels like outside.
  float humidity_level; //Current humidity.
  string description; //Description of weather conditions.
}

Forecast
{
  float high_tomorrow; //High temp for tomorrow.
  float low_tomorrow; //Low temp for tomorrow.
  float upcoming_humidity; //Humidity level for tomorrow.
  int chance_of_rain; //Percentage chance of rain for tomorrow.
  int snow_inches; //Inches of snow expected tomorrow.
  string weather_description; //General weather description of tomorrow.
}
Pseudo-code

```
import urllib2
import json

''' The following module/class handles the retrieving of weather data from the Weather Underground Internet services. Methods have been created for retrieving both forecast data and current outdoor conditions data. More methods may be added later. '''

class WeatherAPI:
    'Class that handles retrieving weather data'
    requestCount = 0  #tracks number of requests made

    def __init__(self, zipcode, key):
        self.apiKey = key  #set weather underground api key

    def GetCurrentForecast(self, zipcode):
        f = urllib2.urlopen('http://api.wunderground.com/api/' + self.apiKey + '/forecast/q/' + zipcode + '.json')
        json_string = f.read()
        forecast_data = json.loads(json_string)
        return forecast_data

    def GetCurrentConditions(self, zipcode):
        f = urllib2.urlopen('http://api.wunderground.com/api/' + self.apiKey + '/conditions/q/' + zipcode + '.json')
        json_string = f.read()
        parsed_json = json.loads(json_string)
        return parsed_json
```

Quality Assurance

The development team will use a combination of white box and black box testing for this module. The black box testing will be used to validate the responses from the Weather API. A request will be sent to the Weather API from a control computer. Then the response will be examined and compared to the API documentation. After that, the ERSC will send the same request to the same API and the response will be compared to the previously requested response. The white box testing will focus on the requests the API Data Processor module sends and the responses the Weather API module provides. The parsed data will be compared to the data that was received by the control computer for discrepancies.
7.3.2 Traffic API

Figure 7-8: Traffic API Module

Prologue

The Traffic API module will be responsible for both requesting and receiving local traffic related data. It will query the list of key destinations acquired from the application and use that data to request an estimate of the traffic level on the quickest route to the destination. The module will return a route object containing various data about the specified route. The module will utilize the Bing traffic and map services to acquire the data. It will be implemented as a class with various methods used to query the traffic service.

Interfaces  

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traffic API</td>
<td>Bing Traffic Server</td>
<td>URL</td>
<td>JSON Response</td>
</tr>
<tr>
<td>API Data Processor</td>
<td>Traffic API</td>
<td>Tuple: Beginning coordinates</td>
<td>RouteData Object</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tuple: Destination coordinates</td>
<td></td>
</tr>
</tbody>
</table>

Table 7-8: Traffic API Interfaces

External Data Dependencies

- Bing Traffic Service: Public traffic service provided by Bing that supplies the module with external traffic data.

Internal Data Dependencies

- Python JSON libraries
Object Descriptions

RouteData
{
    Int estimated_duration;  //Estimated trip duration by car.
    String route_warnings;  //String representing any route warnings
    Boolean Congestion;  //Indicates congestion or not.
    String congestion_severity;  //String representing congestion severity level.
}

Pseudo-code

import urllib2
import json

''' The following class is designed to request and receive traffic information
from the Bing traffic service. '''

class TrafficAPI:
    'Class that handles retrieving weather data'
    requestCount = 0  #tracks number of requests made

    #Class Constructor
    def __init__(self, key):
        self.apiKey = key  #set the api key provided by Bing

    #Method that requests and receives Traffic Conditions along route
    def GetTrafficConditions(self, StartLocation, EndLocation):
        f = urllib2.urlopen('http://dev.virtualearth.net/REST/version/restApi/resourcePath?queryParameters&key=BingMapsKey')  #Request data
        json_string = f.read()  #read the JSON response
        parsed_json = json.loads(json_string)  #parse the JSON
        #extract relevant data and construct route object
        return route_object

        f.close()

Quality Assurance

The development team will use a combination of white box and black box testing for this
module. The black box testing will be used to validate the responses from the Traffic API.
A request will be sent to the Traffic API from a control computer. Then the response will
be examined and compared to the API documentation. After that, the ERSC will send the
same request to the same API and the response will be compared to the previously
requested response. The white box testing will focus on the requests the API Data
Processor module sends and the responses the Traffic API module provides. The parsed
data will be compared to the data that was received by the control computer for
discrepancies.
7.3.3 Calendar API

![Diagram of Calendar API Module]

**Figure 7-9: Calendar API Module**

**Prologue**

The purpose of the Calendar API module is to synchronize and extract data from the user’s Google™ Calendar. It is designed to utilize the Google™ API services to periodically retrieve events from the calendar. Once the events are retrieved they are given to the API Data Processor to be used for output.

**Interfaces** (See Appendix section 12.5 for table legend)

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calendar API</td>
<td>Google™ APIs</td>
<td>String: AuthToken</td>
<td>JSON Response</td>
</tr>
<tr>
<td></td>
<td></td>
<td>String: CalendarID</td>
<td></td>
</tr>
<tr>
<td>API Data Processor</td>
<td>Calendar API</td>
<td>String: CalendarID</td>
<td>String[ ]: CalendarEvents</td>
</tr>
</tbody>
</table>

**Table 7-9: Calendar API Interfaces**

**External Data Dependencies**

- Google™ Calendar: Dependent on external calendar events provided by Google™ Calendar service.

