

Battelle

The Business of Innovation

Smart Phone Control of Advanced Sensor Systems

Michigan State University College of Engineering

ECE480

Design Team 4

December 9th 2011

Design Team:

Jacob Sawicki – Manager

Kevin Gleason – Webmaster

Phillip Horny – Documentation Prep

Andreas Dixon – Lab Coordinator

Thamer Alajlan – Presentation Prep

Executive Summary

In the last decade Smart Phones have increased exponentially in not only popularity, but also technical design. Software of the integrated systems on these handheld devices is under constant development, but is often limited by the increasing demand for compact size. Our sponsor, Battelle Labs, has asked this design team to create a sensor monitoring and control system that can be wirelessly controlled through a smart phone. Through our design process MiWi Radio Frequency (RF) technology was chosen as the communication method. The process of integration on this level has not yet been achieved by the technological community; nonetheless, utilizing this communication method overcomes many of the limitations currently imposed by cellular technology. This design team has laid the foundation for a final product design that meets or exceeds all of Battelle's design requirements.

Acknowledgments

Team 4 would like to thank all of the following individuals involved in helping complete this design. Without them it would not have been possible.

Dr. Tongtong Li
Dr. Christopher Ball
Prof Michael Shanblatt
Dr. Jian Ren
Mark Sawicki
Mark Baynum
David Flowers
Yifeng Yang

Table of Contents

Executive Summary	1
Acknowledgments.....	1
Chapter 1 – Introduction and Background.....	3
1.1 Introduction	3
1.2 Background.....	3
Chapter 2 – Exploring the Solution Space and Selecting a Specific Approach.....	5
2.1 Design Specifications	5
2.2 FAST Diagram	6
2.3 Conceptual Designs	Error! Bookmark not defined.
2.4 Feasibility matrix.....	8
2.5 Gantt Chart and Project Schedule.....	10
Chapter 3 – Technical Description of Work Performed	12
3.1 Hardware Design	12
3.2 Software Design	21
Chapter 4 – Test Data with Proof of Functional Design.....	27
4.1 Zigbee Functional Designs	27
4.2 MiWi The device design can be split into three major sections.....	28
Chapter 5 – Final cost, schedule, summary and conclusions.....	36
5.1 Findings	36
5.2 Suggestions for future projects	37
5.3 Final Costs	37
5.4 Conclusion	38
Appendix 1 – Technical Roles, responsibilities and work accomplished.....	38
Jacob Sawicki – Manager	39
Kevin Gleason – Webmaster	40
Phillip Horny – Documentation Prep	41
Andreas Dixon – Lab Coordinator	41

Thamer Alajlan – Presentation Prep.....	42
Appendix 2 – Literature and website references.....	44
Appendix 3 – Detailed technical attachments.....	45

Chapter 1 – Introduction and Background

1.1 Introduction

Battelle Laboratories assigned Team 4 the task of using an Android based smart-phone to wirelessly control the Resource Effective Bio-Identification sensor, a product of Battelle. This sensor uses Raman Spectroscopy to determine the chemical composition of the air, sending out an alarm when a potentially harmful chemical is detected. To control this sensor, the smart-phone controller must be able to wirelessly send commands, monitor sensor status, and alert the operator if the sensor detects a problem.

Team 4’s approach to this problem is to use an external Radio Frequency transmission technology controlled through the Universal Serial Bus (USB) port on the smart-phone. Currently, there exists no such application of the USB port. This is most likely due to the smart-phone’s plethora of embedded wireless communication technologies, including cellular and Wi-Fi. To understand why Team 4 would want to create yet another form of wireless communication in a smart-phone, one must understand the limitations of the current forms of smart-phone wireless communication.

1.2 Background

Network Limitations

Many current technologies exist that enable smart-phones to use a cellular 3G or Wi-Fi connection to control a multitude of end devices. From consumer computer software to RC cars, smart-phones can control many external technologies when part of a Wi-Fi or cellular network. However, these wireless capabilities require the use of some form of pre-created network. This requirement eliminates the use of these functionalities in areas of the world where the Internet is unavailable. For example, the Resource Effective Bio-Identification sensor from Battelle will be used to detect chemical weapons in overseas battlefields where no Wi-Fi networks exist.

Method of Solution

Team 4’s approach to controlling external devices with a smart-phone aims to overcome these problems by using external Radio Frequency transceiver technology incorporated through the micro USB port that comes standard on many recent smart-phones. This

compact, external hardware plug-in enables the phone to control devices without an internet network by sending commands out of the USB port. These commands go into a micro-controller and then through the use of an RF transmitter, are wirelessly routed to the RF receiver on the end device. This enables the user to wirelessly control his or her device in any location, removing the limiting need for an internet network.

Embedded User Interface Limitations

Many devices that will be wirelessly controlled by Team 4's smart-phone controller can already be controlled with their own wireless controllers. Going back to the RC car example, this product does not necessarily need to be controlled by a smart-phone through Wi-Fi or even Team 4's RF device. The car already has its own remote control. However, creating remote controls even for a standard RC car requires dedicating hardware such as accelerometers and potentiometers and the software needed to control these systems into the controller; as well as the wireless transmission technology.

Method of Solution

Team 4's design simplifies this process by removing the need for such hardware devices. Smart-phones already have many of them built in - including the touch-screen interface that can serve multiple controller purposes. Also, smart-phones have the power to perform data processing such as Fast Fourier Transforms, meaning remotely controlling a system and interpreting the data from it does not require external processors dedicated to these high-level functions. Overall, this means that the design time to create a wireless controller can be cut down by removing the need to hard-code interface hardware and embedded processors.

Also of note is the Bluetooth wireless transmission technology that is embedded in many smart-phones. Bluetooth has limitations that yield it as an impractical choice for a design solution as well. Range is often limited along with data rates when using this communication method. This is obviously a concern when creating a monitoring system for a chemical sensor in a battlefield scenario.

Summary of Limitation Solutions

- Removes smart-phone's need for internet networks to wirelessly control devices
- Cuts down design time of wireless controllers by removing the need to hard-code interface hardware
- High-functionality microprocessors no longer needed due to smart-phone's capability to perform data processing functions

Chapter 2 – Exploring the Solution Space and Selecting a Specific Approach

2.1 Design Specifications

To determine the project specifications Team 4 met with their sponsor from Battelle to discuss their needs. The team decided on the key specifications that need to be taken into account during the design phase. Priority is based on a 1(low) – 5(high) scale.

Power

Priority: 5

The physical location of the sensor is likely to be placed in an area where there is no permanent power source and batteries will be needed to power the sensor. The smart-phone also has a limited battery life so it is important to limit the power that the connected wireless sensor draws from the phone through the micro-USB port.

Size

Priority: 1

The sensor for which the system is being developed is already rather large so the additional hardware to wirelessly communicate between the sensor and the phone will not have a major affect. Since the cell phone needs to be mobile the additional hardware attached to the phone needs to be minimal. However, since the team is working on a proof of concept, this is not a major requirement. The design can be optimized before it is used in the field.

Range

Priority: 5

Range is one of the most important specifications. The main goal for this system is to allow for the user to monitor and control the sensor without the need to actually be at the sensor. The farther away the user can be while still being able to access the sensor, the more useful the system will be to the user.

Speed

Priority: 1

The communication between the sensor and the smart phone will be mostly simple commands and status updates. There is not a lot of data that is being transmitted so it is not a necessity to transmit at a very quick speed.

Adaptability

Priority: 5

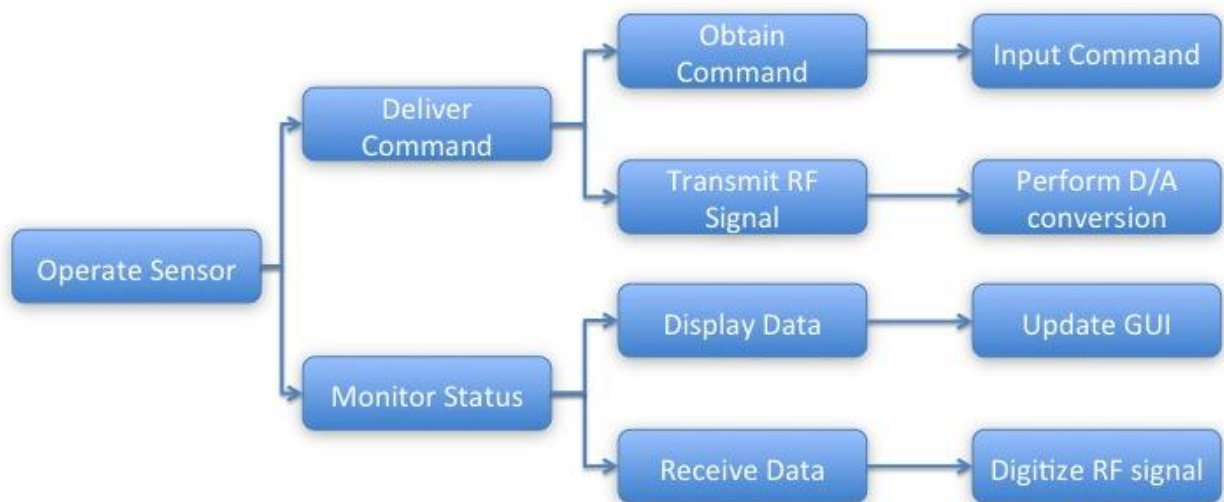
The sensors that will be monitored using this system will be in a variety of different environments. The sensors can be placed indoors or outdoors and may not have an existing network they are connected to. The team does not want to have to rely on an existing network to communicate with the sensor. The design needs to be able to move around without the need to setup an additional network like a Wi-Fi network.

Cost

Priority: 2

Battelle was generous enough to allow us to go over budget if we needed. Nonetheless, the team still strived to design a cheap solution in order to account for future production costs.

2.2 FAST Diagram

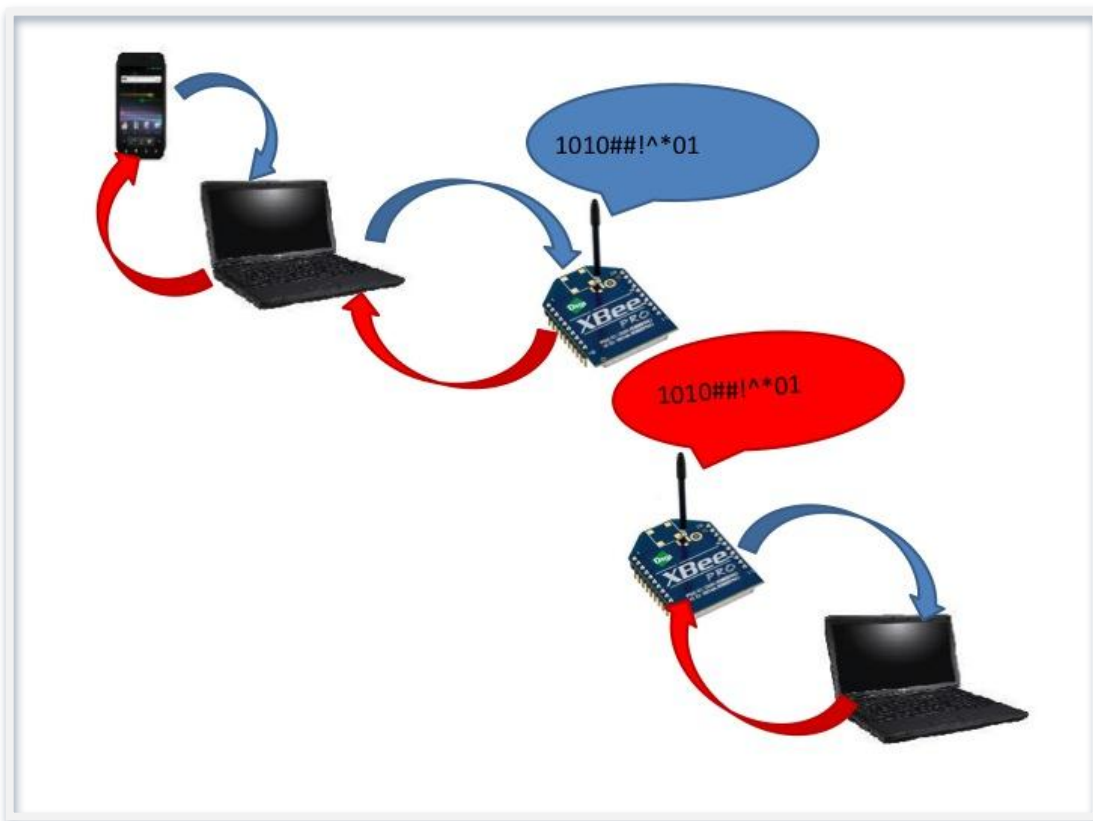


The FAST Diagram is a simple explanation of the HOW and WHY of the group's functional design. From right to left, the reader moves from block to block by asking HOW? From right to left, the reader moves from block to block by asking 'WHY?' For example, if the goal is to Operate Sensor. How to do this would be: Deliver Command and Monitor Status. To Deliver Command, the function must Obtain Command and Transmit RF Signal. To Obtain Command, the function must Input Command. The same logic can be used for moving from right to left.

2.3 Conceptual Designs

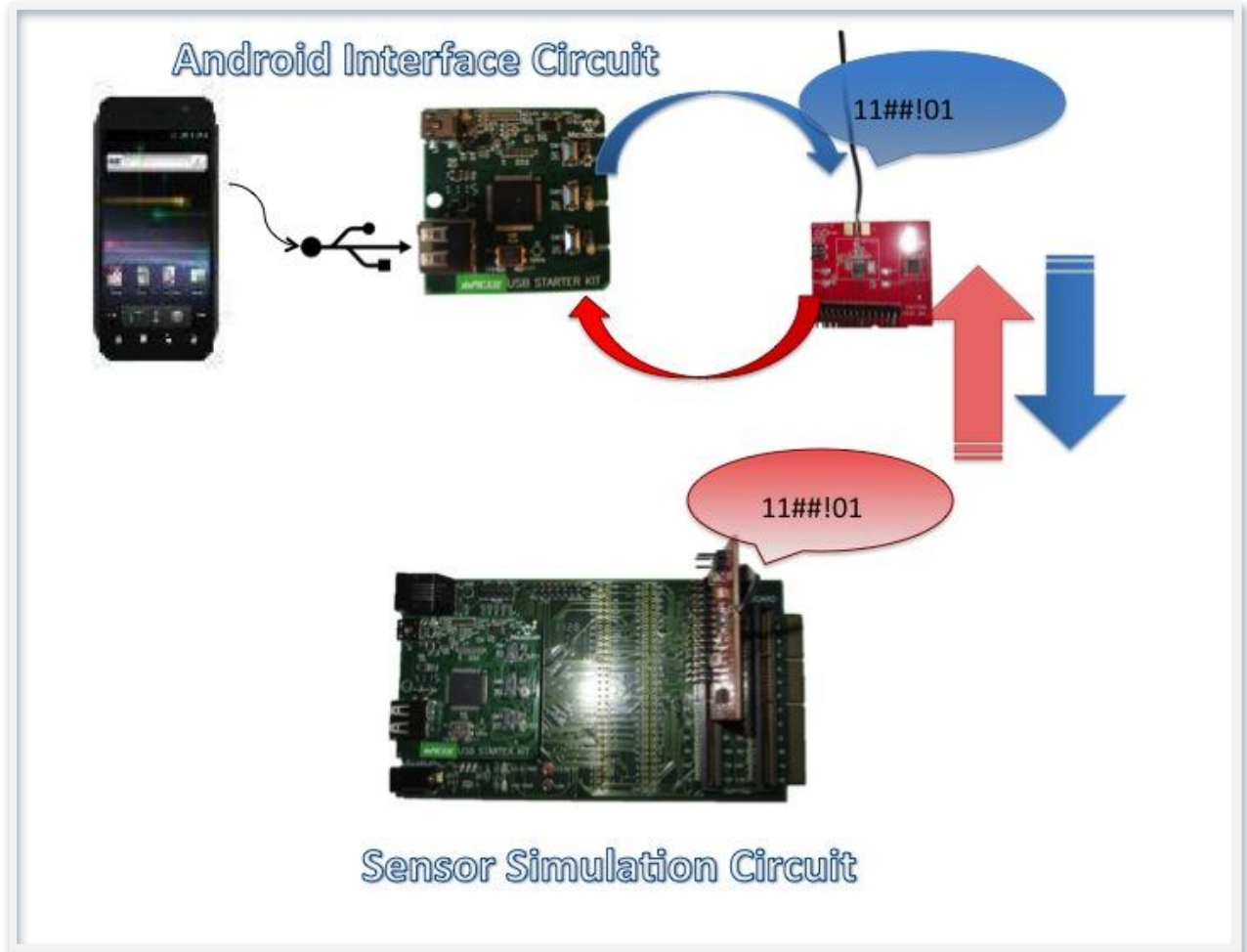
While team 4 conceptualized several designs, the following figures show the block diagrams of the two designs that were pursued. Additionally, the MiWi communication design (figure 2) was chosen as the final design.

Figure 1: Zigbee Communication Block Diagram



The Zigbee communication system design pictured in figure 1 involved the use of a computer to interface the phone to the XBee transceiver. The phone would have been connected via USB or WiFi to the computer for processing. Once the information was processed it would be passed to a terminal program on the computer to be formatted to a language the XBee transceivers could recognize. The sensor would be simulated by a written program.

