Programming Sun™ SPOTs (Small Programmable Object Technology) in Java™

A dissertation submitted by

S. Willisch

in fulfilment of the requirements of

ENG4112 Research Project

towards the degree of

Bachelor of Engineering, (Electrical & Electronic)

Submitted: October, 2009
Abstract

The purpose of this project was to develop a testbed for a simple wireless sensor network (WSN). In a WSN, a collection of sensor nodes, called motes, have the ability to collect information and send this information to a host computer or other motes. The WSN was developed using a Sun™ SPOT Development kit. This development kit contained the hardware and software to develop a WSN. The applications for the motes were written in Squawk Java™ Virtual Machine (VM). This variant of Java™ runs directly on the wire, i.e. without the need for an operating system. To eliminate the need to develop any additional hardware, the eDemoBoard on the Sun™ SPOTs was utilised.

The eDemoBoard included sensors, analogue and digital In/Outputs and user interfaces. Since the development kit only included two Sun™ SPOTs, additional virtual motes were added to the WSN. The virtual motes were launched from Solarium. Solarium is a software application which was delivered with the development kit. Virtual and physical Sun™ SPOTs were able to communicate with each other.

The WSN was implemented as a Home Automation Application (HAA). It was implemented using two different specialised motes - collector and interface motes. The environmental data was sampled and sent by the collector motes. This data was sent as a broadcast over radio communication. The receivers of the broadcast data were the interface motes and the host. The interface motes forwarded the data to the external systems, i.e. climate control, light control, alarm systems, etc.

In addition, the interface motes transmitted the status of the associated external system to the host. The host acted as a user interface, utilising the incoming data from collector and interface motes to provide relevant information to the occupant of the home.
Six testbeds were defined to verify the accuracy and performance of the WSN. The results from testbeds showed that the selected sensors were deemed to be accurate enough for a HAA. In addition, from a performance perspective the testbeds proved that the WSN worked well. An unexpected outcome of the testbeds was the discovery that the host computer became significantly slower if too many virtual motes were running at the same time.

The high cost of the Sun™ SPOT development kit compared to conventional sensors was a limiting factor in deploying a complete WSN using only physical Sun™ SPOTs. However, even though the development kit only included two physical Sun™ SPOTs it was possible to develop a working WSN. A combination of the physical and virtual Sun™ SPOTs provided enough motes for the WSN to cover a small home.
Limitations of Use

The Council of the University of Southern Queensland, its Faculty of Engineering and Surveying, and the staff of the University of Southern Queensland, do not accept any responsibility for the truth, accuracy or completeness of material contained within or associated with this dissertation.

Persons using all or any part of this material do so at their own risk, and not at the risk of the Council of the University of Southern Queensland, its Faculty of Engineering and Surveying or the staff of the University of Southern Queensland.

This dissertation reports an educational exercise and has no purpose or validity beyond this exercise. The sole purpose of the course pair entitled “Research Project” is to contribute to the overall education within the student’s chosen degree program. This document, the associated hardware, software, drawings, and other material set out in the associated appendices should not be used for any other purpose: if they are so used, it is entirely at the risk of the user.

Prof F Bullen
Dean
Faculty of Engineering and Surveying
Certification of Dissertation

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.

I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

S. WILLISCH

0039840087

________________________
Signature

________________________
Date
Acknowledgments

I would like to thank the following people:

My wife Pamela for her support and patience throughout my studies.

Dr. Wei Xiang for his valuable guidance while completing this project.

S. Willisch

University of Southern Queensland

October 2009
# Contents

Abstract i

Acknowledgments v

List of Figures xiii

List of Tables xv

Pseudo & Source Code Listings xvi

Nomenclature xxi

Chapter 1 Introduction 1

1.1 Broad Aims and Specific Objectives .......................... 2

1.2 Background Information ........................................ 3

Chapter 2 Literature Review 5

2.1 MAC Layer ..................................................... 7

2.2 Physical Layer .................................................. 8
CONTENTS

2.3 Network Topology ................................................. 8
2.4 Security .............................................................. 10
2.5 Power supply ......................................................... 11
2.6 Wireless Sensor Node (Mote) ..................................... 12
2.7 SE Java VM and Squawk Java VM .............................. 13

Chapter 3 Methodology ................................................... 17

3.1 Introduction ......................................................... 17
3.2 Select Scenario for WSN .......................................... 18
3.3 Define Testbeds ..................................................... 19
3.4 HAA Design and Implementation ............................... 19
3.5 Verification Accuracy and Performance ....................... 20

Chapter 4 HAA Test Beds .................................................. 21

4.1 Introduction ........................................................... 21
4.2 Overview Test Bed Setup ......................................... 21
   4.2.1 Deploying Application to Physical Motes .................. 23
   4.2.2 Deploying Application to Virtual Motes .................... 23
   4.2.3 Deploying Host Application ................................. 23
   4.2.4 Starting Members of HAA WSN ............................. 24
4.3 Test Beds ............................................................. 24
CONTENTS viii

4.3.1 Test Bed 1: Accuracy of WSN, Internal Sensor . . . . . . . . . . 24

4.3.2 Test Bed 2: Accuracy of WSN, External Sensor . . . . . . . . . . 25

4.3.3 Test Bed 3: Maximum Distances between WSN Members . . . . 26

4.3.4 Test Bed 4: Impact of Noise Sources on WSN . . . . . . . . . . . 26

4.3.5 Test Bed 5: Multifunction Data Collection . . . . . . . . . . . . 27

4.3.6 Test Bed 6: High Volume Data Collection . . . . . . . . . . . . . 27

Chapter 5 Implementation 29

5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

5.2 Design and Development Process . . . . . . . . . . . . . . . . . . . . . 30

5.3 Requirements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

5.3.1 WSN Configuration . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

5.3.2 Data Collection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

5.3.3 Interface to External Systems . . . . . . . . . . . . . . . . . . . . . 40

5.3.4 Data Logging and User Interface on Host . . . . . . . . . . . . . . 46

5.4 Test Scripts . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

5.5 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

5.5.1 General HAA Configurations . . . . . . . . . . . . . . . . . . . . . 52

5.5.2 Implementation of Data Collection . . . . . . . . . . . . . . . . . . 53

5.5.3 Implementation of Interface to External Systems . . . . . . . . . 61

5.5.4 Implementation of Host Application . . . . . . . . . . . . . . . . . 67
Chapter 6  Results  69

6.1 Testbed 1: Accuracy of WSN, Internal Sensor  69

6.2 Testbed 2: Accuracy of WSN, External Sensor  70

6.3 Testbed 3: Maximum Distances between WSN Members  71

6.4 Testbed 4: Impact of Noise Sources on WSN  71

6.5 Testbed 5: Multifunction Data Collection  71

6.6 Testbed 6: High Volume Data Collection  71

6.7 Test Result and Conclusion  73

Chapter 7  Conclusions and Further Work  75

7.1 Achievement of Project Objectives  76

7.2 Further Work  77

References  79

Appendix A  Project Specification  82

Appendix B  Flowcharts  85

B.1 Flowchart Compile and Run Process  86

B.2 Flowchart Main: Collector Mote  87

B.3 Flowchart: Temperature Collecting Thread  88
Appendix C Code Listings

C.1 Example Implement Code ........................................ 93
  C.1.1 Example of Test Script .................................. 93
  C.1.2 Example of the New Method to be Tested ............... 94
C.2 Pseudo Code Collector Motes .................................. 96
  C.2.1 Pseudo Code for Temperature Thread ................... 96
  C.2.2 Pseudo Code for Light Level Thread ................... 97
  C.2.3 Pseudo Code for Battery Capacity Thread ............... 98
  C.2.4 Pseudo Code for Alarm Trigger Thread ................. 99
  C.2.5 Pseudo Code for Carbon Monoxide Thread .............. 101
  C.2.6 Pseudo Code for Collector Mote Main Thread ........... 102
C.3 Code Listings for Collector Mote ............................. 104
  C.3.1 CollectorMote.java ...................................... 104
  C.3.2 CollectorTempMote.java ................................. 108
  C.3.3 CollectorLightMote.java ................................. 111
  C.3.4 CollectorBattMote.java ................................. 114
  C.3.5 CollectorAlarmMote.java ................................. 116
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>File Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.3.6</td>
<td>CollectorCarbonMonoMote.java</td>
<td>117</td>
</tr>
<tr>
<td>C.3.7</td>
<td>CollectMainSPOT.java</td>
<td>120</td>
</tr>
<tr>
<td>C.3.8</td>
<td>TempThread.java</td>
<td>125</td>
</tr>
<tr>
<td>C.3.9</td>
<td>LightThread.java</td>
<td>129</td>
</tr>
<tr>
<td>C.3.10</td>
<td>BatLevelThread.java</td>
<td>134</td>
</tr>
<tr>
<td>C.3.11</td>
<td>AlarmTriggerThread.java</td>
<td>137</td>
</tr>
<tr>
<td>C.3.12</td>
<td>CarMonThread.java</td>
<td>142</td>
</tr>
<tr>
<td>C.4</td>
<td>Pseudo Code for Interface Mote</td>
<td>147</td>
</tr>
<tr>
<td>C.4.1</td>
<td>Pseudo Code for Climate Interface Thread</td>
<td>147</td>
</tr>
<tr>
<td>C.4.2</td>
<td>Pseudo Code for Light Interface Thread</td>
<td>148</td>
</tr>
<tr>
<td>C.4.3</td>
<td>Pseudo Code for Alarm Interface Thread</td>
<td>148</td>
</tr>
<tr>
<td>C.4.4</td>
<td>Pseudo Code for Carbon Monoxide Interface Thread</td>
<td>149</td>
</tr>
<tr>
<td>C.4.5</td>
<td>Pseudo Code for Door Interface Thread</td>
<td>150</td>
</tr>
<tr>
<td>C.4.6</td>
<td>Pseudo Code to Send External System Status Thread</td>
<td>151</td>
</tr>
<tr>
<td>C.4.7</td>
<td>Pseudo Code for Interface Mote Main Thread</td>
<td>152</td>
</tr>
<tr>
<td>C.5</td>
<td>Code Listings for Interface Mote</td>
<td>154</td>
</tr>
<tr>
<td>C.5.1</td>
<td>InterfaceMote.java</td>
<td>154</td>
</tr>
<tr>
<td>C.5.2</td>
<td>ClimateInterfaceMote.java</td>
<td>158</td>
</tr>
<tr>
<td>C.5.3</td>
<td>LightInterfaceMote.java</td>
<td>159</td>
</tr>
<tr>
<td>C.5.4</td>
<td>AlarmInterfaceMote.java</td>
<td>160</td>
</tr>
</tbody>
</table>
List of Figures

1.1 eDemoBoard of Sun™ SPOT ................................. 4

2.1 802.15.4 Topologies ................................. 9

2.2 Comparison Standard Java VM versus Squawk Java VM\textsuperscript{1} ............. 14

4.1 HAA Test Bed ................................. 22

5.1 Host Main Menu ................................. 47

B.1 Flowchart Showing Compile and Run Process ................................. 86

B.2 Flowchart Main Collector Mote ................................. 87

B.3 Flowchart Temperature Thread ................................. 88

B.4 Flowchart Main Interface Mote ................................. 89

B.5 Flowchart Climate Interface Thread ................................. 90

B.6 Flowchart Send External System Status Thread ................................. 91

D.1 Screen Capture Test Script Output ................................. 200

D.2 Screen Capture Test Run Host Output ................................. 201
LIST OF FIGURES

D.3 Screen Capture Test Run Virtual SPOT Output ............... 202

D.4 Screen Capture Solarium with Virtual SPOT ............... 203

D.5 Screen Capture HAA Test Bed with Six Virtual SPOT ....... 204

E.1 Home Layout ...................................................... 206
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>802.15.4. Channel Assignment</td>
<td>9</td>
</tr>
<tr>
<td>5.1</td>
<td>Port Allocation</td>
<td>33</td>
</tr>
<tr>
<td>5.2</td>
<td>Message Type Identifier Collector Motes</td>
<td>34</td>
</tr>
<tr>
<td>5.3</td>
<td>HAA Locations</td>
<td>35</td>
</tr>
<tr>
<td>5.4</td>
<td>Collected Environmental Data</td>
<td>35</td>
</tr>
<tr>
<td>5.5</td>
<td>Receivers of Collected Data</td>
<td>35</td>
</tr>
<tr>
<td>5.6</td>
<td>External System Identification for Interface Motes</td>
<td>41</td>
</tr>
<tr>
<td>5.7</td>
<td>CollectorMote.java class</td>
<td>56</td>
</tr>
<tr>
<td>5.8</td>
<td>CollectorTempMote.java class</td>
<td>57</td>
</tr>
<tr>
<td>5.9</td>
<td>CollectorLightMote.java class</td>
<td>57</td>
</tr>
<tr>
<td>5.10</td>
<td>CollectorBattMote.java class</td>
<td>58</td>
</tr>
<tr>
<td>5.11</td>
<td>CollectorAlarmMote.java class</td>
<td>59</td>
</tr>
<tr>
<td>5.12</td>
<td>CollectorCarbonMonoMote.java class</td>
<td>60</td>
</tr>
<tr>
<td>5.13</td>
<td>InterfaceMote.java class</td>
<td>63</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>5.14</td>
<td>ClimateInterfaceMote.java class</td>
<td>64</td>
</tr>
<tr>
<td>5.15</td>
<td>LightInterfaceMote.java class</td>
<td>64</td>
</tr>
<tr>
<td>5.16</td>
<td>AlarmInterfaceMote.java class</td>
<td>64</td>
</tr>
<tr>
<td>5.17</td>
<td>CarbMonInterfacMote.java class</td>
<td>65</td>
</tr>
<tr>
<td>5.18</td>
<td>DoorInterfacMote.java class</td>
<td>65</td>
</tr>
<tr>
<td>5.19</td>
<td>RadiogramCommClass.java class</td>
<td>66</td>
</tr>
<tr>
<td>5.20</td>
<td>CommClassHost.java class</td>
<td>68</td>
</tr>
<tr>
<td>6.1</td>
<td>Results Testbed 1, Light Level Accuracy</td>
<td>70</td>
</tr>
<tr>
<td>6.2</td>
<td>Results Testbed 1, Temperature Accuracy</td>
<td>70</td>
</tr>
<tr>
<td>6.3</td>
<td>Results Testbed 6, Time Delay, 2 seconds between samples</td>
<td>72</td>
</tr>
<tr>
<td>6.4</td>
<td>Results Testbed 6, Time Delay, 1 second between Samples</td>
<td>73</td>
</tr>
<tr>
<td>6.5</td>
<td>Results Testbed 6, Time Delay, 1 second between Samples, two Motes</td>
<td>73</td>
</tr>
</tbody>
</table>
Pseudo & Source Code Listings

2.1 Measure Temperature in Celsius .......................... 14
2.2 Opening a connection ................................. 15
2.3 Receiving a Stream ................................ 15
2.4 Get IEEE Address .................................. 15
2.5 Sample of Datagram ................................. 15
2.6 Sample of Broadcast Datagram ....................... 16
3.1 Set Time on WSN Members .............................. 20
5.1 Example Pseudo Code ................................ 31
5.2 Pseudo Code for Main Class of Collector Motes .............. 36
5.3 Pseudo Code for Temperature Thread ..................... 36
5.4 Pseudo Code for Light Level Thread ..................... 37
5.5 Pseudo Code for Battery Capacity Thread ................. 38
5.6 Pseudo Code for Alarm Trigger Thread ................. 38
5.7 Pseudo Code for Carbon Monoxide Detector Thread ........ 39
| C.6 | Pseudo Code Alarm Trigger Thread | 99 |
| C.7 | Pseudo Code Carbon Monoxide Thread | 101 |
| C.8 | Pseudo Code Collector Main Thread | 102 |
| C.9 | Generic Functions of a Collector Mote | 104 |
| C.10 | Special Functions of a Temperature Collector Mote | 108 |
| C.11 | Special Functions of a Light Collector Mote | 111 |
| C.12 | Special Functions of a Battery Collector Mote | 114 |
| C.13 | Special Functions of a Alarm Trigger Mote | 116 |
| C.14 | Special Functions of a Carbon Monoxide Collector Mote | 117 |
| C.15 | Main Thread of Collector Mote | 120 |
| C.16 | Temperature Collector Thread | 125 |
| C.17 | Light Collector Thread | 129 |
| C.18 | Battery Collector Thread | 134 |
| C.19 | Alarm Trigger Thread | 137 |
| C.20 | Carbon Monoxide Collector Thread | 142 |
| C.21 | Pseudo Code Climate Interface Thread | 147 |
| C.22 | Pseudo Code Light Interface Thread | 148 |
| C.23 | Pseudo Code Alarm Interface Thread | 148 |
| C.24 | Pseudo Code Carbon Monoxide Interface Thread | 149 |
| C.25 | Pseudo Code Door Interface Thread | 150 |
### Nomenclature

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Analog to Digital Conversion</td>
</tr>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASK</td>
<td>Amplitude Shift Keying</td>
</tr>
<tr>
<td>BPSK</td>
<td>Binary Phase Shift Keying</td>
</tr>
<tr>
<td>CA</td>
<td>Collision Avoidance</td>
</tr>
<tr>
<td>CC2420</td>
<td>Radio transceiver used by Sun™ SPOT</td>
</tr>
<tr>
<td>CD</td>
<td>Collision Detection</td>
</tr>
<tr>
<td>CDMA</td>
<td>Code Division Multiple Access</td>
</tr>
<tr>
<td>CSMA</td>
<td>Carrier Sense Multiple Access</td>
</tr>
<tr>
<td>CSS</td>
<td>Chirp Spread Spectrum</td>
</tr>
<tr>
<td>DC</td>
<td>Direct Current</td>
</tr>
<tr>
<td>DDL</td>
<td>Data Link Layer</td>
</tr>
<tr>
<td>DG</td>
<td>Datagram</td>
</tr>
<tr>
<td>DSSS</td>
<td>Direct Sequence Spread Spectrum</td>
</tr>
<tr>
<td>ECC</td>
<td>Elliptic Curve Cryptography</td>
</tr>
<tr>
<td>ECDLP</td>
<td>Elliptic Curve Discrete Logarithmic Problem</td>
</tr>
<tr>
<td>ED</td>
<td>Energy Detection</td>
</tr>
<tr>
<td>GPIO</td>
<td>General Purpose Digital In/Output</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
</tr>
<tr>
<td>GTS</td>
<td>Guaranteed Time Slot</td>
</tr>
<tr>
<td>HAA</td>
<td>Home Automation Application</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>I/O</td>
<td>In / Output</td>
</tr>
<tr>
<td>ISM</td>
<td>Industrial, Scientific, Medical Band</td>
</tr>
<tr>
<td>LDO</td>
<td>Low Drop Out</td>
</tr>
<tr>
<td>LED</td>
<td>Light Emitting Diode</td>
</tr>
<tr>
<td>LQI</td>
<td>Link Quality Indication</td>
</tr>
<tr>
<td>LR-WPAN</td>
<td>Low-Rate Wireless Personal Area Network</td>
</tr>
<tr>
<td>MAC</td>
<td>Medium Access Control</td>
</tr>
<tr>
<td>Mote</td>
<td>Wireless Network Node</td>
</tr>
<tr>
<td>NO</td>
<td>Normal Open</td>
</tr>
<tr>
<td>OO</td>
<td>Object Orientated</td>
</tr>
<tr>
<td>O-QPSK</td>
<td>Offset Quadrature Phase Shift Keying</td>
</tr>
<tr>
<td>OSI</td>
<td>Open Systems Interconnection</td>
</tr>
<tr>
<td>OTA</td>
<td>Over The Air Deployment</td>
</tr>
<tr>
<td>PAN</td>
<td>Personal Area Network</td>
</tr>
<tr>
<td>PHY</td>
<td>Physical Layer</td>
</tr>
<tr>
<td>pSRAM</td>
<td>Pseudo-static Random Access Memory</td>
</tr>
<tr>
<td>PSSS</td>
<td>Parallel Sequence Spread Spectrum</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>SE</td>
<td>Standard Edition</td>
</tr>
<tr>
<td>SIG</td>
<td>Special Interest Group</td>
</tr>
<tr>
<td>SPI</td>
<td>Serial Peripheral Bus</td>
</tr>
<tr>
<td>SPOT</td>
<td>Small Programmable Object Technology</td>
</tr>
<tr>
<td>SPST</td>
<td>Single Pole Single Throw</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver/Transmitter</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
<tr>
<td>UWB</td>
<td>Ultra Wide Band</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>WPAN</td>
<td>Wireless Personal Area Network</td>
</tr>
<tr>
<td>WSN</td>
<td>Wireless Sensor Network</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Wireless devices have been used for a considerable time. Examples of wireless devices are radio, television and cordless phones. However, wireless devices only recently have been developed to interact with other wireless devices without the need for additional infrastructure. Wireless devices like radios and televisions receive data over a certain frequency and use the information embedded in the frequency to deliver audio and/or video. However, a television does not have the capability to communicate with other televisions. Cordless phone are able to receive and transmit voice signals via a phone network. The two phones require this phone network so the callers can communicate with each other. Recent development in wireless technologies have expanded the capabilities of wireless devices. Wireless devices are now able to interact with other devices. For example, a Bluetooth®-enabled mobile phone can communicate with a Bluetooth®-enabled GPS device. The two devices communicate directly without the need of other supporting communication networks or devices.

The original reason for the development of Bluetooth® was to remove the cables which connect input devices to a computer (Prasad & Deneire 2006). This allowed the user to move the input device without being restricted by the cable length. Unlike many other wireless devices, these new devices were designed for small ranges. The range of these devices has become known as Personal Area Networks (PAN). The development of Bluetooth® was used as a foundation for the IEEE 802.15.4.1 standard.
The concept of two way communication between wireless devices was expanded to create wireless sensor networks. Here a collection of sensor nodes, called motes, have the ability to collect information and send this information to a host computer or other motes. The motes may also have the ability to forward information from other motes or the host.

For this project the Small Programmable Object Technology (SPOT) developed by Sun Microsystems was utilised. Sun SPOT implements Wireless Personal Area Networks (WPAN) technology with the IEEE 802.15.4 standard. Currently a Sun SPOT is the size of a matchbox. When fully developed a Sun SPOT will be the size of a speck of dust.

1.1 Broad Aims and Specific Objectives

'This project aims to experiment with a Sun SPOT hardware development kit so as to set up a prototype testbed for a simple wireless sensor network' (Xiang 2009). The development kit included two Sun SPOTs, a base station and Sun SPOT manager. The Sun SPOT manager included the application Solarium which allowed the launching of virtual Sun SPOTs. The development kit also included the necessary documentation to develop Sun SPOT applications. The plan was to create a Home Automation Application (HAA) which would utilise physical and virtual motes.

In general a WSN requires code written for the host and separate code for the motes. In the case of a Sun SPOT, the programming language for the motes is Squawk Java VM. Once the application on the Sun SPOT is running, the mote collects data and then send the data to the host or other motes. The motes can be configured to forward any transmissions from other motes. The programming language for applications running on the host is Standard Edition (SE) Java VM. When the host application is started, it listens for any new data from the motes and processes the received data. If required, the host can send information to one or more motes in the network.

To successfully design and deploy the simple WSN the capabilities and limitations of
the different elements that make up the WSN needed to be understood. Specifically the hardware, software and the 802.15.4 standard needed to be investigated to successfully complete the project.

1.2 Background Information

A WSN depends on the interaction of hardware, software and network protocol. For this project the hardware consisted of a host computer, a base station and two Sun™ SPOTs. The host computer was an Intel® based Microsoft® Windows desktop computer. The base station had a main board and each Sun™ SPOT consisted of a main board and an eDemoBoard. The base station was connected to the desktop computer via a Universal Serial Bus (USB) cable and acted as the gateway to the WSN.

The main board on Sun™ SPOT contained the processor, memory, radio communication. (Sun 2008c, p.2). The main board had one power and one activity Light Emitting Diode (LED) and a power / reset switch. The eDemoBoard on the Sun™ SPOT held the internal sensors, user interfaces, analogue inputs and high current outputs (Sun 2008b, p.33).

Communication between the host and a Sun™ SPOT was achieved by USB or wireless communication. The wireless communication range of Sun™ SPOT using the CC2420 radio transceiver and antenna should be about 80 meters (Smith 2007).

A Sun™ SPOT is 40 mm wide, 70 mm long and 23 mm deep. When fully developed Sun™ expects to shrink them to the size of a speck of dust. Currently two research projects ("Smart Dust" and "Speckled Computing") are being conducted by Sun™ which focus on wireless networks sensor with size of 1 mm³ or less (Horan, Bush, Nolan & Cleal 2007). The battery on the Sun™ SPOT can be recharged via the USB mini Type B charger. A Sun™ SPOT does not have an operating system. The Java™ code runs directly on the hardware without the need for an operating system.

Figure 1.1 shows the eDemoboard of a Sun™ SPOT.
1.2 Background Information

A Sun™ SPOT operates in three modes:

- deep sleep;
- idle; and
- run.

The Sun™ SPOTs use the 802.15.4 standard to communicate. This standard is based on the original specification for Bluetooth® (Prasad & Deneire 2006, p.83). However, the 802.15.4 standard only "specifies the medium access control and the physical layer for low rate WPAN's" (Prasad & Deneire 2006, p.83, point 4). This standard is for low rate and low power applications, where ease of installation, reliable data transfer, short range operations, low cost and reasonable battery life is required (IEE 2006).

The applications for the host were written in SE Java™ VM and in Squawk Java™ VM for the Sun™ SPOTs. A comparison of the two is shown in Figure 2.2. All that was required to create, compile and deploy applications was a text editor and Software Development Kit (SDK) (current version jdk1.6.0_07) environment. However, to simplify some steps, NetBeans was utilised during the project since it offered an Integrated Development Environment (IDE).
Chapter 2

Literature Review

An important event for WPANs was the creation of a special interest group (SIG) in 1998 (Bluetooth 2009). The aim of the SIG was to eliminate communication cables (RS232, etc.) and utilise wireless communication instead.

The SIG developed Bluetooth®. The specification for Bluetooth® was used as a basis for 802.15.1. However, only certain elements of the Bluetooth® specification were standardised. Further work by task groups resulted in additional standards. One of these four original standards included the 805.15.4 standard. This standard specifies the medium access control (MAC) and physical layer (PHY) for Low Rate WPANs (LR-WPAN). Since 2006 the standard specifies four PHYs (IEE 2006).

The latest standard for IEEE 802.15.4 was released in 2006 with an amendment in 2007.

From (IEE 2006, p. 13) and IEEE 802.15.4a 2007 amendment:

"The main objectives of an LR-WPAN are ease of installation, reliable data transfer, short-range operation, extremely low cost, and a reasonable battery life, while maintaining a simple and flexible protocol. Some of the characteristics of an LR-WPAN are as follows:

- Over-the-air data rates of 851 kb/s, 250 kb/s, 100 kb/s, 40 kb/s, and
20 kb/s

- Star or peer-to-peer operation
- Allocated 16-bit short or 64-bit extended addresses
- Optional allocation of guaranteed time slots (GTSs)
- Carrier sense multiple access with collision avoidance (CSMA-CA) or ALOHA [ultra-wide band (UWB)] channel access
- Fully acknowledged protocol for transfer reliability
- Low power consumption
- Energy detection (ED)
- Link quality indication (LQI)
- 16 channels in the 2450 MHz band, 30 channels in the 915 MHz band, and 3 channels in the 868 MHz band, 14 overlapping chirp spread spectrum (CSS) channels in the 2450 MHz band, and 16 channels in three UWB bands (500 MHz and 3.1 GHz to 10.6 GHz)”

Wireless communication is more susceptible to noise and is less reliable than wired communication (Tanenbaum 2003, p.297). To make the transmission of data more robust the 802.15.4 uses Direct Sequence Spread Spectrum (DSSS). DSSS is a modulation technique which operates similarly to Code Division Multiple Access (CDMA). In CDMA (Ball 2008, p.2.24) a large number of users are allocated a large bandwidth and the users are allowed to transmit at the same time. Each user is assigned a unique spreading code.

In the case of IEEE 802.15.4, communication is achieved in a similar way. Each bit of data is transmitted via 11 chips (sub-bits) and using a Barker Sequence (Tanenbaum 2003, p.294). The Barker Sequence (Daintith 2004) is

“a sequence of symbols (binary or q-ary) that, when embedded in a string of randomly chosen symbols (from the same alphabet), has zero autocorrelation except in the coincidence position. Barker sequences are used to check, and if necessary to correct, the synchronization and framing of received data.”
2.1 MAC Layer

The chips do not carry any binary data (Zahariadis 2003, p.75). The protocol to identify which device is next to transmit is called CSMA and is implemented in the MAC layer (Prasad & Deneire 2006).

The IEEE 802.15.4 standard only specifies the MAC and PHY layers. These layers will be discussed now.