**Internal Data Dependencies**

- Python JSON libraries
- Google™ Data API libraries: Python libraries provided by Google™ to access their APIs.

**Object Descriptions**

N/A
Pseudo-code

```python
import gflags
import httplib2
import urllib2
import json
from apiclient.discovery import build
from oauth2client.file import Storage
from oauth2client.client import OAuth2WebServerFlow
from oauth2client.tools import run

def GetCalendarEvents(CalendarID):
    FLAGS = gflags.FLAGS

    # Set up a Flow object to be used if we need to authenticate. This
    # sample uses OAuth 2.0, and we set up the OAuth2WebServerFlow with
    # the information it needs to authenticate. Note that it is called
    # the Web Server Flow, but it can also handle the flow for native
    # applications
    # The client_id and client_secret can be found in Google Developers Console
    FLOW = OAuth2WebServerFlow(
        client_id='128722555584
    7s20mgsqonnt9h36eth2vp6pr5fk19j2.apps.googleusercontent.com',
        client_secret='XH1nU
        ggVHg7HgqsSOPfw7kXj',
        scope='https://www.googleapis.com/auth/calendar',
        user_agent='ERSC System 1.0')

    # To disable the local server feature, uncomment the following line:
    # FLAGS.auth_local_webserver = False

    # If the Credentials don't exist or are invalid, run through the native client
    # flow. The Storage object will ensure that if successful the good
    # Credentials will get written back to a file.
    storage = Storage('calendar.dat')
    credentials = storage.get()
    if credentials is None or credentials.invalid == True:
        print 'credentials error'
        credentials = run(FLOW, storage)

    # Create an httplib2.Http object to handle our HTTP requests and authorize it
    # with our good Credentials.
    http = httplib2.Http()
    http = credentials.authorize(http)

    # Build a service object for interacting with the API. Visit
    # the Google Developers Console
    # to get a developerKey for your own application.
    service = build(serviceName='calendar', version='v3', http=http,
        developerKey='AIzaSyCbp6fDdTuRMXStRmVZCjlxzEigxm3Hf6WE')

    # ================== Authentication was successful ==================
    calendar_events = service.calendars().getEvents(calendarId=CalendarID).execute()

    # construct string array using api response
    return event_array #return the String array containing events

Based on Google’s Android™ documentation:
https://developers.google.com/google-apps/calendar/instantiate
```
Quality Assurance

The development team will use a combination of white box and black box testing for this module. The black box testing will be used to validate the responses from the Calendar API. A request will be sent to the Calendar API from a control computer. Then the response will be examined and compared to the API documentation. After that, the ERSC will send the same request to the same API and the response will be compared to the previously requested response. The white box testing will focus on the requests the API Data Processor module sends and the responses the Calendar API module provides. The parsed data will be compared to the data that was received by the control computer for discrepancies.
8. File Storage Layer

The purpose of the File Storage Layer is to abstract the details involved with both storing and receiving data from the system’s SD card. This layer will receive requests from the Data Processing Layer asking for specific stored data to be retrieved. It will also accept data that needs to be stored and push the data to the SD card. This section of the document will introduce and elaborate on the various modules associated with the File Storage Layer subsystems.

8.1 File Formatter

The File Formatter subsystem is tasked with receiving requests from the Data Processing Layer for either data retrieval or data storage. If the Data Processing Layer requires certain data to be stored, the File Formatter subsystem will ensure that the data is properly formatted or classified before it is sent to File System Manager subsystem for storage. Likewise, if certain data is requested, the data will be encapsulated in an object by the File Formatter subsystem and sent back to the Data Processing Layer.

8.1.1 SD Card Formatter

Figure 8-1: SD Card Formatter Module

Prologue

The main function of the SD Card Formatter is to receive data that needs to be stored and format the data accordingly before it is sent to the Data Storage module. The formatting will vary depending on the specific type of data that needs to be stored. For example, alarm data received from the Data Processing Layer may be formatted in the following manner: (Time, Alarm Type). This newly formatted data would then be sent to the Data Storage module to store the data on the SD card.

Interfaces  (See Appendix section 12.5 for table legend)

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request Handler</td>
<td>SD Card Formatter</td>
<td>InterfaceRequest Object</td>
<td>N/A</td>
</tr>
<tr>
<td>SD Card Formatter</td>
<td>Data Storage</td>
<td>String: Formatted data Int: Data type flag</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 8-1: SD Card Formatter Interfaces
External Data Dependencies

N/A

Internal Data Dependencies

N/A

Object Descriptions

InterfaceRequest
{
    String source_module; //Name of the module making the request.
    String destination_module; //Name of destination module.
    String request_type; //Either data retrieval, data storage, or output.
    Boolean response_expected; //Is return data expected?
    Byte[] data; //Data to be sent.
}
Detailed Design Specification

Environmentally Responsive Smart Clock

Pseudo-code

class StorageFormatter:
    'Class that handles formatting of data'