Figure 2: MiWi Communication Block Diagram



The MiWi communication design shown in figure 2 uses a dsPIC33E USB starter kit to interface the Android smart-phone to the MiWi transceiver. The starter kit is connected to an I/O expansion board that provides the link to the MiWi transceiver. Communication is established wirelessly between the two transceivers and information is sent back and forth. The sensor simulation circuit contains a dsPIC33E USB starter kit, an I/O expansion board, a MRF49XA transceiver and a potentiometer that simulates the hazardous chemical level.

2.4 Feasibility matrix

	Battelle
Operational Feasibility	The purpose of Team 4's project was to develop a mechanism in which an Android smart-phone could be used as a station that receives signals from several sensors through a determined method of communication (MiWi Protocols).
Technical Feasibility	The design solution chosen by the team uses MiWi transmission. It is conceptually possible to connect an Android smart-phone to a dsPIC33E USB starter kit and control one of Microchip's MiWi compatible transceivers through the use of their PIC24 I/O expansion board. However, this has never been done. MiWi protocol stacks are designed to be a simple wireless solution for many developers and include several prewritten demos to aid the developers in writing their own program. Theoretically, the only work to be done was to configure the hardware profiles of the prewritten demos for the dsPIC33E USB starter kit and rewrite the main program for a specific function. Nevertheless, it had not yet been proven to be possible.
Economic Feasibility	The team's capital is \$500. Due to these limited resources, the team was unable to obtain the actual sensor designed by Battelle. Instead, a dsPIC33E USB starter kit with an I/O expansion board was used to simulate the sensor. Budget constraints limited our ability to expand our research and use a high method of technology that would most likely give us more detailed results.
Schedule Feasibility	The team ran into difficulties with scheduling when in the ninth week the initial design was realized to be impractical due to the necessity of a computer to interface the smart-phone to the transceiver. Once this flaw was revealed, the team was forced to search for different solutions and discovered Microchip Technology's MiWi communication technology the following week. This was a devastating disturbance to the team's schedule. Under normal design circumstances the predetermined finish date of 12/9/2011 might have delayed. However, this being a semester long design class, the finish date is fixed and cannot be delayed.

2.5 Gantt Chart and Project Schedule

The following figure is a screen shot of Team 4's Gantt chart. It depicts the team's schedule after modifying the design of their project. In week nine of the semester, the team ran into a critical issue. The XBee parts that were ordered went out of stock. This occurred after the team had been waiting for three weeks. During this time it was discovered that by using Microchip's MiWi technology, it might be possible to wirelessly control the sensor with an Android smart-phone via a USB connection to a dsPIC33E USB starter kit. The following day a representative from Microchip confirmed this possibility. Since this was a more practical solution than using a computer to interface the phone to the wireless network, the team decided to change to the MiWi design solution. The old Gantt chart is shown in figure 4 of appendix 3.

Figure3: Gantt Chart 1

		Task Mode	Task Name	Duration	Start	Finish	Predecessors
1	☒		Develop Android Application	51 days	Mon 9/26/11	Mon 12/5/11	
2	☒		Obtain smartphone from ece shop	3 days	Mon 9/26/11	Wed 9/28/11	
3	☒		Obtain the code from last semester's team and link to application development tutorial	1 day	Mon 9/26/11	Mon 9/26/11	
4	☒		Take tutorial to become familiar with developing an application for a smartphone< http://developer.android.com/resources/tutorials/hello-world.html >	1 day	Mon 10/3/11	Mon 10/3/11	3
5	☒		Become familiar with old code	5 days	Mon 10/10/11	Wed 10/19/11	3,4
6	☒		Optimize/write new code for our application	9 days	Thu 10/20/11	Fri 11/4/11	5,15
7	☒		Test code using Microchip's Basic Android Accessory Demo firmware	3 days	Thu 11/24/11	Fri 12/2/11	6,12
8	☒		Smartphone application complete	0 days	Mon 12/5/11	Mon 12/5/11	7
9	☒		Get Familiar with MiWi Technology	5 days	Mon 11/14/11	Fri 11/18/11	
10	☒		Research MiWi Technology	2 days	Mon 11/14/11	Tue 11/15/11	
11	☒		Order MiWi/Android Accessory Tool Set	4 days	Tue 11/15/11	Fri 11/18/11	10
12	☒		Download programs/compiler/application library necessary for programming dsPIC33E	3 days	Tue 11/15/11	Fri 11/18/11	10
13	☒		Develop Android Interface to Accessory	10 days	Mon 11/21/11	Fri 12/2/11	
14	☒		Download Microchip Demo Application to Android	0 days	Mon 11/21/11	Mon 11/21/11	
15	☒		Program and Run Basic Android Accessory Firmware	3 days	Tue 11/22/11	Thu 11/24/11	14
16	☒		Clean up Basic Accessory Demo source and header files	5 days	Thu 11/24/11	Wed 11/30/11	15
17	☒		Write program's main android interface program	3 days	Wed 11/30/11	Fri 12/2/11	16
18	☒		Android Interface Program Complete	0 days	Mon 12/5/11	Mon 12/5/11	17
19	☒		Develop Program for MiWi Communication	13 days	Wed 11/16/11	Fri 12/2/11	
20	☒		Read Application notes on MiMac/MiApp/MiWi	4 days	Wed 11/16/11	Mon 11/21/11	10
21	☒		Read Data sheet for MRF49XA/dsPIC33E	4 days	Tue 11/15/11	Fri 11/18/11	10
22	☒		Familiarize ourselves with MiWi Demo code for the Explorer 16 board	3 days	Mon 11/21/11	Wed 11/23/11	12
23	☒		Using schematics, remap hardware profile and related files for the dsPIC33E	4 days	Thu 11/24/11	Tue 11/29/11	22
24	☒		Debug Code	3 days	Wed 11/30/11	Fri 12/2/11	23
25	☒		MiWi Communication Program complete	0 days	Mon 12/5/11	Mon 12/5/11	24
26	☒		Implement Final Design	3 days	Mon 12/5/11	Wed 12/7/11	
27	☒		Combine Android and MiWi programs for the interface to the phone	2 days	Mon 12/5/11	Tue 12/6/11	24
28	☒		Modify MiWi Program to simulate a sensor	2 days	Wed 12/7/11	Thu 12/8/11	27
29	☒		Final Design complete	0 days	Fri 12/9/11	Fri 12/9/11	28

2.6 Budget

Final Budget			
Quantity	Description	Cost(each)	Cost(Total)
3	Microchip RF transceivers	-----	Free
3	Microchip PIC24 I/O Expansion Board	-----	Free
2	Microchip USB starter kit (dsPIC33E)	\$64.99	\$129.88
2	Enclosures for hardware components	\$24.00	\$48.00
2	Microchip USB starter kit II (PIC32)	\$55.00	\$110.00
Total Cost = \$287.88			

Chapter 3 – Technical Description of Work Performed

3.1 Hardware Design

Before a wireless transmission technology could be chosen, a trade study had to be completed to find the proper technology to meet the project needs. The following table summarizes the findings for the major technologies researched. These include Wi-Fi, Bluetooth, Zigbee, and regular Radio Frequency Transmission (PRF).

Figure 4: Hardware Comparison

	Wi-Fi	Bluetooth	ZigBee	PRF
Frequencies	2.4GHz and/or 5GHz	2.45GHz	915MHz(US) 868MHz (EU) 2.4GHz (global)	900MHz(US) 868MHz(EU) or 2.4MHz (global)
Channels	16 @ 2.4GHz 80 @ 8GHz	79	10 @ 915MHz 26 @ 2.4GHz	16 to 79 (can be customized)
Range (Indoor)	70m	Class1=1m Class2=10m Class3=100m	20m	1000m
Range (Outdoor)	160m	100m	100m	40 miles (with high-gain antenna)
Data Rate (Max)	54Mbits/s (with12Mbits/s typical)	3Mbits/s	250Kbit/s	721Kbit/s to 72Mbit/s
Transmission Scheme	DSSS	Adaptive FHSS	DSSS	FHSS or DSSS
Power Sources	Wired	Battery/ Wired	Battery/ Wired	Battery/ Wired (industrial power source compatible)
Uses	Cable replacement, large data transfer, networking	Short distance cable replacement	Monitoring and Controlling	Cable replacement, Monitoring, Controlling, Data Transfer
Max Nodes	32	8	>64,000	Variable
Stack Size	1000KB	720KB	250KB	Variable
Transmit Power (Max)				1W

*Chart obtained from http://www.bb-elec.com/bb-elec/literature/tech/Industrial%20Wireless-WP18-r0_3007.pdf

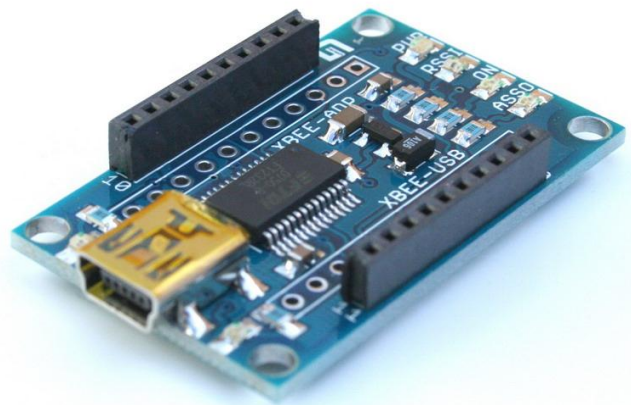
3.1.1 First Design – Zigbee

The first communication technology Team 4 decided to use for the smart-phone control of a sensor was Zigbee. This technology is made for low power and low data rate. Each XBee transceiver module has the capability to communicate with either a computer through an external USB connection, or with a microprocessor. An XBee transceiver runs at 915 MHz, 868MHz, or 2.5GHz. The hardware components for a Zigbee network that would meet our application needs are shown below.

Figure 5: XBee Transceiver 1



Figure 6: XBee to USB Adapter 1



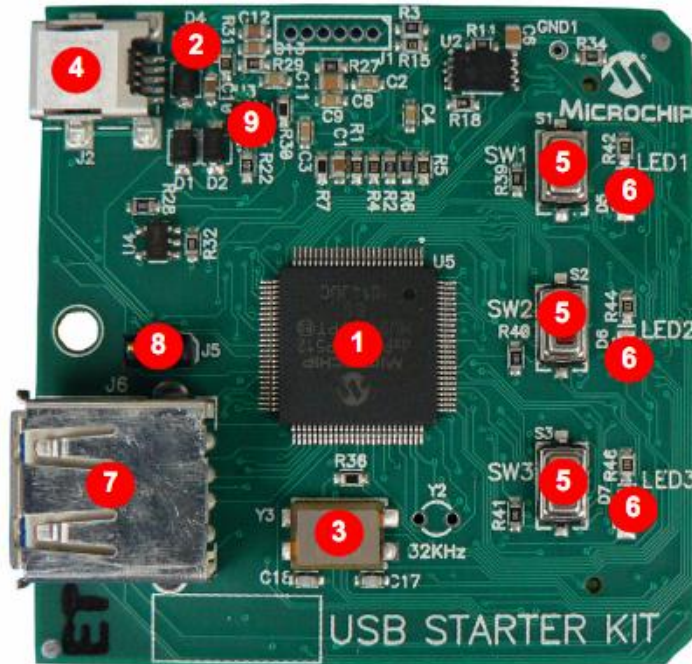
Though Zigbee satisfied the first two design requirements, connecting the XBee modules to the Android smart-phone was not a feasible task for the time allotted. More information on the Zigbee design can be found in Chapter 4.

3.1.2 Final Design – MiWi

A similar technology to Zigbee is Microchip’s MiWi Wireless Protocol. This protocol is useable solely by Microchip proprietary hardware. The hardware used by Team 4 for MiWi wireless interaction is shown below.

DsPIC33E USB Starter Kit

Figure 7: dsPIC33E USB Starter Kit Top View

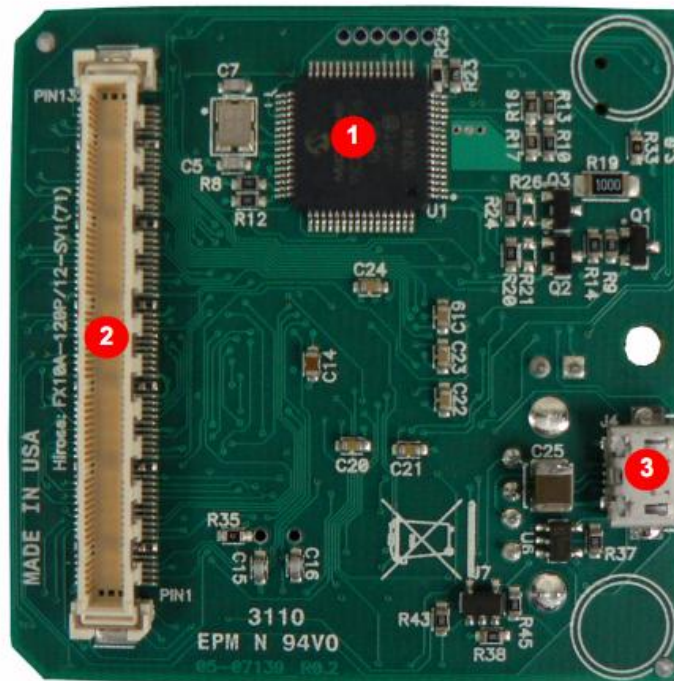


The top assembly dsPIC33E USB Starter Kit includes these key features:

1. dsPIC33EP512MU810 16-bit DSC device (dsPIC33E USB Starter Kit)
2. Green power indicator LED (D4).
3. 8 MHz crystal (Y3) for precision microcontroller clocking.
4. USB connectivity for on-board debugger communications (J2).
5. Three push button switches (SW1, SW2, and SW3) for user-defined inputs.
6. Three user-defined indicator LEDs (LED1, LED2, LED3).
7. USB Type A receptacle (J6) connectivity for dsPIC33E/PIC24E USB host-based applications.
8. HOST mode power jumper (J5).
9. Regulated +3.3V power supply for powering the starter kit via USB or an expansion board

**Figure and Text from dsPIC33E USB Starter Kit and PIC24E USB Starter Kit User's Guide, pg. 12*

Figure 8: dsPIC33E USB Starter Kit Bottom View



The bottom assembly of the DsPIC33E USB Starter Kit includes these key features:

- PIC24FJ256GB106 USB microcontroller (U1) for on-board debugging.
- Connector (J3) for various expansion boards such as the Multimedia Expansion Board (MEB) or the I/O Expansion Board.
- USB Type micro-B receptacle (J4) for USB Device connectivity for dsPIC33E/PIC24E USB device-based applications

**Figure and Information from dsPIC33E USB Starter Kit and PIC24E USB Starter Kit User's Guide, pg. 13*

Application Specific Features:

The DsPIC33E USB Starter Kit was chosen for its USB capabilities. This kit is capable of connecting to an Android based smart-phone through pre-compiled code from Microchip. The implementation of this feature minimized the time required to complete the project.

MRF49XA Wireless Transceiver

Figure 9: MRF49XA Front and Back View

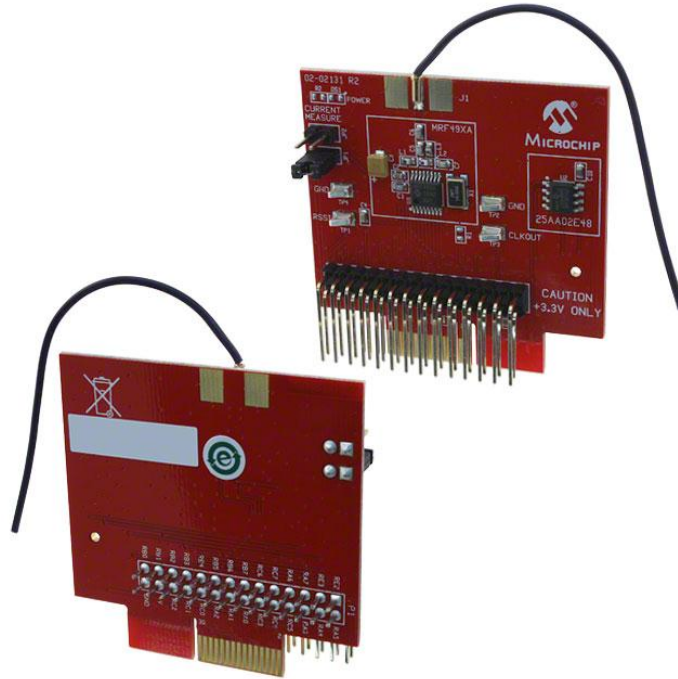


*Figure taken from <http://parts.digikey.com/1/parts/1719877-daughter-board-mrf49xa-ac164137-2.html>

The MRF49XA provides the following features:

- Fully Integrated Sub-GHz Transceiver
- Supports Proprietary Sub-GHz Wireless Protocols
- 4-Wire Serial Peripheral Interface (SPI)
- Compatible Interface
- CMOS/TTL Compatible I/Os
- Clock and Reset Signals for Microcontroller
- Integrated 10 MHz Oscillator Circuitry
- Supports Power-Saving modes
- Operating Voltage: 2.2V–3.8V
- Low-Current Consumption, Typically:
 - 11 mA in RX mode
 - 15 mA in TX mode
 - 0.3 μ A in Sleep mode
- Industrial Temperature Range
- 16-Pin TSSOP Packag

*Information from [MRF49XA Data Sheet](#)

Application Specific Features:

The MRF49XA was chosen for its Sub-GHz Transceiver capabilities and Low Power consumption capabilities. It is also already capable of being used in the MiWi Development Environment.