2.1 MAC Layer

The MAC Layer is a sublayer of the Data Link Layer (DLL) in the Open Systems Interconnection (OSI) Reference Model (Tanenbaum 2003). The MAC Layer needs 'to allocate a single broadcast channel among competing users' (Tanenbaum 2003). The task of the MAC layer is to establish which device will be the next one to access the communication channel. CSMA is a protocol to improve performance in a wireless network (Tanenbaum 2003, p.255). The CSMA type used in 802.15.4 is slotted (Prasad & Deneire 2006, p.101) and "· · · works as follows" (Tanenbaum 2003, p.256):

- station is ready to send;
- station senses the channel;
- if the channel is idle; and
- transmit with probability $p$.

To reduce the probability of collisions, CSMA can be implemented by either using CA or collision detection (CD). Since wireless radios usually can’t send and receive at the same time 802.15.4 specifies CSMA-CA (IEE 2006, p.iii) Tanenbaum (2003) notes that CSMA-CA makes the nodes in a WSN more polite, since the node will back off and wait a random amount of time before attempting to send the data. However this creates a problem if urgent data needs to be transmitted.
2.2 Physical Layer

As mentioned above, the standard for IEEE 802.15.4 defines four PHYs. From (IEE 2006)

"The standard now includes two optional physical layers (PHYs) yielding higher data rates in the lower frequency bands and, therefore, specifies the following four PHYs:

- An 868/915 MHz direct sequence spread spectrum (DSSS) PHY employing binary phase-shift keying (BPSK) modulation
- An 868/915 MHz DSSS PHY employing offset quadrature phase-shift keying (O-QPSK) modulation
- An 868/915 MHz parallel sequence spread spectrum (PSSS) PHY employing BPSK and amplitude shift keying (ASK) modulation
- A 2450 MHz DSSS PHY employing O-QPSK modulation"

The Sun™ SPOTs operate in the 2450 MHz Industrial, Scientific, Medical (ISM) band (Sun 2008c). For 802.15.4 communication the band is divided into 16 channels with each occupying 5 MHz (Sun 2009b, p.16). The frequencies are listed in Table 2.1. The data rate is 250 kB/s.

There is a downside in using the 2450 MHz band. This band is also used by 802.11b/g devices, cordless phones and microwave ovens. With the increasing number of devices deployed in this frequency range, channel conflict is becoming a frequent problem (Chen, Sun & Gerla 2006).

2.3 Network Topology

One characteristics of the IEEE 802.15.4 standard is that networks can be configured as star or peer-to-peer (Figure 2.1). Both network configurations may utilise a PAN Coordinator. In the star topology the PAN Coordinator is used to initiate, terminate,
2.3 Network Topology

<table>
<thead>
<tr>
<th>Channel</th>
<th>Centre Frequency (MHz)</th>
<th>Channel</th>
<th>Centre Frequency (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>2405</td>
<td>19</td>
<td>2445</td>
</tr>
<tr>
<td>12</td>
<td>2410</td>
<td>20</td>
<td>2450</td>
</tr>
<tr>
<td>13</td>
<td>2415</td>
<td>21</td>
<td>2455</td>
</tr>
<tr>
<td>14</td>
<td>2420</td>
<td>22</td>
<td>2460</td>
</tr>
<tr>
<td>15</td>
<td>2425</td>
<td>23</td>
<td>2465</td>
</tr>
<tr>
<td>16</td>
<td>2430</td>
<td>24</td>
<td>2470</td>
</tr>
<tr>
<td>17</td>
<td>2435</td>
<td>25</td>
<td>2475</td>
</tr>
<tr>
<td>18</td>
<td>2440</td>
<td>26</td>
<td>2480</td>
</tr>
</tbody>
</table>

Table 2.1: 802.15.4. Channel Assignment

Star Topology

Peer-to-peer Topology

Figure 2.1: 802.15.4 Topologies

or route communication around the network’ (IEE 2006, p.14). However, in a peer-to-peer network, the individual devices can communicate directly without any interaction with the PAN. Regardless of network topology each Sun\textsuperscript{TM} SPOT has a unique 64-bit address.

The functions for ad-hoc and self-healing networks are implemented in higher layers. As noted above, the 802.15.4 standard only considers the MAC and PHY layers. Therefore
it does not specify how a peer-to-peer network is to be implemented.

2.4 Security

There are a number of security issues which can have an impact on the integrity of the WSN. Olariu and Xu (2005) note that ‘a wireless sensor network is only as good as the information it produces’.

The IEEE 802.15.4 provides a minimum standard of security in the MAC layer. The CC2420 radio in the Sun™ SPOTs can provide additional security through Advanced Encryption Standard (AES) 128 hardware encryption. In addition ‘developers have implemented an Elliptic Curve Cryptography’ (ECC) (Boyle & Newe 2008) to increase security of the Sun™ SPOT. ‘ECC is a public-key cryptosystem that operates over points on an elliptic curve’ (Boyle & Newe 2008, p.291). Like other public-key cryptosystems ECC relies on a public and a private key. In ECC the Elliptic Curve Discrete Logarithmic Problem (ECDLP) is utilised to make ECC more efficient than other public-key cryptosystems.

(Boyle & Newe 2008, p.291) state that if:

\[ Q = kP \] (2.1)

Then, according to the ECDLP, if \( P \) and \( Q \) are known it is difficult to find \( k \). In the case of ECC \( k \) is the private key.

Since the HAA will operate in a reasonably safe environment the security of the actual sensor is not as critical. However, if the HAA will also have sensors which help to secure the home, the sensors will need to be installed so they cannot be lost, inadvertently removed or stolen.
2.5 Power supply

The Sun™ SPOTs in the development kit are powered by batteries, which need to be recharged on a regular basis. The challenge will be to build a power source for the motes when they are the size of a speck of dust. Chalasani & Conrad (2008) note that the following alternative sources for power are being considered.

- mechanical vibration;
- piezoelectric materials;
- solar cells; and
- thermoelectric.

Operating Modes:
A Sun™ SPOT has three operating modes. Based on which components on the Sun™ SPOT are required, the Sun™ SPOT can be placed into these different operating modes. The Owner’s Manual lists and explains the different modes (Sun 2008b, p.9).

**Run**  Basic operation with all processors and radio running. Power required by Sun™ SPOT is 70 to 120 mA. Power required by the daughterboard (if it is enabled) 400 mA.

**Idle**  ARM9 clock and the radio is turned off. Power requirement is approximately 24 mA.

**Deep-Sleep**  All regulators are off. The standby Low Drop Out (LDO), the power control of the Atmega and the pSDRAM are on. Power required is 32 µA. Typical startup time is 2 to 10 milliseconds.

Waking up the processor from deep-sleep can be achieved by the alarm, external interrupt or pressing the button power / reset button on the Sun™ SPOT (Sun 2008b).

To conserve the battery, the motes of the HAA are only running while data is collected and sent. For the remainder of the time the motes are in the idle or sleep mode.
2.6 Wireless Sensor Node (Mote)

The main board on a Sun™ SPOT contains the following main components: (Sun 2008c, p.2)

- main processor, Atmel AT91RM9200 system;
- memory, 4 MB Flash Memory and 512 KB pSRAM;
- power management circuit;
- CC2420 radio transceiver (802.15.4) and antenna;
- battery connector;
- daughter board connector;
- power and activity LEDs; and
- power / reset switch.

The eDemoBoard is a daughterboard and incorporates sensors, user interfaces, analogue inputs and high current outputs (Sun 2008b, p.33). The eDemoBoard has an 'Atmel Atmega88 micro controller and it communicates with the main board ARM9 over the Serial Peripheral Bus (SPI) channel as a slave device' (Sun 2008b).

The eDemoBoard fulfills the following functions (Sun 2009b, p.20):

**Internal Sensors:**

- temperature;
- acceleration (in three dimensions); and
- light level.
User Interfaces:

- two momentary single pole single throw (SPST) normal open (NO) push buttons; and
- eight tri-colour LEDs.

General Purpose Digital I/O (GPIO):
Four GPIOs can be configured as inputs or outputs. At room temperature the source current is 16 mA and the sink current is 24 mA (Sun 2008b).

Analogue Inputs:
Four analogue inputs 0 Volt to 3 Volt dc. (10 bit)

High Current Outputs:
Four high current outputs 125 mA.

Analogue to Digital Conversion (ADC):
The ADC is used to convert the analog signal of the accelerometer, light and temperature sensor into digital values (Sun 2008b).

2.7 SE Java VM and Squawk Java VM

As previously mentioned the PAN and the motes run on different applications. The applications on the PAN are written in SE Java™ VM. This programming language "is a general-purpose, concurrent, strongly typed, class-based object-oriented language." (Gosling, Joy, Steele & Bracha 1996).

The applications on the motes are written in Squawk Java™ VM. The aim of Squawk is to "write as much of the virtual machine as possible in Java" (The Squawk Virtual Machine n.d.). Figure 2.2 shows a comparison of the two virtual machines.

The Java™ VM is the technology which makes Java™ hardware and operating system independent (Lindholm & Yellin 1997). It is normally compiled to the bytecode
2.7 SE Java VM and Squawk Java VM

Figure 2.2: Comparison Standard Java VM versus Squawk Java VM

The Sun™ SPOT Owner’s Manual accompanying the development kit shows a number of simple examples. One such example shows the code to measure the temperature in Celsius.

```
package com.sun.spot.sensorboard;

public class EDemoBoard {
    public static void main(String[] args) {
        ITemperatureInput ourTempSensor = EDemoBoard.getCelsius();
        double celsiusTemp = ourTempSensor.getCelsius();
    }
}
```

Listing 2.1: Measure Temperature in Celsius

The example above only executes the collection of the data. After the last statement above has been executed the mote needs to send the information to other devices over the network. This task is implemented by using radio communication. There are two protocols to transmit information. The `RadioStreamConnection` interface is a 'buffered stream-based I/O between two devices' (Sun 2008b). The `RadiogramConnection` interface 'allows the exchange of packets between two devices' (Sun 2008b).

Before any data can be sent through a radiostream the connection needs to be est-

---

1Used with permission of Sun™ Microsystems.
established. Again a simple example from the Sun™ SPOT Owner’s Manual is shown. Comments have been added to help understand the statements.

```java
// Open connection
// -----------------------------------------------------------
StreamConnection conn = (StreamConnection)
// nnnn.nnnn.nnnn.nnnn -> unique 64-bit IEEE Address
// xxx -> port no. 0–255, 0–31 reserved for system services
Connector.open("radiostream:\\nnnn.nnnn.nnnn.nnnn:xxx");
```

Listing 2.2: Opening a connection

All devices need to open up the same port and each will need to list the unique IEEE address (Sun 2008b). It is possible however, for a mote to have more than one port open at a time. At this point in the code the device has established the connection. To receive the stream the following statement can be used:

```java
DataInputStream dataIn = conn.openDataInputStream();
```

Listing 2.3: Receiving a Stream

The device can establish the IEEE address with the statement:

```java
String myAdress = System.getProperty("IEEE_ADDRESS");
```

Listing 2.4: Get IEEE Address

As mentioned above the exchange of packets between devices is executed via the radiogram protocol. A simple example is shown below:

```java
DatagramConnection conn = (DatagramConnection)
Connector.open("radiogram://" + myAddress + ":100");
Datagram dg = conn.newDatagram(conn.getMaximumLength());
dg.writeUTF("Hello World!");
conn.send(dg);
```

Listing 2.5: Sample of Datagram

The radiogram protocol can broadcast messages across a network. There is no verification if the packet has been delivered (Sun 2008b). To broadcast the second statement in the example above is replaced with the following statement:
2.7 SE Java VM and Squawk Java VM

Listing 2.6: Sample of Broadcast Datagram

```
Connector.open("radiogram://broadcast:100");
```

The Sun™ SPOT Owner’s Manual recommends to keep datagrams smaller than 200 byte when using broadcast mode. The manual also recommends to send important data as unicast via radiograms or radiostreams. Figure B.1 in Appendix B shows how an application is compiled, deployed and started on the host while a Sun™ SPOT is connected to the host via USB.
Chapter 3

Methodology

3.1 Introduction

The aim of the project was to develop a simple WSN application. To fulfill the project aim and the project specification in Appendix A, the following methodology was used:

1. select scenario to implement SunSPOT testbed;
2. define testbeds within selected scenario;
3. design and implement WSN;
4. verify accuracy of data collected by the testbeds;
5. verify performance (time delays, etc) of the testbeds; and
6. document findings and results.

This project focused on the task of programming in Java\textsuperscript{TM}. Therefore, throughout the project the main focus was on program development and not hardware development. Hardware development was only considered if it was not possible to complete a test bed with the hardware provided in the development kit.

Each of the above mentioned steps will be discussed in the following sections.
3.2 Select Scenario for WSN

The project aim did not outline any specific scenario for the testbed. Therefore the first task was to determine a suitable task for the WSN. Two main limiting factors needed to be considered during the selection process. These factors were:

- the development kit had only two Sun\textsuperscript{TM} SPOTs; and
- the physical characteristics of the motes supplied with the development kit.

A main advantage however, was the fact that the eDemoBoard had built-in sensors which had the ability to measure environmental data. This meant that a number of different environmental samples could be collected without the need for additional hardware. The accuracy of the sensor on the eDemoBoard needed to be verified by using suitable testbeds.

Through a process of elimination the idea of setting up a WSN in a home was selected. Figure E.1 in Appendix E shows the layout of the home selected as a base for the implementation of the HAA and the definition of the testbeds. The abilities of the eDemoBoard allowed the measurement of temperature and light levels without the need to develop any additional hardware. A number of other scenarios where discarded since significant additional hardware was required or the physical characteristics of the mote prevented a realistic scenario. Some of these discarded scenarios are listed below.

- roller door remote control;
- sensor to establish if an elderly or disabled person has fallen;
- remote to turn lights on or off in a room;
- establish how quickly liquid is flowing in a pipe;
- establish moisture content in soil to optimise irrigation; and
- exercise tool for rehabilitation of arm or hand movement.
3.3 Define Testbeds

Once the decision had been made to develop the WSN in a home environment, suitable testbeds had to be selected. Each testbed had to simulate a different aspect of the HAA.

The design and implementation of the computer code however, needed to ensure that the different testbeds could be implemented without the need of major code changes. In addition, each testbed needed to generate a result. Therefore each testbed had to be selected so data could be collected and used to investigate the performance and the accuracy of the WSN. The testbeds for the HAA were as follows:

- Testbed 1: Accuracy of WSN, Internal Sensor.
- Testbed 2: Accuracy of WSN, External Sensor.
- Testbed 3: Maximum Distances between WSN Members.
- Testbed 4: Impact of Noise Sources on WSN.
- Testbed 5: Multifunction Data Collection.
- Testbed 6: High Volume Data Collection.

Testbed 2 was only defined in case the results in testbed 1 were deemed to be not accurate enough for the HAA.

Chapter 4 will describe the individual testbeds in greater detail.

3.4 HAA Design and Implementation

As stated above the application had to be designed in such a way that all the testbeds could be executed with the same code. A detailed description of the design and implementation process is shown in chapter 5. In addition the chapter includes pseudo code for the main functionalities.
3.5 Verification Accuracy and Performance

Once the HAA had been implemented the various testbeds were configured and data collected to gather the results. To verify the accuracy of the data collected by the WSN external independent measuring instruments were used. The light levels were verified with a light meter. The temperature data were verified with a temperature meter.

To verify the performance the time on the host, base station and the motes were synchronised using the command shown in Listing 3.1 before each test. The command was executed from the command line interface on the host computer.

\texttt{ant settime}

Listing 3.1: Set Time on WSN Members

After synchronisation a reference measurement was taken to establish how long it took to:

- take one sample;
- send sample via broadcast over the WSN; and
- display the sample on the host computer terminal.

Chapter 6 will discuss these results and draw conclusions.
Chapter 4

HAA Test Beds

4.1 Introduction

This chapter will discuss the test bed configurations. Due to the limited output capabilities of the physical Sun® SPOTs only screen captures of virtual Sun® SPOTs were possible. During the early stages of setting up the test bed the limitations of the processing power of the desktop computer were discovered. It became quite clear that deploying too many virtual Sun® SPOTs put a noticeable strain on the performance of the desktop computer.

4.2 Overview Test Bed Setup

The experimental test beds for the HAA were set up as shown in Figure 4.1. The testbed consisted of the following hardware:

- one desktop computer;
- two Sun® SPOTs; and
- one base station connected to the desktop computer via USB.
The desktop ran multiple applications to fulfill various tasks:

- **NetBeans IDE 6.5.1** to launch HAA;
- Sun SPOT Manager to launch Solarium;
- Solarium, to deploy and run the virtual Sun SPOTs; and
- HAA, the actual application which collected data and interacted with the other members of the WSN via the base station.

The number of virtual Sun SPOTs deployed during the tests were limited by the processing power of the desktop computer. The screen capture D.5 in Appendix D shows the performance via the Microsoft Windows XP Task Manager. The screen capture shows six virtual motes. Five virtual motes were configured to collect only one environmental data type and transmit the data via broadcast to one virtual interface mote. As specified in Section 5.3 a sample was forwarded every ten seconds from each virtual collector mote.

To reduce the possibility of complications during the tests, the number of virtual motes was limited to six. The process to set up and deploy the motes for the WSN is described below.
4.2 Overview Test Bed Setup

4.2.1 Deploying Application to Physical Motes

1. define configuration data in MANIFEST.MF file;
2. connect mote via USB to desktop computer;
3. deploy the application to the mote;
4. run application while mote was still connected to desktop computer;
5. verification that the mote was working properly;
6. disconnect USB; and
7. turn off mote.

4.2.2 Deploying Application to Virtual Motes

To configure the virtual motes, Solarium was launched from the Sun\textsuperscript{TM} SPOT Manger.

1. define configuration data in MANIFEST.MF file;
2. create a new virtual mote in Solarium by:
   - selecting [Emulator] >> [New Virtual SPOT];
   - right-click on virtual mote and select [Deploy MIDlet bundle...];
   - navigate to and select required build.xml file; and
   - click [Open]
3. right-click on virtual SPOT to display sensor panel and / or output window.

4.2.3 Deploying Host Application

The host application was launched from NetBeansIDE6.5.1.
4.2.4 Starting Members of HAA WSN

Once all the members of the WSN had been configured they were deployed in the order shown below:

1. launch host application;
2. launch interface motes; and
3. launch collector motes.

4.3 Test Beds

4.3.1 Test Bed 1: Accuracy of WSN, Internal Sensor

As mentioned in Chapter 2, the WSN is only as good as the data it delivers. Therefore the first testbed verified the accuracy of the collected data. The configuration for this testbed required two motes to collect environmental data. Each mote collected a different type of environmental data. Due to the limited user interface capabilities of the Sun\textsuperscript{TM} SPOTs two possible methods to show the collected data were considered:

1. Connect the data collecting mote via the USB cable to the host computer. The accuracy of the collected data would be verified by the data pushed directly to the terminal by the mote.
2. Design and implement a simple application on the host computer which only pushes the received data to the terminal.

The second method was considered to be more realistic, since the data had to travel via the WSN to reach the host. It was also possible to move the motes to different parts of the home to establish if the accuracy changed if the distance to the host computer was increased.
Next the type of environmental data to be collected was selected. The obvious choice was to utilise the internal sensors on the eDemoBoard. Therefore, for this testbed the temperature and light level were to be measured and sent via broadcast to the host and motes. Some independent means however, were required to verify the accuracy of the collected data.

Based on the considerations above the requirements for the first testbed were:

- Data to be collected: One Sun\textsuperscript{TM} SPOT to measure temperature and one Sun\textsuperscript{TM} SPOT to measure the light level.
- Time between samples: 20 seconds.
- Receivers: Host and virtual motes.
- Independent means for verification of collected data: light and temperature meter.
- Result: Identify errors in collected data.

### 4.3.2 Test Bed 2: Accuracy of WSN, External Sensor

This test bed was only going to be executed if the accuracy of the internal sensors were deemed not accurate enough for the HAA. In that case, an external level sensor would have to be developed and then connected to the Sun\textsuperscript{TM} SPOT in order to increase accuracy. All the remaining requirements remained the same as defined for the first testbed. Therefore based on the considerations above the requirements for this testbed were:

- Data to be collected: One Sun\textsuperscript{TM} SPOT to measure temperature or light level with an external sensor.
- Time between samples: 20 seconds.
- Receivers: Host and virtual motes.
- Independent means for verification of collected data: light or temperature meter.
- Result: Identify any errors in collected data.
4.3 Test Beds

4.3.3 Test Bed 3: Maximum Distances between WSN Members

This testbed expanded the requirements for the previous testbeds. In this testbed the maximum possible distance between members of the WSN was to be investigated. As before one physical mote needed to collect a specific environmental data and send the collected data to the host and/or motes. While the data was being collected the mote was to be moved throughout the home. After each relocation of the mote it was to be verified if data was still being received by the host. If data was still being received then the mote was to be moved further away from the host. Based on these considerations the requirements for this testbed were:

- Data to be collected: One Sun™ SPOT to measure environmental data.
- Receivers: Host and virtual motes.
- Measuring tape to establish the distance between the mote and the host.
- Result: Maximum possible distance between mote and host.

4.3.4 Test Bed 4: Impact of Noise Sources on WSN

This testbed had to establish if other electronic devices operating in the ISM frequency band had a negative impact on the performance of the WSN. Major other equipment which operates in the ISM band are microwave ovens, wireless networks and cordless phones. To establish if these devices had an impact on the performance of the WSN a Sun™ SPOT was to be placed adjacent to these devices.

The details of the available noise sources are:

- GE Microwave, Spacemaker XL1800, output frequency 2450 MHz; and
- LINKSYS Wireless Router, WRT54G, frequency 2.4 GHz.

Based on these considerations the requirements for this testbed were:
4.3 Test Beds

- Data to be collected: One Sun\textsuperscript{TM} SPOT to measure environmental data.
- Mote placed adjacent to microwave and wireless router.
- Receivers: Host and virtual motes.
- Independent means for verification of collected data.
- Result: Performance impact if WSN is operated near the above mentioned noise sources.

4.3.5 Test Bed 5: Multifunction Data Collection

Since the eDemoBoard had multiple sensors, it was deemed to be practical for a mote to collect more than one type of data. Therefore, if a mote could collect and send more than one type of data the receiving host or mote would need to distinguish between different types of environmental data. Hence the computer code for data collection, data transmission and data receiving needed to be implemented for multiple data types.

Based on these considerations the requirements for this testbed were:

- Data to be collected: One Sun\textsuperscript{TM} SPOT to measure more than one environmental type of data.
- Receivers: Host and virtual motes.
- Result: The data collected, sent and received needs to be kept separate and not misread by the receivers.

4.3.6 Test Bed 6: High Volume Data Collection

This testbed investigated the capability of receivers in the WSN. A number of physical and virtual motes were to send data at progressively shorter intervals to the host. The testbed investigated how much data could be received by a single member before delays in processing the data were noticeable.

Based on these considerations the requirements for this testbed were:
• Data to be collected: Single data type collected by as many as possible physical and virtual motes.

• Receivers: Host.

• Result: Delay in processing and displaying the data.
Chapter 5

Implementation

5.1 Introduction

During the initial familiarisation in programming applications for Sun™ SPOTs it became quite clear that a runtime error could prevent access via the USB port or Over The Air (OTA) because the Sun™ SPOT was constantly resetting. Therefore in addition to using virtual motes to extend the number of members in the WSN, virtual motes were used to test any new applications or changes to existing applications. The process of deploying and running an application on a virtual SPOTs is shown in Chapter 4.

As previously mentioned, the Sun™ SPOTs in the development kit included an eDemoBoard. The eDemoBoard has sensor and user interfaces which can be accessed via an application running on the Sun™ SPOT. The capabilities of the eDemoBoard were utilised for this project. Therefore, no additional hardware had to be designed. There was a possibility that the internal sensors could be affected by other components on the eDemoBoard. If the test beds would show that this was the case then an external sensor would have to be designed. The sensors and user interfaces of the eDemoBoard were also available on the virtual motes. The capabilities of the eDemoBoard are listed in Chapter 2.

To ensure that all the different components of the WSN properly interacted with each
other clear requirements were needed. Clear requirements were also necessary as a basis for testing and implementation of the applications. Therefore this chapter will include the following sections:

- design and development process;
- requirements;
- test scripts; and
- implementation.

## 5.2 Design and Development Process

The code for the motes was implemented using the following process:

1. define requirement(s);
2. write pseudo code based on requirement(s);
   - if required, draw flowchart;
3. write a test script which verifies access to instance variables and method;
4. write the actual code;
5. run test script in Solarium and improve if necessary;
6. add new instance variables and methods to new or existing mote class;
7. deploy and run the real code on virtual Sun™ SPOTs in Solarium; and
8. deploy and run the real code on physical Sun™ SPOTs.

A simple method will be used to show the above process:

1. Requirement.
   Before a collector mote sends data it needs to verify that the number of digits after the decimal point is limited to two.
5.2 Design and Development Process

2. Pseudo code.

```java
double reformatDataToBeSent(double valueIn)
   Task: remove excess digits after decimal point before sending
   Input: sampled data to be sent
   Output: number limited to 2 digits after the decimal point

   Define local variables
   double valueOut

   valueIn --> Move decimal point 2 digits to right --> assign toValueIn
   valueIn --> Round up or down --> assign to valueIn
   valueIn --> Move decimal point 2 digits to left --> assign toValueOut

   return valueOut
end of reformatTempToBeSent()
```

Listing 5.1: Example Pseudo Code

3. Test script.

   The test script is listed in the Appendix C.1.

4. Write actual code.

   The actual code is listed in the Appendix C.2.

5. Run the test script in Solarium.

   The screen capture of output from the test script is shown in Appendix D.1.

6. Add new instance variables and methods to new or existing class.

   For this example the tested method was added to the collector.java class. The listing for the common functionality of a collector mote is shown in Appendix C.9.

7. Deploy and run the actual code on virtual Sun SPOTs in Solarium.

   The screen capture of the host output is shown in Appendix D.2.

   The screen capture of the virtual Sun SPOT output is shown in Appendix D.3.

8. Deploy and run the actual code to physical Sun SPOTs.

   For this method it was only possible to verify that the method operated correctly on a physical mote by analysing the data received at the host. All data received by the host was to have no more than two digits after the decimal point.
5.3 Requirements

In broad terms the HAA had to include the following functions:

1. collect environmental data;
2. forward data to external systems;
3. log events and environmental data on host computer; and
4. provide user interface on host computer.

The HAA was not going to be able to make a proper assessment of the current status in the home unless a number of Sun\textsuperscript{TM}SPOTs were deployed to gather environmental data. The WSN was be made up of two types of motes. The main task of each type of mote were:

**Collector mote**: Collects environmental data and forwards to host or other motes.

**Interface mote**: Receives data from host or other motes and forwards to external systems.

5.3.1 WSN Configuration

The Sun\textsuperscript{TM}SPOTs were to be configured to accept OTA commands and to enable OTA downloads (Sun 2009\textsuperscript{a}, p.22). Every Sun\textsuperscript{TM}SPOT deployed for the HAA had to have mesh routing enabled. If mesh routing is enabled, the motes will rebroadcast packets destined for other members of the WSN. This will instruct the SPOT to forward any packet intended for other SPOTs (Sun 2009\textsuperscript{a}, p.26).

Each member can have multiple ports open at any one time. Therefore the ports shown in Table 5.1 were used to communicate between the different members of the WSN.
5.3 Requirements

<table>
<thead>
<tr>
<th>Sender</th>
<th>Type of Communication</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Host</strong></td>
<td>Send IEEE Address (broadcast)</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>Listen for environmental data</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>Listen for status</td>
<td>68</td>
</tr>
<tr>
<td><strong>Interface Mote</strong></td>
<td>Listen for host</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>Listen for environmental data</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>Send status to host (unicast)</td>
<td>68</td>
</tr>
<tr>
<td><strong>Collector Mote</strong></td>
<td>Send environmental data (broadcast)</td>
<td>67</td>
</tr>
</tbody>
</table>

Table 5.1: Port Allocation

5.3.2 Data Collection

Most of the data will come from different parts of the home. It is a requirement however that the outside temperature and light levels were also captured by the motes.

To streamline the deployment of the application to the individual motes a single multifunctional application was to be designed for the data collection motes.

The data to be collected was as follows:

- temperature;
- light Level;
- battery capacity;
- alarm trigger; and
- carbon monoxide level.

Each data collection function was to run in a separate thread. In addition a data collection function (thread) was only to be started if required by the configuration file. For the Sun™ SPOTs the configuration file is called the MANIFEST.MF file. The developer guide (Sun 2009a, p. 17) states that
In addition the MANIFEST.MF file can be used to define properties which the Sun\textsuperscript{TM} SPOT application can access at runtime. The details of the MANIFEST.MF file will be further explained in section 5.5.