    #Class Constructor
    def __init__(self, key):
        self.apiKey = key  #set the api key provided by Bing

    def InterfaceRequestReciever(self, InterfaceRequest my_request):
        data = my_request.getData()  #extract data from Interface object
        if data == destination_data:
            formatted_string = '#call destination data formatting method
            StoreData(formatted_string, destination_data)
        elif data == alarm_data:
            formatted_string = '#call alarm data formatting method
            StoreData(formatted_string, alarm_data)
        elif data == config_data:
            formatted_string = '#call config data formatting method
            StoreData(formatted_string, config_data)

        #Method that accepts a destination object and formats data into a string
        def FormatDestinationData(self, destination):
            formatted_string = '(' + destination.latitude, destination.longitude, destination.description + ')
            return formatted_string

        #Method that accepts an alarm object and formats it into a string
        def FormatAlarmData(self, alarm):
            formatted_string = '(' + alarm.starttime, alarm.type + ')
            return formatted_string

        #Method that accepts a miscellaneous config data object and formats it into a string
        def FormatConfigurationData(self, Config):
            formatted_string = '(' + Config.current_location + ')
            return formatted_string

Quality Assurance

The development team will use white box testing to validate this module. Dummy data will be sent to the module mimicking data being sent from the Request Handler. The module will parse and format the data provided by the Request Handler then compared to the control formatted data. After formatting, the data will be packaged and sent to the Data Storage module. The module will be valid if the formatted data is equal to the control and the newly formatted data is sent to the Data Storage module.
8.1.2 Data Request Formatter

Figure 8-2: Data Request Formatter Module

Prologue

The Data Request Formatter will listen for data retrieval requests from the Data Processing Layer. It will notify the Data Retrieval module of the incoming request and wait for the data to be retrieved. Once the data is retrieved, the module will then proceed to place the requested data in a format that is easily usable by the Data Processing Layer modules. More specifically, it will encapsulate the requested data (currently in string form) in an object before sending it back to the Data Processing Layer.

Interfaces (See Appendix section 12.5 for table legend)

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request Handler</td>
<td>Data Request Formatter</td>
<td>InterfaceRequest Object</td>
<td>RequestStatus Object</td>
</tr>
<tr>
<td>Data Request Formatter</td>
<td>Storage Retrieval</td>
<td>String: File path</td>
<td>Raw String Data</td>
</tr>
</tbody>
</table>

Table 8-2: Data Request Formatter Interfaces

External Data Dependencies

N/A

Internal Data Dependencies

N/A
Object Descriptions

InterfaceRequest
{
    String source_module; //Name of the module making the request.
    String destination_module; //Name of destination module.
    String request_type; //Either data retrieval, data storage, or output.
    Boolean response_expected; //Is return data expected?
    Byte[] data; //Data to be sent.
}

RequestStatus
{
    String destination_module; //Module that requested the data.
    Boolean successful; //Indicates whether request was serviced.
    Byte[] return_data; //Houses any data returned to the request initiator.
}

Pseudo-code

class Data Retrieval Formatter:
    '''Class that handles formatting of data retrieved from storage. It takes the retrieved data in string form and uses that string to construct an object or list that the Data Processing Layer can more easily handle.''
    
    # Class Constructor
    def __init__(self):
        # do nothing

    # method that formats the retrieved SD card data.
    def Request Stored Data(self, Interface Request Object my_request):
        file_path = my_request.getData() # extract path of file containing data
        data_string = Retrieve Data(file_path) # get string data stored in SD card

        if data_string == Destinations: # destination data requested
            DestinationList myKeyDestinations = new DestinationList(data_string)
            RequestResponse response = new RequestResponse (myKeyDestinations)
            return response # return requested data

        elif data_string == Alarms: # alarm data requested
            AlarmList myCurrentAlarms = new AlarmList(data_string)
            RequestResponse response = new RequestResponse (myCurrentAlarms)
            return response # return requested data

        elif data_string == Config Data: # config data requested
            ConfigData myConfigData = new ConfigData(data_string)
            RequestResponse response = new RequestResponse (myConfigData)
            return response # return requested data
Quality Assurance

The development team will use white box testing to validate this module. Dummy data will be sent to the module mimicking data being sent from the Request Handler. This data will be parsed and sent to the Data Retrieval module. The data that is returned from the Data Retrieval module will be converted into an object. To be valid, the module must match the control request and the object that is created must also match the control object.
8.2 File System Manager

The File System Manager is tasked with both storing and retrieving various types of system data stored in the SD card. This subsystem will directly interface with the SD card to satisfy any storage or data retrieval requests initiated by the Data Processing Layer. The File System Manager consists of two modules: a Data Retrieval module and a Data Storage module.

8.2.1 Data Storage

Prologue

The Data Storage module receives system data that requires storage and places the data in the appropriate file in the SD card. It employs the use of the Raspbian OS file storage libraries to interface with the SD card. It takes the data that needs to be stored (previously formatted by the SD Card Formatter module) along with the file path, and proceeds to store the data in the file. Text files will be used to handle most of the system data.