I/O Expansion Board

Figure 10: I/O Expansion Board 1

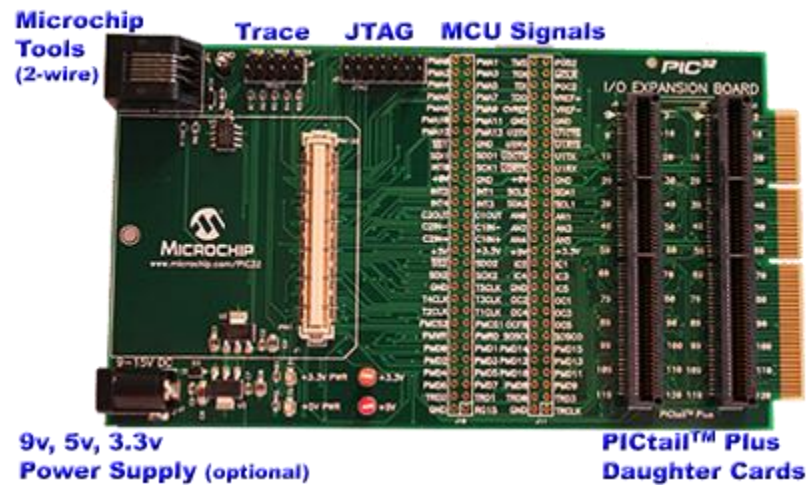


*Figure Obtained from

http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2615&dDocName=en535444

Features:

- The I/O Expansion Board is capable of connecting the MRF49XA to the DsPIC33E expansion board
- Removes need to hardwire DsPIC33E USB Starter Kit to the MRF49XA

Complete Hardware Design

Figure 11: I/O Expansion with dsPIC33E USB Starter Kit and

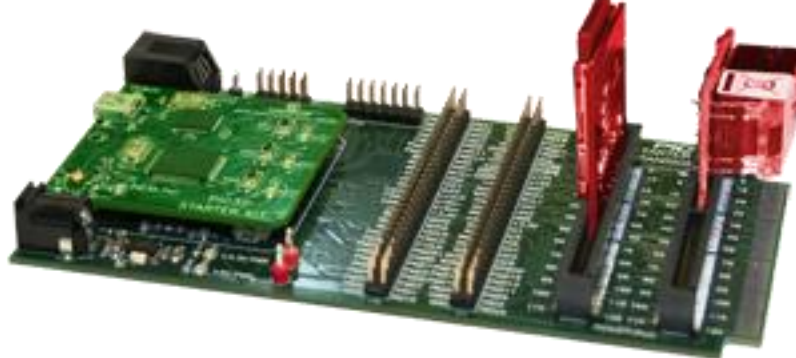


*Figure Obtained from

http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2615&dDocName=en535444

Figure 15 portrays the final hardware connections made between each separate component. The DsPIC33E USB Starter Kit is input to J1 on the I/O Expansion Board. The MRF49XA is input to SPI slot 2 in J2 on the I/O Expansion Board.

3.1.3 Enclosure Design

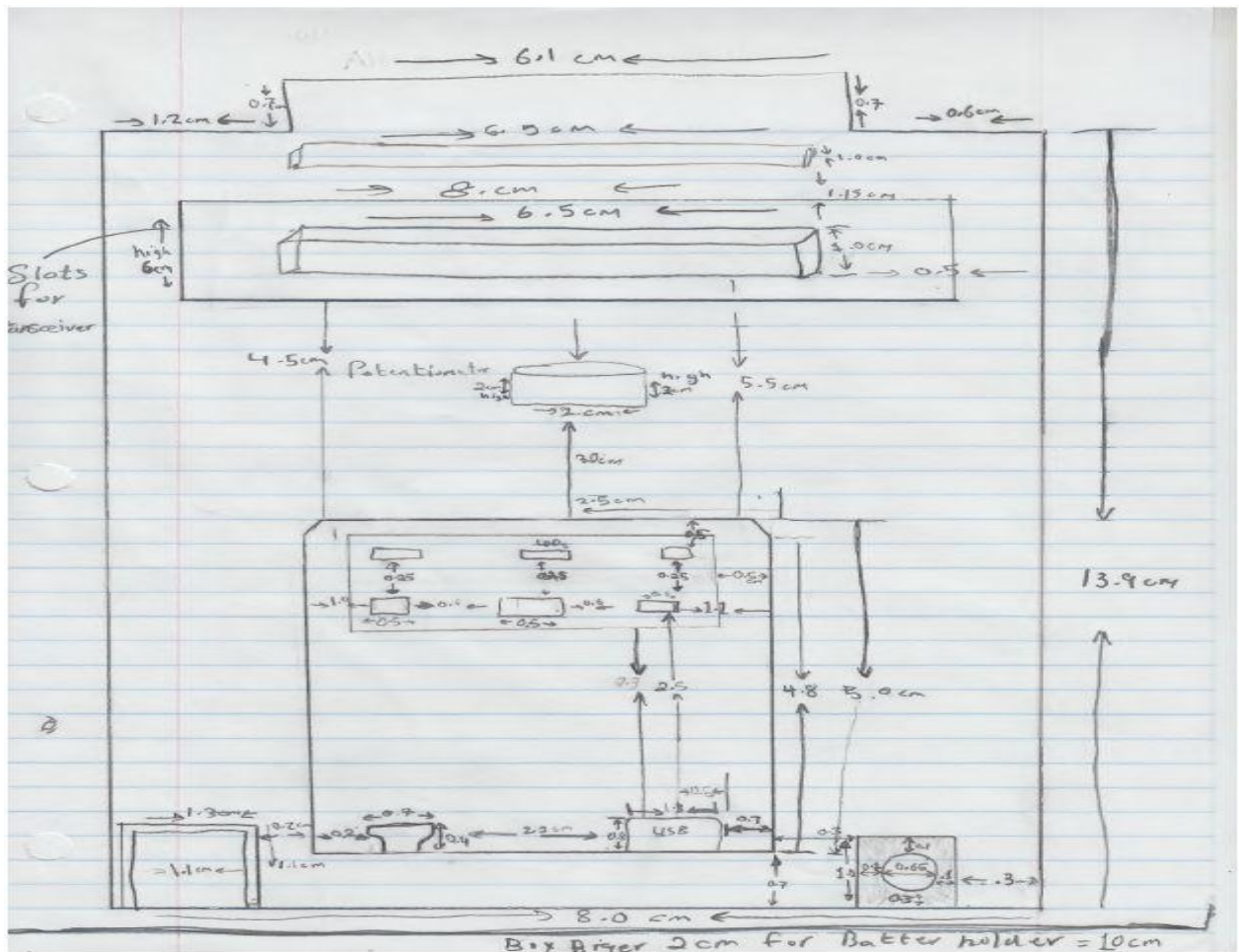
Enclosure design features:

1. Opening at type A and B USB ports as well as a designed battery holder
2. Slot for the transceiver
3. Window to look at the LED's
4. Three push buttons and a hole for potentiometer

The team used a program called AutoCAD (Computer Aided Design) which allowed them to make the enclosure design sketch in two dimensions. In order to get a matching enclosure to fit the hardware components three steps were performed.

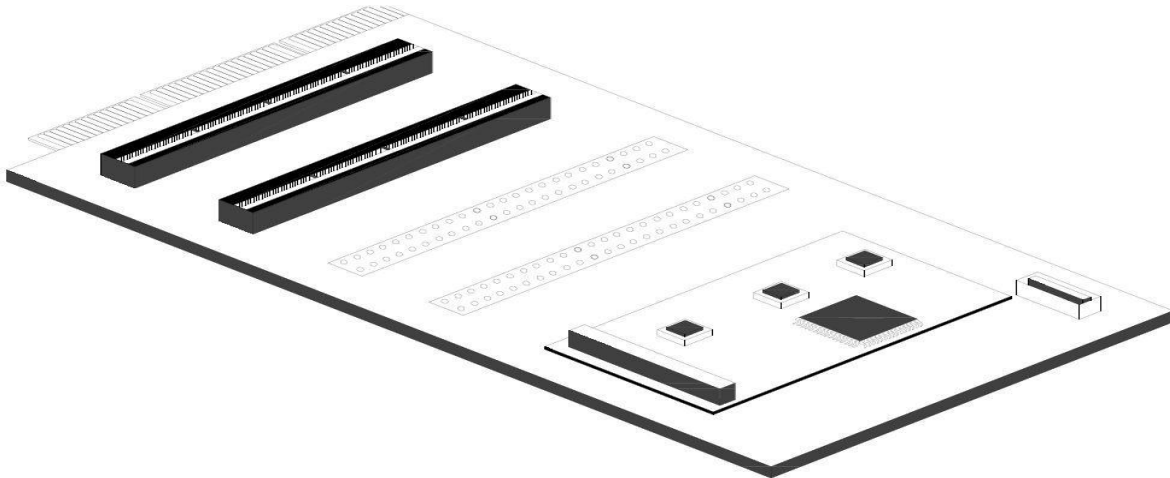
Step 1 : **Dimensions:** These were obtained using a rough sketch with the exact measurements of the dsPIC33E starter kit and I/O expansion board.

Figure 12: Dimensions Sketch



Stage 2: Convert Rough sketch to AutoCAD. 2-D drawing.

Figure 13: AutoCAD Drawing with Dimensions



Stage 3: Design the enclosure

Figure 14: Model 1

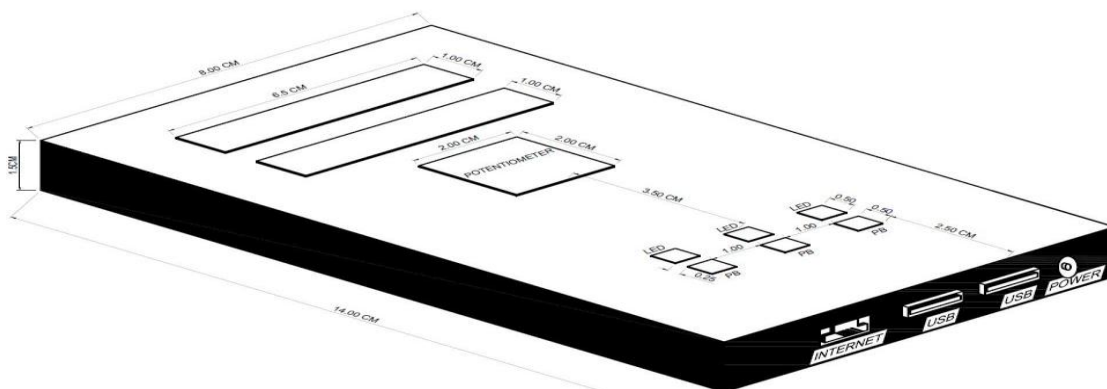


Figure 15: Model 2

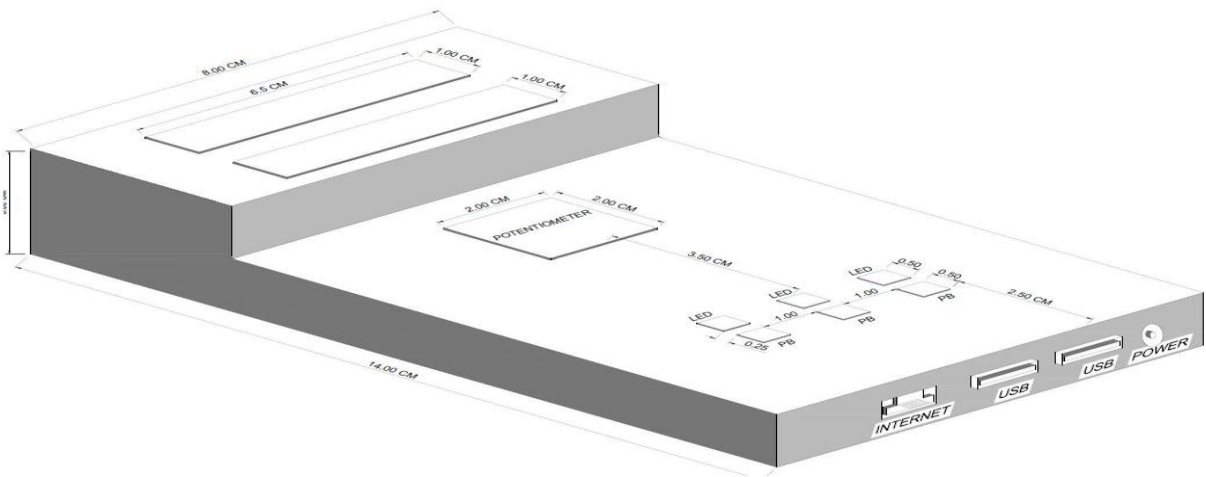
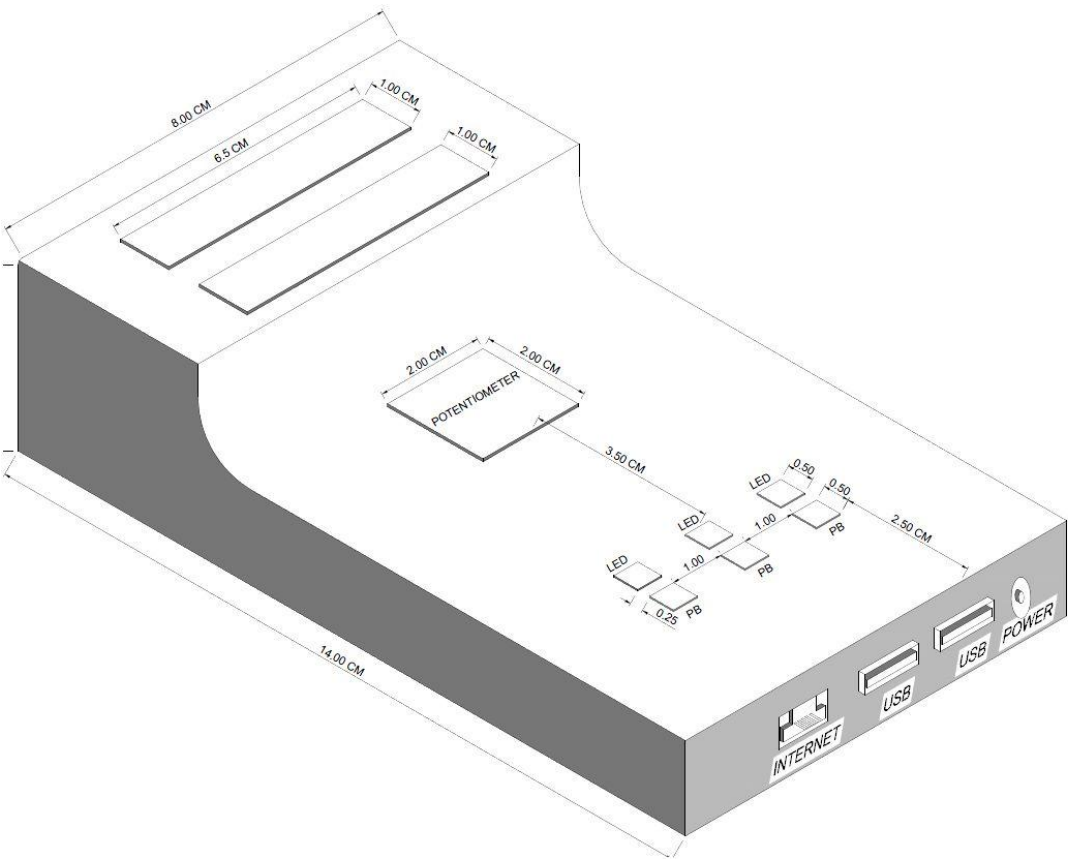


Figure 16: Model 3



3.2 Software Design

Using the hardware shown above as an integrated system required that many pieces of software be brought together. On their website, Microchip provides many free, pre-written demo program examples that aided in, but certainly did not complete, the overall design. The order the final code was developed in required a few more steps than many projects would when attempting to use Microchip's user friendly products. Microchip has intended for most consumers to reuse their demo code with the exception of added functionality in the application layer and peripheral connections. However, because of our signal path complexity and choice of microprocessor, no demos were written that matched our needs. When stepping through the path of the physical signal, code implementation from our design team can be seen all along the way.

To begin from a reasonable level of code hierarchy the group had to start with adapting the demos to the hardware setup. It should be noted that initially the hardware definitions excluded the phone connection; it was integrated into the final program in the final steps. The first signal noticeable by a user is at the phone. It has a graphics interface designed by the team for the user to interact with (see 3.2.3). The next step is for the phone to form the USB connection. This includes stacks, protocols, and hardware definitions from the phone to the microchip in order for signals to be passed (see 3.2.2). From there, the microchip kit had to recognize the I/O expansion board. In addition, all of the peripheral devices including the potentiometer, LED's, and transceivers had to be mapped through the expansion board to the microchip. All of these parts then had to be made to work in harmony with other nodes and their hardware through the application of the transmitting and receiving focused program sections (see 3.2.1). The majority of this conjoining was done in MPLAB IDE v8.80 and coded in C. The Android SDK plugin was also utilized with Eclipse IDE for windows to aid in the development of the previously mentioned GUI.

3.2.1 MiWi Software Design

Requirements for the wireless technology include low power, low data rate, and ability to access a smart-phone through USB. MiWi, from Microchip, satisfies each of these requirements through their MiWi Wireless Protocol and the pre-written demo code for the Android smart-phone interface.

The MiWi software design for this project includes four main steps.