To identify the message type at the receiver, data sent by the collector motes had to be identified with unique value of type "byte". This would allow the other members of the WSN to establish what type of information is included in the packet. The message types are defined in table 5.2.

In addition the collector mote also needed to send the location to inform the receivers where the data was collected. This allowed the interface motes and the host to establish where the data was sampled. The location information had to be of type "String". The locations were defined in table 5.3.

To conserve power the data collection motes would be awake only if a sample was to be taken. In addition data was to be transmitted only if the new sampled value differed from the previous measurement. For trouble shooting purposes however, any sampling or sending events were to be indicated by using the LEDs located on the eDemoBoard. This would have an impact on how quickly the battery drained, but helped during testing of the different modules in the computer code.

The measured environmental data was to be sent via broadcast datagram to other

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Message Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Temperature</td>
</tr>
<tr>
<td>2</td>
<td>Light</td>
</tr>
<tr>
<td>3</td>
<td>Battery</td>
</tr>
<tr>
<td>4</td>
<td>Alarm</td>
</tr>
<tr>
<td>5</td>
<td>Carbon Monoxide</td>
</tr>
</tbody>
</table>

Table 5.2: Message Type Identifier Collector Motes

"the MANIFEST in the resource/META-INF directory contains information used by the Squawk VM to run the application. In particular it contains the name of the initial class".
members of the WSN. Table 5.4 shows how often the data was to be sampled. When sending a sample the sample time had to be included. The flowchart B.3 in Appendix B provides an simplified overview of the temperature collection thread.

Table 5.5 shows how often samples are to be sent via the WSN and defines the recipients of the data. However the newest data is only sent if it differs from the previous sample. The data to be sent is to be limited to two digits after the decimal point.

<table>
<thead>
<tr>
<th>Data</th>
<th>Sample Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temperature</td>
<td>1 min.</td>
</tr>
<tr>
<td>Light Level</td>
<td>1 min.</td>
</tr>
<tr>
<td>Battery Capacity</td>
<td>30 min.</td>
</tr>
<tr>
<td>Alarm Trigger</td>
<td>1 sec.</td>
</tr>
<tr>
<td>Carbon monoxide</td>
<td>10 sec.</td>
</tr>
</tbody>
</table>

Table 5.4: Collected Environmental Data

<table>
<thead>
<tr>
<th>Data</th>
<th>Receivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temperature</td>
<td>Host and Climate Control Interface Mote</td>
</tr>
<tr>
<td>Light Level</td>
<td>Host and Light Control Interface Mote</td>
</tr>
<tr>
<td>Battery Capacity</td>
<td>Host</td>
</tr>
<tr>
<td>Alarm Trigger</td>
<td>Host and Alarm System Interface Mote</td>
</tr>
<tr>
<td>Carbon monoxide</td>
<td>Host and Carbon Mono Control Interface Mote</td>
</tr>
</tbody>
</table>

Table 5.5: Receivers of Collected Data
5.3 Requirements

Based on the considerations mentioned above the pseudo code for the main functionalities of the collectors motes were developed. The complete pseudo codes are listed in Appendix C.

The Pseudo code for the main class of a Collector Mote is shown in Listings 5.2. The main task for this class is to configure the data collecting mote based on the configuration file. Once the threads have started the main class waits until all other threads have finished. The Flowchart B.2 in Appendix B provides a simplified overview of this thread.

<table>
<thead>
<tr>
<th>Pseudo Code for Main Class of a Collector Mote</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance Variables</td>
</tr>
<tr>
<td>Variables for MANIFEST.MF properties (Strings)</td>
</tr>
<tr>
<td>Variables for MANIFEST.MF properties</td>
</tr>
<tr>
<td>Define boolean variables to set functions of mote, set to FALSE</td>
</tr>
<tr>
<td>protected void startApp()</td>
</tr>
<tr>
<td>Monitor the USB Port (if connected) &amp; recognize commands from host</td>
</tr>
<tr>
<td>Make a new instance of possible functions of mote and set to NULL</td>
</tr>
<tr>
<td>Get configuration data from Manifest file</td>
</tr>
<tr>
<td>Convert properties of MANIFEST file to appropriate data type</td>
</tr>
<tr>
<td>Set required functions of mote based on properties of MANIFEST.MF</td>
</tr>
<tr>
<td>Start new threads on a SunSPOT depending Manifest file</td>
</tr>
<tr>
<td>Wait for functions (threads) to end</td>
</tr>
<tr>
<td>END of startApp</td>
</tr>
<tr>
<td>END of class</td>
</tr>
</tbody>
</table>

Listing 5.2: Pseudo Code for Main Class of Collector Motes

Based on the configuration file one to five threads to collect data are started. The pseudo codes for each thread are listed over the next few pages.

Temperature Collection Function

The pseudo code for the temperature functionality is as follows:

<table>
<thead>
<tr>
<th>Pseudo Code for Temperature Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>TempThread implements Runnable</td>
</tr>
<tr>
<td>public void run()</td>
</tr>
<tr>
<td>Start a new temperature collector mote</td>
</tr>
<tr>
<td>Initial set up for measurement of temperature</td>
</tr>
<tr>
<td>Initial set up for communication</td>
</tr>
</tbody>
</table>
5.3 Requirements

Get and push IEEE Address to terminal
Open up a broadcast connection
WHILE (true)
    Get the current time when sample is taken
    Get the current temperature reading to send in datagram
    Reformat tempToBeSent to ****.
    IF (has Temperature Changed)
        Update time and temperature value to be sent
        Send current sample
    END IF (has Temperature changed)
    Move current temperature sample to previous sample variable
    Go to sleep to conserve battery
END of WHILE
END of run()
END of class

Listing 5.3: Pseudo Code for Temperature Thread

The Flowchart B.3 in Appendix B provides a simplified overview of the temperature thread.

Light Level Collection Function

The pseudo code for the light level functionality is as follows:

Pseudo Code for Light Level Thread
LightThread implements Runnable
public void run()
    Start a new light level collector mote
    Initial set up for measurement of light
    Initial set up for communication
    Get and push IEEE Address to terminal
    Open up a broadcast connection
    WHILE (true)
        Get the time when sample is taken
        Get the current light reading to send in datagram
        Reformat lightToBeSent to ****.
        IF (has Light Changed)
            Update time and light value to be sent
            Send current sample
        END IF (has Light changed)
        Move current light sample to previous sample variable
5.3 Requirements

Go to sleep to conserve battery
END of WHILE
END of run()
END of class

Listing 5.4: Pseudo Code for Light Level Thread

Battery Capacity Function

The pseudo code for the battery capacity functionality is as follows:

```
Pseudo code for battery capacity Thread
BatLevelThread implements Runnable
public void run()
   Start a new battery level collector mote
   Initial set up for measurement of battery level
   Initial set up for communication
   Get and push IEEE Address to terminal
   Open up a broadcast connection
   WHILE (true)
      Get the time when sample is taken
      Get the current battery reading to send in datagram
      IF (battery level has changed)
         Update time and battery value to be sent
         Send current sample
      END IF (battery level has changed)
      Move current battery sample to previous sample variable
      Go to sleep to conserve battery
   END of WHILE
END of run()
END of class
```

Listing 5.5: Pseudo Code for Battery Capacity Thread

Alarm Trigger Function

The pseudo code for the alarm trigger functionality is as follows:

```
Pseudo code for Alarm Trigger Thread
AlarmThread implements Runnable
public void run()
   Start a new alarm trigger mote
   Initial set up for alarm trigger
```

5.3 Requirements

Initial set up for communication
Get and push IEEE Address to terminal
Open up a broadcast connection

WHILE (true) {
    WHILE(no Alarm, i.e. input is low)
        Go to sleep for specified time
        Get the time when sample is taken
    END of WHILE (no Alarm, input is low)
    Send message to notify that alarm has been triggered
    Update alarm values to be sent
    Package time & alarm trigger into a radio datagram & send
    Stop the thread until the input has been set to low
        Only send "alarm triggered" once
    Wait for pin to change state
    When pin has changed reenable pin change interrupt
    IF (confirm no Alarm)
        Get the current time
        Update alarm values to be sent
        Package time & alarm trigger into radio datagram & send
    END IF (confirm no Alarm)
END of WHILE (true)
END of run()
END of class

Listing 5.6: Pseudo Code for Alarm Trigger Thread

Carbon Monoxide Detector Function

To implement the carbon monoxide detection an anlogue input was utilised. No actual
hardware was designed or implemented. It was however assumed that the carbon
monoxide detector would provide the carbon monoxide levels by varying the voltage at
the output. This change in output would be read by the carbon monoxide function on
a data collecting mote.

The pseudo code for the Carbon Monoxide detection functionality is as follows:

Pseudo code for carbon monoxide detector thread
CarMonThread implements Runnable
public void run()
    Start a new carbon monoxide detector collector mote
    Initial set up for measurement of carbon monoxide
5.3 Requirements

Initial set up for communication
Get and push IEEE Address to terminal
Open up a broadcast connection
WHILE (true)
  Get the time when sample is taken
  Get the current carbon mono reading to send in datagram
  Convert Sample to volt
  Reformat sample to be sent to ***.** format
  IF (has Carbon Monoxide Level Changed)
    Update time and carbon monoxide value to be sent
    Send current sample
  END of IF (has Carbon Monoxide Level Changed)
  Move current carbon level to previous carbon level variable
  Go to sleep to conserve battery
END of WHILE
END of run()
END of class

Listing 5.7: Pseudo Code for Carbon Monoxide Detector Thread

5.3.3 Interface to External Systems

For the interface motes the following assumptions were made:

- The motes would be located near the external system.
- The motes would communicated via Universal Asynchronous Receiver/Transmitter (UART) with the external system.
- The motes would have reliable power to make them less reliant on the battery.
- The motes would be running for most of the time to listen for incoming data.

The assumption that the communication to the external system was via UART was made to reduce the number of possible ways to communicate with external systems. An exception was made for the door interface. Here it was assumed that one of the
5.3 Requirements

<table>
<thead>
<tr>
<th>Identifier</th>
<th>External System Identification</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Climate Control</td>
</tr>
<tr>
<td>12</td>
<td>Light Control</td>
</tr>
<tr>
<td>13</td>
<td>Not used</td>
</tr>
<tr>
<td>14</td>
<td>Alarm System</td>
</tr>
<tr>
<td>15</td>
<td>Carbon Monoxide Detection System</td>
</tr>
<tr>
<td>16</td>
<td>Door Interface</td>
</tr>
</tbody>
</table>

Table 5.6: External System Identification for Interface Motes

Analogue or digital outputs would be used to drive a solenoid to unlock the door. All interface motes however, should announce themselves to the WSN and communicate the status of the external interface. This will allow the host to keep track if the external system is operating.

The interface motes are to have the following capabilities:

- receive data from data collection motes;
- output the received information to the external system;
- trigger message if status of external system had changed;
- announce presence to other WSN members;
- send a periodic signal to host to indicate status of external system; and
- listen for messages from host.

Once an interface mote has been deployed it will listen for datagrams. If a datagram is received from a collector SPOT the interface SPOT verifies that the correct message type is included. If the message type is correct the interface SPOT will forward the data to the external system. In addition the interface will listen for broadcast from the host. If a broadcast message from the host is received, the interface mote will communicate to the host via unicast radiogramm connection and report the status of the external system in regular intervals.
5.3 Requirements

The pseudo code for an interface mote is as follows:

```java
public class InterfaceMainSPOT extends MIDlet

Instance variables

protected void startApp()

    Monitor the USB (if connected) and recognize commands from host
    Get configuration data from Manifest file to configure mote
    Convert Strings to appropriate data type
    Find external system status id based on MANIFEST.MF file
    Start new thread on a SunSPOT depending on MANIFEST.MF file

    SWITCH (external System Id)
        case 11:
            new ClimateInterfaceThread
            new sendStatus thread
            Wait for threads to finish
        case 12:
            new LightingInterfaceThread
            new sendStatus thread
            Wait for thread to finish
        case 14:
            new AlarmInterfaceThread
            new sendStatus thread
            Wait for thread to finish
        case 15:
            new CarbMonInterfaceThread
            new sendStatus thread
            Wait for thread to finish
        case 16:
            new DoorInterfaceThread
            new sendStatus thread
            Wait for thread to finish

    END of SWITCH (external System Id)

    END of startApp()

END of class
```

Listing 5.8: Pseudo Code for Main Class of Interface Mote

The Flowchart B.4 in Appendix B provides a simplified overview of the main interface mote thread.
5.3 Requirements

Climate Control Interface

The pseudo code for the climate control interface is as follows:

```java
Pseudo code for Climate Control Interface Thread
ClimateInterfaceThread implements Runnable
public void run() {
    Start a new climate interface mote
    Get and push IEEE address
    Initial set up for communication
    Open up a broadcast connection
    Initial set up for local communication with external system
    WHILE (listen for New Datagram)
        Listen for new data packet
        IF (data is for Climate Control)
            Prepare data to send to external system via serial comm.
            Push new data to external system
        END of (data is for Climate Control)
    END of WHILE
    END run()
END of class
```

Listing 5.9: Pseudo Code for Climate Control Interface Thread

The Flowchart B.5 in Appendix B provides a simplified overview of the climate control interface mote thread.

Light Control Interface

The pseudo code for the light control interface is as follows:

```java
Pseudo Code for Light Control Thread
LightingInterfaceThread implements Runnable
public void run() {
    Start a new light interface mote
    Get and push IEEE address
    Initial set up for communication
    Open up a broadcast connection
    Initial set up for local communication with external system
    WHILE (listen for New Datagram)
        Listen for new data packet
        Verify that packet has light data
        IF (data is for light control)
```

Listing 5.9: Pseudo Code for Climate Control Interface Thread
5.3 Requirements

Prepare data to send to external system via serial comm.
Push new data to external system
END of IF (listen for new data packet)
END of WHILE
END run()
END class

Listing 5.10: Pseudo Code for Light Control Interface Thread

Alarm System Interface

The pseudo code for the alarm system interface is as follows:

Pseudo Code for Alarm System Thread
AlarmInterfaceThread implements Runnable
public void run()
   Start a new alarm interface mote
   Get and push IEEE address
   Initial set up for communication
   Open up a broadcast connection
   Initial set up for local communication with external system
   WHILE (listen For New Datagram)
   Listen for new data packet
   IF (data is for Alarm System)
      Prepare data to send to external system via serial comm.
      Push new data to external system
   END of IF (data is for Alarm System)
   END of WHILE
END run()
END class

Listing 5.11: Pseudo Code for Alarm System Interface Thread

Carbon Monoxide Control Interface

The pseudo code for the Carbon Monoxide Control interface is as follows:

Pseudo Code for Carbon Monoxide Control Thread
CarbMonInterfaceThread implements Runnable
public void run()
   Start a new carbon monoxide interface mote
   Get and push IEEE address
   Initial set up for communication
5.3 Requirements

Open up a broadcast connection
Initial set up for local communication with external system
WHILE (listen For New Datagram)
    Listen for new data packet
    IF (data is for CarbonMono control)
        Prepare data to send to external system via serial comm.
        Push new data to external system
    END of IF (data is for CarbonMono control)
END of WHILE
END of run()
END of class

Listing 5.12: Pseudo Code for Carbon Monoxide Control Interface Thread

Door Interface

The pseudo code for the door interface is as follows:

Pseudo Code for Door Interface
DoorInterfaceThread implements Runnable
public void run() {
    Start a new door interface mote
    Get and push IEEE address
    Initial set up for communication
    Open up a broadcast connection
    Initial set up for local communication with external system
    WHILE (listen For New Datagram)
        Listen for new data packet
        IF (data is for door)
            Prepare data to send to the external system via GPIO output
            Push new data to external system
        END of IF (data is for door)
    END of WHILE
END run()
END class

Listing 5.13: Pseudo Code for Door Interface Thread

External System Status

The pseudo code for the external system status is as follows:

Pseudo Code to send external system status
sendStatus implements Runnable

public void run()
{
    Create object to send status to host
    Initial setup for communication (listening)
    Open up a broadcast connection to listen for host
    WHILE (hostAddressNotKnown){
        Listen for host broadcast
        IF (broadcast is from host)
            Get host address
            Set flag to false (i.e. address is now known)
            Sleep for some time
        END of IF (broadcast is from host)
    END of WHILE
    Create object to send status to host
    Initial setup for communication (sending)
    Open up a radiogram connection to host
    Initial setup to verify status of external system
    WHILE (true){
        Check status of external system
        Send status to host
        Sleep
    END of WHILE
}
END of sendStatus
END of class

Listing 5.14: Pseudo Code for External System Status Thread

The Flowchart B.6 in Appendix B provides a simplified overview of the external system status thread.

5.3.4 Data Logging and User Interface on Host

The main tasks of the host is to act as the interface between the occupant of the home and the WSN. In addition, the host will log data at regular intervals and keep a log of events encountered by the WSN.

The full version of the host will provide a main menu as shown in Figure 5.1. The buttons along the top show the status of the external systems connected to the WSN.
5.3 Requirements

The tree directory on the left side allowed the user to access each room or type of environmental data from all collector motes in the WSN. The three buttons along the bottom allow the user to either turn off the HAA, unlock the front door or access the system tools.

The host logs the temperature, light level, battery capacity, alarm trigger and carbon monoxide. This provides a history of the collected data.

![Figure 5.1: Host Main Menu](image)

The full version of the host application has the following additional features:

- graphical user interface;
- incoming data is stored; and
- door unlock function / system tools.

For the testbeds a simplified version of the host application was used. The host application for the testbed has three main tasks:
5.3 Requirements

- listen for environmental data;
- listen for external system status updates; and
- broadcast IEEE Address to other members of WSN.

These tasks are started in individual threads by the main class of the application. Once the threads have started the main class waits until the threads are finished. The pseudo code for the main class is shown below:

```java
public class IdentMessByByteOnHost

Instance variables

public static void main(String[] args)

Start new broadcast thread
    new BroadcastHostAddress("BroadcastHostAddress", Port);

Start new thread to listen for environmental data
    new HostIdentifyByByte("EnvironmentalData", Port);

Start new thread to listen for external system status
    new HostIdentifyByByte("ExtSysData", port);

Wait for threads to finish

END of main()

END of class
```

Listing 5.15: Pseudo Code Main Class Host Testbed Version

The requirements for the two listening tasks are very similar and can be implemented in the same class. However each task is to be started in a different thread. The pseudo code is shown below:

```java
public class HostIdentifyByByte implements Runnable

Instance variables

public void run()

Start a new listener on host
    call new CommClassHost();

Initial set up for communication
```

```java
```
5.3 Requirements

```java
    call initBroadcastCom();
    call setPortNumber(portNumber);

    Open up a unicast (broadcast) connection
    call tryToOpenBroadcastConnection(false);

    WHILE (true)
        listen for new data packet
        call listenFor();

        Get data id from byte data in packet
        call getReadDataByte();

        IF (portNumber == 67) // environmental data
            Push time, delay and location to terminal
            call pushTimeAndLocationToTerminal(true);

            Identify message based on message identifier, data
            call identifyMessageAndPushToTerminal(messageId);

        ELSE IF (portNumber == 68) // ext. sys. status data
            Push time and location to terminal
            call pushTimeAndLocationToTerminal(false);

            Identify message based on message identifier
            call identifyMessageAndPushToTerminal(messageId);

        END of ELSE IF
    END WHILE
    END of run()
    END of class
```

Listing 5.16: Pseudo Code Listen for Environmental Data

The third task repeatedly sends the IEEE address of the host. This thread does not listen for any incoming data. The pseudo code is shown below:

```java
Pseudo code to broadcast host address
class BroadcastHostAddress implements Runnable
    Instance variables
    public void run()
```
Get host address
    call RadioFactory.getRadioPolicyManager().getIEEEAddress();

Initial setup for communication
    new CommClassHost();
    call initBroadcastCom();
    call setPortNumber(portNumber);
    call setLocation("Host");
    call setWriteDataLong(hostAddress);
    call setWriteDataByte(hostId); // id of host

Open up a broadcast connection
    call tryToOpenBroadcastConnection(true);

WHILE (true){
    Send IEEE Address
    call packageAndSendRadiogram();

    Wait for 30 seconds
    Sleep(30000);

END of WHILE
END of run()
END of class

Listing 5.17: Pseudo Code Broadcast Host Address

5.4 Test Scripts

Implementing test scripts for Sun™ SPOTs created some difficulties, because of the
limited user interfaces on the physical motes. To help with trouble shooting, the code
included statements to print information to a terminal. However, these statements
were only printed on the screen if virtual motes were used or if physical motes were
connected to the host via the USB connection. These statements remained in the final
code. An example is shown below:
Where possible, the computer code for the motes included statements to activate / deactivate the LEDs located on the eDemoBoard. Except for the events listed in 5.3, all other LED indications were removed after testing and trouble shooting was completed.

As mentioned previously any new code was tested using Solarium before deploying the code to a real Sun™ SPOT. If there was a problem with the new application it was easier to terminate the faulty application by deleting the virtual Sun™ SPOT. In addition, each mote was associated with an output window to see the print statements included throughout the code.

5.5 Implementation

This section of the chapter will focus on the implementation of the HAA based on the requirements outlined in 5.3. The data collector motes, external systems interface motes and the host performed different functions in the WSN. To ensure that these three main areas functioned properly a clear understanding on what type of data was to be passed from one area to the other was needed.

To implement the required code Application Programming Interfaces (API) were utilised. An API is a guide to find the appropriate class in a package. The APIs are available online and can be searched via the package name or alphabetical by class. Three APIs were used during the design and development of the HAA.
5.5 Implementation

- Java™ SE API (Systems 2009a);
- Sun™ SPOT Host API (Systems 2009b); and
- Sun™ SPOT API (Systems 2009c).

The APIs were visited on a regular basis between March and October 2009. No actual code was copied from these websites. All code was typed in NetBeans. The auto complete feature in NetBeans was enabled to reduce typing and syntax errors.

In addition two books were used as reference

- Java™ Head First (Sierra & Bates 2005); and
- Java™: The Complete Reference (Schildt 2007).

5.5.1 General HAA Configurations

Configuring Mesh Routing and OTA Commands

The requirements in 5.3 state that mesh routing is to be enabled. The easiest way to enable this function was to connect each physical Sun™ SPOT via USB to the host computer and type the following command in a command line prompt (Sun 2009a, p.26).
5.5 Implementation

The applications on the host and motes needed to include statements for the maximum number of hops when broadcasting. The default value for the maximum number of hops is two. This value was increased to four due to the fact that communication between the host and the base station is one hop (Sun 2009a, p.19).

An additional requirement defined in 5.3 was that OTA commands were to be enabled so commands from the host could be executed. To enable this feature the following command needed to be entered at the command line prompt (Sun 2009a, p.21).

ant enableota

Listing 5.20: Enable Over-the-Air (OTA) commands on Sun™ SPOT

5.5.2 Implementation of Data Collection

This section will outline the implementation of the collector motes. First the different classes will be shown. These classes acted as building blocks for the different functions assigned to the motes. Once these classes have been shown the implementation for the different collector functions will be shown. Each of these functions will run in a separate thread. A thread will only run if required by the configuration data in the MANIFEST.MF file.

Once the threads have been shown the main class of the collector mote will be outlined. This class essentially extracts the data from the MANIFEST.MF file and configures the mote. When all the required threads have been started the main class waits for the threads to finish.

Configuring Collector Mote Settings

The MANIFEST.MF was utilised to instruct the collector SPOTs which environmental data was to be collected. The properties in the MANIFEST.MF file were edited with

ant set-system-property Dkey=spot.mesh.enable Dvalue=true

Listing 5.19: Enable Mesh Routing on Sun™ SPOT
a text editor before an application was deployed to a mote.

As an example the MANIFEST.MF file for a collector SPOTs is shown below:

<table>
<thead>
<tr>
<th>Line</th>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MIDlet−Name: CollectorSPOT</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>MIDlet−Version: 1.0.0</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>MIDlet−Vendor: Sun Microsystems Inc</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>MIDlet−1: CollectMainSPOT, org.sunspotworld.CollectMainSPOT</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>MicroEdition−Profile: IMP−1.0</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>MicroEdition−Configuration: CLDC−1.1</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>Location: Kitchen</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>IdentTemp: 1</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>Temperature: 1</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>IdentLight: 2</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>LightLevel: 1</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>IdentBat: 3</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>Battery: 0</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>IdentAlarm: 4</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>Alarm: 0</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>IdentCarbonMono: 5</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>CarbonMono: 0</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>SampleTimeTemp: 10</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td>SampleTimeLight: 10</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>SampleTimeBattery: 10</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>SampleTimeAlarm: 1</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>SampleTimeCarbonMono: 1</td>
<td>22</td>
</tr>
</tbody>
</table>

Listing 5.21: MANIFEST.MF file for Collector Mote

A brief explanation of the MANIFEST.MF file follows. The entries in lines 1 to 6 are included for any application to be run on a Sun™ SPOT. The entry in line four is the name of the initial class for the mote. It is the most important entry in any MANIFEST file.

The entries in lines 7 to 22 are specific properties to configure a Sun™ SPOT for the HAA.

The location states in which room the Sun™ SPOT is located. The HAA locations are defined in Table 5.3. The properties Temperature, LightLevel, Battery, Alarm and CarbonMono hold either a '1' or a '0'. 
5.5 Implementation

- '1' indicates that this environmental information is to be collected and sent.
- '0' indicates that this particular mote does not sample this information.

The entries in lines 18 to 22 set the time between samples depending on the type of sample to be taken.

The data in the MANIFEST.MF file can be accessed at run time (Sun 2009a, p.17). The data is read by the application as a String and then converted to the appropriate data type for further use.

**Implementation Collector Mote Class**

Based on the requirements in Section 5.3 and the fact that sections of the pseudo codes were similar, the decision was made to combine the generic functions of a collector mote into one class. An overview of the collector.java class is given in Table 5.7.

To implement the various functions, the collector.java class was extended to include the special functions outlined with an overview for each given in Tables 5.8 to 5.12.

**Temperature Collection Class**

The main task of the temperature collection class was to initialise the temperature sensor on the eDemoBoard. Once the temperature sensor was initialised a temperature sample was taken based on the sample time defined in the MANIFEST.MF file. In addition this class verified that the temperature changed. The new sample was only sent if the temperature had changed. Table 5.8 gives an overview of this class.

**Light Collection Class**

The main task of the light collection class was to initialise the light sensor on the eDemoBoard. Once the light sensor was initialised the average of 17 light sample was taken based on the sample time defined in the MANIFEST.MF file. As with the temperature collection class a new sampled value was only sent via radio communication if the sample had changed compared to the previous sample. Table 5.9 gives an overview of this class.

**Battery Capacity Collection Class**

The main task of the battery capacity collection class was to initialise the battery
### Class: CollectorMote.java

<table>
<thead>
<tr>
<th>Instance Variables</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>private RadiogramConnection rCon</td>
<td>Setter methods for Instance Variables</td>
</tr>
<tr>
<td>private Datagram dg</td>
<td>Getter methods for Instance Variables</td>
</tr>
<tr>
<td>private String location</td>
<td>void initBroadcastCom ()</td>
</tr>
<tr>
<td>private int portNumber</td>
<td>void getPushAddress()</td>
</tr>
<tr>
<td>private long writeDataLong</td>
<td>void tryToOpenBroadcastConnection()</td>
</tr>
<tr>
<td>private byte writeDataByte</td>
<td>void packageAndSendRadiogram()</td>
</tr>
<tr>
<td>private int writeDataInt</td>
<td>void sleepForSomeTime(long samplePeriod,</td>
</tr>
<tr>
<td>private double writeDataDouble</td>
<td>long lastTimeToBeSent)</td>
</tr>
<tr>
<td></td>
<td>long getCurrentTime()</td>
</tr>
<tr>
<td></td>
<td>double reformatDataToBeSent(double valueIn)</td>
</tr>
</tbody>
</table>

Table 5.7: CollectorMote.java class
5.5 Implementation

<table>
<thead>
<tr>
<th>Class: CollectorTempMote.java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends: CollectorMote.java</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instance Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>private double currentTemp</td>
</tr>
<tr>
<td>private double previousTemp</td>
</tr>
<tr>
<td>private ITemperatureInput</td>
</tr>
<tr>
<td>temperatureSensor</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setter methods for Instance Variables</td>
</tr>
<tr>
<td>Getter methods for Instance Variables</td>
</tr>
<tr>
<td>void initTemperatureMeasurement()</td>
</tr>
<tr>
<td>double takeTemperatureSample()</td>
</tr>
<tr>
<td>boolean hasTemperatureChanged()</td>
</tr>
<tr>
<td>void moveCurrentTempToPreviousTemp()</td>
</tr>
</tbody>
</table>

Table 5.8: CollectorTempMote.java class

<table>
<thead>
<tr>
<th>Class: CollectorLightMote.java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends: CollectorMote.java</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instance Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>private int currentLightLevelAverage</td>
</tr>
<tr>
<td>private int previousLightLevelAverage</td>
</tr>
<tr>
<td>private ILightSensor lightSensor</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setter methods for Instance Variables</td>
</tr>
<tr>
<td>Getter methods for Instance Variables</td>
</tr>
<tr>
<td>void initLightMeasurement()</td>
</tr>
<tr>
<td>double takeLightSample()</td>
</tr>
<tr>
<td>boolean hasLightChanged()</td>
</tr>
<tr>
<td>void moveCurrentLightToPreviousLight()</td>
</tr>
</tbody>
</table>

Table 5.9: CollectorLightMote.java class
### 5.5 Implementation

<table>
<thead>
<tr>
<th>Class: CollectorBattMote.java</th>
<th>Extends: CollectorMote.java</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instance Variables</strong></td>
<td>private int currentBattLevel</td>
</tr>
<tr>
<td></td>
<td>private int previousBattLevel</td>
</tr>
<tr>
<td></td>
<td>private IBattery battStatus</td>
</tr>
<tr>
<td><strong>Methods</strong></td>
<td>Setter methods for Instance Variables</td>
</tr>
<tr>
<td></td>
<td>Getter methods for Instance Variables</td>
</tr>
<tr>
<td></td>
<td>void initBattMeasurement()</td>
</tr>
<tr>
<td></td>
<td>double takeBattSample()</td>
</tr>
<tr>
<td></td>
<td>boolean hasBattChanged()</td>
</tr>
<tr>
<td></td>
<td>void moveCurrentBattToPreviousBatt()</td>
</tr>
</tbody>
</table>

Table 5.10: CollectorBattMote.java class

measurement. When initialised, a sample was taken based on the sample time defined in the MANIFEST.MF file. As with the temperature collection class, a new sample is only sent via radio communication if the sample had changed compared to the previous sample. Table 5.10 gives an overview of this class.