Interfaces (See Appendix section 12.5 for table legend)

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD Card Formatter</td>
<td>Data Storage</td>
<td>String: Formatted data</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Int: Data type flag</td>
<td></td>
</tr>
<tr>
<td>Data Storage</td>
<td>SD Card</td>
<td>String: Formatted data</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>String: File path</td>
<td></td>
</tr>
</tbody>
</table>

Table 8-3: Data Storage Interfaces

External Data Dependencies

- SD Card: The module is dependent on data stored on the SD card

Internal Data Dependencies

- Raspbian libraries: The module is dependent on OS file system libraries
Object Descriptions

N/A

Pseudo-code

''' The following module/function handles the retrieval of locally stored data from the SD card. The function accepts a request object that contains details on the specific type of data that is being requested'''

def StoreData(Formatted_Data_String, dataType):
    if dataType == destination_data:
        #open and write to the destination data text file
        #return nothing
    elif dataType == alarm_data:
        #open and write to the alarm data text file
        #return nothing
    elif dataType == config_data:
        #open and write to the configuration data text file
        #return nothing

Quality Assurance

The development team will use white box testing to validate this module. Dummy data will be sent to the module mimicking data being sent from the SD Card Formatter. The module will parse and sort the data based on its destination file and then write the data. The module will be valid if the updated files match the control files.
8.2.2 Data Retrieval

Prologue

The Data Retrieval module will interact with the SD card directly to retrieve any stored data needed by the system. The module will accept requests for data from the Data Request Formatter module and will retrieve the data from the appropriate file using the supplied file path. The retrieved data will be sent back to the Data Request Formatter for encapsulation into an object. This module will be implemented as a function.

Interfaces  (See Appendix section 12.5 for table legend)

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Request Formatter</td>
<td>Data Retrieval</td>
<td>String: File path</td>
<td>Data in String form</td>
</tr>
<tr>
<td>Data Retrieval</td>
<td>SD Card</td>
<td>String: File path</td>
<td>File contents in String form</td>
</tr>
</tbody>
</table>

Table 8-4: Data Retrieval Interfaces

External Data Dependencies

- SD Card: The module is dependent on data stored on the SD card

Internal Data Dependencies

- Raspbian libraries: The module is dependent on OS file system libraries

Object Descriptions

N/A
Pseudo-code

''' This function uses the supplied file path to read a text file containing system data. It reads the data and into a string variable before returning it to the calling routine.''

def RetrieveData(file_path):
    file = open(file_path, 'r')  # open the file for reading
    string_data = file.read()  # reads the file data in string form
    return string_data

Quality Assurance

The development team will use white box testing to validate this module. Dummy data will be sent to the module mimicking data being sent from the Data Request Formatter. The data will be parsed and the module will open the desired file for reading. Data will be read in from the file and packaged before it is sent back to the Data Request Formatter. To be valid, the request string and the return object must match the control request string and return object.
9. Output Layer

The purpose of the Output Layer is to handle all interactions involving output between the system and its hardware peripherals. This layer will accept data from the Data Processing Layer and distribute it to the proper formatter to be output to the user. The formatters are separated into three individual modules which will output to the radio, speaker, and display. Each formatter performs its own set of unique actions on the data that is passed into the module.

9.1 Output Manager

The Output Manager subsystem’s job is to gather information from the Processing Controller subsystem and send it to the next step needed to output the data. The Output Manager consists of a single module that is used to determine what action is needed to output a particular type of data. The Dispatcher module will analyze the data for the intended output and send it to the Radio Formatter, Speaker Formatter, or the Display Formatter based on the analysis.

9.1.1 Dispatcher

Figure 9-1: Dispatcher Module
Detailed Design Specification  

Environmentally Responsive Smart Clock

**Prologue**

The Dispatcher is responsible for accepting data from the Request Module and determining where the data must be sent in order to output that particular information. Once the Dispatcher has received the data object, it is processed and sent to a formatter that corresponds with its intended output. The Dispatcher consists of a series of predetermined possible cases for output options.

**Interfaces**

(See Appendix section 12.5 for table legend)

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request Handler</td>
<td>Dispatcher</td>
<td>InterfaceRequest Object</td>
<td>N/A</td>
</tr>
<tr>
<td>Dispatcher</td>
<td>Radio Formatter</td>
<td>Float: Radio frequency</td>
<td>N/A</td>
</tr>
<tr>
<td>Dispatcher</td>
<td>Speaker Formatter</td>
<td>String: Audio file path</td>
<td>N/A</td>
</tr>
<tr>
<td>Dispatcher</td>
<td>Display Formatter</td>
<td>String: Display data</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Table 9-1: Dispatcher Interfaces**

**External Data Dependencies**

N/A

**Internal Data Dependencies**

N/A

**Object Descriptions**

InterfaceRequest

```java
{
    String source_module;  //Name of the module making the request.
    String destination_module;  //Name of destination module.
    String request_type;  //Either data retrieval, data storage, or output.
    Boolean response_expected;  //Is return data expected?
    Byte[] data;  //Data to be sent.
}
```
Pseudo-code

```python
'''
This module will use the destination of the object to determine
which formatter the data must be sent to.
'''
def Manager (interfaceRequest_Object):
    object_data = interfaceRequest_Object.getData

    if interfaceRequest_Object.destination_module == "Radio_Formatter":
        #send object_data to Radio_Formatter
    elif interfaceRequest_Object.destination_module == "Speaker_Formatter":
        #send object_data to Speaker_Formatter
    elif interfaceRequest_Object.destination_module == "Display_Formatter":
        #send object_data to Display_Formatter
    else:
        #invalid destination
```