1. Re-Mapping I/O pins from dsPIC33F to dsPIC33E
2. MiWi Application Configuration
3. Creating the Sensor Node wireless software
4. Creating the Android Node wireless software

Re-Mapping I/O Pins from dsPIC33F to dsPIC33E

MiWi Development Environment demo code could be accessed through the free Microchip Applications Library located on their website. However, this code was written for the dsPIC33F and not the dsPIC33E. Re-mapping the pins to the I/O Expansion Board and the dsPIC33E USB Starter Kit required changing the file HardwareProfile.h, located

in Appendix 3, Figures 4 - 7. Extensive reading of the data sheets and user guides for hardware component was required to complete this task. These included:

MRFX49A Data Sheet
dsPIC33F to dsPIC33E Migration Guide
dsPIC33E Data Sheet
dsPIC33E USB Starter Kit

The guide for mapping one MRFX49A pin to one dsPIC33E is located in Appendix 3, Figure 7. The general guidance to completing this task was given to the group by Microchip's technical support. The hardware pin change definitions are as follows:

nCSCON		RB1 -> RB9
INT(49XA only)		RB3 -> RA9
FINT or IRQ1		RE9 -> RA15
EE_CS		RF1 -> RG1
INT	24J40	RE8 -> RA14
IRQ0	89XA	(same as above)
IRO	49XA	(same as above)
SS2		RB2 -> RG9
SCK2		RF6 -> RG6
SDI2		RF7 -> RG7
SDO2		RF8 -> RG8
nReset		RG2 -> RA2
WAKE		RG3 -> RA3

[*Information obtained from Microchip Support Ticket ID: 218122](#)

General interface from the dsPIC33E to peripheral units such as transceivers is done through Serial Peripheral Inputs (SPI). Microchip uses a feature called Peripheral Pin Select (PPS) to easily access the SPIs. For each pin, the latch/port and tri-state bits had to be configured. This general configuration is below. This example shows mapping of the RF_INT_PIN to pin A14 on the dsPIC33E.

```
#define RF_INT_PIN PORTAbits.RA14
#define RF_INT_PIN TRISAbits.TRISA14
```

The next step in the re-mapping process is setting each pin for its particular function. This is done through the modification of the function HardwareProfile.c. Setting an SPI pin for peripheral use requires setting the pin register labeled RPINRx for input only pins and RPNRx for input/output pins. Each register's contents can be found in the dsPIC33E Data Sheet. The tri-state for some pins must also be set. For example, setting an LED for an output the register LED_TRIS must be set to 0 and setting a push button for input the

user must set Button_TRIS to 1. Other board initiations are set up in HardwareProfile.c, including clock settings.

MiWi Application Configuration

Using the MiWi development environment, setting up a wireless network using the MiWi protocol can be completed in a few steps. The first step is to configure the particular MiWi application in the file ConfigApp.h. This is done by commenting in or out the definitions for specific protocol functions. By doing this, the user can control the wireless protocol (Peer-to-Peer, MiWi, and MiWi PRO) and the transceiver used. Team 4's application defined the MRF49XA. Also, the type of roll that the node being set up will take must be decided. This is done with the function NWK_ROLE_COORDINATOR. The role of specific nodes will be described in each node's MiWi software design.

The MiWi development environment allows users to control many different functions depending on the application. The following table summarizes each function and defines the ones used by Team 4.

Figure 18: ConfigApp.h Functions

Function	Description	Used
ENABLE_HAND_SHAKE	Enables multiple nodes to connect to the same network before communicating with each other.	YES
ENABLE_SLEEP	Enables the device to enter a low-power sleep mode and to be asked to wake up from sleep.	NO
ENABLE_ED_SCAN	Allows the device to do an energy scan to find the channel with the least noise and operate on that channel.	NO
ENABLE_ACTIVE_SCAN	Lets the device check for networks already in existence.	YES
ENABLE_SECURITY	Allows security functions to be set.	NO
ENABLE_INDIRECT_MESSAGE	Enables the device to store wireless packets for sleeping devices temporarily.	NO
ENABLE_FREQUENCY_AGILITY	Enables the device to change frequency to avoid sudden noise increases on the current operating channel.	NO

**Information for table obtained from MiApp Application Note*

It may be seen from the table that Team 4 did not enable many device functions beyond the minimum required to make a simple wireless connection. However, the functions chosen have a large impact on the overall functionality of the application. First off, ENABLE_HAND_SHAKE was added to the application so that each node would only send messages when a proper network connection had occurred. This prevents wasted power consumption caused by un-needed transmissions. ENABLE_ACTIVE_SCAN is

used by the Android Node to check for existing networks before attempting to connect to one. This saves power by disallowing the Android to attempt a connection when no other nodes exist.

Creating the Sensor Node wireless software

For full code implementation, see Figure 9. The simulated sensor was defined as the network coordinator. This node will always establish a new wireless network when activated. Since the large Resource Effective Bio-Identification sensor could possibly be plugged into a power supply or a large battery, having it continuously be connected to a network will not hurt power usage as much as a smart-phone, lowering the phone's power consumption. Also, this design is feasible because the sensor will rarely be turned on and off by hand.

Through the use of the function `GetPushButton`, the sensor used for demonstration will activate the LED corresponding to the button pressed and then wirelessly transmit this data to the smart-phone. Activating push button 1 simulates a harmful chemical detection. When pressed, data is written to the transmission buffer (`TxBuffer`) with the function `MiApp_WriteData` and then broadcasted with the function `MiApp_BroadcastMessage` to all nodes on the sensor's network. When the Android Node receives this wireless packet and reads it using `MiApp_ReceiveMessage`, it causes all of the LEDs on the Android to light up and a warning message to appear on the application. This causes the all of the LEDs on the Android Node to light up and a "Harmful Chemical Detection" warning to emerge on the application window.

Creating the Android Node wireless software

For full code implementation see Figure 10. The Android Node is defined to connect to any existing networks on the channel set in the `Node2.c` program. This also allows for the smart-phone to hop on and off of the sensor's network with ease when power is returned to the `dsPIC33E` by means of the phone's USB port. Also, when no network is detected the Android Node will not attempt to create a new network in the application's current standing. However, more advanced networks will have the capability to create a network allowing the Android Node to wake up sleeping Sensor Nodes. This would be a requirement for Sensor Nodes that are powered by batteries or for general-purpose power saving.

When control commands are implemented on the Android interface, LEDs corresponding to the application illuminate on the `dsPIC33E` board. When this occurs, data is written to the transmission buffer (`TxBuffer`) through the function `MiApp_WriteData`. Then data gets broadcasted to the Sensor Node through the function `MiApp_BroadcastMessage`. When the Sensor Node receives this message using the function `MiApp_ReceiveMessage`, the LED corresponding to the command on the Android interface illuminates.

Though the Android to dsPIC33E was code created by Microchip, applying this connection code to the MiWi wireless technology has not been completed before making the Android Node software created by Team 4 a brand new technology.

Challenges

Mapping the SPI pins was one of the most time consuming problems. The first main concern was mapping each SPI to slot 1 on the I/O Expansion Board. Microchip's support team created code to help do this, but as found by much trial and error, was not complete. During this mapping process it was concluded that slot 1 did not contain the correct amount of pins necessary to use the MRF49XA. Therefore, SPI slot 2 was used. The results of this re-mapping have not yet been concluded, as will be described in Chapter 4. The mapping of SPI pins for use with the MRF49XA has never been done with the dsPIC33E before this project. This only added to the complexity of this process for it is unknown whether the physical configuration of the DsPIC33E allows for such processes to exist electrically.

3.2.2 Android to microprocessor code Design

The team inherited a Nexus S Android smart-phone from the previous team that Battelle sponsored in the Spring 2011 term. After meeting with an applications engineer from Microchip Technology, the team determined that their dsPIC33E USB starter kit could be used to interface the Nexus S to a MiWi compatible transceiver. Microchip's free applications library contains a "Basic Android Accessory Demo" that is written for the dsPIC33E USB starter kit. Using MPLAB IDE the demo was programmed onto the starter kit and the Nexus S was connected via USB. Contrary to expectations, the Android phone did not recognize that the accessory was connected. At this point the team was unsure if the problem was hardware or software related. A representative from Microchip was contacted to discuss the issue and it was confirmed that the demo code should work correctly without any modifications. Thus, the focus was directed towards determining the part of the hardware that was malfunctioning.

There are two types of USB configurations available for android devices. USB host mode and USB accessory mode. These two modes differ in the way the USB receives power. In USB host mode the Android device powers the USB and in USB accessory mode the accessory powers the USB. Only android devices that run the API 3.1 support USB host mode and the Nexus S runs API 2.3.4. Therefore, the dsPIC33E USB starter kit must provide power to the USB. In this configuration, charge can be conserved on the phones rechargeable batteries while the board's batteries can be replaced.

In order to make the trouble shooting more efficient the problem was broken down into smaller parts. A conversation with Dr. Ren uprooted several possible causes of the problem, the information may never be reaching the USB, information could be getting passed through the USB but the phone may not recognize the language, the USB might not be receiving power or the phone's USB input could be damaged. The last possibility,

that the phone's USB input could be damaged, was quickly eliminated since the phone was able to correctly operate as a USB storage device.

Next, a simple test hex file was obtained from an applications engineer at Microchip that writes a test text file to a USB storage device. This test was used to determine if the USB was passing any sort of information. Surprisingly, the test was successful. Since it would seem that this test also eliminated the possibility that the USB wasn't getting power, the only possible issue left was that the phone didn't understand the information the starter kit was passing to it. Subsequently, it was noticed that when the connection was made to the starter kit, the Android was not being charged. The charging icon was blinking on and off meaning that the connection was being made and dropped continuously. Accordingly, the current was monitored while connected and found to be jumping from around 170mA to around 270mA which is not enough to sustain a connection. With the starter kit schematic as a reference and help from Microchip, the problem was located. There was a bad current limiting diode on the board. Luckily, at this time the second USB starter kit arrived and was programmed. Everything worked correctly.

3.2.3 Android Software Design

The Android application software was written in the Eclipse Integrated Development Environment since it is the most popular and highly recommended IDE to develop Android applications. Eclipse was used with the Android Development Tools plugin for eclipse. The first step was setting up a new Android project that was compatible with the Nexus S. The team used Google APIs version 10 as the SDK version for the application. This was because the Nexus S runs Android 2.3.4 and Google APIs are needed to take advantage of USB accessory mode.

Microchip included a USB accessory demo application with the starter kit that the team received. This allowed the Android application to be able to interact with the dsPIC33E. This code was used as a base for the Android development. Since the team did not have access to the real sensor, a dsPIC33E to simulate the sensor. The required features were low battery indicator, device functioning status indicator, dangerous chemical level warning and power on and off control. To simulate this, the team decided to have push buttons to send signals to the phone to simulate if the device was functioning and if the device had low power. The power on and off control turns on an LED when the sensor is on and turns the LED off when the sensor is off. When pushbutton 1 is pressed it simulates that the sensor is not functioning properly. When pushbutton 2 is pressed it simulates that the sensor has low power. Push button 3 triggers a notification that alerts the user of dangerous power levels. Since a dangerous chemical level is very important, the designed the notification to sound an alarm that repeats until the user turns it off to ensure that it is noticed.

The layout was edited through the main.xml layout file. The team decided on a one-page layout that displayed all of the information and controls to simplify the experience for the user. There are 2 status indicators that are always displayed. When the sensor is

functioning properly the application reads, “Yes” with a green background and when it is not functioning properly (simulated by pushbutton 1 pressed down) the application displays, “No” with a red background. Similarly, when the sensor has sufficient battery power left the application displays, “Good Battery” with a green background and when it is low (simulated by pushbutton 2 pressed down) the application displays, “Low Battery” with a red background. When the chemicals levels tested are determined to be unsafe (simulated by push button 3 pressed down) the application shows the word, “Dangerous” under the chemical levels heading with a red background. Additionally, a notification appears in the notification tray when chemicals have reached a dangerous level. At the bottom there is a power on/off icon that turns green or red when the power is on or off respectively.

Chapter 4 – Test Data with Proof of Functional Design

4.1 Zigbee Functional Designs

The methodology for this design solution gave the group a basic foundation for the final design using Microchip’s product line. As such, the team found this design noteworthy and so it has been included in this report. Conceptually the only difference is the adaptation of the XBee 802.14.4 Starter Kit rather than using the dsPIC33E in conjunction with the I/O board and transceivers. The phone in the final design would require a PC to interface it to the XBee module, making it a less practical design. Other XBee modules would be linked to a PC running a sensor simulation program. The XBee network would consist of at least two of these sensors, with the possibility of more being included to form an XBee mesh network. RF antennas on each XBee module could also be added to the system. Provided there exists a mesh network, multiple sets of antennas would be used. This would yield increased range as the direction of conduction for the antennas can be narrowed and thusly elongated. The block diagram level design for the Zigbee network can be found in Appendix 3, Figure 8.

4.2 MiWi

The device design can be split into three major sections.

4.2.1 Android Node Application Functionality

4.2.2 Sensor Node Application Functionality

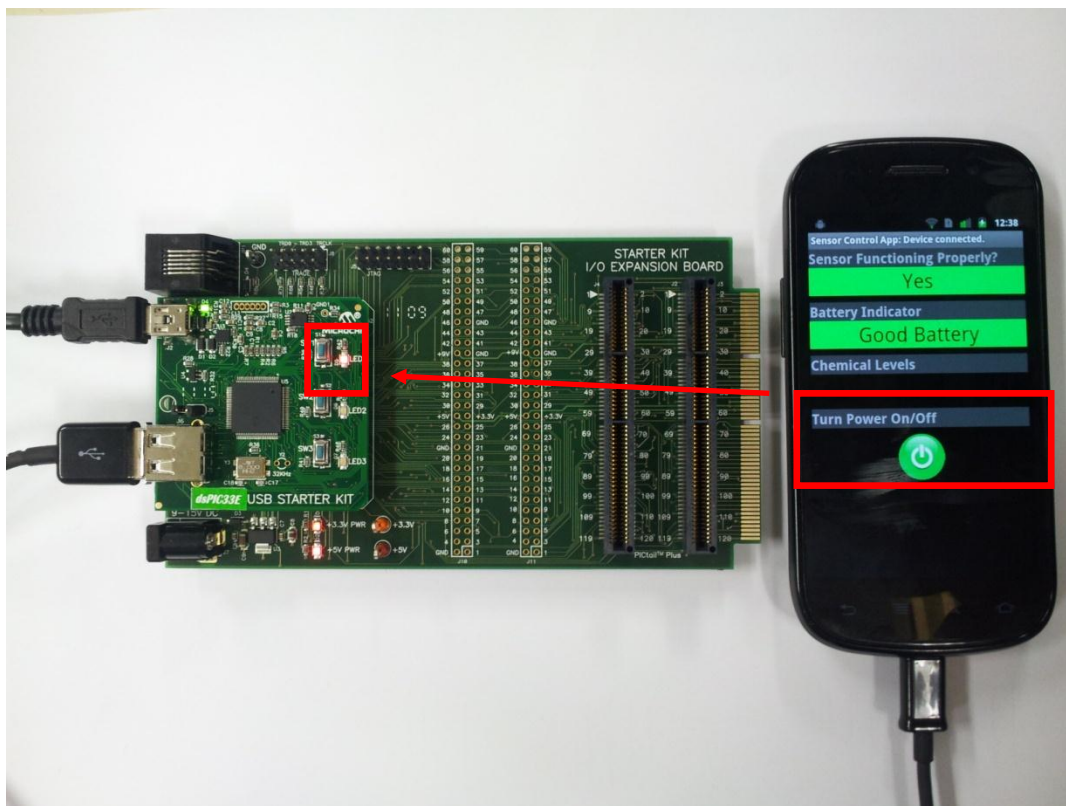
4.2.3 Wireless Connectivity Functionality

4.2.1 Android Node Application Functionality

The Android application functions fully as described in section 3.2.3 Android Software Design. The following photos depict the phone functionality.

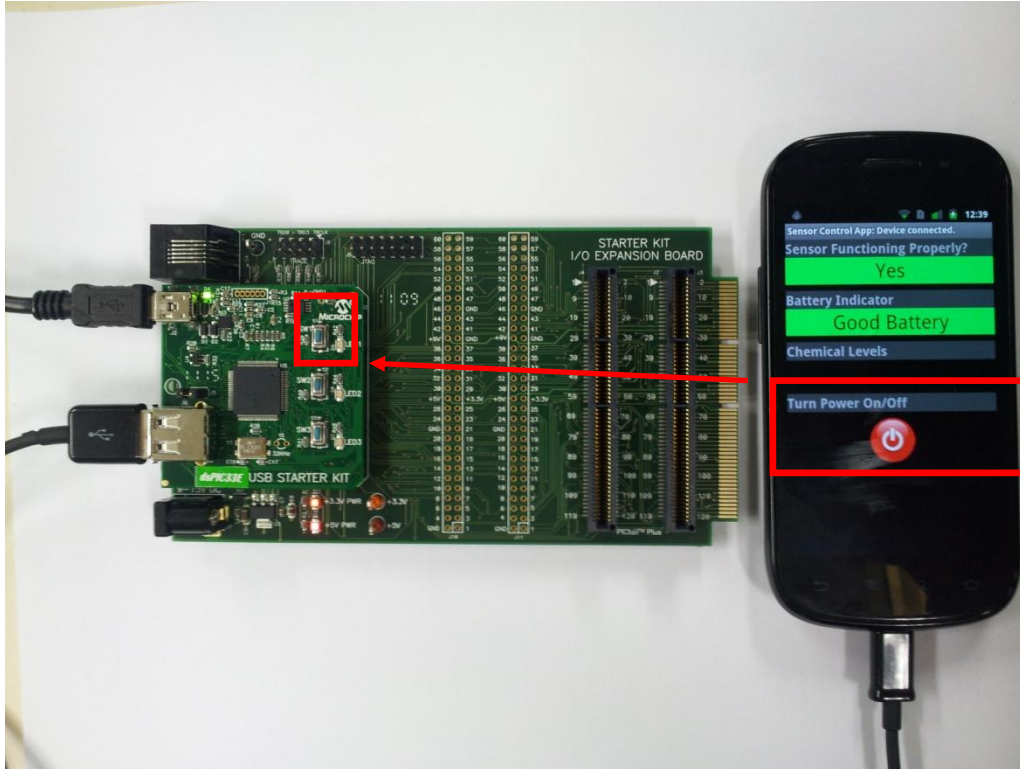
Function A) Pressing the red ON/OFF button on the Android application turns on LED1 on the dsPIC33E. With a wireless connection, this would also send a wireless message to the Sensor Node telling it to resume its normal sensor functionalities. Since we do not have the actual sensor, this is simulated by turning on LED1 on the Sensor Node's dsPIC33E board.