**Alarm Trigger Class**

The main task for this class was to initialise one of the digital I/O to be used as the alarm trigger. First the input to be monitored had to be defined. After selection of the input the pin direction had to be set to “input”. To avoid repeated broadcasting of an alarm, the pin interrupt was disabled until the input was returned to normal. Table 5.11 gives an overview of this class.

**Carbon Monoxide Level Class**

The main task of this class is to initialise one of the analog I/O to be used to measure the input signal. Table 5.12 gives an overview of this class. First the input to be monitored had to be defined. The voltage levels were read by using the ADT7411 ADC converter (Sun 2009b, p.22). Characteristics of the Analog inputs and the Analog to Digital Converter are:
5.5 Implementation

**Class:** CollectorAlarmMote.java  
**Extends:** CollectorMote.java

| Instance Variables | Setter methods for Instance Variables  
|---------------------|---------------------------------------------------------------------------------|  
| private PinDescriptor AlarmTriggerInput  
| private EDemoBoard test  
| private EDemoController testController  
| private InputPin pin | Setter methods for Instance Variables  
| Getter methods for Instance Variables  
| void initAlarm()  
| boolean pinStatus()  
| void setEnablePinChangeInterruptsToTrue() |

Table 5.11: CollectorAlarmMote.java class
5.5 Implementation

<table>
<thead>
<tr>
<th>Class: CollectorCarbonMonoMote.java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends: CollectorMote.java</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Instance Variables</strong></td>
</tr>
<tr>
<td>private double currentCarbonMonoLevel</td>
</tr>
<tr>
<td>private double previousCarbonMonoLevel</td>
</tr>
<tr>
<td>private int currentSample</td>
</tr>
<tr>
<td>private EDemoBoard carbonMonoSensor</td>
</tr>
<tr>
<td>private IScalarInput analogInputs[]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Methods</strong></td>
</tr>
<tr>
<td>Setter methods for Instance Variables</td>
</tr>
<tr>
<td>Getter methods for Instance Variables</td>
</tr>
<tr>
<td>void initCarbonMonoMeasurement()</td>
</tr>
<tr>
<td>int takeCarbonMonoSample()</td>
</tr>
<tr>
<td>double convertCarbonMonoSampleToVolt(int currentSample)</td>
</tr>
<tr>
<td>boolean hasCarbonMonoLevelChanged()</td>
</tr>
<tr>
<td>void moveCurrentCarbonMonoToPreviousCarbonMono()</td>
</tr>
</tbody>
</table>

Table 5.12: CollectorCarbonMonoMote.java class

- voltage range at input is 0 to 3 Volt Direct Current (DC)
- The resolution is:

\[
ADC = \frac{V_{in} \times 1024}{V_{ref}}
\]  

(5.1)

To forward the actual voltage level to the external system the actual voltage was calculated as shown

\[
V_{in} = \frac{ADC \times V_{ref}}{1024}
\]  

(5.2)

The calculation was implemented in the method

\[
\text{double convertCarbonMonoSampleToVolt(int currentSample)}
\]

Having outlined the classes available for a collector mote the associated threads will be listed now. As mentioned above the threads will start based on the properties set in the MANIFEST.MF file (see Listing 5.21).
Temperature Collection Thread
The detailed pseudo code for this thread is shown in Appendix C.3. It was expanded from the pseudo code in Listing 5.3

Light Collection Thread
The detailed pseudo code for this thread is shown in Appendix C.4. It was expanded from the pseudo code in Listing 5.4

Battery Capacity Thread
The detailed pseudo code for this thread is shown in Appendix C.5. It was expanded from the pseudo code in Listing 5.5

Alarm Trigger Thread
The detailed pseudo code for this thread is shown in Appendix C.6. It was expanded from the pseudo code in Listing 5.6

Carbon Monoxide Collection Thread
The detailed pseudo code for this thread is shown in Appendix C.7. It was expanded from the pseudo code in Listing 5.7.

Collector Mote Main Thread
The detailed pseudo code for this thread is shown in Appendix C.8. It was expanded from the pseudo code in Listing 5.2

5.5.3 Implementation of Interface to External Systems

Configuring Interface Mote Settings
As with the collector motes the MANIFEST.MF file will be used to configure the interface motes. The additional entries were required to identify the external system. Table 5.6 shows all possible identifiers for interface motes.

The entries in line 6 and 7 were the properties required for the configuration of an interface mote. As an example the configuration for a mote used as a door interface is shown.
The entry in line 7 is used at runtime to verify that the data in the packet is appropriate for this interface mote. The entry in line 8 is only required for a door interface mote. This property indicates how long the door is to remain unlocked. The value is given in milliseconds.

**Implementation Interface Mote Class**

Based on the requirements in 5.3 and the fact that sections of the pseudo codes were similar the decision was made to combine the generic functions of an interface mote into one class. An overview of the collector.java class is given in Table 5.13. As with the collector mote the various functions of the interface motes were implemented by extending the generic functions class InterfaceMote.java. These are outlined in Tables 5.14 to 5.18.

**Climate Interface Class**

The main task of this class was to prepare the data intended for the climate control system. The input data which included location and value was packaged into a single string with the two entries separated by a comma. Table 5.14 gives an overview of this class.

**Light Interface Class**

The main task of this class was to prepare the data intended for the lighting control system. The input data which included location and value was packaged into a single string with the two entries separated by a comma. Table 5.15 gives an overview of this class.
### 5.5 Implementation

#### Class: InterfaceMote.java

<table>
<thead>
<tr>
<th>Instance Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>private String systemInterface</td>
</tr>
<tr>
<td>private int portNumber</td>
</tr>
<tr>
<td>private RadiogramConnection rCon</td>
</tr>
<tr>
<td>private Datagram dg</td>
</tr>
<tr>
<td>private long readDataLong</td>
</tr>
<tr>
<td>private byte readDataByte</td>
</tr>
<tr>
<td>private String readDataUTF</td>
</tr>
<tr>
<td>private int readDataInt</td>
</tr>
<tr>
<td>private double readDataDouble</td>
</tr>
<tr>
<td>EDemoBoard serialComm</td>
</tr>
<tr>
<td>EDemoBoard.getInstance()</td>
</tr>
<tr>
<td>private int baud</td>
</tr>
<tr>
<td>private int databits</td>
</tr>
<tr>
<td>private int parity</td>
</tr>
<tr>
<td>private int stopbits</td>
</tr>
<tr>
<td>private String externalDataString</td>
</tr>
<tr>
<td>private ITriColorLED[] sendCommIndication</td>
</tr>
<tr>
<td>EDemoBoard.getInstance().getLEDs()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setter methods for Instance Variables</td>
</tr>
<tr>
<td>Getter methods for Instance Variables</td>
</tr>
<tr>
<td>void initInterfaceMoteCommunication()</td>
</tr>
<tr>
<td>void getPushAddress()</td>
</tr>
<tr>
<td>void tryToOpenBroadcastConnection()</td>
</tr>
<tr>
<td>void listenFor()</td>
</tr>
<tr>
<td>void sleepForSomeTime(long samplePeriod, long getCurrentTime())</td>
</tr>
<tr>
<td>void initExternalCommunication()</td>
</tr>
<tr>
<td>void sendDataToExternalSystem()</td>
</tr>
</tbody>
</table>

Table 5.13: InterfaceMote.java class
5.5 Implementation

### Climate Interface Mote

<table>
<thead>
<tr>
<th>Class:</th>
<th>ClimateInterfaceMote.java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends:</td>
<td>InterfaceMote.java</td>
</tr>
<tr>
<td>Instance Variables</td>
<td>none</td>
</tr>
<tr>
<td>Methods</td>
<td>packageTempDataToSendToClimateControl(String location, double value)</td>
</tr>
</tbody>
</table>

Table 5.14: ClimateInterfaceMote.java class

### Light Interface Mote

<table>
<thead>
<tr>
<th>Class:</th>
<th>LightInterfaceMote.java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends:</td>
<td>InterfaceMote.java</td>
</tr>
<tr>
<td>Instance Variables</td>
<td>none</td>
</tr>
<tr>
<td>Methods</td>
<td>packageLightDataToSendToLightControl(String location, int value)</td>
</tr>
</tbody>
</table>

Table 5.15: LightInterfaceMote.java class

#### Alarm Interface Class

The main task of this class was to prepare the data intended for the alarm system. The input data which included location and value was packaged into a single string with the two entries separated by a comma. Table 5.16 gives an overview of this class.

#### Carbon Monoxide Interface Class

The main task of this class was to prepare the data intended for the carbon monoxide control system. The input data which included location and value was packaged into a single string with the two entries separated by a comma. Table 5.17 gives an overview of this class.

### Alarm Interface Mote

<table>
<thead>
<tr>
<th>Class:</th>
<th>AlarmInterfaceMote.java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends:</td>
<td>InterfaceMote.java</td>
</tr>
<tr>
<td>Instance Variables</td>
<td>none</td>
</tr>
<tr>
<td>Methods</td>
<td>packageAlarmDataToSendToAlarmSystem(String location, int value)</td>
</tr>
</tbody>
</table>

Table 5.16: AlarmInterfaceMote.java class
5.5 Implementation

<table>
<thead>
<tr>
<th>Class: CarbMonInterfaceMote.java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends: InterfaceMote.java</td>
</tr>
<tr>
<td>Instance Variables: none</td>
</tr>
<tr>
<td>Methods:</td>
</tr>
<tr>
<td>packageCarbonMonoDataToSendToCarbMonSystem</td>
</tr>
<tr>
<td>(String location, double value)</td>
</tr>
</tbody>
</table>

Table 5.17: CarbMonInterfaceMote.java class

<table>
<thead>
<tr>
<th>Class: DoorInterfaceMote.java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends: InterfaceMote.java</td>
</tr>
<tr>
<td>Instance Variables:</td>
</tr>
<tr>
<td>private PinDescriptor unlockDoorOutput</td>
</tr>
<tr>
<td>private EDemoBoard doorCommand</td>
</tr>
<tr>
<td>private EDemoController doorCommandController</td>
</tr>
<tr>
<td>private InputPin pin</td>
</tr>
<tr>
<td>private ITriColorLED[] indicateDoorUnlocked</td>
</tr>
<tr>
<td>EDemoBoard.getInstance().getLEDs()</td>
</tr>
<tr>
<td>Methods:</td>
</tr>
<tr>
<td>Setter methods for Instance Variables</td>
</tr>
<tr>
<td>Getter methods for Instance Variables</td>
</tr>
<tr>
<td>void initDoorInterface()</td>
</tr>
<tr>
<td>void unlockTheDoor(long unlockDoorTime)</td>
</tr>
</tbody>
</table>

Table 5.18: DoorInterfaceMote.java class

Door Interface Class
The main task of this class was to use one of the digital I/Os to lock / unlock a door. One of the pins was configured as an output. The output was set to high for the time set given by the variable unlockDoorTime. After the time had elapsed the output was set to low to lock the door. Table 5.18 gives an overview of this class.

Radiogram Comm Class
The main task of this class was to periodically send the status of an external system. The class also included methods to set up a radiogram unicast connection. Table 5.19 gives an overview of this class.
### Class: RadiogramCommClass.java

<table>
<thead>
<tr>
<th>Instance Variables</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>private RadiogramConnection rCon</td>
<td>Setter methods for Instance Variables</td>
</tr>
<tr>
<td>private Datagram dg</td>
<td>Getter methods for Instance Variables</td>
</tr>
<tr>
<td>private PinDescriptor statusExternalSystem</td>
<td>void initRadiogramCom()</td>
</tr>
<tr>
<td>private EDemoBoard status</td>
<td>void tryToOpenRadiogramConnection()</td>
</tr>
<tr>
<td>private EDemoController statusController</td>
<td>void initExternalSystemStatus()</td>
</tr>
<tr>
<td>private InputPin pin</td>
<td>boolean errorExternalSystem()</td>
</tr>
<tr>
<td>private boolean extSysStatus</td>
<td>void packageAndSendRadiogram()</td>
</tr>
<tr>
<td>private String location</td>
<td></td>
</tr>
<tr>
<td>private String hostAddress</td>
<td></td>
</tr>
<tr>
<td>private int portNumber</td>
<td></td>
</tr>
<tr>
<td>private long writeDataLong</td>
<td></td>
</tr>
<tr>
<td>private byte writeDataByte</td>
<td></td>
</tr>
<tr>
<td>private int writeDataInt</td>
<td></td>
</tr>
<tr>
<td>private double writeDataDouble</td>
<td></td>
</tr>
<tr>
<td>private boolean writeDataBool</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.19: RadiogramCommClass.java class
5.5.4 Implementation of Host Application

Only a brief overview of the implementation of the host application used with the testbeds will be given. The host application for the testbeds only pushed the data to the output window of NetBeans IDE 6.5.1.

All functionalities of the host were implemented in one class. Table 5.20 gives an overview of this class. The code for the host is shown in Listings C.42 to C.45.

5.6 Conclusion Implementation

The chapter outlined the implementation of the design and development of the HAA. The requirements were defined for the collector motes, interface motes and the host applications. Based on these requirements the classes and MANIFEST.MF files were developed. The next chapter will show the results obtained from the different testbeds.
| Class: CommClassHost.java | private RadiogramConnection rCon  
| | private Datagram dg  
| | private String location  
| | private int portNumber  
| | private long writeDataLong  
| | private byte writeDataByte  
| | private int writeDataInt  
| | private double writeDataDouble  
| | private boolean writeDataBool  
| | private long readDataLong  
| | private byte readDataByte  
| | private String readDataUTF  
| | private int readDataInt  
| | private double readDataDouble  
| | private boolean readDataBool  
| | DateFormat fmt = DateFormat.getTimeInstance()  
| | long timeOfReading  
| | long nowOnHost  
| | long delaySampleToTerminal  
| Instance Variables | Setter methods for Instance Variables  
| | Getter methods for Instance Variables  
| | void initBroadcastCom()  
| | void tryToOpenBroadcastConnection()  
| | void packageAndSendRadiogram()  
| | void listenFor()  
| | void pushTimeAndLocationToTerminal  
| | void identifyMessageAndPushToTerminal  
| Methods | Table 5.20: CommClassHost.java class
Chapter 6

Results

This chapter documents the results from the six testbeds. The following calibrated instruments were used to verify the accuracy of the data collected by the WSN testbeds.

- EXTECH 401036 Light Meter;
- FLUKE 115 True RMS Multimeter; and
- ΩEOMEGA RH 101 Temperature Humidity Meter.

6.1 Testbed 1: Accuracy of WSN, Internal Sensor

Table 6.1 shows the accuracy of the light level measurements taken by the WSN in testbed 1 compared to light meter. The light meter and the motes were placed next to each other during the test.

Table 6.2 shows the accuracy of the temperature measurements taken by the WSN in testbed 1 compared to temperature meter. Before taking the measurements both the mote and the instrument were placed in position to adjust to the current ambient temperature.
## 6.2 Testbed 2: Accuracy of WSN, External Sensor

<table>
<thead>
<tr>
<th>Sample No.</th>
<th>WSN (Lux)</th>
<th>Instrument (Lux)</th>
<th>Error (Lux)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>400</td>
<td>417</td>
<td>+17</td>
</tr>
<tr>
<td>2</td>
<td>414</td>
<td>417</td>
<td>+3</td>
</tr>
<tr>
<td>3</td>
<td>396</td>
<td>417</td>
<td>+21</td>
</tr>
<tr>
<td>4</td>
<td>428</td>
<td>417</td>
<td>-11</td>
</tr>
<tr>
<td>5</td>
<td>416</td>
<td>417</td>
<td>+1</td>
</tr>
<tr>
<td>6</td>
<td>412</td>
<td>416</td>
<td>+4</td>
</tr>
<tr>
<td>7</td>
<td>410</td>
<td>417</td>
<td>+7</td>
</tr>
<tr>
<td>8</td>
<td>404</td>
<td>415</td>
<td>+11</td>
</tr>
<tr>
<td>9</td>
<td>402</td>
<td>415</td>
<td>+13</td>
</tr>
<tr>
<td>10</td>
<td>414</td>
<td>414</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.1: Results Testbed 1, Light Level Accuracy

<table>
<thead>
<tr>
<th>Sample No.</th>
<th>WSN (°C)</th>
<th>Instrument (°C)</th>
<th>Error (°C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>27.5</td>
<td>28</td>
<td>-0.5</td>
</tr>
<tr>
<td>2</td>
<td>27.25</td>
<td>28</td>
<td>-0.25</td>
</tr>
<tr>
<td>3</td>
<td>27.5</td>
<td>28</td>
<td>-0.5</td>
</tr>
<tr>
<td>4</td>
<td>27.25</td>
<td>28</td>
<td>-0.25</td>
</tr>
<tr>
<td>5</td>
<td>27.5</td>
<td>28</td>
<td>-0.5</td>
</tr>
<tr>
<td>6</td>
<td>27.25</td>
<td>28</td>
<td>-0.25</td>
</tr>
<tr>
<td>7</td>
<td>27.25</td>
<td>28</td>
<td>-0.25</td>
</tr>
<tr>
<td>8</td>
<td>27.5</td>
<td>28</td>
<td>-0.5</td>
</tr>
<tr>
<td>9</td>
<td>27.25</td>
<td>28</td>
<td>-0.25</td>
</tr>
<tr>
<td>10</td>
<td>27.5</td>
<td>28</td>
<td>-0.5</td>
</tr>
</tbody>
</table>

Table 6.2: Results Testbed 1, Temperature Accuracy

### 6.2 Testbed 2: Accuracy of WSN, External Sensor

This testbed was not completed, since the results for testbed 1 were deemed to be accurate for the HAA.
6.3 Testbed 3: Maximum Distances between WSN Members

The host computer was located in the study (see Figure E.1). Except of the main bedroom, it was possible for a mote to communicate directly with the host. This distance between the main bedroom and the study was approximately 16 meters. Therefore, regardless of location within the apartment no more than two hops were required to reach any other Sun™ SPOTs in the WSN. To reach the host from the main bedroom 3 hops were required.

6.4 Testbed 4: Impact of Noise Sources on WSN

Neither of the identified noise sources had an impact on the Sun™ SPOTs. The motes were placed around all accessible areas of the noise source. The microwave was built-in to the kitchen cabinets. Therefore the motes could only be placed above, on the sides and below the microwave. During this test, the environmental data to be collected was not changed. The data received by the host was always exactly the same.

6.5 Testbed 5: Multifunction Data Collection

The code for the collector motes, interface motes and host were implemented by using identifiers. These identifiers indicate to the receivers what type of data is being received. Each type of data to be collected also ran in a separate thread. At no time was any incorrect data received by the host or interface mote.

6.6 Testbed 6: High Volume Data Collection

To establish how long it took from sampling at the mote to when the data was received a small application was written for the host. This application did not have any user
6.6 Testbed 6: High Volume Data Collection

Table 6.3: Results Testbed 6, Time Delay, 2 seconds between samples

<table>
<thead>
<tr>
<th>Sample No.</th>
<th>Delay (ms)</th>
<th>Sample No.</th>
<th>Delay (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2273</td>
<td>6</td>
<td>2406</td>
</tr>
<tr>
<td>2</td>
<td>2382</td>
<td>7</td>
<td>2419</td>
</tr>
<tr>
<td>3</td>
<td>2286</td>
<td>8</td>
<td>2527</td>
</tr>
<tr>
<td>4</td>
<td>2300</td>
<td>9</td>
<td>2495</td>
</tr>
<tr>
<td>5</td>
<td>2409</td>
<td>10</td>
<td>2537</td>
</tr>
</tbody>
</table>

Table 6.3: Results Testbed 6, Time Delay, 2 seconds between samples

interface. The application was launched from NetBeans IDE 6.5.1. The output window was utilised to show:

- the incoming data;
- time when sample was taken;
- time when sample was received by host; and
- time delay.

The reference measurement showed that the delay was 2288 ms for one sample to be received by the host. The mote was only collecting carbon monoxide data via the analogue input.

Table 6.3 shows the delay for 10 samples. The sample time was set to 2000 ms. The table shows that no significant increase in delay was experienced when the sample time was reduced to 2 seconds.

Table 6.4 shows the delay for 10 samples. The sample time was set to 1000 ms. Again the table shows that no increase in delay was experienced when the sample time was reduced to 1 second.

Table 6.5 shows the delay for five samples each from the two physical motes. The sample time was set to 1000 ms on both motes. The applications on both motes were started at the same time. The table shows that even after only five samples there was a significant delay from when the sample was taken to when it was received by the
### 6.7 Test Result and Conclusion

<table>
<thead>
<tr>
<th>Sample No.</th>
<th>Delay (ms)</th>
<th>Sample No.</th>
<th>Delay (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2073</td>
<td>6</td>
<td>2253</td>
</tr>
<tr>
<td>2</td>
<td>2087</td>
<td>7</td>
<td>2251</td>
</tr>
<tr>
<td>3</td>
<td>2101</td>
<td>8</td>
<td>2265</td>
</tr>
<tr>
<td>4</td>
<td>2225</td>
<td>9</td>
<td>2325</td>
</tr>
<tr>
<td>5</td>
<td>2223</td>
<td>10</td>
<td>2339</td>
</tr>
</tbody>
</table>

Table 6.4: Results Testbed 6, Time Delay, 1 second between Samples

<table>
<thead>
<tr>
<th>Mote 1</th>
<th>Mote 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample No.</td>
<td>Delay (ms)</td>
</tr>
<tr>
<td>1</td>
<td>2115</td>
</tr>
<tr>
<td>2</td>
<td>3145</td>
</tr>
<tr>
<td>3</td>
<td>4158</td>
</tr>
<tr>
<td>4</td>
<td>5173</td>
</tr>
<tr>
<td>5</td>
<td>6172</td>
</tr>
</tbody>
</table>

Table 6.5: Results Testbed 6, Time Delay, 1 second between Samples, two Motes

host. Therefore with an increase in motes in a WSN delay may become significant if the sample time is not carefully selected.

This testbed turned out to be somewhat challenging to execute. As mentioned in Section 3.5, the time needed to be synchronised before each test. This reference value was then used for comparison during the actual tests. It was not actually possible however, to confirm if all the times of the WSN members were exactly the same. During initial testing while setting up this testbed delay times of less than one second were recorded.

### 6.7 Test Result and Conclusion

As specified each testbed provided a result. Testbed 6 provided some minor inconclusive results, since it was not possible to establish if all members of the WSN had exactly
the same time. It was however, possible to show that multiple Sun\textsuperscript{TM} SPOTs sending data in short intervals could cause significant delays. The original sample intervals in Table 5.4 were increased based on the results obtained in Testbed 6.

The fully completed and tested code for the Sun\textsuperscript{TM} SPOTs was very stable. Hence, it was possible to run the applications on the Sun\textsuperscript{TM} SPOTs for extended periods of time.
Chapter 7

Conclusions and Further Work

The aim of the project was to develop a testbed for a simple WSN. The WSN was implemented using a Sun™ SPOT development kit from Sun™ Microsystems.

Even though the development kit only included two physical motes, it was possible to create a WSN with the available Sun™ SPOTs and the use of virtual motes. The limited number of physical motes however, prevented the deployment of a complete WSN for the proposed HAA.

Due to the limited user interfaces on the physical Sun™ SPOTs, challenges were encountered during the implementation and testing of the applications. Generous use of LEDs on the eDemoBoard during testing of the code overcame some of these challenges. On completion of trouble shooting and testing most of the LED indications were removed from the final code to maximise the time between battery charges.

The applications for the motes were written in Squawk Java™ VM. The application for the host was written in SE Java™ VM. The final code for both the motes and the host computer, were very stable.

Six testbeds were selected to simulate realistic events within the WSN. Each testbed aimed to verify a different aspect of a deployed HAA. The selected testbeds provided results in regards to:
• accuracy of the collected data;

• performance; and

• maximum distance between members of the WSN.

Minor performance issues were encountered on the host computer if too many virtual Sun™ SPOTs were deployed. The performance issues were managed by limiting the number of virtual Sun™ SPOTs running on the host to six.

## 7.1 Achievement of Project Objectives

The research aspects of the project specification were addressed in Chapter 2. The design, develop and implementation aspects of the project specification were addressed in Chapters 3, 4 and 5. Two points in regards to the HAA were not fully satisfied.

1. Item 5.2 of the Project Specification was not completed to full satisfaction. It was not possible to establish if the power consumption of the home would be reduced by using the HAA. To investigate this objective a testbed containing approximately twenty physical Sun™ SPOTs would have to be deployed and interfaced to the external systems.

To further increase the potential for energy savings the alarm functionality could have been adapted to show if a person was present in a room. This would have allowed the HAA to adjust lighting or temperature levels.

2. Due to time constraints, item 5.3 of the test specification was implemented without the use of fully developed additional hardware. This meant that the carbon monoxide feature was implemented using an analogue input on the eDemoBoard. The change in carbon monoxide levels was simulated by using a three terminal potentiometer.
7.2 Further Work

Since WSNs rely on the interaction of software, hardware and networking a great number of projects and further tests could be conducted. Additional projects could therefore be defined to focus on any or all of these specific technical aspects.

By using the eDemoBoard it was possible to implement a WSN without the need to develop any additional hardware. As mentioned in Section 3.1 however, a number of scenarios were discarded due to the physical characteristics of the Sun™ SPOTs in the development kit. Developing specific motes to overcome the limitations of the Sun™ SPOTs would allow the implementation of some of these scenarios and therefore further the understanding of WSNs.

Another limiting factor was the small number of physical motes available in the development kit. It was relatively easy to deploy a new application to the two physical motes. This process was completed within a few minutes and was completed by connecting the Sun™ SPOTs via USB connection to the host computer.

If however, a large number of motes (e.g. 10,000) were deployed in a WSN even a simple task like deploying a new application to all motes could become challenging.

In addition, projects could focus on applications which could only be implemented if the motes are fully developed. The small size of the fully developed motes could also be used as a base to investigate ethical implications. Due to the very small size of the motes an individual or group of people may not be aware that they are being observed.

Based on the above consideration a number of projects are proposed below. Some of these projects could be implemented by using the discarded ideas in Section 3.1.

- Performance impact within a WSN, if a large number of physical motes are deployed.
- Maintain a large WSN after initial deployment.
- Develop power supplies for very small motes.
7.2 Further Work

- Determine how to replace a system of wired sensors with a WSN in an existing control system.

- Investigate performance and security issues if multiple WSN are deployed within the same geographical area.

- Develop hardware for single purpose motes.

- Implement a WSN to constantly measure the blood pressure or other vital signs of a patient.

- Investigate the ethical implications if motes are of size $1 \text{ mm}^3$ or less.

Overall, despite the limitations of the development kit, a WSN was deployed and various testbeds were investigated. This project was a first step to gain understanding of WSNs and the positive results obtained by the testbeds could be used as a foundation for future studies.
References


REFERENCES

IEE (2006), *Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)*.


Olariu, S. & Xu, Q. (2005), Information assurance in wireless sensor networks, *in ‘19th IEEE International and Distributed Processing Symposium (IPDPS’05)’*.