Quality Assurance

The development team will use white box testing to validate this module. Dummy data will be sent to the module mimicking data being sent from the Request Handler module. The data will test that the module sends data to the Display Formatter, Speaker Formatter, and Radio Formatter. To validate this module, we will verify the paths the dummy data was sent on and compare them to the expected paths.
9.2 Output Formatter

The purpose of the Output Formatter is to receive information sent by the Output Manager and format the data so that it can be used by its intended output hardware. The Output Formatter consists of three modules, each corresponding to a specific output. The modules included in the Output Formatter subsystem are the Radio Formatter, Speaker Formatter, and Display Formatter. The Dispatcher module in the Output Manager subsystem sends objects to the appropriate formatter module in order to output data to the system.

9.2.1 Radio Formatter

**Figure 9-2: Radio Formatter Module**

**Prologue**

The Radio Formatter accepts an object that contains the desired radio frequency for the FM radio. Data is sent from the Dispatcher module to the Radio Formatter. The information is formatted into a usable data format and sent directly to the FM Radio hardware component.

**Interfaces** (See Appendix section 12.5 for table legend)

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispatcher</td>
<td>Radio Formatter</td>
<td>Float: Radio frequency</td>
<td>N/A</td>
</tr>
<tr>
<td>Radio Formatter</td>
<td>FM Radio Receiver</td>
<td>A five byte binary value</td>
<td>N/A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispatcher</td>
<td>Radio Formatter</td>
<td>Float: Radio frequency</td>
<td>N/A</td>
</tr>
<tr>
<td>Radio Formatter</td>
<td>FM Radio Receiver</td>
<td>A five byte binary value</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 9-2: Radio Formatter Interfaces
External Data Dependencies

N/A

Internal Data Dependencies

- Python SMBus library: Python library that allows access to devices on the I2C bus.

Object Descriptions

N/A

Pseudo-code

```python
This module will communicate directly with the Radio Receiver. It will take in the frequency as a float and perform calculations based on that frequency. It will then output a five byte binary value to the receiver.

```

```python
class Radio:
    def __init__(self, lastStation, presets):
        # initialize all variables needed
    def getStatus(self):
        # get the current status of the radio system
    def listPresets(self):
        # list all of the defined presets
    def muteAudio(self):
        # mute the audio
    def tuneUp(self):
        # tune the radio to a higher frequency
    def tuneDown(self):
        # tune the radio to a lower frequency
    def savePreset(self, presetBank):
        # save a given preset
    def loadPreset(self, presetBank):
        # load a saved preset
    def byteEncode(self):
        # encode the data to get a byte array for the radio system
    def byteDecode(self):
        # decode the data to gather information from the radio system
    def i2cRead(self):
        # read using the i2c bus
    def i2cWrite(self, radioConfig):
        # write using the i2c bus
```

Quality Assurance

The development team will use white box testing to validate this module. Dummy data will be sent to the module mimicking data being sent from the Dispatcher module. The data will be parsed and the appropriate methods will be fired to edit current radio settings. The module will be valid if the correct changes are made to the radio.
9.2.2 Speaker Formatter

![Diagram of Output Layer with Output Manager, Output Data Formatter, and Speaker Formatter]

**Figure 9-3: Speaker Formatter Module**

**Prologue**

The Speaker Formatter module accepts a string from the Dispatcher module that contains information regarding the sound to be played. The string will contain details such as its file name and path. This information will be used by the module to gather the data needed to output the requested sound to the physical speaker.

**Interfaces** (See Appendix section 12.5 for table legend)

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispatcher</td>
<td>Speaker Formatter</td>
<td>String: Audio file path</td>
<td>N/A</td>
</tr>
<tr>
<td>Speaker Formatter</td>
<td>Speaker</td>
<td>Byte sound data</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Table 9-3: Speaker Formatter Interfaces**

**External Data Dependencies**

- mpg321: Used to output the sound to the speakers.

**Internal Data Dependencies**

- OS library: Provides command line system access in the OS.
Object Descriptions

N/A

Pseudo-code

```
This module will accept a string which contains the location of the sound file to be played. The module will then make a system call in order to interface with the speaker hardware component.
'''

music = os.system('mpg321 /home/testfolder/test.mp3 &', 'w')

Quality Assurance

The development team will use white box testing to validate this module. Dummy data will be sent to the module mimicking data being sent from the Dispatcher module. The data will be parsed and the module will play the specified audio file referenced in the request. If the speaker outputs the correct sound then the module will be considered valid.
9.2.3 Display Formatter

Prologue

The Display Formatter module will accept a string from the Dispatcher module containing information regarding what should be displayed on the LCD Screen. The string may be comprised of data such as the text needed for the weather, calendar events, time, date, temperature, and any background graphic that is to be displayed to the screen. The module will configure the data with the font, size, location, and color of the text for each specific data category, if necessary. Once the proper data has been gathered and formatted it is sent directly to the LCD Screen.

Interfaces (See Appendix section 12.5 for table legend)

<table>
<thead>
<tr>
<th>Source</th>
<th>Sink</th>
<th>Input to Sink</th>
<th>Return from Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispatcher</td>
<td>Display Formatter</td>
<td>String: Display data</td>
<td>N/A</td>
</tr>
<tr>
<td>Display Formatter</td>
<td>LCD Display</td>
<td>String: Formatted display data</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 9-4: Display Formatter Interfaces

External Data Dependencies

N/A
Internal Data Dependencies

- Tkinter module: Python's standard interface to display information on the LCD.

Object Descriptions

N/A

Pseudo-code