Figure 19: Function A.1



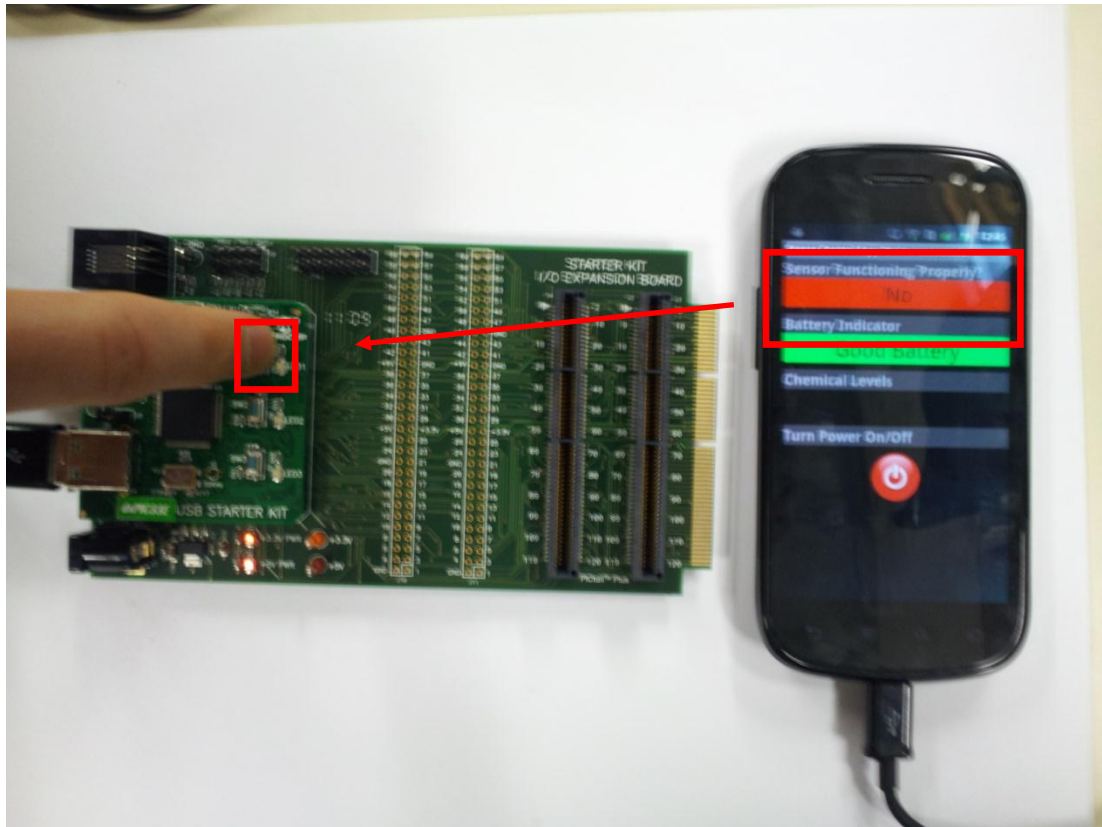
Pressing the now green ON/OFF button on the Android Application turns off LED1.

Figure 20: Function A.2



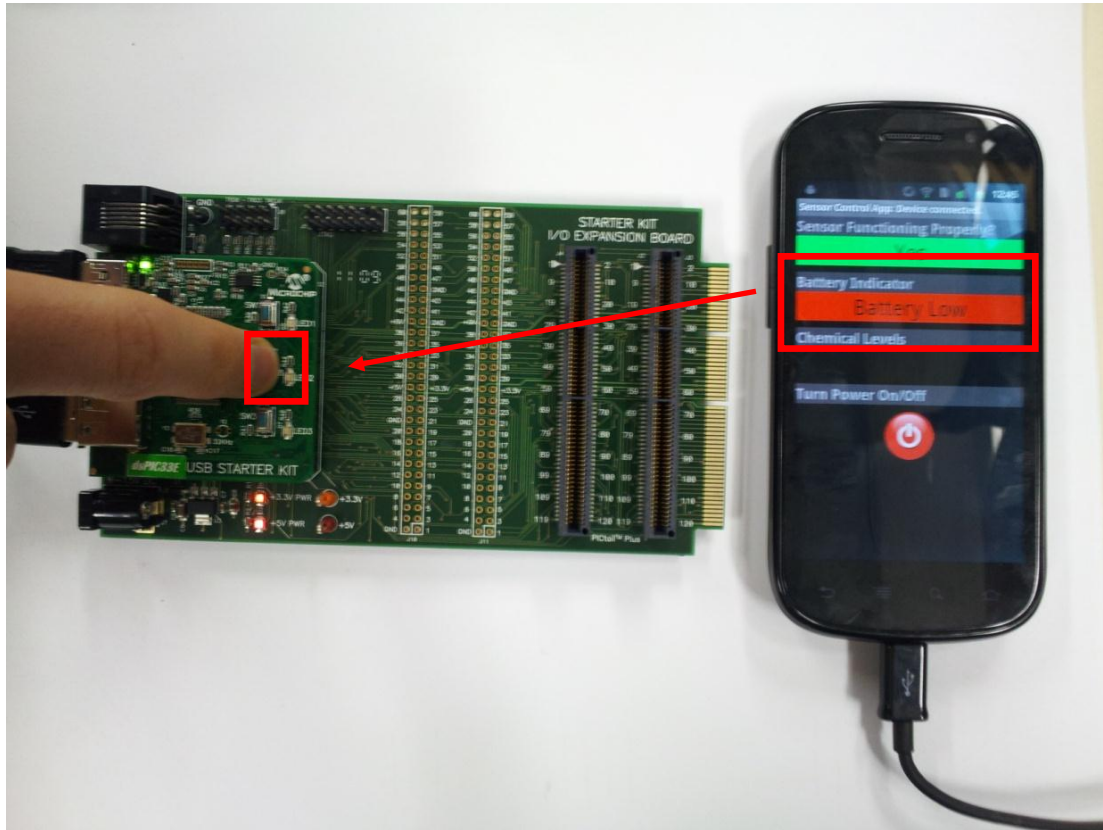
Function B) Pressing push button 1 on the dsPIC33E simulates that the sensor is not detected as being functional. If a full wireless connection were made, pressing push button 1 on the Sensor Node would do wirelessly complete the same functionality.

Figure 21: Function B



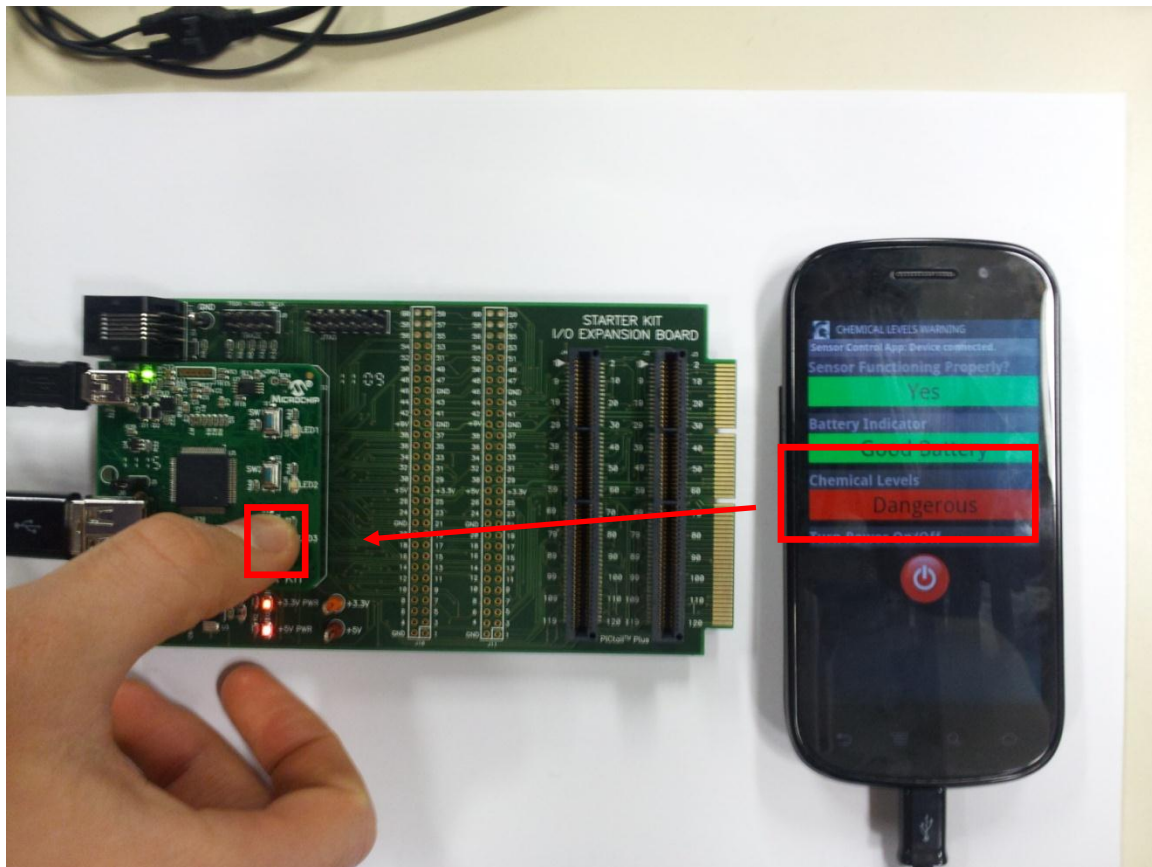
Function C) Pressing push button 2 on the dsPIC33E simulates a low battery indicator from the sensor. With a full wireless connection pressing push button 2 on the Sensor Node will wirelessly complete the same functionality.

Figure 22: Function C



Function D) Pressing Push Button 3 on the Android Node dsPIC33E will simulate a hazardous chemical detection by the sensor. This lights up a “Dangerous” warning light as well as creates a beeping notification on the phone. The notification will not shut off until the user manually turns it off. With a full wireless connection pressing push button 3 on the Sensor Node would cause the same functionality.

Figure 23: Function D



4.2.2 Sensor Node Application Functionality

Using the same push button control functions as in the Android Node, the Sensor Node is capable of controlling the LEDs on its corresponding dsPIC33E board. With a wireless connection, each button press would send data wirelessly to the Android Node as specified in the Sensor Node Software Design section found in Chapter 3. Demonstration is similar to the Android Node so it is not specifically depicted.

4.2.3 Wireless Connectivity Functionality and Testing

Functionality

Team 4 was unable to wirelessly connect the dsPIC33E boards to each other through the MiWi wireless connection. However, various tests were performed to narrow down the cause. Unfortunately, a solution was not found before the Final Report deadline.

LED Debugging

One way to test if the team's code was reaching the proper MiWi initializations was to place calls to turn on LEDs after major steps were completed. On the Sensor Node, one LED turn on was placed after the code for network creation. This showed the group that the Sensor Node was ready to accept new connections. On the Android Node, one LED turn on would be called if this node detected any networks to join. Another LED turn on was placed after the code for connecting to a network. Since the program was set to not continue further into the program until it joined a network, this LED turn on placement allowed the group to know when the Android Node was ready to send commands. The first attempts at running each of these programs lit the LED indicating that the Sensor Node was creating a network, but the Android Node could not detect a network was present. This analysis made the group question whether the transceiver was actually outputting a signal. The next documented attempt at proper MiWi wireless connectivity is as follows.

Re-Mapping I/O Pins and Removing Unused Functions

The group went through and removed the LCDBlocking.c file and all functions related to it in the main program, as well as the other transceiver files (for the 89J40) but had errors pertaining to the second SPI slot (in functions SPI_Put2 and SPI_Get2 ---> SPI_SDO2 and SPI_SDK2 not defined). This problem was caused by the incorrect definition to SUPPORT_TWO_SPI. In order to get the program to build, the group removed this definition in HardwareProfile.h. Next, the team mapped SDO2 to RG8 and SDK2 to RG6.

Results: When the team ran the program there was still no communication between the two transceivers.

Removal of LED 1

The group noticed that LED_1 was mapped to the same pin (RD0) on the dsPIC33E as the digital output for the SPI. It was determined that the LED was loading the pin and causing the data rate to slow. They then removed both LED_1 from the HardwareProfile.h code and physically de-soldered it from each of the boards, leaving the connection open to eliminate the loading on the SDO pin.

Results: There was no change in network connectivity.

Output Signal Testing

Then, the DC voltage level at the digital input on the transceiver was measured. This was done because the group questioned whether the connection at the Digital Input was opened. This was still getting power (switching between 3.3V and something in the mV range). The team then decided to take the board into the anechoic chamber to see if the transceiver was emitting a radiation pattern from the antenna. It did not appear to be the pattern expected at 915MHz so the center frequency of the antenna was measured with a network analyzer. The results of this test showed that the antenna was resonating at around 770 MHz, much less than the desired frequency.

The team's next step was to shorten the antenna slightly to improve the center frequency, this was successful and a new center freq of 918 MHz for one antenna and 909 MHz for the other was achieved. After this adjustment was performed, the radiation pattern looked more like it should (with the exception of noise from the computer). The results of this test would appear to confirm that the transceiver is setting up some kind of network.

Results: The shortening of this antenna wire did not allow for a successful wireless connection. In retrospect, the results shown from the network analyzer test could have detected noise from the computer components located in the lab.

Clock Frequency Analysis

The next step of the debugging process was to ensure the correct clock frequency was defined for communication with the MRF49XA. This was done by including the following code in the program directly after the MiWi Protocol is initiated.

```

/*****
MIWI_TICK t1, t2;

t1 = MiWi_TickGet();
While(1)
{
    t2 = MiWi_TickGet();
    If( MiWi_TickGetDiff(t2, t1) > ONE_SECOND )
    {
        LED_1 ^= 1;
        t2.Val = t1.Val;
    }
}
*****/
```

This function causes an LED to blink in compliance with the clock frequency. If the frequency of the transceiver and the frequency of the dsPIC33E are equal, then the LED blinks ON and OFF exactly one time in two seconds.

Results: When the group implemented this function, the LED did not blink as expected, showing that one of our major problems was defining the correct clock frequency. The correct frequency to use is still being researched, but setting the new frequency will require multiplying the 8 MHz Crystal Oscillator on the dsPIC33E USB Starter Kit to obtain the correct value.

Testing Conclusions

Using the MRF49XA in SPI slot 1 requires the use of I/O pins that the DsPIC33E ties to USB functionality. Therefore, mapping for use in SPI slot 2 is required. Team 4 accomplished this task through the mapping pin diagrams found in Appendix 3.

To set these newly mapped pins for Input or Output functionality, the pin's RPINR/RPOR register must be configured for the correct mapping. Upon further evaluation, however, it appears that the register corresponding to SPI2, RPINR22, has been removed by Microchip.

Table A-1: MAJOR SECTION UPDATES

Section 11.0 "I/O Ports" ----->Removed RPINR22 register

Information Obtained from dsPIC33EPXXXMU806/810/814 and PIC24EPXXGU810/814 Data Sheet, p. 553

Microchip has been contacted regarding this issue, but no response has been made before the completion of this Final Report. The result could be that it is impossible to connect the DsPIC33E USB Starter Kit to the MRF49XA through the I/O Expansion Board due to the use of SPI1 pins by USB functions on the DsPIC33E and the removal of the RPINR22 control register needed to set the SPI2 pins for input and output.

Chapter 5 – Final cost, schedule, summary and conclusions

5.1 Findings

When evaluating the outcome of our final design solution a picture of success can easily be painted. The goal of this design was to unite a phone with a sensor, utilizing a form of communication that didn't rely on anything except battery power. As can be seen in this report even without a functional prototype all of the foundation has been laid for a final product. Although the design could have been more eloquent, it gave a proper demonstration of how simpler technology, in conjunction with new products, can yield a very robust system.

With that said, the process from start to finish was hardly an easy one. It is the opinion of this team that when attempting to integrate; phones, RF communication, and sensor systems, using a fully integrated system with an external self-powered antenna would yield best results in practical application. This however, was in no way feasible for our team with the allotted resources.

Not to be discouraged our team designed a prototype intentionally created in a manner that sets the foundation for a final product. The hardware of our design is not only separable but also interchangeable. With only a few minor code changes, our design can be made to work with other Microchip products. The concept was born from the decision to replace XBee in our final design. Due to its limited functionality, and its requirement for a vast amount of design needed in order to improve upon this, our team made it a point to find a more cohesive system that still took on the same principles of our last. The ability to 'upgrade as you go' makes this prototype invaluable for a final product design process.