REFERENCES


Replace with project specification
Replace with project specification
Appendix B

Flowcharts
B.1 Flowchart Compile and Run Process

Figure B.1: Flowchart Showing Compile and Run Process
B.2 Flowchart Main: Collector Mote

Figure B.2: Flowchart Main Collector Mote
B.3 Flowchart: Temperature Collecting Thread

Figure B.3: Flowchart Temperature Thread
B.4 Flowchart Main: Interface Mote

![Flowchart Main Interface Mote](image)

Figure B.4: Flowchart Main Interface Mote
B.5 Flowchart: Climate Interface Thread

Figure B.5: Flowchart Climate Interface Thread
B.6 Flowchart: Send External System Status Thread

Figure B.6: Flowchart Send External System Status Thread
Appendix C

Code Listings
C.1 Example Implement Code

C.1.1 Example of Test Script

```java
package org.sunspotworld;
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;

/**
   * The startApp method of this class is called by the VM to start the
   * application.
   *
   * The manifest specifies this class as MIDlet-1, which means it will
   * be selected for execution.
   */
public class testChangeDataFormat extends MIDlet {

    protected void startApp() throws MIDletStateChangeException {
        // variables
        double valueIn = 0.0;
        double valueOut = 0.0;

        ChangeDataFormat newTest = new ChangeDataFormat();

        System.out.println("\n");
        valueIn = 125.5678789657456456456456456;
        valueOut = newTest.reformatDataToBeSent(valueIn);
        System.out.println(valueIn + " becomes " + valueOut);

        System.out.println("\n");
        valueIn = -25.5678789657456456456456456;
        valueOut = newTest.reformatDataToBeSent(valueIn);
        System.out.println(valueIn + " becomes " + valueOut);

        System.out.println("\n");
        valueIn = 5.917;
        valueOut = newTest.reformatDataToBeSent(valueIn);
        System.out.println(valueIn + " becomes " + valueOut);

        System.out.println("\n");
        valueIn = -5.997;
```
valueOut = newTest.reformatDataToBeSent(valueIn);
System.out.println(valueIn + " becomes " + valueOut);

Listing C.1: Test Code Example Implementation

C.1.2 Example of the New Method to be Tested

```java
package org.sunspotworld;
import com.sun.squawk.util.MathUtils;
public class ChangeDataFormat{

    double reformatDataToBeSent(double valueIn){

        // local variable
        double valueOut = 0;

        // For trouble shooting only
        System.out.println("Print input value: "+valueIn);
    }
}
```
// Move decimal point 2 digits to the right
valueIn = valueIn * 100;

// For trouble shooting only
System.out.println("After * operator: " + valueIn);

// Round up or down
valueIn = MathUtils.round(valueIn);

// For trouble shooting only
System.out.println("After rounding: " + valueIn);

// Move decimal point 2 digits to the left
valueOut = valueIn / 100;

// For trouble shooting only
System.out.println("After / operator: " + valueOut);

System.out.println("Data to be returned: " + valueOut);
return valueOut;

} // end of reformatDataToBeSent()
} // end of class
C.2 Pseudo Code Collector Motes

C.2.1 Pseudo Code for Temperature Thread

```java
Pseudo Code for Temperature Thread
TempThread implements Runnable

Instance variables
String location;
Thread t;
hostPortNumber;
dataIdentifier;
long sampleTime;
long timeToBeSent;
double tempToBeSent;

Methods to access instance variables, setter methods
Methods to access instance variables, getter methods

public void run()
Start a new temperature collector mote
new CollectorTempMote

Initial setup for measurement of temperature
call initTemperatureMeasurement();

Initial setup for communication
call initBroadcastCom();
call setPortNumber(hostPortNumber);
call setLocation(location);
call setWriteDataByte(dataIdentifier);

Get and push IEEE Address to terminal
call getPushAddress();

Open up a broadcast connection
call tryToOpenBroadcastConnection();

WHILE (true)
Get the sample time
call getCurrentTime();

Indicate sample event (start, i.e. LED on)
call sampleEventIndication();

Get the current temperature reading to send in datagram
call takeTemperatureSample();

Indicate sample event (end, i.e. LED off)
call sampleEventIndication();

Reformat tempToBeSent to ****
call reformatDataToBeSent(tempToBeSent);

IF (temperature has changed)
Update time and temperature value to be sent
call setWriteDataLong(timeToBeSent);
call setWriteDataDouble(tempToBeSent);```
Send current sample
   call packageAndSendRadiogram();
END of IF (temperature has changed)

Move current temp sample to previous sample variable
   call moveCurrentTempToPreviousTemp();

Go to sleep to conserve battery
   call sleepForSomeTime(sampleTime, timeToBeSent);

END of WHILE loop
END of run()
END of class

Listing C.3: Pseudo Code Temperature Collection Thread

C.2.2 Pseudo Code for Light Level Thread

Pseudo code for light thread
LightThread implements Runnable
Instance variables
   private String location;
   private Thread t;
   private int hostPortNumber;
   private byte dataIdentifier;
   private long sampleTime;
   private long timeToBeSent;
   private int lightToBeSent;

Methods to access instance variables, setter methods
Methods to access instance variables, setter methods

public void run () {
   Start a new light level collector mote
      new CollectorLightMote();

   Initial set up for measurement of light level
      call initLightMeasurement();

   Initial set up for communication
      call initBroadcastCom();
      call setPortNumber(hostPortNumber);
      call setLocation(location);
      call setWriteDataByte(dataIdentifier);

   Get and push IEEE Address to terminal
      call getPushAddress();

   Open up a broadcast connection
      call tryToOpenBroadcastConnection();

   WHILE (true)
      Get the sample time
         call getCurrentTime();

      Indicate sample event (start, i.e LED on)
C.2 Pseudo Code Collector Motes

Listing C.4: Pseudo Code Light Collection Thread

call sampleEventIndication();

Get the current light reading to send in datagram
call takeLightSample();

Indicate sample event (end, i.e LED off)
call sampleEventIndication();

IF (light has changed)
Update time and light value to be sent
call setWriteDataLong(timeToBeSent);
call setWriteDataInt(lightToBeSent);

Send current sample
call packageAndSendRadiogram();
END of IF (light has changed)

Move current light sample to previous sample variable
call moveCurrentLightToPreviousLight();

Go to sleep to conserve battery
call sleepForSomeTime(sampleTime, timeToBeSent);

END of WHILE
END of run()
END of class

C.2.3 Pseudo Code for Battery Capacity Thread

Pseudo Code for Battery Capacity
BatLevelThread implements Runnable

Instance Variables
private String location;
private Thread t;
private int hostPortNumber;
private byte dataIdentifier;
private long sampleTime;
private long timeToBeSent;
private int battToBeSent;

Methods to access instance variables, setter methods
Methods to access instance variables, setter methods

public void run() {
    Start a new battery level collector mote
    new CollectorBattMote();

    Initial set up for measurement of battery level
call initBattMeasurement();

    Initial set up for communication
call initBroadcastCom();
call setPortNumber(hostPortNumber);
call setLocation(location);
Listing C.5: Pseudo Code Battery Capacity Thread

C.2.4 Pseudo Code for Alarm Trigger Thread

Pseudo code for alarm trigger thread
AlarmTriggerThread implements Runnable
  Instance variables
    private String location;
    private Thread t;
    private int hostPortNumber;
    private byte dataIdentifier;
    private long sampleTime;
    private long timeToBeSent;

Methods to access instance variables, setter methods
Methods to access instance variables, setter methods
public void run() {
    Start a new alarm trigger mote
    new CollectorAlarmMote();

    Initial set up for measurement for alarm trigger
    call initAlarm();

    Initial set up for communication
    call initBroadcastCom();
    call setPortNumber(hostPortNumber);
    call setLocation(location);
    call setWriteDataByte(dataIdentifier);

    Get and push IEEE Address to terminal
    call getPushAddress();

    Open up a broadcast connection
    tryToOpenBroadcastConnection();

    WHILE (true)
        WHILE (is the input pin low)
            Since alarm is not triggered go to sleep for specified time
            call sleepForSomeTime(sampleTime, timeToBeSent);
            Get the current time. i.e. time when sample is taken
            call getCurrentTime();
            END of WHILE (is the input pin low)

            Update alarm values to be sent
            call setWriteDataLong(timeToBeSent);
            call setWriteDataInt(1); // 1 = alarm triggered

            Package time & alarm trigger into a radio datagram & send
            call packageAndSendRadiogram();

            Stop the thread until the input has been set to low
            call getPinStatus().waitForChange();
            call setEnablePinChangeInterruptsToTrue();

        IF (input pin is not triggered)
            Get the current time
            call getCurrentTime();

            Update alarm values to be sent
            call setWriteDataLong(timeToBeSent);
            call setWriteDataInt(2); // 2 = alarm not triggered

            Package time & alarm trigger into radio datagram & send
            call packageAndSendRadiogram();
            END of IF (input pin is not triggered)

    END of WHILE
    END of run()
    END of class
C.2.5 Pseudo Code for Carbon Monoxide Thread

```java
Pseudo Code for Carbon Mono thread
class CarMonThread implements Runnable {
    Instance variables
    private String location;
    private Thread t;
    private int hostPortNumber;
    private byte dataIdentifier;
    private long sampleTime;
    private long timeToBeSent;
    private double carbonMonoToBeSent;
    private int currentCarbonMonoSample;

    Methods to access instance variables, setter methods
    Methods to access instance variables, getter methods

    public void run() {
        Start a new carbon mono collector mote
        new CollectorCarbonMonoMote();

        Initial set up for measurement of carbon mono
        call initCarbonMonoMeasurement();

        Initial set up for communication
        call initBroadcastCom();
        call setPortNumber(hostPortNumber);
        call setLocation(location);
        call setWriteDataByte(dataIdentifier);

        Get and push IEEE Address to terminal
        call getPushAddress();

        Open up a broadcast connection
        call tryToOpenBroadcastConnection();

        WHILE (true)
        Get the sample time
        call getCurrentTime();

        Indicate sample event (start, i.e LED on)
        call sampleEventIndication();

        Get the current carbon mono reading to send in datagram
        call takeCarbonMonoSample();

        Indicate sample event (end, i.e LED off)
        call sampleEventIndication();

        Convert Sample to volt
        call moteCarMon.convertCarbonMonoSampleToVolt(
            currentCarbonMonoSample);

        Reformat sample to be sent to ****.** format
        call reformatDataToBeSent(carbonMonoToBeSent);

        IF (Carbon Mono Level has Changed)
        Update time and light value to be sent
        call setWriteDataLong(timeToBeSent);
        call setWriteDataDouble(carbonMonoToBeSent);
    }
}
```
C.2 Pseudo Code Collector Motes

C.2.6 Pseudo Code for Collector Mote Main Thread

Listing C.7: Pseudo Code Carbon Monoxide Thread

Pseudo Code for Main Class of a Collector Mote

Instance Variables

Define constants
int HostPort
long DefaultSampleTime

Define variables for MANIFEST.MF properties (Strings)
MANI_LOCATION, MANI_IDENT_TEMP, MANI_TEMPERATURE, MANI_IDENT_LIGHT,
MANI_LIGHT_LEVEL, MANI_IDENT_BAT, MANI_BATTERY, MANI_IDENT_ALARM,
MANI_ALARM, MANI_IDENT_CARBON_MONO, MANI_CARBON_MONO

Define variables for MANIFEST.MF properties
byte
identByteTemp, identByteLight, identByteBat, identByteAlarm,
identByteCarbonMono
int
tempInt, lightInt, batInt, alarmInt, carbonInt

Define boolean variables to set functions of mote, set to FALSE
tempFlag, lightFlag, batFlag, alarmFlag, carbonMonoFlag

protected void startApp() {
Monitor the USB Port (if connected) & recognize commands from host
Make a new instance of possible functions of mote and set to NULL
BatLevelThread
TempThread
LightThread
AlarmTriggerThread
CarMonThread

Get configuration data from Manifest file

Convert properties of MANIFEST file to appropriate data type

Set required functions of mote based on properties of MANIFEST.MF
Set flags for functions, if 1 —> true —> function required
Start new threads on a mote depending Manifest file
IF (tempFlag)
   new TempThread
IF (lightFlag)
   new LightThread
IF (batFlag)
   new BatLevelThread
IF (alarmFlag)
   new AlarmTriggerThread
IF (carbonMonoFlag)
   new CarMonThread

Wait for functions (threads) to end

END of startApp()
END of class

Listing C.8: Pseudo Code Collector Main Thread
C.3 Code Listings for Collector Mote

C.3.1 CollectorMote.java

```java
package org.sunspotworld;

import com.sun.spot.io.j2me.radiogram.RadiogramConnection;
import com.sun.spot.sensorboard.EDemoBoard;
import com.sun.spot.sensorboard.peripheral.ITriColorLED;
import com.sun.spot.util.Utils;
import com.sun.squawk.util.MathUtils;
import java.io.IOException;
import javax.microedition.io.Connector;
import javax.microedition.io.DataGram;

class CollectorMote {

    // Instance variables
    
    private RadiogramConnection rCon;
    private Datagram dg;
    private String location;
    private int portNumber;
    
    private long writeDataLong = 0;
    private byte writeDataByte = 0;
    private int writeDataInt = 0;
    private double writeDataDouble = 0;
    private boolean writeDataBool = false;
    
    private ITriColorLED[] sampleEvent = EDemoBoard.getInstance().getLEDs();

    // methods to access instance variables, setter methods
    
    void setLocation(String setLoc){
        location = setLoc;
    }
}
```
```c
void setPortNumber(int setPort) {
    portNumber = setPort;
}

void setWriteDataLong(long setDataLong) {
    writeDataLong = setDataLong;
}

void setWriteDataByte(byte setDataByte) {
    writeDataByte = setDataByte;
}

void setWriteDataInt(int setDataInt) {
    writeDataInt = setDataInt;
}

void setWriteDataDouble(double setDataDouble) {
    writeDataDouble = setDataDouble;
}

void setWriteDataBoolean(boolean setDataBoolean) {
    writeDataBool = setDataBoolean;
}

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// methods to access instance variables, getter methods
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
String getLocation() {
    return location;
}

int getPortNumber() {
    return portNumber;
}

long getWriteDataLong() {
    return writeDataLong;
}

byte getWriteDataByte() {
    return writeDataByte;
}

int getWriteDataInt() {
    return writeDataInt;
}

double getWriteDataDouble() {
    return writeDataDouble;
}

boolean getWriteDataBoolean() {
    return writeDataBool;
}
```

C.3 Code Listings for Collector Mote
C.3 Code Listings for Collector Mote

void initBroadcastCom (){
    // Initial setup for communication
    rCon = null;
    dg = null;
}

void getPushAddress (){
    // get and push IEEE Address to terminal. Troubleshoot only
    String ourAddress = System.getProperty("IEEE_ADDRESS");
    System.out.println("Starting app. on " + ourAddress + ", ourAddress = " + ourAddress + "};
}

void tryToOpenBroadcastConnection (){ try {
    // Open up a broadcast connection to the host port where the ...
    // ... the host will be listening.
    rCon = (RadiogramConnection) Connector.open("radiogram://broadcast:" + portNumber);
    dg = rCon.newDatagram(rCon.getMaximumLength());
    // For troubleshooting only
    System.out.println("Broadcast is open.");
} catch (IOException ex) { ex.printStackTrace(); }
}

void packageAndSendRadiogram (){
    // package data into datagram and send
    dg.reset();
    try {
        dg.writeLong(writeDataLong);
        dg.writeByte(writeDataByte);
        dg.writeUTF(location);
        dg.writeInt(writeDataInt);
        dg.writeDouble(writeDataDouble);
        dg.writeBoolean(writeDataBool);
    }
```java
rCon.send(dg);
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// For troubleshooting only
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
System.out.println("Data packet sent.");
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
}
} catch (IOException ex) {
    ex.printStackTrace();
} // end try / catch
} // end packageAndSendRadiogram()

void sleepForSomeTime(long samplePeriod, long lastTimeToBeSent) {
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // let the mote rest to conserve battery
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    Utils.sleep(samplePeriod - (System.currentTimeMillis() - lastTimeToBeSent));
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
} // end of sleepForSomeTime()

long getCurrentTime() {
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // take time of sample
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    return System.currentTimeMillis();
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
} // end getCurrentTime()

double reformatDataToBeSent(double valueIn) {
    /* ***********************************************************************
    * Purpose : Method changes format of data of type long.
    * Purpose : Limit data to 2 digits after decimal point
    * Input   : Data to be changed
    * Outputs : Data limited to two decimals
    * ***********************************************************************
    */
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // local variable
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    double valueOut = 0;
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // For troubleshooting only
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    System.out.println("Print input value: "+valueIn);
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // Move decimal point 2 digits to the right
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    valueIn = valueIn * 100;
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
```
// For trouble shooting only
System.out.println("After * operator: "+valueIn);

// Round up or down
valueIn = MathUtils.round(valueIn);

// For trouble shooting only
System.out.println("After rounding: "+valueIn);

// Move decimal point 2 digits to the left
valueOut = valueIn / 100;

// For trouble shooting only
System.out.println("After / operator: "+valueOut);
System.out.println("Data to be returned: "+valueOut);
return valueOut;

} // end of reformatTempToBeSent()

void sampleEventIndication (boolean turnIndicationOn, int i) {
  if (turnIndicationOn){
    sampleEvent[i].setRGB(0, 0, 255);
    sampleEvent[i].setOn(); // LED on
  } else {
    sampleEvent[i].setOff(); // LED off
  } // end of IF/ ELSE

} // end of sampleEventIndication ()

} // end CollectorMote class

Listing C.9: Generic Functions of a Collector Mote

C.3.2 CollectorTempMote.java

// *******************************************************************************
* Author: Severin Willisch
* email: d9840087@mail.connect.usq.edu.au
package org.sunspotworld;

import com.sun.spot.sensorboard.EDemoBoard;
import com.sun.spot.sensorboard.peripheral.ITemperatureInput;
import java.io.IOException;

class CollectorTempMote extends CollectorMote{
    // Instance variables

    private double currentTemp = 0.0; // current temperature sample
    private double previousTemp = 0.0; // previous temperature sample
    private ITemperatureInput temperatureSensor;

    // methods to access instance variables, setter methods
    void setCurrentTemp(double setCurTemp){
        currentTemp = setCurTemp;
    }

    void setPreviousTemp(double setPrevTemp){
        previousTemp = setPrevTemp;
    }

    // methods to access instance variables, getter methods
    double getCurrentTemp(){
        return currentTemp;
    }

    double getPreviousTemp(){
        return previousTemp;
    }

    // All other methods
    void initTemperatureMeasurement(){
        // initialise temperature measurement capability of mote
    }
}
C.3 Code Listings for Collector Mote

```java
double takeTemperatureSample() {
    try {
        // take sample of current temperature
        currentTemp = temperatureSensor.getCelsius();
        // For trouble shooting only
        System.out.println("Temperature sample taken: " + currentTemp);
        System.out.println(" Previous temp: " + previousTemp);
    } catch (IOException ex) {
        ex.printStackTrace();
    }
    return currentTemp;
} // end takeTemperatureSample()

boolean hasTemperatureChanged() {
    // For trouble shooting only
    System.out.println("Check if temp has changed.");
    // return FALSE if previous temp equals current temp
    return currentTemp != previousTemp;
} // end hasTemperatureChanged()

void moveCurrentTempToPreviousTemp() {
    // For trouble shooting only
    System.out.println("Move current temp to previous temp.");
    // move current temp sample to previous sample
    previousTemp = currentTemp;
} // end moveCurrentTempToPreviousTemp()
```
C.3 Code Listings for Collector Mote

C.3.3 CollectorLightMote.java

```java
package org.sunspotworld;

import com.sun.spot.sensorboard.EDemoBoard;
import com.sun.spot.sensorboard.peripheral.ILightSensor;
import java.io.IOException;

class CollectorLightMote extends CollectorMote {

    // Instance variables

    private int currentLightLevelAverage = 0; // current light level
    private int previousLightLevelAverage = 0; // previous light level
    private ILightSensor lightSensor;

    // methods to access instance variables, setter methods

    void setCurrentLight(int setCurLight) {
        currentLightLevelAverage = setCurLight;
    }

    void setPreviousLight(int setPrevLight) {
        previousLightLevelAverage = setPrevLight;
    }

    // methods to access instance variables, getter methods

    int getCurrentLight() {
        return currentLightLevelAverage;
    }

    int getPreviousLight() {
        return previousLightLevelAverage;
    }

    // All other methods
```
void initLightMeasurement()
{
    // initialise light measurement capability of mote
    lightSensor = EDemoBoard.getInstance().getLightSensor();
}

int takeLightSample()
{
    try {
        // take sample of current light level
        currentLightLevelAverage = lightSensor.getAverageValue();
        //
        // For troubleshooting only
        //
        System.out.println("Light sample taken: "+
                          currentLightLevelAverage);
        System.out.println("Previous light: "+
                          previousLightLevelAverage);
        //
    } catch (IOException ex) {
        ex.printStackTrace();
    }
    return currentLightLevelAverage;
}

int convertRawDataToLux()
{
    return 2 * currentLightLevelAverage; // Lux = 2 times raw data
}

boolean hasLightChanged()
{
    // For troubleshooting only
    //
    System.out.println("Check if light level has changed.");
    //
    // return FALSE if previous light level equals current light level
    return currentLightLevelAverage != previousLightLevelAverage;
}

void moveCurrentLightToPreviousLight()
{ // For troubleshooting only
}
System.out.println("Move current light level to " +
"previous light level.");

// move current light sample to previous sample
previousLightLevelAverage = currentLightLevelAverage;

} // end moveCurrentTempToPreviousTemp()

} // end of CollectorLightMote class

Listing C.11: Special Functions of a Light Collector Mote
C.3 Code Listings for Collector Mote

C.3.4 CollectorBattMote.java

```java
package org.sunspotworld;

import com.sun.spot.peripheral.IBattery;
import com.sun.spot.peripheral.Spot;

class CollectorBattMote extends CollectorMote {
    // Instance variables
    private int currentBattLevel = 0;  // current battery level
    private int previousBattLevel = 0;  // previous battery level
    private IBattery battStatus;  // init battery capacity;

    // methods to access instance variables, setter methods
    void setCurrentBatt(int setCurBatt){
        currentBattLevel = setCurBatt;
    }

    void setPreviousBatt(int setPrevBatt){
        previousBattLevel = setPrevBatt;
    }

    // methods to access instance variables, getter methods
    int getCurrentBatt(){
        return currentBattLevel;
    }

    int setPreviousBatt(){
        return previousBattLevel;
    }

    // All other methods
    void initBattMeasurement(){
```

// initialise battery measurement capability of mote
//
battStatus = Spot.getInstance().getPowerController().getBattery();

} // end initLightMeasurement()

int takeBattSample() {
    //
    // take sample of current battery capacity
    //
currentBattLevel = battStatus.getBatteryLevel();
    //
    // For trouble shooting only
    //
    System.out.println("Battery sample taken: "+
currentBattLevel);
    System.out.println("Previous temp: "+
previousBattLevel);
    //

    return currentBattLevel;

} // end takeBattSample()

boolean hasBattChanged() {
    //
    // For trouble shooting only
    //
    System.out.println("Check if battery capacity has changed.");
    //
    // return FALSE if previous batt level equals current batt level
    //
    return currentBattLevel != previousBattLevel;
    //

} // end hasBattChanged()

void moveCurrentBattToPreviousBatt() {
    //
    // For trouble shooting only
    //
    System.out.println("Move current battery level to previous batt.");
    //
    // move current temp sample to previous sample
    //
    previousBattLevel = currentBattLevel;

} // end moveCurrentBattToPreviousBatt()

} // end of CollectorBattMote
C.3 Code Listings for Collector Mote

Listing C.12: Special Functions of a Battery Collector Mote

C.3.5 CollectorAlarmMote.java

```java
package org.sunspotworld;

import com.sun.spot.sensorboard.EDemoBoard;
import com.sun.spot.sensorboard.EDemoController;
import com.sun.spot.sensorboard.io.InputPin;
import com.sun.spot.sensorboard.io.PinDescriptor;

class CollectorAlarmMote extends CollectorMote {
    // Instance variables
    //
    private PinDescriptor AlarmTriggerInput;
    private EDemoBoard test;
    private EDemoController testController;
    private InputPin pin;
    //
    // methods to access instance variables, getter methods
    //
    InputPin getPinStatus(){
        return pin;
    }
    //
    // methods to access instance variables, setter methods
    //
    void setPinStatus(InputPin setPin){
        pin = setPin;
    }
    //
}
```
C.3 Code Listings for Collector Mote

```java
void initAlarm()
{
    // initialise alarm capability of mote
    AlarmTriggerInput = EDemoController.D2;
    test = EDemoBoard.getInstance();
    testController = test.getEDemoController();
    pin = new InputPin(AlarmTriggerInput, testController);
    testController.setPinDirection(AlarmTriggerInput, false);
    testController.enablePinChangeInterrupts(pin);
}

boolean pinStatus()
{
    // Check if input is low on D2. Low means no alarm triggered
    return (pin.isLow());
}

void setEnablePinChangeInterruptsToTrue()
{
    testController.enablePinChangeInterrupts(pin);
}

Listing C.13: Special Functions of a Alarm Trigger Mote

C.3.6 CollectorCarbonMonoMote.java

```
import com.sun.spot.sensorboard.io.IScalarInput;
import java.io.IOException;

class CollectorCarbonMonoMote extends CollectorMote {
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // Instance variables
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    private double currentCarbonMonoLevel = 0; // in volt
    private double previousCarbonMonoLevel = 0; // in volt
    private int currentSample = 0; // converted to volt
    private EDemoBoard carbonMonoSensor;
    private IScalarInput analogInputs[];
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // methods to access instance variables, setter methods
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    void setCurrentCarbonMono(double setCurCarbonMono) {
        currentCarbonMonoLevel = setCurCarbonMono;
    }
    void setPreviousCarbonMono(double setPrevCarbonMono) {
        previousCarbonMonoLevel = setPrevCarbonMono;
    }
    void setCurrentSample(int setCurSamp) {
        currentSample = setCurSamp;
    }
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // methods to access instance variables, getter methods
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    double getCurrentCarbonMono() {
        return currentCarbonMonoLevel;
    }
    double getPreviousCarbonMono() {
        return previousCarbonMonoLevel;
    }
    int setCurrentSample() {
        return currentSample;
    }
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // All other methods
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    void initCarbonMonoMeasurement() {
        // initialise carbon mono measurement capability of mote
        carbonMonoSensor = EDemoBoard.getInstance();
        analogInputs = carbonMonoSensor.getScalarInputs();
    } // end of initCarbonMonoMeasurement()

    int takeCarbonMonoSample() {
        // ...
try {
    // take sample of current carbon mono
    currentSample = analogInputs[0].getValue();

    // For troubleshooting only
    System.out.println("Carbon mono sample taken: "+
currentSample);
}

} catch (IOException ex) {
    ex.printStackTrace();
}

return currentSample;

// end takeCarbonMonoSample()

double convertCarbonMonoSampleToVolt(int currentSample) {
    // For troubleshooting only
    System.out.println("Current Sample: "+ currentSample);

    // Convert sample to volt (see Theory of Operation, blue (p. 19))
    currentCarbonMonoLevel = (double) currentSample * 3 / 1024;

    // For troubleshooting only
    System.out.println("Sample in Volt: "+ currentCarbonMonoLevel);

    return currentCarbonMonoLevel;

} // end convertCarbonMonoSampleToVolt()

boolean hasCarbonMonoLevelChanged() {
    // For troubleshooting only
    System.out.println("Check if carbon mono level has changed.");

    // return FALSE if previous temp equals current temp
    return currentCarbonMonoLevel != previousCarbonMonoLevel;
} //
C.3 Code Listings for Collector Mote

Listing C.14: Special Functions of a Carbon Monoxide Collector Mote

C.3.7 CollectMainSPOT.java

```
/**
 * *****************************************************************
 * * Author : Severin Willisch
 * * email : d9840087@mail.connect.usq.edu.au
 * * Student No. : 0039840087
 * * Purpose : Project for Semester 1 & 2, 2009
 * * Purpose : 09-036 Programming Sun SPOTs in JAVA
 * * *****************************************************************
 */

/**
 * *****************************************************************
 * * Description : Main class for the collector SPOTs.
 * * Inputs : MANIFEST.MF file properties
 * * Outputs : None
 * * Purpose : Reads properties from MANIFEST.MF file & configures ...
 * * Purpose : ... collector SPOT based on the MANIFEST.MF file.
 * * Purpose : After the appropriate functions (threads) have been ...
 * * Purpose : ... started the class waits for all threads to finish.
 * * Purpose : When all threads are finished, class finishes.
 * * *****************************************************************
 */

package org.sunspotworld;

import com.sun.spot.util.*;
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;

public class CollectMainSPOT extends MIDlet {
    // Instance variables
    //
```
C.3 Code Listings for Collector Mote