```
This module will format the information that must be sent to the screen. It uses python's built-in gui library's methods.

w = Canvas(myFrame, width=700, height=400)
w.pack(fill = "both", expand = "yes")

if temp_data:
    # display outdoor temp text
    outdoor_temp_font = tkFont.Font(size = "20", family = "Helvetica")
    outdoor_temp_text = w.create_text(480, 225, text = "95 F", font = outdoor_temp_font)

elif time_data:
    # display time text
    time_font = tkFont.Font(size = "45", weight = "bold")
    timeText = w.create_text(525, 100, text = "9:37 P.M.", font = time_font)

elif background_data:
    # draw background image
    photo=PhotoImage(file='C:\Users\Jorge Zavala\Documents\CSE 4317\Implementation\ERSC Source Workspace\Images\sunny-day-landscape.gif')
    w.create_image(75, 75, image=photo)

elif date_data:
    # display date text
    date_font = tkFont.Font(size = "20")
    date_text = w.create_text(540, 190, text = "Wednesday, March 28", font = date_font)

elif calendar_data:
    # display calendar event
    event_description_font = tkFont.Font(size = "14", weight="bold")
    date_description_text = w.create_text(45, 275, text = "6:00 p.m. - SD Meeting", font = event_description_font, width = "300", anchor = "w")