5.2 Suggestions for future projects

Pending completion of the current project, there could be several improvements. The MiWi network could be configured to support multiple phones and sensors. The network configuration that the team suggests is a mesh network. A mesh network allows for all of the Full Functional Devices (FFDs) to route messages to any end device in the mesh. The sensors would be the FFDs and the smart-phones would be the end devices.

Additionally, the MiApp layer of the MiWi protocol stacks contains useful functions that can enable security and allow the RF transceiver to use an external power amplifier to boost the range of the signal. Both of these functionalities would be an excellent addition to the overall project.

Furthermore, Microchip has a Wi-Fi module that can be programmed to create its own Wi-Fi network and web page. After connecting to the module's Wi-Fi network through the smart-phone, entering the module's IP Address into the search bar connects the phone to the customized web page. From here the user can control LEDs and monitor push buttons on the circuit board. It may be possible to utilize this product to connect to the MiWi transceiver wirelessly. It is Team 4's suggestion that future projects should explore this possibility. A wireless connection to the MiWi transceivers would be much more practical than a USB connection.

Moreover, it has been proven that MiWi technology can be used to control a remote control car. Hence, a further improvement to the sensor system would be to place it on a remote control cart or robot that would enable the sensor to move. This would both make it more difficult to disable and increase the area that it could monitor.

5.3 Final Costs

The final cost from our project includes 2 USB starter kits, 2 enclosures, and 2 USB starter kit IIs.

Total Cost = \$287.88

*3 Transceivers and 3 I/O boards were donated by Microchip. Of which 2 each were used in the final design.

5.4 Conclusion

What Worked

A program for connecting a smart-phone to a microprocessor through the micro-USB port has been created by Microchip. Team 4 adapted this code to create a working application that can control the DsPIC33E.

Using the commands from the Android application, wireless commands can be sent out from the DsPIC33E or other applicable processor if a wireless connection between two or more nodes has been established.

What Did Not Work

The creation of a wireless connection between the Android Node and the Sensor Node was not completed. This problem was not due to impossibility, but instead due to a lack of time to complete the research needed to fully utilize the SPI pin selection features on top of learning the commands to control the MiWi Development Environment.

Summary

Though the completion of the task assigned by Battelle Laboratories to create an Android based smart-phone wireless sensor control for the group was unable to establish, the proof of concept is deemed as FUNCTIONAL. The creation of a working wireless network was the only step not completed as outlined in Team 4's design plan. However, recent contact with Microchip has given the group the documents and knowledge needed to complete this task. The last step would be to correctly map the DsPIC33E to the MRF49XA.

Final Thoughts

With the completion of the final prototype only one step away, Team 4 is excited to introduce a new Android smart-phone functionality, one of which has not been attempted before. Getting to this far in the project completion first involved using basic engineering design principals to research a communication technology compatible with the Android phone. Then, through much technical research and professional communications, the group was able to narrow down the final steps for prototype completion. Thanks to the hard work of the team and the helpful staff at Microchip, completion of the first USB powered Android smart-phone sensor control device is only a few steps away.

Appendix 1 – Technical Roles, responsibilities and work accomplished.

Jacob Sawicki – Manager



Over the course of this term Jacob's work contributed to the completion of several phases of the project. In the early stages he and the rest of the team were tasked with researching several forms of wireless communication. He collaborated with Andreas Dixon to perform research directed towards the capabilities of RF transmission. This research ultimately led to the decision of which wireless technology to push forward with.

Initially, Jacob's major task was to familiarize himself with the previous Battelle team's sensor simulation program and modify it for the current team's specifications. However, it was later suggested that the sensor could instead be modeled using circuitry. This

option was adopted and Jacob's task changed. Jacob's focus was then directed towards interfacing the Android application to the microprocessor in order to provide a link between the MiWi transceivers and the smart-phone. It was a seemingly simple task, but due to unforeseen complications an extensive amount of debugging was required. Jacob's debugging included a deep look into the example code provided by Microchip, the verification that the smart-phone's hardware was not damaged and using a test program to verify that the starter kit's USB was working correctly. This debugging ultimately led to the discovery that the starter kit's hardware was damaged. Much to his surprise, the second board he tried also malfunctioned. While theoretically the problem could be solved by shorting two of the pins on the USB connector, he instead decided to test the third starter kit that they had ordered as a back up. It had arrived the day after the problem was located and thankfully functioned correctly.

Jacob also spent much of his time providing support to Andreas Dixon and Phillip Horny who were working on the MiWi program. My support was utilized during the various stages of debugging. These stages included the remapping of the MiWi transceiver to the dsPIC33E starter kit, configuring MiWi for a P2P network, testing the resonance of the transceiver's antenna and measuring the radiation pattern of the transceiver.

Kevin Gleason – Webmaster



Throughout the semester Kevin worked on a variety of aspects of the project. The first thing that he helped out on was researching the various communication methods to decide on what communication method would be used to communicate between the sensor and smart phone.

Kevin's primary contribution to the group was the design and development of the Android application. Kevin has the most programming experience out of the group which is why he was chosen for this task. The first task was familiarizing himself with the Android development environment. Once he was familiar with Android development he reviewed the demo code that was included with the dsPIC33E starter kit. After Kevin setup the

included demo code it was able to communicate to the dsPIC33E. After reviewing the code he cut out the parts of the code that were not going to be used in order to make the application more efficient. Kevin then implemented the required functions that were needed to control the simulated sensor. He modified the LED control class that Microchip created to allow the phone to turn on and off the simulated sensor. He then used the push buttons functions to simulate alerts for low battery or alerts if the sensor is not functioning. Next he made the system alert notification system to alert the user if there are toxic chemicals in the air.

Once the communication between the phone and the dsPIC33E was completed for the design requirements his next task was to create the graphical user interface for the sensor control application. He then designed the user interface to make sure it was very simple to use. The design phase went through couple iterations but based on the low number of features that are need he decided to put the controls and status updates all on one page. After the interface was complete he tested the functionality of the application under a variety of circumstances to make sure it operated correctly and was intuitive to use.

Phillip Horny – Documentation Prep



Phillip's first role in this project was to research the possible utilization of Cellular networks to control the sensor. He concluded that this task was not feasible for the group would have to purchase a service contract. After MiWi Wireless Protocol was chosen as the wireless transmission technology, his next role in this project was creating a wireless connection between the Android and Sensor Nodes. He spent a great deal of time reading through the various user guides for each hardware component listed in the Hardware Design section of Chapter 3. The main step to set up the MiWi Development Environment for wireless communication is to map the pins of the microprocessor to the correct pins on the transceiver. Phil went through many different methods of mapping, taking up the bulk of the semester's time. He even contacted Microchip's support team in order to gain some guidance. Though this task has yet to be successful, he learned a great deal about pin mapping and data sheet reading.

Next, Phil created the Sensor Node program to be used as the simulated Resource Effective Bio-Identification sensor. Creating this program included creating a push button control function to allow the sensor to send commands to the Android Node. Also, Phil had to familiarize himself with the MiApp application layer for MiWi programming. MiApp is the set of functions that allow the user to create networks, send transmissions, read incoming data, and configure individual nodes.

Lastly, Phillip created the MiWi Android Node program. This process involved integrating the Basic Android Accessory Demo with the MiWi MiApp functionality. This allowed him to create a program that will transmit data out of the Android Node when commanded to by the Android phone interface. Before this project, this application integration had not been completed or even considered by Microchip, the creators of MiWi and the Basic Android Accessory Demo.

Andreas Dixon – Lab Coordinator



Andreas' undertaking in the development of this design was one of multi-functionality with a focus on hardware and the software connections needed for it. It began with a collaborative effort in identifying which method of communication would be best, and what hardware to use in order to accomplish this goal. When pushing forward with the project Andreas and Jacob met with several Microchip representatives in order to redesign our project from Xbee to MiWi after it was determined that Xbee was not the most efficient method of signal transfer. Once a hardware setup had been clearly laid out Andreas worked with Phillip to reconfigure the mapping of our overall system.

The hardware and mapping combination at this point in the project was not functioning properly. As such, Andreas in conjunction with Jacob helped forward the troubleshooting of the transceiver antenna including signal strength, radiation pattern and resonant frequency. This was done to ensure that our hardware had been properly manufactured before Team 4 received it, as well as to verify that it was still functioning without physical error. Andreas also aided in further testing that was done directly across our entire system. DC voltage was traced around the hardware system with the project code running and several inconsistencies were found. After a phone conference with a Microchip software engineer it was determined that an LED on the USB starter kit need be removed. With the combined efforts of Jacob and Andreas this LED was removed from each one of Team Fours boards. Another aspect of the hardware design in this project was to create a design enclosure to fit microchips products and a battery power supply. To do this Andreas collaborated with Thamer in creating a design enclosure that would meet our needs as well as give a more professional display to our final design.

Thamer Alajlan – Presentation Prep



The purpose of our project is to develop a method of communication between several sensors through the use of a Smart Phone as a common node. The first step was researching on the communication methods such as (Zigbee, MiWi, WiFi) which allowed us to understand and develop more in the project and the suitable network configuration for each method. Along with his team, Thamer concluded that MiWi protocol is the perfect match for the project due to the ease of use for short range networking and the low-cost of the product. He also conclude that the most suitable configuration for the project will be the star network configuration since they be using less complicated system where they only need to set a wireless network between a smart phone and a sensor.

His second technical Task was building the enclosure design for the hardware components using Auto CAD (Computer Aided Design) software. Before using AutoCAD he had to take all the measurement of the starter kit including the battery holder, two USB ports and the push buttons. Using a rough sketch he took all the measurement using the centimeter unit for accurate so when we get from the factory it fits our hardware components. The second step was converting the rough sketch to a 2-d drawing and then creates the enclosure based on the demanding module. According to Thamer, he was not familiar with the AutoCAD software. However, he did a research on the web along with couple of step by step videos he was able to design three modules with the exact features the group asked for.

His third technical task was engaging with the daily development of the project and research about any problems the team was facing. Along with his teammates, they were

able to solve most of the challenges they faced during this semester by getting together and share the ideas and solutions the team found for a certain problem.

Appendix 2 – Literature and website references

Android Developers *Google*. Google Inc., 2011. Web. 10 Oct 2011.
<<http://www.developer.android.com>>.

DsPIC33F/PIC24H to DsPIC33E/PIC24E Migration and Performance Enhancement Guide *Microchip*. Microchip Technology Inc., 2011. Web. 7 Nov 2011.
<<http://ww1.microchip.com/downloads/en/DeviceDoc/70637C.pdf>>.

DsPIC33EPXXXMU806/810/814 and PIC24EPXXXGU810/814 Data Sheet *Microchip*. Microchip Technology Inc., 2011. Web. 7 Nov 2011.
<<http://ww1.microchip.com/downloads/en/DeviceDoc/70616E.pdf>>.

DsPIC33E USB Starter Kit and User's Guide Kit and PIC24E *Microchip*. Microchip Technology Inc., 2011. Web. 7 Dec 2011.
<http://ww1.microchip.com/downloads/en/DeviceDoc/33E_24EUSB_SK_UserGuide.pdf>.

Microchip Support Ticket ID: 218122. Message to Phillip Horny. 17 Nov 2011. E-mail.

Microchip Application Library v2011-10-18 *Microchip*. Microchip Technology Inc., 2011. Web. 7 Nov 2011. <<http://www.microchip.com/mal>>.

MRF49XA Data Sheet *Microchip*. Microchip Technology Inc., 2011. Web. 7 Nov 2011.
<<http://ww1.microchip.com/downloads/en/DeviceDoc/70590C.pdf>>.

Yang, Yifeng. **Microchip Wireless (MiWi™) Application Programming Interface – MiApp**. AN1284. Microchip Technology Inc., 2009. Web. 7 Nov 2011.
<<http://ww1.microchip.com/downloads/en/AppNotes/01284A.pdf>>.

Yang, Yifeng. **Microchip Wireless (MiWi™) Media Access Controller – MiMAC**. AN1283. Microchip Technology Inc., 2009. Web. 7 Nov 2011.
<<http://ww1.microchip.com/downloads/en/AppNotes/01283a.pdf>>.

Yang, Yifeng. **Microchip MiWi™ P2P Wireless Protocol**. AN1204. Microchip Technology Inc., 2010. Web. 7 Nov 2011.
<<http://ww1.microchip.com/downloads/en/AppNotes/01204B.pdf>>.

Appendix 3 – Detailed technical attachments

Figure 24: Program Flow Chart

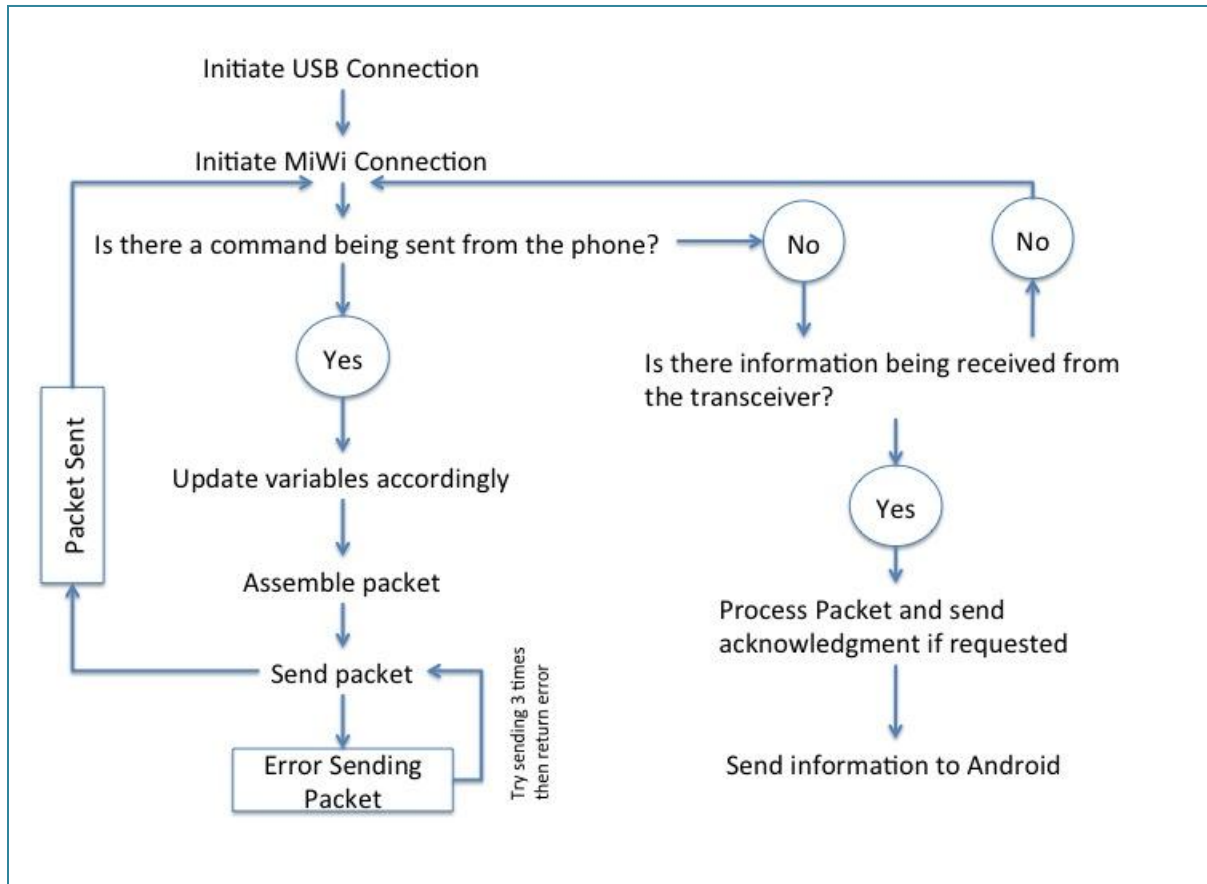


Figure 25: Old Gantt Chart

		Task Mode	Task Name	Duration	Start	Finish	Predecessors
1			Project Definition Task	3 days	Wed 9/14/11	Sun 9/18/11	
2			Initial Team Meeting	1 day	Wed 9/14/11	Wed 9/14/11	
3			Meeting with Sponsor	1 day	Thu 9/15/11	Thu 9/15/11	2
4			Discuss with whole team	1 day	Sun 9/18/11	Sun 9/18/11	3
5			Develop conceptual designs	5 days	Mon 9/19/11	Fri 9/23/11	
6			Develop concepts individually	1 day	Mon 9/19/11	Mon 9/19/11	
7			Assign concepts to push forward with	1 day	Wed 9/21/11	Wed 9/21/11	6
8			Develop evaluation criteria from specs	2 days	Thu 9/22/11	Fri 9/23/11	7
9			Evaluate conceptual designs	1 day	Sun 9/25/11	Sun 9/25/11	8
10			Conceptual design selected	0 days	Sun 9/25/11	Sun 9/25/11	9
11			Design communication circuits	37 days	Mon 10/3/11	Tue 11/22/11	10
12			Research Zigbee Communication	2 days	Mon 10/3/11	Wed 10/5/11	10
13			Order Zigbee components	1 day	Fri 10/7/11	Fri 10/7/11	12
14			Build preliminary communication circuits	3 days	Mon 10/17/11	Fri 10/21/11	13
15			Build and test final communication circuits	4 days	Tue 11/8/11	Fri 11/18/11	
16			Communication circuit complete	0 days	Tue 11/22/11	Tue 11/22/11	15
17			Develop sensor simulation software	41 days	Mon 9/26/11	Mon 11/21/11	10
18			Obtain the code from last semester's team	1 day	Mon 9/26/11	Mon 9/26/11	10
19			Understand code	6 days	Mon 10/3/11	Fri 10/14/11	18
20			Optimize code for our application	7 days	Mon 10/17/11	Wed 11/2/11	19
21			Test code using preliminary communication circuit	5 days	Mon 11/7/11	Mon 11/21/11	20,14
22			Sensor simulation code complete	0 days	Tue 11/22/11	Tue 11/22/11	21
23			Develop communication software for smart phone	42 days	Mon 9/26/11	Tue 11/22/11	10
24			Obtain smartphone from ece shop	1 day	Mon 9/26/11	Wed 9/28/11	10
25			Obtain the code from last semester's team and link to application development tutorial	1 day	Mon 9/26/11	Mon 9/26/11	10
26			Take tutorial to become familiar with developing an application for a smartphone http://developer.android.com/resources/tutorials/hello-world.html >	1 day	Mon 10/3/11	Mon 10/3/11	25
27			Become familiar with old code	6 days	Mon 10/10/11	Wed 10/19/11	25,26
28			Optimize/write new code for our application	5 days	Thu 10/20/11	Fri 11/4/11	27
29			Test code using preliminary communication circuit	5 days	Mon 11/7/11	Mon 11/21/11	28,14
30			Smartphone application complete	0 days	Tue 11/22/11	Tue 11/22/11	29
31			Implement Final Design	11 days	Wed 11/23/11	Wed 12/7/11	
32			Hook up final communication circuitry to laptop and phone	1 day	Tue 11/22/11	Wed 11/23/11	22,30
33			Test range of final design	3 days	Fri 11/25/11	Fri 12/2/11	32
34			Final Design complete	0 days	Mon 12/5/11	Mon 12/5/11	33

The diagram illustrates the pin-to-pin connections for the dsPIC33E/24E microcontroller. It shows the mapping of pins 1 through 50 to pins 51 through 100, detailing the internal modules and external components connected to each pin.