```java
// Define constants

private static final int HOST_PORT = 67; // Broadcast port
private static final long SAMPLE_TIME_DEFAULT = 1000;

// Manifest file configuration information will be saved to the ...
// ... following variables

private String MANILOCATION = null; // location of SPOT
private String MANIDENT_TEMP = null; // IDs temp data
private String MANIDENT_LIGHT = null; // IDs light data
private String MANIDENT_LEVEL = null; // light measure
private String MANIDENT_BAT = null; // IDs bat data
private String MANIDENT_BATTERY = null; // battery measure
private String MANIDENT_ALARM = null; // IDs alarm data
private String MANIDENT_CARBON_MONO = null; // Ids carbmon data
private String MANICARBON_MONO = null; // carbon monoxide

// The data from the manifest file is of type String. Therefore ...
// ... to make further processing easier they will be converted to ...
// ... variables of type byte or int.

private byte identByteTemp = 0;
private byte identByteLight = 0;
private byte identByteBat = 0;
private byte identByteAlarm = 0;
private byte identByteCarbonMono = 0;
private int tempInt = 0;
private int lightInt = 0;
private int batInt = 0;
private int alarmInt = 0;
private int carbonInt = 0;

// Create boolean variables to set functions a SPOT needs. These ...
// ... variables will be set based on the entries in the manifest.

private boolean tempFlag = false; // true if MANITEMPERATURE = 1
private boolean lightFlag = false; // true if MANILIGHT_LEVEL = 1
private boolean batFlag = false; // true if MANIBATTERY = 1
private boolean alarmFlag = false; // true if MANIALARM = 1
private boolean carbonMonoFlag = false; // true if MANICARBON_MONO = 1

protected void startApp() {
    // monitor the USB (if connected) and recognize commands from host
    new BootloaderListener().start();
```
C.3 Code Listings for Collector Mote

// Names of the different functions a SPOT can have. Function...  
// will only start if the manifest file instructs to run.  
// Each function will start a new thread.

BatLevelThread measureBat = null; // place holder for now
TempThread measureTemp = null; // place holder for now
LightThread measureLight = null; // place holder for now
AlarmTriggerThread detectAlarm = null; // place holder for now
CarMonThread detectCarbonMono = null; // place holder for now

// Get configuration data from Manifest file. After data has...  
// ... been read from manifest it will be used to configure the ...
// ... SPOT.

try {
    MANILOCATION = this.getAppProperty("Location");
    MANIIDENT_TEMP = this.getAppProperty("IdentTemp");
    MANITEMPERATURE = this.getAppProperty("Temperature");
    MANIIDENT_LIGHT = this.getAppProperty("IdentLight");
    MANILIGHT_LEVEL = this.getAppProperty("LightLevel");
    MANIIDENT_BAT = this.getAppProperty("IdentBat");
    MANIBATTERY = this.getAppProperty("Battery");
    MANIIDENT_ALARM = this.getAppProperty("IdentAlarm");
    MANIALARM = this.getAppProperty("Alarm");
    MANIIDENT_CARBON_MONO = this.getAppProperty("IdentCarbonMono");
    MANICARBON_MONO = this.getAppProperty("CarbonMono");

    // For trouble shooting only
    System.out.println("Configuration data OK");
}

} catch (NullPointerException e) {
    System.out.println("Problem reading MANIFEST.MF " + e);
} // end try / catch

// For trouble shooting only
System.out.println("The location is: " + MANILOCATION);

// convert Strings to appropriate data type

try {
    identByteTemp = Byte.parseByte(MANIIDENT_TEMP);
    tempInt = Integer.parseInt(MANITEMPERATURE);
    identByteLight = Byte.parseByte(MANIIDENT_LIGHT);
    lightInt = Integer.parseInt(MANILIGHT_LEVEL);
    identByteBat = Byte.parseByte(MANIIDENT_BAT);
    batInt = Integer.parseInt(MANIBATTERY);
    identByteAlarm = Byte.parseByte(MANIIDENT_ALARM);
    alarmInt = Integer.parseInt(MANIALARM);
identByteCarbonMono = Byte.parseByte(MANIIDENT_CARBON_MONO);
carbonInt = Integer.parseInt(MANI_CARBON_MONO);

} catch (NumberFormatException numberFormatException) {
    System.out.println("Problem converting data: "+numberFormatException);
}

// end try / catch

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// Set flags for functions, if 1 −− true −− function required
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
if (tempInt == 1) { tempFlag = true; } // temperature
if (lightInt == 1) { lightFlag = true; } // light
if (batInt == 1) { batFlag = true; } // battery capacity
if (alarmInt == 1) { alarmFlag = true; } // alarm
if (carbonInt == 1) { carbonMonoFlag = true; } // carbon monoxide

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// start new threads on a SunSPOT depending Manifest file
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
if (tempFlag) {
    long SampleTimeTempStatus =
        Long.parseLong(
            this.getAppProperty("SampleTimeTemp")
        );
    measureTemp = new TempThread(
        MANILOCATION, HOST_PORT, identByteTemp ,
        SampleTimeTempStatus * SAMPLE_TIME_DEFAULT
    );
}

// end if (tempFlag)

if (lightFlag) {
    long SampleTimeLightStatus =
        Long.parseLong(
            this.getAppProperty("SampleTimeLight")
        );
    measureLight = new LightThread(
        MANILOCATION, HOST_PORT, identByteLight ,
        SampleTimeLightStatus * SAMPLE_TIME_DEFAULT
    );
}

// end if (lightFlag)

if (batFlag) {
    long SampleTimeBattStatus =
        Long.parseLong(
            this.getAppProperty("SampleTimeBattery")
        );
    measureBat = new BatLevelThread(
        MANILOCATION, HOST_PORT, identByteBat ,
        SampleTimeBattStatus * SAMPLE_TIME_DEFAULT
    );
}

// end if (batFlag)

if (alarmFlag) {
    long SampleTimeAlarmStatus =
        Long.parseLong(
            this.getAppProperty("SampleTimeAlarm")
        );
}
C.3 Code Listings for Collector Mote

detectAlarm = new AlarmTriggerThread(
    MANILOCATION, HOST_PORT, identByteAlarm,
    SampleTimeAlarmStatus * SAMPLE_TIME_DEFAULT
);
} // end if (alarmFlag)

if (carbonMonoFlag) {
    long SampleTimeCarbonMonoStatus =
        Long.parseLong(
            this.getAppProperty("SampleTimeCarbonMono")
        );

detectCarbonMono = new CarMonThread(
    MANILOCATION, HOST_PORT, identByteCarbonMono,
    SampleTimeCarbonMonoStatus * SAMPLE_TIME_DEFAULT
);
} // end if (carbonMonoFlag)

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// Wait for functions (threads) to end
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
try {
    System.out.println("Wait for threads to finish.");
    if (measureTemp != null) {measureTemp.getThread().join();}
    if (measureLight != null) {measureLight.getThread().join();}
    if (measureBat != null) {measureBat.getThread().join();}
    if (detectAlarm != null) {detectAlarm.getThread().join();}
    if (detectCarbonMono != null) {detectCarbonMono.getThread().join();}
}
} catch (InterruptedException e) {
    System.out.println("Main thread interrupted." + e);
} // end try / catch

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// For trouble shooting only
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
System.out.println("Main thread exiting.");
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// cause the midlet to exit
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
notifyDestroyed();
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
} // end startApp()

protected void pauseApp() {
    // This is not currently called by the Squawk VM
} // end pauseApp()

/**
 * Called if the MIDlet is terminated by the system.
 */
protected void destroyApp(boolean arg0) throws
C.3 Code Listings for Collector Mote

Listing C.15: Main Thread of Collector Mote

MIDletStateChangeException {
} // end destroyApp()
} // end class

C.3.8 TempThread.java

/* ******************************************************************************************
 * Author : Severin Willisch
 * email : d9840087@mail.connect.usq.edu.au
 * Student No. : 0039840087
 * Purpose : Project for Semester 1 & 2, 2009
 * Purpose : 09-036 Programming Sun SPOTs in JAVA
 * ******************************************************************************************/

package org.sunspotworld;

class TempThread implements Runnable {
    // Instance variables
    //
    private String location;
    private Thread t;
    private int hostPortNumber;
    private byte dataIdentifier;
    private long sampleTime;
    private long timeToBeSent;
    private double tempToBeSent;
    //
    TempThread(String threadname, int portNo, byte dataType,
               long timeBetweenSamples) {
        location = threadname;
        t = new Thread(this, location);
        hostPortNumber = portNo;
        dataIdentifier = dataType;
        sampleTime = timeBetweenSamples;
        System.out.println("New thread: "+ t);
        t.start();
    }
*/
// methods to access instance variables, setter methods

void setLocThread(String setLoc){
    location = setLoc;
}

void setThread(Thread setThread){
    t = setThread;
}

void setPortThread(int setPort){
    hostPortNumber = setPort;
}

void setDataIdType(byte setDataType){
    dataIdentifier = setDataType;
}

void setTimeBetweenSamples(long setSampleInterval){
    sampleTime = setSampleInterval;
}

void setTimeForSending(long setSendTime){
    timeToBeSent = setSendTime;
}

void setTempForSending(double setSendTemp){
    tempToBeSent = setSendTemp;
}

// methods to access instance variables, getter methods

String getLocThread(){
    return location;
}

Thread getThread(){
    return t;
}

int getPortThread(){
    return hostPortNumber;
}

byte getDataIdType(){
    return dataIdentifier;
}

long getTimeBetweenSamples(){
    return sampleTime;
}

long getTimeForSending(){
    return timeToBeSent;
}
double getTempForSending() {
    return tempToBeSent;
}

public void run() {
    // For troubleshooting only
    System.out.println("Start temp thread: "+ location);

    CollectorTempMote moteTemp = new CollectorTempMote();

    moteTemp.initTemperatureMeasurement();

    moteTemp.initBroadcastCom();
    moteTemp.setPortNumber(hostPortNumber);
    moteTemp.setLocation(location);
    moteTemp.setWriteDataByte(dataIdentifier);

    moteTemp.getPushAddress();

    moteTemp.tryToOpenBroadcastConnection();

    while (true) {
        // Get the sample time
        timeToBeSent = moteTemp.getCurrentTime();
        //
    }
}
// Indicate sample event (start, i.e LED on)
moteTemp.sampleEventIndication(true, 1);

// Get the current temperature reading to send in datagram
(tempToBeSent = moteTemp.takeTemperatureSample();

// Indicate sample event (end, i.e LED off)

moteTemp.sampleEventIndication(false, 1);

// Reformat tempToBeSent to *****

(tempToBeSent = moteTemp.reformatDataToBeSent(tempToBeSent);

// For trouble shooting only

System.out.println("Has temp changed? " +
   moteTemp.hasTemperatureChanged());

if(moteTemp.hasTemperatureChanged()){
   // For trouble shooting only
   System.out.println("Try to send sample.");

   // Update time and temperature value to be sent
   moteTemp.setWriteDataLong(timeToBeSent);
   moteTemp.setWriteDataDouble(tempToBeSent);

   // Send current sample
   moteTemp.packageAndSendRadiogram();

} // end if block

// move current temp sample to previous sample variable
moteTemp.moveCurrentTempToPreviousTemp();
C.3 Code Listings for Collector Mote

Listing C.16: Temperature Collector Thread

```java
// Go to sleep to conserve battery
moteTemp.sleepForSomeTime(sampleTime, timeToBeSent);

} // end of while loop
} // end of run()
} // end of class
```

C.3.9 LightThread.java

```java
package org.sunspotworld;

class LightThread implements Runnable {

    // Instance variables
    private String location;
    private Thread t;
    private int hostPortNumber;
    private byte dataType;
    private long sampleTime;
    private long timeToBeSent;
    private int lightToBeSent;

    LightThread(String threadname, int portNo, byte dataType, long timeBetweenSamples) {
        location = threadname;
        t = new Thread(this, location);
        hostPortNumber = portNo;
        dataType = dataIdenfier = dataType;
    }
```

/* Author : Severin Willisch
 email : d9840087@mail.connect.usq.edu.au
 Student No. : 0039840087
 Purpose : Project for Semester 1 & 2, 2009
 Purpose : 09–036 Programming Sun SPOTs in JAVA
 */

/* Purpose : This class runs if the MANIFEST.MF file has light ... 
 Purpose : measurement set to TRUE.
 Inputs : Location of mote, port used by host, data type, sample time
 Outputs : Environmental data */
sampleTime = timeBetweenSamples;
System.out.println("New thread: " + t);
t.start();
}

// methods to access instance variables, setter methods
void setLocThread(String setLoc)
{
    location = setLoc;
}

void setThread(Thread setThread)
{
    t = setThread;
}

void setPortThread(int setPort)
{
    hostPortNumber = setPort;
}

void setDataIdType(byte setDataType)
{
    dataIdentifier = setDataType;
}

void setTimeBetweenSamples(long setSampleInterval)
{
    sampleTime = setSampleInterval;
}

void setTimeForSending(long setSendTime)
{
    timeToBeSent = setSendTime;
}

void setLightForSending(int setSendLight)
{
    lightToBeSent = setSendLight;
}

String getLocThread()
{
    return location;
}

Thread getThread()
{
    return t;
}

int getPortThread()
{
    return hostPortNumber;
}

byte getDataIdType()
{
    return dataIdentifier;
}

long getTimeBetweenSamples()
{
    return sampleTime;
}
C.3 Code Listings for Collector Mote

```java
long getTimeForSending()
{
    return timeToBeSent;
}

int getLightForSending()
{
    return lightToBeSent;
}

public void run()
{
    // For trouble shooting only
    System.out.println("Start light thread: " + location);

    CollectorLightMote moteLight = new CollectorLightMote();

    moteLight.initLightMeasurement();

    moteLight.initBroadcastCom();
moteLight.setPortNumber(hostPortNumber);
moteLight.setLocation(location);
moteLight.setWriteDataByte(dataIdentifier);

    moteLight.getPushAddress();

    moteLight.tryToOpenBroadcastConnection();

    while (true){
        //
    }
```

// entry point for light thread
//
// Get the current time. I.e. time when sample is taken
//
timeToBeSent = moteLight.getCurrentTime();

// Indicate sample event (start, i.e LED on)
//
moteLight.sampleEventIndication(true, 2);

// Get the current light reading to send in datagram
//
lightToBeSent = moteLight.takeLightSample();

// Indicate sample event (end, i.e LED off)
//
moteLight.sampleEventIndication(false, 2);

// Convert raw data to luminance (lux). Based on Sun SPOT ...
// Theory of Operation [p. 21] Luminance = 2 times raw data
//
lightToBeSent = moteLight.convertRawDataToLux();

// For trouble shooting only
//
System.out.println("Has light changed? " +
    moteLight.hasLightChanged());

if(moteLight.hasLightChanged()){
    // For trouble shooting only
    System.out.println("Try to send sample. ");

    // Update time and light value to be sent
    moteLight.setWriteDataLong(timeToBeSent);
    moteLight.setWriteDataInt(lightToBeSent);

    // Send current sample
    moteLight.packageAndSendRadiogram();
}

// end if block

// move current light sample to previous sample variable
moteLight.moveCurrentLightToPreviousLight();

// Go to sleep to conserve battery
moteLight.sleepForSomeTime(sampleTime, timeToBeSent);

} // end of while loop
} // end of run()
} // end of class
C.3 Code Listings for Collector Mote

C.3.10 BatLevelThread.java

```java
package org.sunspotworld;

import com.sun.spot.peripheral.Battery.*;

class BatLevelThread implements Runnable {
    private String location; // name of thread
    private Thread t;
    private int hostPortNumber;
    private byte dataIdentifier;
    private long sampleTime;
    private long timeToBeSent;
    private int battToBeSent;

    BatLevelThread(String threadname, int portNo, byte dataType,
                    long timeBetweenSamples) {
        location = threadname;
        t = new Thread(this, location);
        hostPortNumber = portNo;
        dataIdentifier = dataType;
        sampleTime = timeBetweenSamples;
        System.out.println("New thread: "+ t);
        t.start();
    }

    // methods to access instance variables, setter methods
    void setLocThread(String setLoc){
        location = setLoc;
    }

    void setThread(Thread setThread){
        t = setThread;
    }

    void setPortThread(int setPort){
        hostPortNumber = setPort;
    }
}
```
void setDataIdType(byte setDataType){
    dataIdentifier = setDataType;
}

void setTimeBetweenSamples(long setSampleInterval){
    sampleTime = setSampleInterval;
}

void setTimeForSending(long setSendTime){
    timeToBeSent = setSendTime;
}

void setBattForSending(int setSendBatt){
    battToBeSent = setSendBatt;
}

String getLocThread(){
    return location;
}

Thread getThread(){
    return t;
}

int getPortThread(){
    return hostPortNumber;
}

byte getDataIdType(){
    return dataIdentifier;
}

long getTimeBetweenSamples(){
    return sampleTime;
}

long getTimeForSending(){
    return timeToBeSent;
}

int getBattForSending(){
    return battToBeSent;
}

public void run() {
    // For trouble shooting only
}
System.out.println("Start battery thread: " + location);
//
// Start a new battery level collector mote
//
CollectorBattMote moteBatt = new CollectorBattMote();
//
// Initial set up for measurement of battery level
//
moteBatt.initBattMeasurement();
//
// Initial set up for communication
//
moteBatt.initBroadcastCom();
moteBatt.setPortNumber(hostPortNumber);
moteBatt.setLocation(location);
moteBatt.setWriteDataByte(dataIdentifier);
//
// get and push IEEE Address to terminal
//
moteBatt.getPushAddress();
//
// Open up a broadcast connection
//
moteBatt.tryToOpenBroadcastConnection();
//
// Get the current battery reading and send in datagram
//
while (true){
    // Get the current time. I.e. time when sample is taken
    timeToBeSent = moteBatt.getCurrentTime();
    //
    // Indicate sample event (start, i.e LED on)
    //
moteBatt.sampleEventIndication(true, 3);
    //
    // Get the current battery reading to send in datagram
    //
battToBeSent = moteBatt.takeBattSample();
    //
    // Indicate sample event (end, i.e LED off)
moteBatt.sampleEventIndication(false, 3);

// For troubleshooting only
System.out.println("Has battery capacity changed? " + moteBatt.hasBattChanged());

if(moteBatt.hasBattChanged()){
    // For troubleshooting only
    System.out.println("Try to send sample.");

    // Update time and battery value to be sent
    moteBatt.setWriteDataLong(timeToBeSent);
    moteBatt.setWriteDataInt(battToBeSent);

    // Send current sample
    moteBatt.packageAndSendRadiogram();
}

// move current battery sample to previous sample variable
moteBatt.moveCurrentBattToPreviousBatt();

// Go to sleep to conserve battery
moteBatt.sleepForSomeTime(sampleTime, timeToBeSent);

} // end of while loop
} // end of run()
} // end of class

Listing C.18: Battery Collector Thread

C.3.11 AlarmTriggerThread.java

/* *********************************************************************
 * Author : Severin Willisch
 * email   : d9840087@mail.connect.usq.edu.au */
C.3 Code Listings for Collector Mote

* Student No. : 0039840087
* Purpose : Project for Semester 1 & 2, 2009
* Purpose : 09-036 Programming Sun SPOTs in JAVA

---

```java
package org.sunspotworld;

class AlarmTriggerThread implements Runnable {

    private String location;
    private Thread t;
    private int hostPortNumber;
    private byte dataIdentifier;
    private long sampleTime;
    private long timeToBeSent;

    AlarmTriggerThread(String threadname, int portNo, byte dataType, long timeBetweenSamples) {
        location = threadname;
        t = new Thread(this, location);
        hostPortNumber = portNo;
        dataIdentifier = dataType;
        sampleTime = timeBetweenSamples;
        System.out.println("New thread: "+t);
        t.start();
    }

    // methods to access instance variables, setter methods
    // ---------------------------------------------
    void setLocThread(String setLoc) {
        location = setLoc;
    }

    void setThread(Thread setThread) {
        t = setThread;
    }
```

/*
 */

/* ***************************
 * Purpose : This class monitors a specific input to trigger an ... 
 * Purpose : alarm. If the level is low everything is normal. ... 
 * Purpose : i.e. no alarm has been activated. The input is ... 
 * Purpose : checked once every second. If an alarm is triggered ... 
 * Purpose : a broadcast message is sent out. After the message ... 
 * Purpose : has been sent, the thread waits until the the D2 pin ... 
 * Purpose : has been set to low. This prevents the SPOT from ... 
 * Purpose : sending out message that alarm has been triggered. ... 
 * Purpose : After input has been set to low the thread continues ... 
 * Purpose : to monitor the D2 input. 
 * Inputs : Location of mote, 
 *          port used by host, 
 *          data type, 
 *          sample time 
 * Outputs : Alarm trigger sent via radio 
 * ***************************
 */
void setPortThread(int setPort)
    { 
    hostPortNumber = setPort ; 
    }

void setDataIdType(byte setDataType)
    { 
    dataIdentifier = setDataType ; 
    }

void setTimeBetweenSamples(long setSampleInterval)
    { 
    sampleTime = setSampleInterval ; 
    }

void setTimeForSending(long setSendTime)
    { 
    timeToBeSent = setSendTime ; 
    }

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// methods to access instance variables, setter methods
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
String getLocThread()
    { 
    return location ; 
    }

Thread getThread()
    { 
    return t ; 
    }

int getPortThread()
    { 
    return hostPortNumber ; 
    }

byte getDataIdType()
    { 
    return dataIdentifier ; 
    }

long getTimeBetweenSamples()
    { 
    return sampleTime ; 
    }

long getTimeForSending()
    { 
    return timeToBeSent ; 
    }

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// entry point for alarm trigger thread
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
public void run()
    { 
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // For trouble shooting only
    // = = =
    System.out.println("Start alarm thread: " + location);
    // = = = = = = =
    //
```java
// start a new alarm trigger mote
CollectorAlarmMote moteAlarm = new CollectorAlarmMote();

// Initial setup for alarm trigger
moteAlarm.initAlarm();

// Initial setup for communication
moteAlarm.initBroadcastCom();
moteAlarm.setPortNumber(hostPortNumber);
moteAlarm.setLocation(location);
moteAlarm.setWriteDataByte(dataIdentifier);

// get and push IEEE Address to terminal
moteAlarm.getPushAddress();

// Open up a broadcast connection
moteAlarm.tryToOpenBroadcastConnection();

while (true) {

    // Check if input is low. Low means no alarm triggered

    while (moteAlarm.pinStatus()) {

        // if alarm is not triggered go to sleep for specified time
        moteAlarm.sleepForSomeTime(sampleTime, timeToBeSent);

        // Get sample time
        timeToBeSent = moteAlarm.getCurrentTime();

        System.out.println("timeToBeSent : " + timeToBeSent);
        System.out.println("sampleTime : " + sampleTime);
    }

} // end of while (pin.isLow()) loop

// The application only gets to here because pin is no longer...
// ... low. Therefore, send message to notify that an alarm ...
// ... has been triggered.
```
// For troubleshooting only
System.out.println("Send data");

// Update alarm values to be sent
moteAlarm.setWriteDataLong(timeToBeSent);
moteAlarm.setWriteDataInt(1); // 1 = alarm triggered

// Package time & alarm triggered into a radio datagram & send
moteAlarm.packageAndSendRadiogram();

// For troubleshooting only
System.out.println("Data sent");

// Stop the thread until the input has been set to low. Only ... send "alarm triggered" once. Wait for pin D2 to change ... state. Once the pin has changed reenable pin change ... interrupt (listen for new alarm)
moteAlarm.getPinStatus().waitForChange();
moteAlarm.setEnablePinChangeInterruptsToTrue();

// For troubleshooting only
System.out.println("After waitForChange() statement");

// The input has changed back to normal. Therefore send info ... to notify of return to "No alarm". First verify that the ... the input is back to normal position, then send datagram.
if (moteAlarm.pinStatus()) {
    try {
        // For troubleshooting only
        System.out.println("Send data");

        // Get the current time
        timeToBeSent = moteAlarm.getCurrentTime();
    }
C.3 Code Listings for Collector Mote

Listing C.19: Alarm Trigger Thread

```java
// Update alarm values to be sent
moteAlarm.setWriteDataLong(timeToBeSent);
moteAlarm.setWriteDataInt(2); // 2 = alarm not triggered

// Package time & alarm trigger into radio datagram & send
moteAlarm.packageAndSendRadiogram();

// For trouble shooting only
System.out.println("Data sent");
```

C.3.12 CarMonThread.java

```java
package org.sunspotworld;

class CarMonThread implements Runnable {

    // This class runs if the MANIFEST.MF file has carbon ...
    // mono measurement set to TRUE.
    // Inputs : Location of mote,
    //          port used by host,
    //          data type,
    //          sample time
    // Outputs : Environmental data collected

    // ******************************************************************************
    // Author : Severin Willisch
    // email : d9840087@mail.connect.usq.edu.au
    // Student No. : 0039840087
    // Purpose : Project for Semester 1 & 2, 2009
    // Purpose : 09-036 Programming Sun SPOTs in JAVA
    // ******************************************************************************
    */
```
C.3 Code Listings for Collector Mote

// Instance variables
private String location;
private Thread t;
private int hostPortNumber;
private byte dataIdentifier;
private long sampleTime;
private long timeToBeSent;
private double carbonMonoToBeSent;
private int currentCarbonMonoSample;

CarMonThread(String threadname, int portNo, byte dataType,
    long timeBetweenSamples) {
    location = threadname;
    t = new Thread(this, location);
    hostPortNumber = portNo;
    dataIdentifier = dataType;
    sampleTime = timeBetweenSamples;
    System.out.println("New thread: " + t);
    t.start();
}

// methods to access instance variables, setter methods
void setLocThread(String setLoc) {
    location = setLoc;
}
void setThread(Thread setThread) {
    t = setThread;
}
void setPortThread(int setPort) {
    hostPortNumber = setPort;
}
void setDataIdType(byte setDataType) {
    dataIdentifier = setDataType;
}
void setTimeBetweenSamples(long setSampleInterval) {
    sampleTime = setSampleInterval;
}
void setTimeForSending(long setSendTime) {
    timeToBeSent = setSendTime;
}
void setTempForSending(double setSendTemp) {
    carbonMonoToBeSent = setSendTemp;
}

// methods to access instance variables, getter methods
C.3 Code Listings for Collector Mote

```java
String getLocThread() {
    return location;
}

Thread getThread() {
    return t;
}

int getPortThread() {
    return hostPortNumber;
}

byte getDataIdType() {
    return dataIdentifier;
}

long getTimeBetweenSamples() {
    return sampleTime;
}

long getTimeForSending() {
    return timeToBeSent;
}

double getTempForSending() {
    return carbonMonoToBeSent;
}
```

```java
public void run() {
    // For trouble shooting only
    System.out.println("Start carbon mono thread: " + location);
    //
    // start a new carbon monoxide collector mote
    CollectorCarbonMonoMote moteCarMon = new CollectorCarbonMonoMote();
    //
    // Initial set up for measurement of carbon monoxide
    moteCarMon.initCarbonMonoMeasurement();
    //
    // Initial set up for communication
    moteCarMon.initBroadcastCom();
    moteCarMon.setPortNumber(hostPortNumber);
    moteCarMon.setLocation(location);
}
```
moteCarMon.setWriteDataByte(dataIdentifier);

// get and push IEEE Address to terminal
moteCarMon.getPushAddress();

// Open up a broadcast connection
moteCarMon.tryToOpenBroadcastConnection();

// read the current value on the A0 input

while (true) {
    // Get the current time. I.e. time when sample is taken
    timeToBeSent = moteCarMon.getCurrentTime();

    // Indicate sample event (start, i.e LED on)
    moteCarMon.sampleEventIndication(true, 5);

    // Get the current carbon mono reading to send in datagram
    currentCarbonMonoSample = moteCarMon.takeCarbonMonoSample();

    // Indicate sample event (end, i.e LED off)
    moteCarMon.sampleEventIndication(false, 5);

    // Convert Sample to volt
    carbonMonoToBeSent = moteCarMon.convertCarbonMonoSampleToVolt(currentCarbonMonoSample);

    // Reformat sample to be sent to ****.** format
    carbonMonoToBeSent = moteCarMon.reformatDataToBeSent(carbonMonoToBeSent);

    // For trouble shooting only
System.out.println("Has carbon mono level changed? " + moteCarMon.hasCarbonMonoLevelChanged());

// if (moteCarMon.hasCarbonMonoLevelChanged()) {

// For trouble shooting only
System.out.println("Try to send sample.");

// Update time and carbon mono value to be sent
moteCarMon.setWriteDataLong(timeToBeSent);
moteCarMon.setWriteDataDouble(carbonMonoToBeSent);

// Send current sample
moteCarMon.packageAndSendRadiogram();

} // end if block

// move current carbon level to previous carbon level variable
moteCarMon.moveCurrentCarbonMonoToPreviousCarbonMono();

// Go to sleep to conserve battery
moteCarMon.sleepForSomeTime(sampleTime, timeToBeSent);

} // end of while loop
} // end of run()

Listing C.20: Carbon Monoxide Collector Thread
C.4 Pseudo Code for Interface Mote

C.4.1 Pseudo Code for Climate Interface Thread