elif aler_data:
    # display an alert
    alert_font = tkFont.Font(size = "14", weight="bold")
    alert_text = w.create_text(415, 370, text = "Severe Thunderstorm Watch", font = alert_font, width = "300", anchor = "w")
```
Quality Assurance

The development team will use white box testing to validate this module. Dummy data will be sent to the module mimicking data being sent from the Dispatcher module. The data string will be parsed and the module will either format the data or load a graphic to be displayed. If the Display Formatter displays the graphic or data then it will be considered valid.
10. Requirements Mapping

The tables below show which modules meet the requirements of the smart clock. This will help show which modules play a more critical role in the system’s design. While some modules do not completely satisfy a requirement, they may play a role in meeting that requirement by working together with other modules. Each module plays its own part to satisfy the overall purpose of the project; even though some may handle more requirements than others each module is important in its own way. Without each module doing what it is supposed to do then the entire system would not function as desired.

10.1 Application Input Layer

<table>
<thead>
<tr>
<th>Requirement Number</th>
<th>Requirement Name</th>
<th>GUI Listener</th>
<th>Event Classifier</th>
<th>Bluetooth™ Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Telling Time</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3.2</td>
<td>LCD Based Screen</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Calendar Synchronization</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>Alarm</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3.5</td>
<td>The Date</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3.6</td>
<td>Display Indoor/Outdoor Temperature</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3.7</td>
<td>Time Digits Change Color: Temperature</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.8</td>
<td>Weather Alerts</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.9</td>
<td>Get Traffic Data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.10</td>
<td>Adjust Alarm Time</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3.11</td>
<td>FM Radio</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3.12</td>
<td>Weather Graphic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.13</td>
<td>Basic Voice Input</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.14</td>
<td>Alarm Sounds</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3.15</td>
<td>Lights-Out Mode</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3.16</td>
<td>LCD Touch Screen</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.17</td>
<td>Auxiliary Input</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.18</td>
<td>Mute on Phone Call</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3.19</td>
<td>Phone Notifications</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>3.20</td>
<td>Basic Voice Output</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 10-1: Requirements Mapping for Application Input Layer
## 10.2 Hardware Input Layer

<table>
<thead>
<tr>
<th>Requirement Number</th>
<th>Requirement Name</th>
<th>PyShutdown</th>
<th>Jasper</th>
<th>Light Sensor</th>
<th>Temperature Sensor</th>
<th>Sensor Data Packager</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Telling Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td>LCD Based Screen</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Calendar Synchronization</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>Alarm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>The Date</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Display</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Indoor/Outdoor Temperature</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Time Digits Change</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Color: Temperature</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.8</td>
<td>Weather Alerts</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.9</td>
<td>Get Traffic Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.10</td>
<td>Adjust Alarm Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.11</td>
<td>FM Radio</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.12</td>
<td>Weather Graphic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.13</td>
<td>Basic Voice Input</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.14</td>
<td>Alarm Sounds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.15</td>
<td>Lights-Out Mode</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.16</td>
<td>LCD Touch Screen</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.17</td>
<td>Auxiliary Input</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.18</td>
<td>Mute on Phone Call</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.19</td>
<td>Phone Notifications</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.20</td>
<td>Basic Voice Output</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 10-2: Requirements Mapping for Hardware Input Layer
### 10.3 Data Processing Layer

<table>
<thead>
<tr>
<th>Requirement Number</th>
<th>Requirement Name</th>
<th>Bluetooth™ Server</th>
<th>Application Request Processor</th>
<th>Hardware Event Processor</th>
<th>Hardware Sensor Processor</th>
<th>API Data Processor</th>
<th>Weather API</th>
<th>Traffic API</th>
<th>Calendar API</th>
<th>Request Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Telling Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td>LCD Based Screen</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Calendar Synchronization</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>Alarm</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>The Date</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Display Indoor/Outdoor Temperature</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Time Digits Change Color: Temperature</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.8</td>
<td>Weather Alerts</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.9</td>
<td>Get Traffic Data</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.10</td>
<td>Adjust Alarm Time</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.11</td>
<td>FM Radio</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.12</td>
<td>Weather Graphic</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.13</td>
<td>Basic Voice Input</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.14</td>
<td>Alarm Sounds</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.15</td>
<td>Lights-Out Mode</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.16</td>
<td>LCD Touch Screen</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.17</td>
<td>Auxiliary Input</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.18</td>
<td>Mute on Phone Call</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>3.19</td>
<td>Phone Notifications</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.20</td>
<td>Basic Voice Output</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 10-3: Requirements Mapping for Data Processing Layer**
## 10.4 File Storage Layer

<table>
<thead>
<tr>
<th>Requirement Number</th>
<th>Requirement Name</th>
<th>SD Card Formatter</th>
<th>Data Request Formatter</th>
<th>Data Storage</th>
<th>Data Retrieval</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Telling Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td>LCD Based Screen</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Calendar Synchronization</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3.4</td>
<td>Alarm</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3.5</td>
<td>The Date</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>Display Indoor/Outdoor Temperature</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>Time Digits Change Color: Temperature</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.8</td>
<td>Weather Alerts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.9</td>
<td>Get Traffic Data</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3.10</td>
<td>Adjust Alarm Time</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>3.11</td>
<td>FM Radio</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3.12</td>
<td>Weather Graphic</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>3.13</td>
<td>Basic Voice Input</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.14</td>
<td>Alarm Sounds</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3.15</td>
<td>Lights-Out Mode</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3.16</td>
<td>LCD Touch Screen</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.17</td>
<td>Auxiliary Input</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.18</td>
<td>Mute on Phone Call</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.19</td>
<td>Phone Notifications</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.20</td>
<td>Basic Voice Output</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 10-4: Requirements Mapping for File Storage Layer**
### 10.5 Output Layer

<table>
<thead>
<tr>
<th>Requirement Number</th>
<th>Requirement Name</th>
<th>Dispatcher</th>
<th>Radio</th>
<th>Display</th>