Pin 1 to 50:

- 1: R015
- 2: +3.3V_TARGET
- 3: PMPD5/PED5
- 4: PMPD6/PED6
- 5: PMPD7/PED7
- 6: T2CLK/R01
- 7: T3CLK/R02
- 8: T4CLK/R03
- 9: S011/T2CLK/R04
- 10: PMPA5/SCM20/CN0A/R00
- 11: PMPA4/SCM24/CN0B/R07
- 12: PMPA3/SCM28/CN10/R08
- 13: R014
- 14: PMPA2/SCM20/CN11/R09
- 15: +3.3V_TARGET
- 16: TMS/R00
- 17: INT1/R05
- 18: INT2/R06
- 19: VBUSON/CIN1+/AN0/CN7/R05
- 20: USBDON/CIN1-/AN4/CN6/R04
- 21: CIN1+/AN3/CN5/R03
- 22: CIN1-/AN2/CN4/R02
- 23: PGC1/AN0/CN3/R01
- 24: PGC1/AN0/CN2/R00
- 25: I2SP_P0REF_TARGET
- 26: I2SP_P0REF_TARGET
- 27: PMPA7/VREF-/R08
- 28: PMPA6/VREF+/R09
- 29: +3.3V_TARGET
- 30: C1OUT/AN0/R00
- 31: C2OUT/AN0/R00
- 32: PMPA13/CVREF/AN10
- 33: PMPA12/AN11/RB11
- 34: +3.3V_TARGET
- 35: TCK/RA1
- 36: SCHN0/BCLK2/RF13
- 37: SCHN0/RF12
- 38: PMPA11/AN12/RB12
- 39: PMPA10/AN13/RB13
- 40: PMPA1/AN14/RB14
- 41: PMPA0/AN16/CURB/CN12
- 42: +3.3V_TARGET
- 43: CH20/SGM10/RD14
- 44: CH21/SGM10/RD14
- 45: PMPA8/SCM34/CN17/RF4
- 46: PMPA8/SCM38/CN10/RF5

Pin 51 to 100:

- 51: PMPD4/RE4
- 52: PMPD3/RE3
- 53: PMPD2/RE2
- 54: R010/R013
- 55: R011/R012
- 56: TRD2/R014
- 57: PMPD1/RE1
- 58: PMPD0/RE0
- 59: TRD3/RA7
- 60: TRCLK/RA0
- 61: PMPD0/R00
- 62: PMPD0/R01
- 63: PMPD10/RF1
- 64: PMPD11/RF0
- 65: +3.3V_TARGET
- 66: VDDCAP
- 67: C3NAR07
- 68: C3NAR08
- 69: PMPD0R02
- 70: PMPA0/R00
- 71: R013
- 72: R012
- 73: PMPD0R03
- 74: VDDCAP
- 75: VDD
- 76: PGC1/S0G0R0C13
- 77: PGC1/S0G0R0C13
- 78: INT0R00
- 79: PMPD0R01
- 80: ASCL1PMPD0R00
- 81: ASDA1R00
- 82: RTCCR00
- 83: RA10
- 84: RA14
- 85: VDD
- 86: G000R010
- 87: G000R012
- 88: VDD
- 89: T0CRPA0
- 90: T0CRPA4
- 91: ASDA3R00
- 92: ASCL2RA0
- 93: D-R00
- 94: D-R00
- 95: VDD
- 96: VDD
- 97: NC
- 98: R010
- 99: USBDON
- 100: PMPD4/RE4

External Components:

- C14: 10uF
- C15: 11pF
- Y2: 32 KHz
- C16: 11pF
- C17: 20pF
- Y3: 8 Mhz
- C18: 20pF
- R35: 0R
- R36: 1M

dsPIC33E/24E

Figure 27: Hardware connection map for the USB starter kit to I/O board

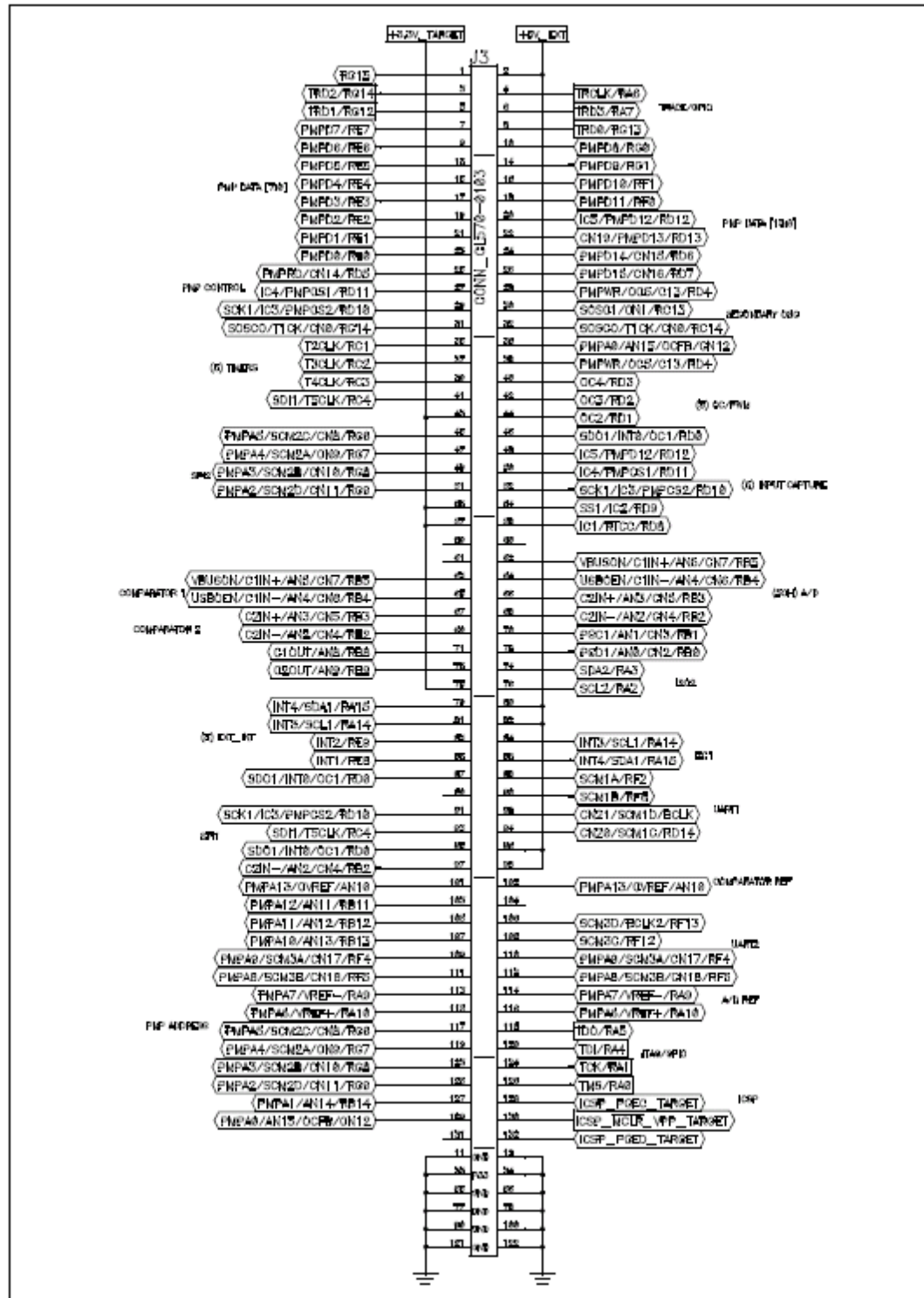


Figure 28: Hardware connection map for the I/O board

Starter Kit Connector (J3)			Multimedia Expansion Board	
Pin	Description	Pin Type	Device	Description
25	PMP Data <7:0>	I/O	Graphics Controller (SSD1926)	8-bit or 16-bit Data Bus
23		I/O		
21		I/O		
19		I/O		
17		I/O		
15		I/O		
13		I/O		
9		I/O		
7		I/O		
10	PMP Data <15:8>	I/O		
14		I/O		
16		I/O		
18		I/O		
20		I/O		
22		I/O		
24		I/O		
26		I/O		
8	RG13	O		Chip Select
101	RB10	O		Register Select
39	RC3	I		Wait Line
115	RA10	O		Reset
103	RB11	I/O	Touch Screen	X+
105	RB12	O		Y-
107	RB13	O		X-
127	RB14	I/O		Y+
72	RB0/CN2	I	Joystick	Left
70	RB1/CN3	I		Up
66	RB3/CN5	I		Down
64	RB4/CN6	I		Right
36	RB15/CN12	I		Fire
44	RD1	O	LEDs	LED1
42	RD2	O		LED2
40	RD3	O		LED3
35	RC1	O		LED4
37	RC2	O		LED5

Starter Kit Connector (J3)			Multimedia Expansion Board	
Pin	Description	Pin Type	Device	Description
74	SDA2	I	I ² C bus for BMA150, MCHP24LC08 and WM8731	I ² C Bus
76	SCL2	O		
91	SCK1	O	SPI Bus for WM8731	SPI Bus
93	SDI1	I		
95	SDO1	O		
4	RA6	O	CPLD	Control Pins
6	RA7	O		
5	RG12	O		
3	RG14	O		
45	SCK2	O		
47	SDI2	I		SPI Bus
49	SDO2	O		
51	RG9	O		Chip Select
106	SCK3A	O		SPI Bus
110	SDI3A	I		
112	SDO3A	O		Chip Select
108	RF12	O		
97	SS1	O		Codec DACLRC
54	RD9	O		
81	INT3	I	MRF24WB0MA	External Interrupt
115	RA10	O		Reset
71	RB8	O		Chip Enable
76	SCL2	I/O	P1Ctail J5	Pin 3
74	SDA2	I/O		Pin 5
47	SDI2	I/O		Pin 7
49	SDO2	I/O		Pin 9
45	SCK2	I/O		Pin 11
51	RG9	I/O		Pin 13
88	U1RX	I/O		Pin 15
90	U1TX	I/O		Pin 17
92	U1RTS	I/O		Pin 19
94	U1CTS	I/O		Pin 21
73	RB9	I/O		Pin 23
115	RA10	O		Pin 25
85	INT1	I/O		Pin 27
84	SCL1	I/O		Pin 4
86	SDA1	I/O		Pin 6
97	SS1	I/O		Pin 8
110	U2RX	I/O		Pin 16
112	U2TX	I/O		Pin 18
106	U2RTS	I/O		Pin 20
108	U2CTS	I/O		Pin 22

Figure 29: Hardware connection map for the MRF49XA transceivers

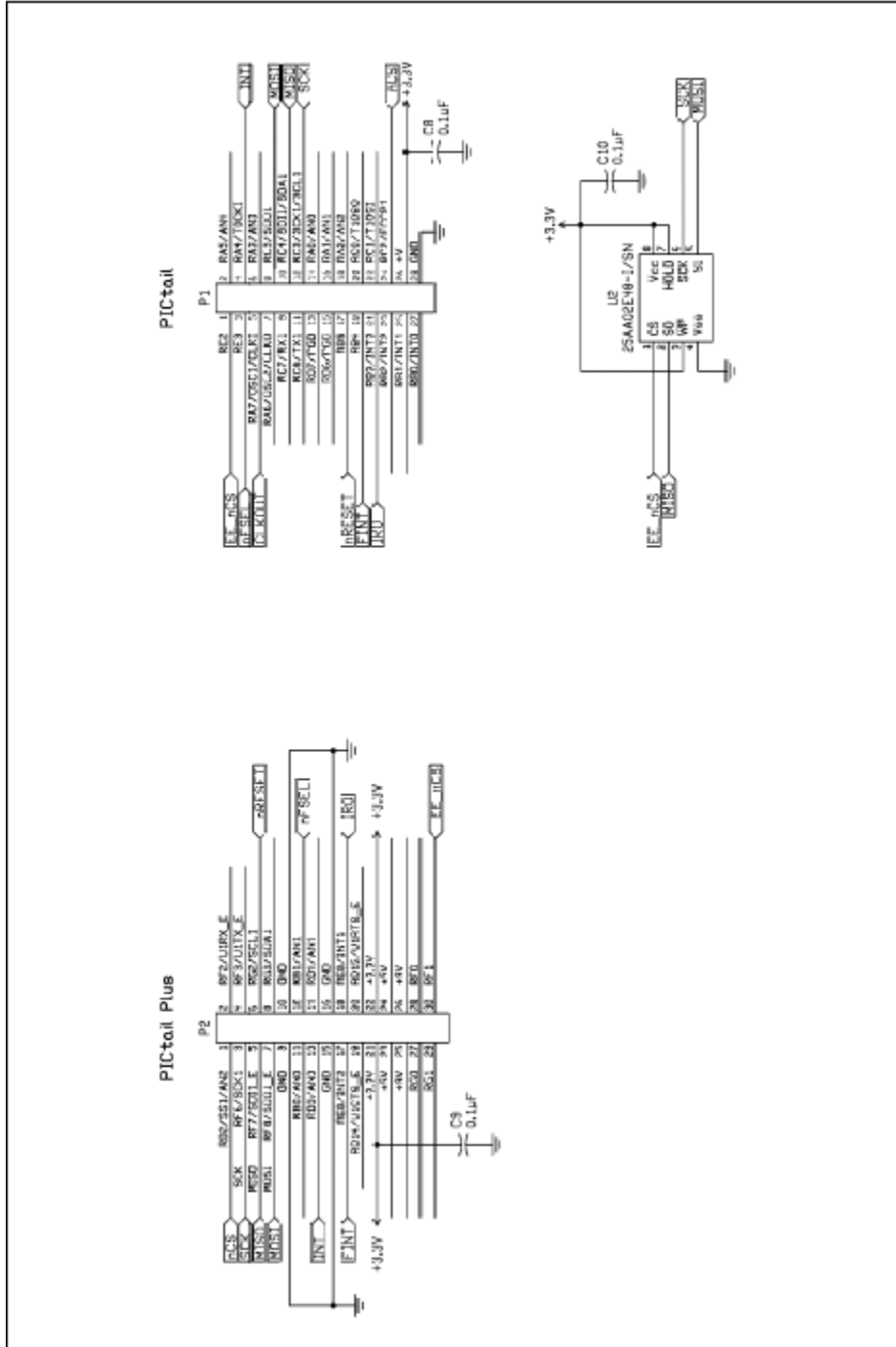


Figure 30: Zigbee Functional Design 1

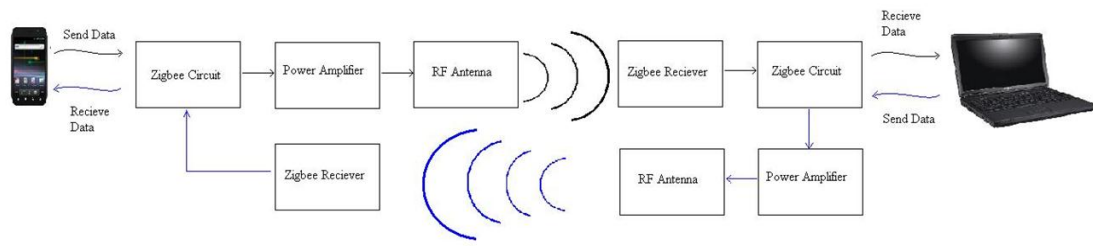


Figure 31: Sensor_Node.c

```

/*****
* FileName:                Sensor_Node.c
* Dependencies: none
* Processor:  dsPIC33E
* Compiler:  Microchip C18 v3.04 or higher
*                Microchip C30 v2.03 or higher
*                Microchip C32 v1.02 or higher
*****/
* File Description:
* Sensor_Node acts as a simulated Resource Effective Bio Identification
* sensor. This node will transmit status based on commands received from
* the Android Node.
* It will also output warnings to the Android Node when a simulated Hazardous
* Chemical is detected.
*****/

/***** HEADERS *****/

#include "WirelessProtocols/Console.h"
#include "ConfigApp.h"
#include "HardwareProfile.h"
#include "WirelessProtocols/MCHP_API.h"
#include "WirelessProtocols/LCDBlocking.h"

#include "WirelessProtocols/SymbolTime.h"

/***** VARIABLES *****/
#define LIGHT  0x01
#define SWITCH 0x02

#if ADDITIONAL_NODE_ID_SIZE > 0
    BYTE AdditionalNodeID[ADDITIONAL_NODE_ID_SIZE] = {LIGHT};
#endif

BYTE myChannel = 0;

/*****
* Function:    void main(void)
*
* PreCondition: none
*
* Input:        none
*
* Output:        none
*
* Side Effects: none
*
* Overview:      Initializes board and MiWi P2P wireless connection.
*                If a message is received, LED's corresponding to the
*                proper command will light up.
*                If a button is pressed, commands will be sent out
*                wirelessly from the Sensor Node to the Android Node.
*
* Note:
*****/
#ifdef __18CXX
void main(void)
#else
int main(void)
#endif
{
    int j;
    BYTE i;
    BYTE TxSynCount = 0;
    BYTE TxSynCount2 = 0;