```java
Pseudo Code for Temperature Interface
ClimateInterfaceThread implements Runnable
Instance variables
    private String systemInterface;
    private Thread t;
    private int hostPortNumber;
    private byte interfaceIdAsByte;
    private byte dataIdAsByte = 1; // identifies temp data
    private boolean listenForNewDatagram = true;

Methods to access instance variables, setter methods
Methods to access instance variables, getter methods

public void run() {
    Start a new climate interface mote
    call new ClimateInterfaceMote();

    Get and push IEEE address
    call getPushAddress();

    Initial set up for communication
    call initInterfaceMoteCommunication();
    call setPortNumber(hostPortNumber);

    Open up a broadcast connection
    call tryToOpenBroadcastConnection();

    Initial set up to communicate with external system
    call initExternalCommunication();

    WHILE (listenForNewDatagram)
        Listen for new data packet
        call listenFor();

        IF (is data for Climate Interface)
            Prepare data to send to external system via serial comm.
            call packageTempDataToSendToClimateControl(
                call getReadDataUTF(),
                call getReadDataDouble());

            Push new data to external system
            call sendDataToExternalSystem();
        ELSE
            call sendExternalSystemStatus();
        END IF (is data for Climate Interface)

    END of WHILE
END of run()
END of class
```

Listing C.21: Pseudo Code Climate Interface Thread
C.4.2 Pseudo Code for Light Interface Thread

LightingInterfaceThread implements Runnable

```java
public void run()
{
    Start a new climate interface mote
    new LightInterfaceMote();

    Get and push IEEE address
    call getPushAddress();

    Initial set up for communication
    call initInterfaceMoteCommunication();
    call setPortNumber(hostPortNumber);

    Open up a broadcast connection
    call tryToOpenBroadcastConnection();

    Initial set up to communicate with external system
    call initExternalCommunication();

    WHILE (listenForNewDatagram)
    Listen for new data packet
    call listenFor();

    IF (data is for Light Interface)
    Prepare data to send to external system via serial comm.
    call packageLightDataToSendToLightControl(getReadDataUTF(),
    getReadDataInt());
    Push new data to external system
    call sendDataToExternalSystem();
    END of IF (data is for Light Interface)

    END of WHILE
    END of run()
```

Listing C.22: Pseudo Code Light Interface Thread

C.4.3 Pseudo Code for Alarm Interface Thread

AlarmInterfaceThread implements Runnable

```java
```
C.4 Pseudo Code for Interface Mote

**Instance variables**
- private String systemInterface;
- private Thread t;
- private int hostPortNumber;
- private byte interfaceIdAsByte;
- private byte dataIdAsByte = 4; // identifies temp data
- private boolean listenForNewDatagram = true;

Methods to access instance variables, setter methods
Methods to access instance variables, getter methods

```
public void run() {
    Start a new alarm interface mote
    new AlarmInterfaceMote();

    Get and push IEEE address
    call getPushAddress();

    Initial set up for communication
    call initInterfaceMoteCommunication();
    call setPortNumber(hostPortNumber);

    Open up a broadcast connection
    call tryToOpenBroadcastConnection();

    Initial set up to interface with external system
    call initExternalCommunication();

    WHILE (listenForNewDatagram) {
        Listen for new data packet
        call listenFor();

        IF (data is for Alarm Interface)
            Prepare data to send to external system via serial comm.
            call packageAlarmDataToSendToAlarmSystem(getReadDataUTF(),
                getReadDataInt());
            Push new data to external system
            call sendDataToExternalSystem();
        END of IF (data is for Alarm Interface)
    }

    END of WHILE
    END of run()
END of class
```

Listing C.23: Pseudo Code Alarm Interface Thread

C.4.4 Pseudo Code for Carbon Monoxide Interface Thread

```
Pseudo code for Carbon monoxide Interface
CarbMonInterfaceThread implements Runnable

Instance variables
- private String systemInterface;
- private Thread t;
- private int hostPortNumber;
- private byte interfaceIdAsByte;
```
private byte dataIdAsByte = 5; // identifies CarMon data
private boolean listenForNewDatagram = true;

Methods to access instance variables, setter methods
Methods to access instance variables, getter methods

public void run() {
    Start a new carbon monoxide interface mote
    new CarbMonInterfaceMote();

    Get and push IEEE address
    call getPushAddress();

    Initial set up for communication
    call initInterfaceMoteCommunication();
    call setPortNumber(hostPortNumber);

    Open up a broadcast connection
    call tryToOpenBroadcastConnection();

    Initial set up to interface with external system
    call initExternalCommunication();

    WHILE (listenForNewDatagram)
        Listen for new data packet
        call listenFor();

        IF (data is for Carbon Mono Interface)
            Prepare data to send to external system via serial comm.
            call packageCarbonMonoDataToSendToCarbMonSystem(
                getReadDataUTF(),
                getReadDataDouble());

            Push new data to external system
            call sendDataToExternalSystem();
        END of IF (data is for Carbon Mono Interface)
    END of WHILE

END of run()
END of class

Listing C.24: Pseudo Code Carbon Monoxide Interface Thread

C.4.5 Pseudo Code for Door Interface Thread

Pseudo Code for Door Interface
DoorInterfaceThread implements Runnable

Instance variables
    private String systemInterface;
    private Thread t;
    private int hostPortNumber;
    private long unlockTime;
    private byte interfaceIdAsByte;
    private byte dataIdAsByte = 16; // identifies unlock door
    private boolean listenForNewDatagram = true;
C.4 Pseudo Code for Interface Mote

### Methods to access instance variables, setter methods

### Methods to access instance variables, getter methods

```java
public void run() {
    Start a new climate interface mote
    new DoorInterfaceMote();

    Get and push IEEE address
    call getPushAddress();

    Initial set up for communication
    call initInterfaceMoteCommunication();
    call setPortNumber(hostPortNumber);

    Open up a broadcast connection
    call tryToOpenBroadcastConnection();

    Initial set up to interface with external system
    call initDoorInterface();

    WHILE (listenForNewDatagram)
        Listen for new data packet
        call listenFor();

        IF (data is for door interface)
            Push new data to external system
            call unlockTheDoor(unlockTime);
        END of IF (data is for door interface)
    END of WHILE

END of run()
END of class
```

Listing C.25: Pseudo Code Door Interface Thread

---

C.4.6 Pseudo Code to Send External System Status Thread

### Pseudo Code to send external system status

`sendStatus implements Runnable`

**Instance variables**

- `private String systemInterface;`
- `private Thread t;`
- `private int portNumberSend;`
- `private int portNumberListen;`
- `private byte statusIdentification;`
- `private String hostAddress;`
- `private boolean extSystemOK = false;`
- `private boolean hostAddressNotKnown = true;`

**Methods to access instance variables, setter methods**

**Methods to access instance variables, getter methods**

```java
public void run() {
    Create object to send status to host
```
new InterfaceMote();

Initial setup for communication (listening)
call initInterfaceMoteCommunication();
call setPortNumber(portNumberListen);

Open up a broadcast connection to listen for host
call tryToOpenBroadcastConnection();

WHILE (hostAddressNotKnown)
{
    Listen for host broadcast
call listenFor();

    IF (broadcast is from host)
        Get host address
call getReadDataLong();
        Set flag to false (i.e. address is now known)
        Sleep for some time
        END of IF (broadcast is from host)
    END of WHILE

Create object to send status to host
new RadiogramCommClass();

Initial setup for communication (sending)
call initRadiogramCom();
call setPortNumber(portNumberSend);
call setLocation(systemInterface);
call setHostAddress(hostAddress);
call setWriteDataByte(statusIdentification);

Open up a radiogram connection to host
call tryToOpenRadiogramConnection();

Initial set up to verify status of external system
call initExternalSystemStatus();

WHILE (true)
{
    Check status of external system
call errorExternalSystem();
call setWriteBoolean(extSystemOK);

    Send status to host
call packageAndSendRadiogram();

    Sleep
END of WHILE

END of sendStatus

END of class

Listing C.26: Pseudo Code Send External System Status Thread

C.4.7 Pseudo Code for Interface Mote Main Thread
Pseudo Code for Interface Mote

InterfaceMainSPOT extends MIDlet

Instance variables
- private static final int HOST_PORT = 67;  // Broadcast port
- private String INTERFACE = null;
- private String MANI_DOOR_OPEN_TIME = null;
- private byte interfaceByteId = 0;
- private long doorOpenTime = 0;
- private String interfaceStringId = null;

protected void startApp()
    Monitor the USB (if connected) and recognize commands from host
    new BootloaderListener().start();

    Get configuration data from Manifest file
    Convert Strings to appropriate data type

    Find external system status id based on MANIFEST.MF file
    add 100 to interfaceByteId

    Start new thread depending on MANIFEST.MF file
    SWITCH (Identification of interface)
    CASE 11:
        new ClimateInterfaceThread
        new sendStatus
        Wait for thread to finish
        BREAK;

    CASE 12:
        new LightingInterfaceThread
        new sendStatus
        Wait for thread to finish
        BREAK;

    CASE 14:
        new AlarmInterfaceThread
        new sendStatus
        Wait for thread to finish
        BREAK;

    CASE 15:
        new CarbMonInterfaceThread
        new sendStatus
        Wait for thread to finish
        BREAK;

    CASE 16:
        new DoorInterfaceThread
        new sendStatus
        Wait for thread to finish
        BREAK;

    END of SWITCH (Identification of interface)
END of startApp()
END of class

Listing C.27: Pseudo Code Interface Main Thread
C.5 Code Listings for Interface Mote

C.5.1 InterfaceMote.java

```java
package org.sunspotworld;

import com.sun.spot.io.j2me.radiogram.RadiogramConnection;
import com.sun.spot.sensorboard.EDemoBoard;
import com.sun.spot.sensorboard.peripheral.ITriColorLED;
import java.io.IOException;
import javax.microedition.io.Connector;
import javax.microedition.io.Datagram;
import com.sun.spot.util.Utils;

class InterfaceMote {
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // Instance variables
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    private String systemInterface;
    private int portNumber;
    private String ourAddress;
    private RadiogramConnection rCon;
    private Datagram dg;
    private long readDataLong = 0;
    private byte readDataByte = 0;
    private String readDataUTF = null;
    private int readDataInt = 0;
    private double readDataDouble = 0;
    private boolean readDataBool = false;

    EDemoBoard serialComm = EDemoBoard.getInstance();
    private int baud = 19200;
    private int databits = 8;
    private int parity = 1;  // 0 = none, 1 = odd, 2 = even
    private int stopbits = 1;
    private String externalDataString = null;

    private ITriColorLED[] sendCommIndication = EDemoBoard.getInstance().getLEDs();
```

C.5 Code Listings for Interface Mote

// methods to access instance variables, setter methods
void setSystemInterface(String setLoc) {
    systemInterface = setLoc;
}

void setPortNumber(int setPort) {
    portNumber = setPort;
}

void setBaud(int baudValue) {
    baud = baudValue;
}

void setDatabits(int databitsValue) {
    databits = databitsValue;
}

void setParity(int parityValue) {
    parity = parityValue;
}

void setStopbits(int stopbitsValue) {
    stopbits = stopbitsValue;
}

void setExternalDataString(String dataOut) {
    externalDataString = dataOut;
}

// methods to access instance variables, getter methods
String getSystemInterface() {
    return systemInterface;
}

String getOurAddress() {
    return ourAddress;
}

int getPortNumber() {
    return portNumber;
}

long getReadDataLong() {
    return readDataLong;
}

byte getReadDataByte() {
    return readDataByte;
}

String getReadDataUTF() {
return readDataUTF;
}

int getReadDataInt(){
    return readDataInt;
}

double getReadDataDouble(){
    return readDataDouble;
}

boolean getReadDataBoolean(){
    return readDataBool;
}

int getBaud (){}
    return baud;
}

int getDatabits(){
    return databits;
}

int getParity (){{
    return parity;
}

int getStopbits (){{
    return stopbits;
}

String getExternalDataString (){{
    return externalDataString;
}

// end initBroadcastCom

void initInterfaceMoteCommunication (){{
    // Initial set up for communication
    rCon = null;
    dg = null;
    } // end initBroadcastCom

void getPushAddress (){{
    // get and push IEEE Address to terminal. Troubleshoot only
    ourAddress = System.getProperty("IEEE_ADDRESS");
    System.out.println("Starting app. on " + ourAddress + " ...");
    } // end getPushAddress()
C.5 Code Listings for Interface Mote

```java
void tryToOpenBroadcastConnection()
{
    try {
        // Open up a radiogram connection to listen for data from ... collector motes.
        rCon = (RadiogramConnection) Connector.open("radiogram://:" + portNumber);
        dg = rCon.new Datagram(rCon.getMaximumLength());
        //
        // For trouble shooting only
        //
        System.out.println("Radiogram connection is open.");
    }
    catch (IOException ex) {
        ex.printStackTrace();
    } // end try / catch
} // end tryToOpenBroadcastConnection()

void listenFor()
{
    //
    // listen for new datagram data
    //
    try {
        rCon.receive(dg);
        readDataLong = dg.readLong();
        readDataByte = dg.readByte();
        readDataUTF = dg.readUTF();
        readDataInt = dg.readInt();
        readDataDouble = dg.readDouble();
        readDataBool = dg.readBoolean();
        //
        // For trouble shooting only
        //
        System.out.println("Data packet received. " + readDataByte);
    }
    catch (IOException ex) {
        ex.printStackTrace();
    } // end try / catch
} // end listenFor()

void sleepForSomeTime(long samplePeriod, long lastTimeToBeSent) {
    //
    // let the mote rest to conserve battery, probably never called ...
    // in interface mote.
    //
    Utils.sleep(samplePeriod - (System.currentTimeMillis() - lastTimeToBeSent));
    //
```
C.5 Code Listings for Interface Mote

Listing C.28: Generic Functions of an Interface Mote

```java
} // end of sleepForSomeTime()

long getCurrentTime(){
  // take time of sample
  return System.currentTimeMillis();
} // end getCurrentTime()

void initExternalCommunication(){
  // For troubleshooting only
  System.out.println("Baud\t: " + baud);
  System.out.println("Databits\t: " + databits);
  System.out.println("Parity\t: " + parity);
  System.out.println("Stop bits\t: " + stopbits);

  // Init serial communication to external system
  serialComm.initUART(baud, databits, parity, stopbits);
} // end of initExternalCommunication()

void sendDataToExternalSystem(){
  // Send data via serial communication to external system
  for (int i = 0; i<8; i++){
    sendCommIndication[i].setRGB(0, 0, 255);
    sendCommIndication[i].setOn();
  } // set each LED to blue and turn on
  serialComm.writeUART(externalDataString);

  for (int i = 7; i >= 0; i--){
    sendCommIndication[i].setOff();
  } // turn LEDs off

  System.out.println("Data sent via serial comm to external system");
} // end of sendDataToExternalSystem()

} // end CollectorMote class
```

C.5.2 ClimateInterfaceMote.java

```java
/* ******************************************
 * Author : Severin Willisch
 */
```
/* Purpose : This class includes all the specific functions of a ... */

package org.sunspotworld;

class ClimateInterfaceMote extends InterfaceMote{

    void packageTempDataToSendToClimateControl(String location, double value) {
        // Prepare data to send via serial communication
        // Required data: >> location --> String
        //                >> value --> double
        StringBuffer prepareOutputString = new StringBuffer(location);
        prepareOutputString.append(", ");
        prepareOutputString.append(value);
        setExternalDataString(prepareOutputString.toString());
        // For trouble shooting only
        System.out.println(getExternalDataString());
    }

} // end of class

Listing C.29: Special Functions of Climate Interface Mote

C.5.3 LightInterfaceMote.java

/* Purpose : This class includes all the specific functions of a ... */
C.5 Code Listings for Interface Mote

* Purpose : light control interface mote.
* ***************************************************************************/
package org.sunspotworld;

class LightInterfaceMote extends InterfaceMote {
    void packageLightDataToSendToLightControl(String location, int value) {
        // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
        // Prepare data to send via serial communication
        // Required data: >> location --> String
        //    >> value --> int
        // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
        StringBuffer prepareOutputString = new StringBuffer(location);
        prepareOutputString.append(",");
        prepareOutputString.append(value);
        setExternalDataString(prepareOutputString.toString());
        // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
        // For trouble shooting only
        // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
        System.out.println(getExternalDataString());
    } // end of packageDataToSendToExternalSystem()
} // end of class

Listing C.30: Special Functions of Light Interface Mote

C.5.4 AlarmInterfaceMote.java

/*/ ***************************************************************************/
 package org.sunspotworld;

class AlarmInterfaceMote extends InterfaceMote{
    void packageAlarmDataToSendToAlarmSystem(String location, int value) {
        // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
        // Prepare data to send via serial communication
        // Required data: >> location --> String
        // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
C.5 Code Listings for Interface Mote

Listing C.31: Special Functions of Alarm Interface Mote

```java
// value --> int
StringBuffer prepareOutputString = new StringBuffer(location);
prepareOutputString.append(", ");
prepareOutputString.append(value);
setExternalDataString(prepareOutputString.toString());

// For trouble shooting only
System.out.println(getExternalDataString());
```

C.5.5 CarbMonInterfaceMote.java

```java
/* ******************************************
 * Author : Severin Willisch
 * email : d9840087@mail.connect.usq.edu.au
 * Student No. : 0039840087
 * Purpose : Project for Semester 1 & 2, 2009
 * Purpose : 09-036 Programming Sun SPOTs in JAVA
 * *******************************************/

package org.sunspotworld;

class CarbMonInterfaceMote extends InterfaceMote{

   void packageCarbonMonoDataToSendToCarbMonSystem(String location, double value) {
      // Prepare data to send via serial communication
      // Required data: >> location --> String
      // >> value --> double
      StringBuffer prepareOutputString = new StringBuffer(location);
      prepareOutputString.append(", ");
      prepareOutputString.append(value);
      setExternalDataString(prepareOutputString.toString());
   }

   // For trouble shooting only
```
C.5 Code Listings for Interface Mote

Listing C.32: Special Functions of Carbon Monoxide Interface Mote

C.5.6 DoorInterfaceMote.java

```java
package org.sunspotworld;

import com.sun.spot.sensorboard.EDemoBoard;
import com.sun.spot.sensorboard.EDemoController;
import com.sun.spot.sensorboard.io.InputPin;
import com.sun.spot.sensorboard.io.PinDescriptor;
import com.sun.spot.sensorboard.peripheral.ITriColorLED;
import com.sun.spot.util.Utils;

class DoorInterfaceMote extends InterfaceMote{
    // Instance variables
    private PinDescriptor unlockDoorOutput;
    private EDemoBoard doorCommand;
    private EDemoController doorCommandController;
    private InputPin pin;

    private ITriColorLED[] indicateDoorUnlocked = EDemoBoard.getInstance().getLEDs();

    void setPinStatus(InputPin setPin){
        pin = setPin;
    }
}
```

Purpose: This class includes all the specific functions of a door interface mote. A digital output will be used to forward the command to unlock the door.
// methods to access instance variables, getter methods
//
InputPin getPinStatus() {
    return pin;
}
//
// All other methods
//
void initDoorInterface() {
    // initialise door unlocking function of mote
    unlockDoorOutput = EDemoController.D2;
    doorCommand = EDemoBoard.getInstance();
    doorCommandController = doorCommand.getEDemoController();
    pin = new InputPin(unlockDoorOutput, doorCommandController);
    doorCommandController.setPinDirection(unlockDoorOutput, true);
}

void unlockTheDoor(long unlockDoorTime) {
    // Activate one LED to show that door is unlocked
    indicateDoorUnlocked[1].setRGB(0, 0, 255);
    indicateDoorUnlocked[1].setOn();

    // Set pin to unlock(true)
    doorCommandController.setPinValue(unlockDoorOutput, true);

    // For troubleshooting only
    System.out.println("Door unlocked");

    Utils.sleep(unlockDoorTime); // door unlocked while mote sleeps

    // Set pin to lock(false)
    doorCommandController.setPinValue(unlockDoorOutput, false);

    // Deactivate LED to show that door is locked
    indicateDoorUnlocked[1].setRGB(0, 0, 255);
    indicateDoorUnlocked[1].setOff();
C.5 Code Listings for Interface Mote

Listing C.33: Special Functions of Door Interface Mote

```java
// For troubleshooting only
System.out.println("Door locked");

}} // end of pushDataToExternalSystem()
}} // end of class
```

C.5.7 RadiogramCommClass.java

```java
/* ********************************************
 * Author : Severin Willisch
 * email : d9840087@mail.connect.usq.edu.au
 * Student No. : 0039840087
 * Purpose : Project for Semester 1 & 2, 2009
 * Purpose : 09-036 Programming Sun SPOTs in JAVA
 * ********************************************
 */

/* ********************************************
 * Purpose : This class includes all the special functionalities ...
 * Purpose : to send the status of an external system to the host.
 * ********************************************
 */
package org.sunspotworld;

import com.sun.spot.io.j2me.radiogram.RadiogramConnection;
import com.sun.spot.sensorboard.EDemoBoard;
import com.sun.spot.sensorboard.EDemoController;
import com.sun.spot.sensorboard.io.InputPin;
import com.sun.spot.sensorboard.io.PinDescriptor;
import java.io.IOException;
import javax.microedition.io.Connector;
import javax.microedition.io.Datagram;

class RadiogramCommClass {
    // Instance variables
    private RadiogramConnection rCon;
    private Datagram dg;
    private PinDescriptor statusExternalSystem;
    private EDemoBoard status;
    private EDemoController statusController;
    private InputPin pin;
    private boolean extSysStatus = false;
    private String location;
    private String hostAddress;
    private int portNumber;
}```
private long writeDataLong = 0;
private byte writeDataByte = 0;
private int writeDataInt = 0;
private double writeDataDouble = 0;
private boolean writeDataBool = false;

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// methods to access instance variables, setter methods
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

InputPin getPinStatus() {
    return pin;
}

void setLocation(String setLoc) {
    location = setLoc;
}

void setPortNumber(int setPort) {
    portNumber = setPort;
}

void setHostAddress(String hostAddr) {
    hostAddress = hostAddr;
}

void setWriteDataLong(long setDataLong) {
    writeDataLong = setDataLong;
}

void setWriteDataByte(byte setDataByte) {
    writeDataByte = setDataByte;
}

void setWriteDataInt(int setDataInt) {
    writeDataInt = setDataInt;
}

void setWriteDataDouble(double setDataDouble) {
    writeDataDouble = setDataDouble;
}

void setWriteBoolean(boolean setDataBoolean) {
    writeDataBool = setDataBoolean;
}

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// methods to access instance variables, getter methods
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

String getLocation() {
    return location;
}

int getPortNumber() {
    return portNumber;
}

String getHostAddress() {
```java
    return hostAddress;
}

long getWriteDataLong() {
    return writeDataLong;
}

byte getWriteDataByte() {
    return writeDataByte;
}

int getWriteDataInt() {
    return writeDataInt;
}

double getWriteDataDouble() {
    return writeDataDouble;
}

boolean getWriteDataBoolean() {
    return writeDataBool;
}

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// All other methods
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
void initRadiogramCom() {
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // Initial set up for communication
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    rCon = null;
    dg = null;
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
} // end initRadiogramCom()

void tryToOpenRadiogramConnection() {
    try {
        // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
        // Open up a radiogram connection to the host
        // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
        rCon = (RadiogramConnection) Connector.open(
"radiogram://" + hostAddress + ":" + portNumber);
        ((RadiogramConnection) rCon).setMaxBroadcastHops(4);
        dg = rCon.newDatagram(rCon.getMaximumLength());
        // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
        // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
        // For troubleshooting only
        // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
        System.out.println("Broadcast is open.");
        // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    } catch (IOException ex) {
        ex.printStackTrace();
    } // end try / catch
} // end tryToOpenRadiogramConnection()
```
void initExternalSystemStatus() {
    statusExternalSystem = EDemoController.D2;
    status = EDemoBoard.getInstance();
    statusController = status.getEDemoController();
    pin = new InputPin(statusExternalSystem, statusController);
    statusController.setPinDirection(statusExternalSystem, false);
    statusController.enablePinChangeInterrupts(pin);

    // For trouble shooting only
    System.out.println("External system status initialised.");

} // end of initExternalSystemStatus()

boolean errorExternalSystem() {
    // Check if input is low on D2. Low = external System OK
    extSysStatus = pin.isLow();

    // For trouble shooting only
    System.out.println("External System Status: " + extSysStatus);

    return extSysStatus;

} // end of errorExternalSystem()

void packageAndSendRadiogram() {

    // package data into datagram and send
    dg.reset();
    try {
        dg.writeLong(writeDataLong);
        dg.writeByte(writeDataByte);
        dg.writeUTF(location);
        dg.writeInt(writeDataInt);
        dg.writeDouble(writeDataDouble);
        dg.writeBoolean(writeDataBool);
        rCon.send(dg);
        // For trouble shooting only
        System.out.println("Data packet sent.");
    }
    catch (IOException ex) {
        ex.printStackTrace();
    } // end try / catch
C.5 Code Listings for Interface Mote

Listing C.34: Radiogram Communication Class

C.5.8 InterfaceMainSPOT.java

```java
package org.sunspotworld;

import com.sun.spot.util.*;
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;

public class InterfaceMainSPOT extends MIDlet {
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // Instance variables
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    private static final int PORT_LISTEN_HOST = 66;
    private static final int PORT_LISTEN_SPOT = 67;
    private static final int PORT_SEND_HOST = 68;

    private String MAN_INTERFACE = null;
    private String MAN_DOOR_OPEN_TIME = null;
    private byte interfaceByteId = 0;
    private byte expandByte = 100;
    private byte statusByteId = 0;
    private long doorOpenTime = 0;
    private String interfaceStringId = null;
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

    protected void startApp() throws MIDletStateChangeException {
        // monitor the USB (if connected) and recognize commands from host
    }
```
new BootloaderListener().start();

// Get configuration data from Manifest file. After data has ...  
// ... been read from manifest it will be used to configure the ...  
// ... SPOT.

try {
    MANI_INTERFACE = this.getAppProperty("Interface");
    MANI_DOOR_OPEN_TIME = this.getAppProperty("DoorOpenTime");

    // For trouble shooting only
    System.out.println("Configuration data OK");
}
catch (NullPointerException e) {
    System.out.println("Problem reading MANIFEST.MF " + e);
} // end try / catch

// convert Strings to appropriate data type

try {
    interfaceByteId = Byte.parseByte(MANI_INTERFACE);
    doorOpenTime = Long.parseLong(MANI_DOOR_OPEN_TIME);
} catch (NumberFormatException numberFormatException) {
    System.out.println("Problem converting data: 
    + numberFormatException
    
} // end try / catch

// Find external system status id based on MANIFEST.MF file

statusByteId = (byte) (interfaceByteId + expandByte);

// start new thread on a SunSPOT depending on MANIFEST.MF file

switch (interfaceByteId){
    case 11:
        interfaceStringId = "Climate Control";
        ClimateInterfaceThread climateInterMote = 
        new ClimateInterfaceThread( 
            interfaceStringId, PORT_LISTEN_SPOT, 
            interfaceByteId);

        // Start thread to send status of external system to host
        
        SendStatus tempSysStatus = new SendStatus("statusClimate", 
                PORT_SEND_HOST, PORT_LISTEN_HOST, statusByteId); 

        //
try {
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // wait for thread to finish
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    climateInterMote.getThread().join();
tempSysStatus.getThread().join();
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
} catch (InterruptedException ex) {
    ex.printStackTrace();
} // end of try / catch
break;

case 12:
    interfaceStringId = "Lighting Control";
    LightingInterfaceThread lightingInterMote =
        new LightingInterfaceThread(
            interfaceStringId, PORT_LISTEN_SPOT,
            interfaceByteId);
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // Start thread to send status of external system to host
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    SendStatus lightSysStatus = new SendStatus("statusLight",
        PORT_SEND_HOST, PORT_LISTEN_HOST, statusByteId);
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    try {
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
        // wait for thread to finish
        // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
        lightingInterMote.getThread().join();
        lightSysStatus.getThread().join();
        // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    } // end of try / catch
    break;

case 14:
    interfaceStringId = "Alarm System";
    AlarmInterfaceThread alarmInterMote =
        new AlarmInterfaceThread(
            interfaceStringId, PORT_LISTEN_SPOT,
            interfaceByteId);
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // Start thread to send status of external system to host
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    SendStatus alarmSysStatus = new SendStatus("statusAlarm",
        PORT_SEND_HOST, PORT_LISTEN_HOST, statusByteId);
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    try {
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
        // wait for thread to finish
        // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
        alarmInterMote.getThread().join();
        alarmSysStatus.getThread().join();
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    } // end of try / catch
catch (InterruptedException ex) {
    ex.printStackTrace();
} // end of try / catch

break;

case 15:
    interfaceStringId = "Carbon Mono System";
    CarbMonInterfaceThread ventInterMote =
        new CarbMonInterfaceThread(
            interfaceStringId, PORT_LISTEN_SPOT,
            interfaceByteId);