<th>Speakers</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Telling Time</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td>LCD Based Screen</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Calendar Synchronization</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>Alarm</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>3.5</td>
<td>The Date</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>3.6</td>
<td>Display Indoor/Outdoor Temperature</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>3.7</td>
<td>Time Digits Change Color: Temperature</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>3.8</td>
<td>Weather Alerts</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>3.9</td>
<td>Get Traffic Data</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.10</td>
<td>Adjust Alarm Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.11</td>
<td>FM Radio</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.12</td>
<td>Weather Graphic</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>3.13</td>
<td>Basic Voice Input</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.14</td>
<td>Alarm Sounds</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>3.15</td>
<td>Lights-Out Mode</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.16</td>
<td>LCD Touch Screen</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.17</td>
<td>Auxiliary Input</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.18</td>
<td>Mute on Phone Call</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>3.19</td>
<td>Phone Notifications</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.20</td>
<td>Basic Voice Output</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 10-5: Requirements Mapping for Output Layer
11. Acceptance Plan

This section describes how the development team will validate the Environmentally Responsive Smart Clock’s functionality and overall completeness. The plan outlines criteria that must be met, acceptance testing, and how the ERSC will be packaged.

11.1 Acceptance Criteria

The ERSC must meet the following criteria in order to be considered complete by the development team and the sponsor. These requirements all have priorities of either 3 - Moderate, 2 – High, or 1 - Critical.

- **3.1 Telling Time:**
  - The ERSC must consistently tell the correct time.
- **3.2 LCD Based Screen:**
  - The ERSC must have a LCD screen.
- **3.3 Calendar Synchronization:**
  - The ERSC must synchronize with the user’s Google™ Calendar.
- **3.4 Alarm:**
  - The ERSC must be able to sound an alarm based on user parameters.
- **3.5 The Date:**
  - The ERSC must display the date.
- **3.6 Display Indoor/Outdoor Temperature:**
  - The ERSC must display both the indoor and outdoor temperatures in degree C or degree F as user decides.
- **3.8 Weather Alerts:**
  - The ERSC must provide severe weather alerts as provided by the weather API.
- **3.9 Get Traffic Data:**
  - The ERSC must get information from the traffic API, analyze said data and alert the user with data concerning his or her route.
- **3.10 Adjust Alarm Time:**
  - The ERSC must auto adjust the alarms in accordance to data that it receives from the APIs.
- **3.11 FM Radio:**
  - The ERSC must have an FM transceiver and play the radio frequencies for the user.
- **3.13 Basic Voice Input:**
  - The ERSC must listen for voice commands from the user and execute them.
- **3.14 Alarm Sounds:**
  - The ERSC must sound alarms that correspond with the current weather data that it receives from the weather API.
- **3.15 Lights-Out Mode:**
  - The ERSC must feature a Lights-Out Mode that suppresses all notifications except for alarms.
- **3.18 Mute on Phone Call:**
  - The ERSC must mute any sounds that it is making when the user gets a phone call.
- **3.20 Basic Voice Output:**
The ERSC must provide basic voice output in the form of reading user events.

4.1 Clock Size:
- The ERSC must be no larger than 14” x 11” x 3”

4.2 Clock Weight:
- The ERSC must have a weight no heavier than 10lbs.

4.3 Clock Housing:
- The ERSC must have a housing that secures the inner compartments.

4.4 Android™ Application:
- There must be a supporting Android™ application.

4.5 AC Adapter:
- There must be a supporting AC adapter packaged with the ERSC.

4.6 User Manual:
- There must be a supporting user manual packaged with the ERSC.

5.1 Android™ Application Startup Delay:
- The application must take no longer than 10 seconds to startup.

5.2 Clock Startup Delay:
- The ERSC must startup in no longer than 1 minute.

5.3 Initial Clock Setup Delay:
- The ERSC must take no longer than 5 minutes to set up initially.

5.4 Android™ Interface Latency:
- The application should take no longer than 2 seconds to respond to user input, and the ERSC must take no longer than 4 seconds to respond to a command from the application.

5.5 Weather API Latency:
- The ERSC’s requests to the weather API should take no longer than 30 seconds to receive a response, but if no response is received an appropriate error will be displayed for the user.

5.6 Traffic API Latency:
- The ERSC’s requests to the traffic API should take no longer than 30 seconds to receive a response, but if no response is received an appropriate error will be displayed for the user.

5.7 Speaker Quality:
- The fidelity of the speakers shall, at minimum, be comparable in sound quality to those found in most low-end (sub-$500) laptops or desktops.

6.1 Electrical Circuit Grounding:
- The ERSC’s wiring must be properly grounded.

6.2 Sound Output Safety:
- The ERSC’s sound output, including alarms and voice output, shall not exceed 80 decibels, the sound level for a typical alarm tone.

6.3 Accurate Weather Forecasting:
- The ERSC must present weather data as accurately as permitted by the API.

6.4 Packaging Safety:
- The ERSC shall not be composed of packaging that includes unusually sharp edges or materials that can induce harm to an individual.

6.5 Operating Temperature:
- The ERSC must operate at a reasonable temperature.
11.2 Acceptance Testing

The ERSC will be tested to ensure all moderate to critical requirements stated in the System Requirements Specification are met as stated in the document. Once all requirements are considered complete the project will be accepted. A detailed plan for testing the requirements will be documented in the System Test Plan.

11.3 Packaging and Installing

The ERSC will be delivered in a box no bigger than 15” x 15” x 15” and will include the pre-assembled smart clock, the Android™ application (loaded onto a CD), a cord for powering the device, and the user manual. To install the Android™ application, the user will need to insert the CD and attach their Android™ smart phone to the same computer. From there, they simply need to copy the APK file from the CD onto the Android™ smart phone. The APK must be run on the phone in order to install it.

After the application is loaded onto the smart phone the user will also need to make sure the ERSC is powered on. This can be done by plugging it in or, if already plugged in, giving the power button a long press. Then the user will only have to go through the startup screen on the application.
12. Appendix

12.1 I2C-Tools

I2C is a multi-master serial single-ended computer bus and is used for attaching low-speed peripherals to a motherboard, embedded system, cellphone, or other digital electronic devices. The smart clock uses the I2C bus on the Raspberry Pi and uses the tools from the I2C-Tool package for Linux. It uses the I2Cdetect to scan an I2C bus for devices and will output a table with the list of detected devices on the specified bus.

12.2 Python-SMBus

Python-SMBus is a Python module that will allow access to the I2C protocol through the System Management Bus. Once the module has been imported, Python code can be used to do various types of reads or writes on the I2C bus. For instance, code can be used to write to registers, read from registers, and call other processes.

12.3 Jasper

Jasper is an open source platform for developing “always-on” voice controlled applications. Since Jasper is “always-on”, it is constantly listening for a specific key phrase. Once the key phrase is spoken, then Jasper will listen for the command that the user wants to execute. Jasper is an external resource that has been integrated into the smart clock’s system. Jasper contains modules that execute specific tasks based on the voice command issued. New modules can also be created so that the commands to be executed are specific to the smart clock’s needs.

12.4 RPi.GPIO

RPi.GPIO is a Python module that allows access to the Raspberry Pi’s general purpose input/output pins. A general-purpose input/output pin is connected to an integrated circuit that the user can control during runtime. The pins can be used to control a variety of devices. For example, the smart clock will use the Raspberry Pi’s general purpose input/output pins to interface with the light sensor and the push button.

12.5 Interface Table Legend

This section will give a brief overview of how the Interface tables under each module’s sections are to be read.

- **Source** – This is the sending interface that will provide input for the sink.
- **Sink** – This is the receiving interface that will accept input from the source.
- **Input to Sink** – Data (Objects, Messages, etc.) that is passed via parameters from the source to the sink.
- **Return from Sink** – Data (Objects, Messages, etc.) that is returned from the sink to the source.