    // Start thread to send status of external system to host
    SendStatus carbMonSysStatus = new SendStatus("statusCarbMon",
        PORT_SEND_HOST, PORT_LISTEN_HOST, statusByteId);

    try {
        // wait for thread to finish
        ventInterMote.getThread().join();
        carbMonSysStatus.getThread().join();
    }
    catch (InterruptedException ex) {
        ex.printStackTrace();
    } // end of try / catch

    break;

case 16:
    interfaceStringId = "Door Interface";
    DoorInterfaceThread doorInterMote =
        new DoorInterfaceThread(
            interfaceStringId, PORT_LISTEN_SPOT, interfaceByteId,
            doorOpenTime);

    // Start thread to send status of external system to host
    SendStatus doorSysStatus = new SendStatus("statusDoor",
        PORT_SEND_HOST, PORT_LISTEN_HOST, statusByteId);

    try {
        // wait for thread to finish
        doorInterMote.getThread().join();
        doorSysStatus.getThread().join();
    }
    catch (InterruptedException ex) {
        ex.printStackTrace();
    } // end of try / catch

    break;
Listing C.35: Main Thread of Interface Mote

C.5.9 ClimateInterfaceThread.java

```java
package org.sunspotworld;

class ClimateInterfaceThread implements Runnable {

    // Instance variables
    private String systemInterface;
    private Thread t;
    private int hostPortNumber;
    private byte interfaceIdAsByte;
    private byte dataIdAsByte = 1; // identifies temp data
    private boolean listenForNewDatagram = true;

    ClimateInterfaceThread(String threadname, int portNo, byte moteType) {
        systemInterface = threadname;
    }
}
```
t = new Thread(this, systemInterface);
hostPortNumber = portNo;
interfaceIdAsByte = moteType;
System.out.println("New thread: " + t);
t.start();
} // end constructor

// methods to access instance variables, setter methods

void setThread(Thread setThread) {
    t = setThread;
}

void setListenForNewDatagram(boolean lisForDgs) {
    listenForNewDatagram = lisForDgs;
}

// methods to access instance variables, getter methods

Thread getThread() {
    return t;
}

boolean getListenForNewDatagram() {
    return listenForNewDatagram;
}

public void run() {

    // For trouble shooting only

    System.out.println("Start interface thread: " + systemInterface + " " + interfaceIdAsByte);

    // start a new climate interface mote

    ClimateInterfaceMote moteClimate = new ClimateInterfaceMote();

    // get and push IEEE address

    moteClimate.getPushAddress();

    // Initial set up for communication

    moteClimate.initInterfaceMoteCommunication();
    moteClimate.setPortNumber(hostPortNumber);

    // Open up a broadcast connection
// tryToOpenBroadcastConnection();

// Initial setup to communicate with external system
moteClimate.initExternalCommunication();

// For troubleshooting only
System.out.println("Start listening for datagrams on port: " +
    hostPortNumber);

while (listenForNewDatagram) {
    // listen for new data packet
    moteClimate.listenFor();

    // Verify that packet has temperature data
    if (moteClimate.getReadDataByte() == dataIdAsByte) {
        // For troubleshooting only
        System.out.println("Temperature data received.");

        // Prepare data to send to external system via serial comm.
        moteClimate.packageTempDataToSendToClimateControl(
            moteClimate.getReadDataUTF(),
            moteClimate.getReadDataDouble());

        // Push new data to external system
        moteClimate.sendDataToExternalSystem();
    }
}

// end of while loop

} // end run()

} // end class

Listing C.36: Climate Interface Thread
C.5 Code Listings for Interface Mote

C.5.10 LightingInterfaceThread.java

/∗ ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗
∗ Author : Severin Willisch
∗ email : d9840087@mail.connect.usq.edu.au
∗ Student No. : 0039840087
∗ Purpose : Project for Semester 1 & 2, 2009
∗ Purpose : 09−036 Programming Sun SPOTs in JAVA
∗ ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗
∗

package org.sunspotworld;

class LightingInterfaceThread implements Runnable {
    // Instance variables
    private String systemInterface;
    private Thread t;
    private int hostPortNumber;
    private byte interfaceIdAsByte;
    private byte dataIdAsByte = 2; // identifies light data
    private boolean listenForNewDatagram = true;

    LightingInterfaceThread(String threadname, int portNo, byte moteType) {
        systemInterface = threadname;
        t = new Thread(this, systemInterface);
        hostPortNumber = portNo;
        interfaceIdAsByte = moteType;
        System.out.println("New thread: " + t);
        t.start();
    }

    // methods to access instance variables, setter methods
    void setThread(Thread setThread) {
        t = setThread;
    }

    void setListenForNewDatagram(boolean lisForDgs) {
        listenForNewDatagram = lisForDgs;
    }

    // methods to access instance variables, getter methods
    Thread getThread() {

return t;
}

boolean getListenForNewDatagram() {
    return listenForNewDatagram;
}

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
public void run() {
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // For trouble shooting only
    //
    System.out.println("Start interface thread: " + systemInterface + " " + interfaceIdAsByte);
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // start a new light interface mote
    //
    LightInterfaceMote moteLight = new LightInterfaceMote();
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // get and push IEEE address to terminal
    //
    moteLight.getPushAddress();
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // Initial set up for communication
    //
    moteLight.initInterfaceMoteCommunication();
    moteLight.setPortNumber(hostPortNumber);
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // Open up a broadcast connection
    //
    moteLight.tryToOpenBroadcastConnection();
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // Initial set up to communicate with external system
    //
    moteLight.initExternalCommunication();
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // For trouble shooting only
    //
    System.out.println("Start listening for datagrams on port: " + hostPortNumber);
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

while (listenForNewDatagram) {
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // listen for new data packet
    //
    moteLight.listenFor();
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// Verify that packet has light data
if (moteLight.getReadDataByte() == dataIdAsByte) {
  // For trouble shooting only
  System.out.println("Light data received.");
  //
  // Prepare data to send to external system via serial comm.
  moteLight.packageLightDataToSendToLightControl(
      moteLight.getReadDataUTF(),
      moteLight.getReadDataInt());
  //
  // Push new data to external system
  moteLight.sendDataToExternalSystem();
  //
} // end of if packet has light data
//
} // end of while loop
}
} // end run()
}
} // end class
package org.sunspotworld;

class AlarmInterfaceThread implements Runnable {

    // Instance variables
    private String systemInterface;
    private Thread t;
    private int hostPortNumber;
    private byte interfaceIdAsByte;
    private byte dataIdAsByte = 4; // identifies temp data
    private boolean listenForNewDatagram = true;

    AlarmInterfaceThread(String threadname, int portNo, byte moteType) {
        systemInterface = threadname;
        t = new Thread(this, systemInterface);
        hostPortNumber = portNo;
        interfaceIdAsByte = moteType;
        System.out.println("New thread: " + t);
        t.start();
    } // end constructor

    void setThread(Thread setThread) {
        t = setThread;
    }

    void setListenForNewDatagram(boolean lisForDgs) {
        listenForNewDatagram = lisForDgs;
    }

    Thread getThread() {
        return t;
    }

    boolean getListenForNewDatagram() {
        return listenForNewDatagram;
    }

    public void run() {
        // For troubleshooting only
        System.out.println("Start interface thread: " +
                systemInterface + " + " + interfaceIdAsByte);

        AlarmInterfaceMote moteAlarm = new AlarmInterfaceMote();
    }
}
// get and push IEEE address
moteAlarm.getPushAddress();

// Initial setup for communication
moteAlarm.initInterfaceMoteCommunication();
moteAlarm.setPortNumber(hostPortNumber);

// Open up a broadcast connection
moteAlarm.tryToOpenBroadcastConnection();

// Initial setup to interface with external system
moteAlarm.initExternalCommunication();

// For troubleshooting only
System.out.println("Start listening for datagrams on port: " + hostPortNumber);

while (listenForNewDatagram) {
    // listen for new data packet
    moteAlarm.listenFor();

    // Verify that packet has alarm data
    if (moteAlarm.getReadDataByte() == dataIdAsByte) {
        // For troubleshooting only
        System.out.println("Alarm data received.");

        // Prepare data to send to external system via serial comm.
        moteAlarm.packageAlarmDataToSendToAlarmSystem(
            moteAlarm.getReadDataUTF(),
            moteAlarm.getReadDataInt());

        // Push new data to external system
C.5 Code Listings for Interface Mote

C.5.12 CarbMonInterfaceThread.java

```java
package org.sunspotworld;

class CarbMonInterfaceThread implements Runnable {

    // Instance variables
    private String systemInterface;
    private Thread t;
    private int hostPortNumber;
    private byte interfaceIdAsByte;
    private byte dataIdAsByte = 5; // identifies CarMon data
    private boolean listenForNewDatagram = true;

    CarbMonInterfaceThread(String threadname, int portNo, byte moteType) {
        systemInterface = threadname;
        t = new Thread(this, systemInterface);
        hostPortNumber = portNo;
        interfaceIdAsByte = moteType;
        System.out.println("New thread: "+t);
    }
}
```

Listing C.38: Alarm Interface Thread
C.5 Code Listings for Interface Mote

void start() {
} // end constructor

// methods to access instance variables, setter methods
void setThread(Thread setThread) {
    t = setThread;
}

void setListenForNewDatagram(boolean lisForDgs) {
    listenForNewDatagram = lisForDgs;
}

// methods to access instance variables, getter methods
Thread getThread() {
    return t;
}

boolean getListenForNewDatagram() {
    return listenForNewDatagram;
}

public void run() {
    // For troubleshooting only
    System.out.println("Start interface thread: " +
                      systemInterface + " " + interfaceIdAsByte);

    // start a new carbon mono interface mote
    CarbMonInterfaceMote moteCarbMon = new CarbMonInterfaceMote();

    // get and push IEEE address
    moteCarbMon.getPushAddress();

    // Initial set up for communication
    moteCarbMon.initInterfaceMoteCommunication();
    moteCarbMon.setPortNumber(hostPortNumber);

    // Open up a broadcast connection
    moteCarbMon.tryToOpenBroadcastConnection();

// Initial set up to interface with external system
moteCarbMon.initExternalCommunication();

// For troubleshooting only
System.out.println("Start listening for datagrams on port: " + hostPortNumber);

while (listenForNewDatagram) {
    // listen for new data packet
    moteCarbMon.listenFor();

    // Verify that packet has carbon mono data
    if (moteCarbMon.getReadDataByte() == dataIdAsByte) {
        // For troubleshooting only
        System.out.println("Carbon mono data received.");

        // Prepare data to send to external system via serial comm.
        moteCarbMon.packageCarbonMonoDataToSendToCarbMonSystem(
            moteCarbMon.getReadDataUTF(),
            moteCarbMon.getReadDataDouble());

        // Push new data to external system
        moteCarbMon.sendDataToExternalSystem();
    }
}

} // end of while loop

} // end run()
}
// end class

Listing C.39: Carbon Monoxide Thread
C.5 Code Listings for Interface Mote

C.5.13 DoorInterfaceThread.java

```java
package org.sunspotworld;

class DoorInterfaceThread implements Runnable{
    // Instance variables
    private String systemInterface;
    private Thread t;
    private int hostPortNumber;
    private long unlockTime;
    private byte interfaceIdAsByte;
    private byte dataIdAsByte = 16; // identifies unlock door
    private boolean listenForNewDatagram = true;

    DoorInterfaceThread(String threadname, int portNo, byte moteType,
            long doorTime){
        systemInterface = threadname;
        t = new Thread(this, systemInterface);
        hostPortNumber = portNo;
        interfaceIdAsByte = moteType;
        unlockTime = doorTime;
        System.out.println("New thread: " + t);
        t.start();
    }

    void setThread(Thread setThread) {
        t = setThread;
    }

    void setListenForNewDatagram(boolean lisForDgs) {
        listenForNewDatagram = lisForDgs;
    }
}
```

/*
** Author: Severin Willisch
** email: d9840087@mail.connect.usq.edu.au
** Student No.: 0039840087
** Purpose: Project for Semester 1 & 2, 2009
** Purpose: 09−036 Programming Sun SPOTs in JAVA
*/

/*
** Purpose: This class runs if the MANIFEST.MF file has door ...
** Purpose: interface as a requirement.
** Inputs: External System Type,
** port used by host.
** interface type as a byte
*/
Thread getThread() {
    return t;
}

boolean getListenForNewDatagram() {
    return listenForNewDatagram;
}

public void run() {
    // For troubleshooting only
    System.out.println("Start interface thread: " + systemInterface + " " + interfaceIdAsByte);
    //
    // start a new door interface mote
    DoorInterfaceMote moteDoor = new DoorInterfaceMote();
    //
    // get and push IEEE address
    moteDoor.getPushAddress();
    //
    // Initial setup for communication
    moteDoor.initInterfaceMoteCommunication();
    moteDoor.setPortNumber(hostPortNumber);
    //
    // Open up a broadcast connection
    moteDoor.tryToOpenBroadcastConnection();
    //
    // Initial setup to interface with external system
    moteDoor.initDoorInterface();
    //
    // For troubleshooting only
    System.out.println("Start listening for datagrams on port: " + hostPortNumber);
    //
    while (listenForNewDatagram) {
        // listen for new data packet
C.5.14 SendStatus.java

package org.sunspotworld;

import com.sun.spot.util.IEEEAddress;

class SendStatus implements Runnable{

    // Instance variables

}
private String systemInterface;
private Thread t;
private int portNumberSend;
private int portNumberListen;
private byte statusIdentification;
private String hostAddress;

private boolean extSystemOK = false;
private boolean hostAddressNotKnown = true;

SendStatus(String threadname, int portSend, int portList, 
byte byteId)
{
    systemInterface = threadname;
    t = new Thread(this, systemInterface);
    portNumberSend = portSend;
    portNumberListen = portList;
    statusIdentification = byteId;
    System.out.println("New thread: " + t);
    t.start();
} // end constructor

// methods to access instance variables, setter methods

void setThread(Thread setThread) {
    t = setThread;
} //

// methods to access instance variables, getter methods

Thread getThread() {
    return t;
} //

public void run() {
    // For troubleshooting only
    System.out.println("Start external system status thread: " + systemInterface);
    System.out.println("Listening on port: " + portNumberListen);
    //
    // Create object to send status to host
    //
    InterfaceMote listenForHost = new InterfaceMote();
    //
    // Initial setup for communication (listening)
    //
    listenForHost.initInterfaceMoteCommunication();
    listenForHost.setPortNumber(portNumberListen);
// Open up a broadcast connection to listen for host
listenForHost.tryToOpenBroadcastConnection();

// Wait for host address
while (hostAddressNotKnown){
    listenForHost.listenFor();
    if(listenForHost.getReadDataByte() == 99){
        hostAddress = IEEEAddress.toDottedHex(listenForHost.getReadDataLong());
    }
    System.out.println("Host address received: " + hostAddress);
    hostAddressNotKnown = false;
    try {
        Thread.sleep(2000);
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
} // end of WHILE

// Create object to send status to host
RadiogramCommClass sendExternalSystemStatus = new RadiogramCommClass();

// Initial setup for communication (sending)
sendExternalSystemStatus.initRadiogramCom();
sendExternalSystemStatus.setPortNumber(portNumberSend);
sendExternalSystemStatus.setLocation(systemInterface);
sendExternalSystemStatus.setHostAddress(hostAddress);
sendExternalSystemStatus.setWriteDataByte(statusIdentification);

// Open up a radiogram connection to host
sendExternalSystemStatus.tryToOpenRadiogramConnection();
Listing C.41: Send External System Status Thread

```java
// Initial setup to verify status of external system
sendExternalSystemStatus.initExternalSystemStatus();

while (true) {
    // check status of external system
    extSystemOK = sendExternalSystemStatus.errorExternalSystem();
    sendExternalSystemStatus.setWriteBoolean(extSystemOK);

    // send status to host
    sendExternalSystemStatus.packageAndSendRadiogram();

    try {
        Thread.sleep(30000);
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
}
```

Listing C.41: Send External System Status Thread
C.6 Code Listings for Host

C.6.1 CommClassHost.java

```java
package org.sunspotworld;

import com.sun.spot.i2me.radiogram.RadiogramConnection;
import java.io.IOException;
import java.text.DateFormat;
import java.util.Date;
import javax.microedition.io.Connector;
import javax.microedition.io.Datagram;

class CommClassHost {
    // Instance variables
    private RadiogramConnection rCon;
    private Datagram dg;
    private String location;
    private int portNumber;
    private long writeDataLong = 0;
    private byte writeDataByte = 0;
    private int writeDataInt = 0;
    private double writeDataDouble = 0;
    private boolean writeDataBool = false;
    private long readDataLong = 0;
    private byte readDataByte = 0;
    private String readDataUTF = null;
    private int readDataInt = 0;
    private double readDataDouble = 0;
    private boolean readDataBool = false;

    DateFormat fmt = DateFormat.getTimeInstance();
    long timeOfReading = 0;
    long nowOnHost = 0L;
    long delaySampleToTerminal = 0L;
```
// methods to access instance variables, setter methods

void setLocation(String setLoc) { location = setLoc; }

void setPortNumber(int setPort) { portNumber = setPort; }

void setWriteDataLong(long setDataLong) { writeDataLong = setDataLong; }

void setWriteDataByte(byte setDataByte) { writeDataByte = setDataByte; }

void setWriteDataInt(int setDataInt) { writeDataInt = setDataInt; }

void setWriteDataDouble(double setDataDouble) { writeDataDouble = setDataDouble; }

void setWriteDataBool(boolean setDataBool) { writeDataBool = setDataBool; }

// methods to access instance variables, getter methods

String getLocation() { return location; }

int getPortNumber() { return portNumber; }

long getWriteDataLong() { return writeDataLong; }

byte getWriteDataByte() { return writeDataByte; }

int getWriteDataInt() { return writeDataInt; }

double getWriteDataDouble() { return writeDataDouble; }

long getReadDataLong() { return readDataLong; }

byte getReadDataByte() { return readDataByte; }

String getReadDataUTF() { return readDataUTF; }

int getReadDataInt() { return readDataInt; }

double getReadDataDouble() { return readDataDouble; }

boolean getReadDataBool() { return readDataBool; }

// All other methods

void initBroadcastCom() {
    // Initial set up for communication
    rCon = null;
    dg = null;
}
void tryToOpenBroadcastConnection(boolean broadRequired) {
    try {
        if (broadRequired) {
            // Open up a broadcast connection.
            rCon = (RadiogramConnection) Connector.open("radiogram://broadcast:" + portNumber);
        } else {
            // Open up a unicast connection.
            rCon = (RadiogramConnection) Connector.open("radiogram://:" + portNumber);
        }
        dg = rCon.newDatagram(rCon.getMaximumLength());
        //
        // For trouble shooting only
        //
        System.out.println("Broadcast is open.");
    } catch (IOException ex) {
        ex.printStackTrace();
    }
} // end tryToOpenBroadcastConnection()

void packageAndSendRadiogram() {
    //
    // package data into datagram and send
    //
    dg.reset();
    try {
        dg.writeLong(writeDataLong);
        dg.writeByte(writeDataByte);
        dg.writeUTF(location);
        dg.writeInt(writeDataInt);
        dg.writeDouble(writeDataDouble);
        dg.writeBoolean(writeDataBool);
        rCon.send(dg);
        //
        // For trouble shooting only
        //
        System.out.println("Data packet sent.");
    } catch (IOException ex) {
        ex.printStackTrace();
    }
} // end packageAndSendRadiogram()
```java
void listenFor(){
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // listen for new datagram data
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    try {
        rCon.receive(dg);
        readDataLong = dg.readLong();
        readDataByte = dg.readByte();
        readDataUTF = dg.readUTF();
        readDataInt = dg.readInt();
        readDataDouble = dg.readDouble();
        readDataBool = dg.readBoolean();
        // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
        // For trouble shooting only
        // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
        System.out.println("Data packet received: " + readDataByte);
        // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    } catch (IOException ex) {
        ex.printStackTrace();
    } // end try / catch
} // end listenFor()

void pushTimeAndLocationToTerminal(boolean timeDelay){
    nowOnHost = System.currentTimeMillis();
    System.out.print("Time on Host: " +
        fmt.format(new Date(nowOnHost)) + "");
    if(timeDelay){
        delaySampleToTerminal = nowOnHost - readDataLong;
        System.out.print(delaySampleToTerminal + " ms ");
        System.out.print(fmt.format(new Date(readDataLong)));
    } System.out.print("Location: " + readDataUTF + "\t");
} // end of pushTimeAndLocationToTerminal()

void identifyMessageAndPushToTerminal(int messageType){
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // Identify the data in the packets
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    switch (messageType) {
        case 1: System.out.println("Temperature (Celsius): " +
            readDataDouble); break;
        case 2: System.out.println("Light (lux): " + readDataInt); break;
        case 3: System.out.println("Battery Capacity (%): " +
            readDataInt); break;
        case 4: if (readDataInt == 1) {
            System.out.println("\tAlarm activated");
        } else if (readDataInt == 2) {
```

System.out.println("\tNo alarm");
} else {
    System.out.println("\tProblem, check SPOT");
    break;
    case 5:
    System.out.println("Carbon Monoxide (Volt): " + readDataDouble);
    break;
    case 111:
    case 112:
    case 113:
    case 114:
    case 115:
    case 116:
    case 117:
    case 118:
    case 119:
        if (readDataBool == true) {
            System.out.println(" Status -> OK");
        } else {
            System.out.println(" Status -> not OK");
        }
    } // end switch
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    } // end of identifyMessageAndPushToTerminal()
} // end CollectorMote class

C.6.2 IdentMessByByteOnHost.java

Listing C.42: Host Functions

C.6 Code Listings for Host

// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

package org.sunspotworld;

public class IdentMessByByteOnHost {

    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    // Instance variables

    /* **************************************************************
    * Author          : Severin Willisch
    * email            : d9840087@mail.connect.usq.edu.au
    * Student No.      : 0039840087
    * Purpose          : Project for Semester 1 & 2, 2009
    * Purpose          : 09-036 Programming Sun SPOTs in JAVA
    * **************************************************************
    */

    /* **************************************************************
    * Purpose          : This is the main class for the application used ...
    * Purpose          : during the test beds. The class launches 3 threads.
    * Purpose          : Once threads are launched this class waits until ...
    * Purpose          : all threads are finished. This class runs on the host.
    * **************************************************************
    */
private static final int HOST_PORT_SEND = 66;
private static final int HOST_PORT_LISTEN_DATA = 67;
private static final int HOST_PORT_LISTEN_STATUS = 68;

public static void main(String[] args) {
    // Set the host up to identify information in data packet based.
    BroadcastHostAddress myHostAddress = new BroadcastHostAddress("BroadcastHostAddress", HOST_PORT_SEND);
    HostIdentifyByByte enviroData = new HostIdentifyByByte("EnvironmentalData", HOST_PORT_LISTEN_DATA);
    HostIdentifyByByte statusReport = new HostIdentifyByByte("ExtSysData", HOST_PORT_LISTEN_STATUS);
    try {
        // For trouble shooting only
        System.out.println("Wait for thread to finish.");
        enviroData.t.join();
        statusReport.t.join();
        myHostAddress.t.join();
    } catch (InterruptedException e) {
        System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
}

Listing C.43: Main Thread on the Host

C.6.3 HostIdentifyByByte.java

/* ***************************************************************************/
/* Author: Severin Willisch */
/* email: d9840087@mail.connect.usq.edu.au */
/* Student No.: 0039840087 */
/* Purpose: Project for Semester 1 & 2, 2009 */
/* Purpose: 09–036 Programming Sun SPOTs in JAVA */
/* ***************************************************************************/
/* ***************************************************************************/
/* Purpose: This class is used to establish if data received ... */
/* Purpose: by the host is a status update or environmental data. */
/* ***************************************************************************/
package org.sunspotworld;

import com.sun.spot.io.j2me.radiogram.*;
import javax.microedition.io.*;

class HostIdentifyByByte implements Runnable {

    // Instance variables
    RadiogramConnection rCon = null;
    Datagram dg = null;
    String name; // name of thread
    Thread t;
    int portNumber;
    byte byteData = 0;
    double doubleData = 0.0;
    int intData = 0;
    String addr = null;
    String location = null;
    int messageIdentifier = 0;

    HostIdentifyByByte(String threadname, int portNo) {
        name = threadname;
        t = new Thread(this, name);
        portNumber = portNo;
        System.out.println("New thread: " + t);
        t.start();
    }

    public void run() {
        // For troubleshooting only
        System.out.println("Start new " + name + " thread on " + portNumber);

        // Start a new listener on host
        CommClassHost hostListener = new CommClassHost();

        // Initial set up for communication
        hostListener.initBroadcastCom();
        hostListener.setPortNumber(portNumber);
// Open up a unicast (broadcast) connection
hostListener.tryToOpenBroadcastConnection(false);

// Main data collection loop
while (true) {
    // listen for new data packet
    hostListener.listenFor();
    
    // Set value for switch case
    messageIdentifier = hostListener.getReadDataByte();
    
    if (portNumber == 67) {
        // push time, delay and location to terminal for packets ... from collector SPOTs
        hostListener.pushTimeAndLocationToTerminal(true); 
        
        // Identify message based on message identifier, data
        hostListener.identifyMessageAndPushToTerminal(messageIdentifier); 
    } else if (portNumber == 68) {
        // push time, delay and location to terminal for packets ... from interface hosts
        hostListener.pushTimeAndLocationToTerminal(false); 
        
        // Identify message based on message identifier
        hostListener.identifyMessageAndPushToTerminal(messageIdentifier); 
    }
}

} // end of ELSE IF

} // end while

} // end run()
C.6 Code Listings for Host

C.6.4 BroadcastHostAddress.java

```java
//***************************************************************
* Author : Severin Willisch
* email : d9840087@mail.connect.usq.edu.au
* Student No. : 0039840087
* Purpose : Project for Semester 1 & 2, 2009
* Purpose : 09–036 Programming Sun SPOTs in JAVA
****************************************************************

package org.sunspotworld;

import com.sun.spot.peripheral.radio.RadioFactory;
import com.sun.spot.util.IEEEAddress;
import java.util.logging.Level;
import java.util.logging.Logger;

class BroadcastHostAddress implements Runnable {
    String name; // name of thread
    Thread t;
    int portNumber;

    BroadcastHostAddress(String threadname, int portNo) {
        name = threadname;
        t = new Thread(this, name);
        portNumber = portNo;
        System.out.println("New thread: " + t);
        t.start();
    }

    // Instance variables
    //
    long hostAddress = 0;
    byte hostId = 99;

    // Get and push host address to terminal
    public void run() {
        // Get host address and push to terminal
        hostAddress = RadioFactory.getRadioPolicyManager().getIEEEAddress();
    }
}
```

Listing C.44: Listening Class on Host
System.out.println("Our radio address (long)= " + 
    IEEEAddress.toDottedHex(hostAddress));

// Initial setup for communication
//
CommClassHost newBroadcast = new CommClassHost();
newBroadcast.initBroadcastCom();
newBroadcast.setPortNumber(portNumber);
newBroadcast.setLocation("Host");
newBroadcast.setWriteDataLong(hostAddress);
newBroadcast.setWriteDataByte(hostId); // id of host

// Open up a broadcast connection
//
newBroadcast.tryToOpenBroadcastConnection(true);

while (true){
    newBroadcast.packageAndSendRadiogram();
    // For trouble shooting only
    //
    System.out.println("Host broadcasted address");
    try {
        Thread.sleep(30000);
    } catch (InterruptedException ex) {
        Logger.getLogger(BroadcastHostAddress.class.getName()).
            log(Level.SEVERE, null, ex);
    }
} // end of WHILE

} // end of run()

} // end of class

Listing C.45: Broadcast Host Address Class
Appendix D

Screen Captures
D.1 Example Implementation Code

![Screen Capture Test Script Output](image-url)

Figure D.1: Screen Capture Test Script Output
D.1 Example Implementation Code

Figure D.2: Screen Capture Test Run Host Output
Figure D.3: Screen Capture Test Run Virtual SPOT Output
Figure D.4: Screen Capture Solarium with Virtual SPOT
D.2 Screen Captures HAA Test Bed

Figure D.5: Screen Capture HAA Test Bed with Six Virtual SPOT
Appendix E

Building Layout
Figure E.1: Home Layout