```

```

BYTE TxNum = 0;
BYTE RxNum = 0;
    BYTE tempValue = 0xFF;
    BYTE pushButtonValues = 0xFF;
    BOOL buttonsNeedUpdate = FALSE;
    BYTE command = 0xFF;

/*****
// Initialize the system
*****/
BoardInit();
    InitAllSwitches();
//LED_1 = 0; //JS Commented out
LED_2 = 0;
    LED_3 = 0;

/*****
// The following will initialize the MiWi P2P protocol for Node 1,
// the node that will set up the network.
*****/

    MiApp_ProtocolInit(FALSE);

if( MiApp_SetChannel(myChannel) == FALSE )
{
    LED_2 = 1;
}

MiApp_ConnectionMode(ENABLE_ALL_CONN);

MiApp_StartConnection(START_CONN_DIRECT, 10, 0);
    LED_3 = 1;

/*****
//      while(1) loop continuously checks for received messages.
//      It also will send messages if it is not currently receiving a message
*****/

while(1)
{
    if( MiApp_MessageAvailable() )
    {
        rxMessage.PayloadSize; i++)
        {
            command = *rxMessage.Payload;

            if (command == 0x00)
            {
                MiApp_DiscardMessage();
            }
            else if (command = 0x02) //Indicates command 2 from phone
            {
                LED_2 = 1;
            }
            else if (command = 0x04) //Indicates command 3 from phone
            {
                LED_3 = 1;
            }
            else if (command = 0x01) //Indicates check status, command 1 from phone
            {
                for(i = 0;i<4;i++)
                {
                    for(j = 0;j<10000;j++) //LEDs blink 5 times while status is checked.
                    {
                        LED_2 = !LED_2;
                        LED_3 = !LED_3;
                    }
                }
            }
        }
    }
}

```

```

        }
    }
    MiApp_FlushTx();
    MiApp_WriteData(0x04);
    MiApp_BroadcastPacket(FALSE);
}
else
{
    tempValue = GetPushbuttons();

    if(tempValue != pushButtonValues)
    {
        buttonsNeedUpdate = TRUE;
        pushButtonValues = tempValue;
    }

    if (pushButtonValues == 0x01) // switch( PressedButton )
    {
        LED_2 = 0;
        for(i = 0;i<2;i++)
        {
            for(j = 0;j<10000;j++) //LED_2 blinks 3 times before message is sent
            {
                LED_2 = !LED_2;
            }
        }

        MiApp_FlushTx();
        MiApp_WriteData(0x01);
        MiApp_BroadcastPacket(FALSE);
    }

    if (pushButtonValues == 0x02)
    {
        LED_3 = 0;
        for(i = 0;i<2;i++)
        {
            for(j = 0;j<10000;j++) //LED_3 blinks 3 times before message is sent
            {
                LED_3 = !LED_3;
            }
        }

        MiApp_FlushTx();
        MiApp_WriteData(0x02);
        MiApp_BroadcastPacket(FALSE);
    }
}
}
}

```

/******

Function:

BYTE GetPushbuttons(void)

Summary:

Reads the current push button status.

Description:

Reads the current push button status.

Precondition:

None

Parameters:

None

Return Values:

BYTE - bitmap for button representations (1 = pressed, 0 = not pressed)
bit 0 = button 1

```
bit 1 = button 2  
bit 2 = button 3  
bit 3 = button 4
```

Remarks:

None

*****/

```
static BYTE GetPushbuttons(void)
```

```
{
```

```
    BYTE toReturn;
```

```
    // InitAllSwitches();
```

```
    toReturn = 0xFF;
```

```
    if(Switch1Pressed()){toReturn = 0x01;}
```

```
    if(Switch2Pressed()){toReturn = 0x02;}
```

```
    if(Switch3Pressed()){toReturn = 0x04;}
```

```
    return toReturn;
```

```
}
```

Figure 32: Android Accessory demo

```
/******  
  
    USB Android Accessory basic demo with accessory in host mode  
*****/  
//DOM-IGNORE-BEGIN  
/*****  
  
FileName:    main.c  
Dependencies: None  
Processor:   PIC24/dsPIC30/dsPIC33/PIC32MX  
Compiler:    C30/C32  
Company:     Microchip Technology, Inc.  
  
*****/  
  
// Include files  
#include "USB/usb.h"  
#include "USB/usb_host_android.h"  
#include "Compiler.h"  
#include "HardwareProfile.h"  
#include "HardwareProfile1.h"  
#include "ConfigApp.h"  
#include "WirelessProtocols/MCHP_API.h"  
  
/***** VARIABLES *****/  
#define LIGHT 0x01  
#define SWITCH 0x02  
#if ADDITIONAL_NODE_ID_SIZE > 0  
    BYTE AdditionalNodeID[ADDITIONAL_NODE_ID_SIZE] = {SWITCH};  
#endif  
BYTE myChannel = 0;  
  
// If a maximum current rating hasn't been defined, then define 500mA by default  
#ifndef MAX_ALLOWED_CURRENT  
    #define MAX_ALLOWED_CURRENT (500) // Maximum power we can supply in mA  
#endif  
  
// Define a debug error printing  
#define DEBUG_ERROR(a) Nop(); Nop(); Nop();  
  
// Configuration bits for the device. Please refer to the device datasheet for each device  
// to determine the correct configuration bit settings  
#ifdef __C30__  
    #if defined(__dsPIC33EP512MU810__) || defined(PIC24EP512GU810_PIM)  
        _FOSCSEL(FNOSC_FRC);  
        _FOSC(FCKSM_CSECMD & OSCIOFNC_OFF & POSCMD_XT);  
        _FWDTP(FWDTPEN_OFF);  
    #endif  
#elif defined( __PIC32MX__ )  
  
#pragma config \  
    UPLLEN = ON, \  
    UPLLDIV = DIV_1, \  
    FPLLDIV = DIV_1, \  
    FPLLMUL = MUL_20, \  
    FPLLODIV = DIV_1, \  
    POSCMOD = HS, \  
    FNOSC = PRIPLL, \  
    FPBDIV = DIV_1, \  
    FSOSCEN = OFF, \  
    FWDTPEN = OFF, \  
    WDTPS = PS1, \  
    OSCIOFNC = OFF, \  
    IESO = OFF, \  

```

```

        CP    = OFF,\
        BWP    = OFF,\
        PWP    = OFF

#else
    #error Cannot define configuration bits.
#endif

// C30 and C32 Exception Handlers
// If your code gets here, you either tried to read or write
// a NULL pointer, or your application overflowed the stack
// by having too many local variables or parameters declared.
#if defined(__C30__)
    void _ISR __attribute__((__no_auto_psv__)) _AddressError(void)
    {
        DEBUG_ERROR("Address error");
        while(1){}
    }
    void _ISR __attribute__((__no_auto_psv__)) _StackError(void)
    {
        DEBUG_ERROR("Stack error");
        while(1){}
    }

#elif defined(__C32__)
    void _general_exception_handler(unsigned cause, unsigned status)
    {
        unsigned long address = _CP0_GET_EPC();

        DEBUG_ERROR("exception");
        while(1){}
    }
#endif

//Definitions of the various application commands that can be sent
typedef enum _ACCESSORY_DEMO_COMMANDS
{
    COMMAND_SET_LEDS            = 0x01,
    COMMAND_UPDATE_PUSHBUTTONS = 0x02,
    COMMAND_UPDATE_POT          = 0x03,
    COMMAND_APP_CONNECT         = 0xFE,
    COMMAND_APP_DISCONNECT      = 0xFF
} ACCESSORY_DEMO_COMMANDS;

//Creation of the data packet that is going to be sent. In this example
// there is just a command code and a one byte data.
typedef struct __attribute__((packed))
{
    BYTE command;
    BYTE data;
} ACCESSORY_APP_PACKET;

//Local prototypes
#if defined(__C32__)
static void InitPIC32(void);
#endif

static void SetLEDs(BYTE setting);
static BYTE GetPushbuttons(void);
static BYTE ReadPOT(void);

//local variables
static BYTE read_buffer[64];
static ACCESSORY_APP_PACKET outgoing_packet;
static void* device_handle = NULL;
static BOOL device_attached = FALSE;

static char manufacturer[] = "Microchip Technology Inc.";
static char model[] = "Basic Accessory Demo";
static char description[] = DEMO_BOARD_NAME_STRING;

```

```

static char version[] = "2.0";
static char uri[] = "http://www.microchip.com/android";
static char serial[] = "N/A";

ANDROID_ACCESSORY_INFORMATION myDeviceInfo =
{
    manufacturer,
    sizeof(manufacturer),
    model,
    sizeof(model),
    description,
    sizeof(description),
    version,
    sizeof(version),
    uri,
    sizeof(uri),
    serial,
    sizeof(serial)
};

/*****
Function:
    int main(void)

Summary:
    main function

Description:
    main function

Precondition:
    None

Parameters:
    None

Return Values:
    int - exit code for main function

Remarks:
    None
*****/
int main(void)
{
    DWORD size = 0;
    BOOL responseNeeded;
    BYTE pushButtonValues = 0xFF;
    BYTE potPercentage = 0xFF;
    BOOL buttonsNeedUpdate = FALSE;
    BOOL potNeedsUpdate = FALSE;
    BOOL readyToRead = TRUE;
    BOOL writeInProgress = FALSE;
    BYTE tempValue = 0xFF;
    BYTE errorCode;
    ACCESSORY_APP_PACKET* command_packet = NULL;
    BYTE i;
        BYTE command_dsPIC;

    BOOL connected_to_app = FALSE;
    BOOL need_to_disconnect_from_app = FALSE;

    #if defined(__PIC32MX__)
        InitPIC32();
    #endif

    #if defined(__dsPIC33EP512MU810__) || defined (__PIC24EP512GU810__)

        // Configure the device PLL to obtain 60 MIPS operation. The crystal

```

```

// frequency is 8MHz. Divide 8MHz by 2, multiply by 60 and divide by
// 2. This results in Fosc of 120MHz. The CPU clock frequency is
// Fcy = Fosc/2 = 60MHz. Wait for the Primary PLL to lock and then
// configure the auxilliary PLL to provide 48MHz needed for USB
// Operation.

    PLLFBD = 38;                                /* M = 60 */
    CLKDIVbits.PLLPOST = 0;                      /* N1 = 2 */
    CLKDIVbits.PLLPRE = 0;                      /* N2 = 2 */
    OSCUN = 0;

/*    Initiate Clock Switch to Primary
 *    Oscillator with PLL (NOSC= 0x3)*/

__builtin_write_OSCCONH(0x03);
__builtin_write_OSCCONL(0x01);
while (OSCCONbits.COSC != 0x3);

// Configuring the auxiliary PLL, since the primary
// oscillator provides the source clock to the auxiliary
// PLL, the auxiliary oscillator is disabled. Note that
// the AUX PLL is enabled. The input 8MHz clock is divided
// by 2, multiplied by 24 and then divided by 2. Wait till
// the AUX PLL locks.

ACLKCON3 = 0x24C1;
ACLKDIV3 = 0x7;
ACLKCON3bits.ENAPLL = 1;
while(ACLKCON3bits.APLLCK != 1);

TRISBbits.TRISB5 = 0;
LATBbits.LATB5 = 1;

#endif
/*****/
//Initialize USB
/*****/
USBInitialize(0);
AndroidAppStart(&myDeviceInfo);

responseNeeded = FALSE;
mInitPOT();

/*****/
// Initialize board and MiWi
/*****/
BoardInit();
// InitAllSwitches();
MiApp_ProtocolInit(FALSE);

if( MiApp_SetChannel(myChannel) == FALSE )
{
}
MiApp_ConnectionMode(ENABLE_ALL_CONN);
while( (i = MiApp_EstablishConnection(0xFF, CONN_MODE_DIRECT)) == 0xFF );
DumpConnection(0xFF);

while(1)
{
    //Keep the USB stack running
    USBTasks();

    //If the device isn't attached yet,
    if(device_attached == FALSE)
    {
        buttonsNeedUpdate = TRUE;
        potNeedsUpdate = TRUE;
    }
}

```

```

    need_to_disconnect_from_app = FALSE;
    connected_to_app = FALSE;
    size = 0;

    //Reset the accessory state variables
    InitAllLEDs();

    //Continue to the top of the while loop to start the check over again.
    continue;
}

//If the accessory is ready, then this is where we run all of the demo code

if(readyToRead == TRUE)
{
    errorCode = AndroidAppRead(device_handle, (BYTE*)&read_buffer, (DWORD)sizeof(read_buffer));
    //If the device is attached, then lets wait for a command from the application
    if( errorCode != USB_SUCCESS)
    {
        //Error
        DEBUG_ERROR("Error trying to start read");
    }
    else
    {
        readyToRead = FALSE;
    }
}

size = 0;

if(AndroidAppIsReadComplete(device_handle, &errorCode, &size) == TRUE)
{
    //We've received a command over the USB from the Android device.
    if(errorCode == USB_SUCCESS)
    {
        //Maybe process the data here. Maybe process it somewhere else.
        command_packet = (ACCESSORY_APP_PACKET*)&read_buffer[0];
    }
    else
    {
        //Error
        DEBUG_ERROR("Error trying to complete read request");
    }
}

while(size > 0)
{
    if(connected_to_app == FALSE)
    {
        if(command_packet->command == COMMAND_APP_CONNECT)
        {
            connected_to_app = TRUE;
            need_to_disconnect_from_app = FALSE;
        }
    }
    else
    {
        switch(command_packet->command)
        {
            case COMMAND_SET_LEDS:
                SetLEDs(command_packet->data);
                break;

            case COMMAND_APP_DISCONNECT:
                need_to_disconnect_from_app = TRUE;
                break;

            default:
                //Error, unknown command

```

```

        DEBUG_ERROR("Error: unknown command received");
        break;
    }
}
//All commands in this example are two bytes, so remove that from the queue
size -= 2;
//And move the pointer to the next packet (this works because
// all command packets are 2 bytes. If variable packet size
// then need to handle moving the pointer by the size of the
// command type that arrived.
command_packet++;

if(need_to_disconnect_from_app == TRUE)
{
    break;
}

if(size == 0)
{
    readyToRead = TRUE;
}
/*****
// Check RxBuffer for a received message. If it exists, process the message and
// act accordingly.
*****/
    if( MiApp_MessageAvailable() )
    {
        rxMessage.PayloadSize; i++)
        {
            command_dsPIC = *rxMessage.Payload;

            if (command_dsPIC == 0x01)
            {
                pushButtonValues = 0x04;
                outgoing_packet.command = COMMAND_UPDATE_PUSHBUTTONS;
                outgoing_packet.data = pushButtonValues;

                errorCode = AndroidAppWrite(device_handle,(BYTE*)&outgoing_packet, 2);
                if( errorCode != USB_SUCCESS )
                {
                    DEBUG_ERROR("Error trying to send button update");
                }
            }

            MiApp_DiscardMessage();
        }
/*****
/*****
//Get the current pushbutton settings
tempValue = GetPushbuttons();

//If the current button settings are different than the last time
// we read the button values, then we need to send an update to the
// attached Android device
if(tempValue != pushButtonValues)
{
    buttonsNeedUpdate = TRUE;
    pushButtonValues = tempValue;
}

//Get the current potentiometer setting
tempValue = ReadPOT();

//If it is different than the last time we read the pot, then we need
// to send it to the Android device
if(tempValue != potPercentage)
{
    potNeedsUpdate = TRUE;
    potPercentage = tempValue;
}

```

```

}

//If there is a write already in progress, we need to check its status
if( writeInProgress == TRUE )
{
    if(AndroidAppIsWriteComplete(device_handle, &errorCode, &size) == TRUE)
    {
        writeInProgress = FALSE;
        if(need_to_disconnect_from_app == TRUE)
        {
            connected_to_app = FALSE;
            need_to_disconnect_from_app = FALSE;
        }

        if(errorCode != USB_SUCCESS)
        {
            //Error
            DEBUG_ERROR("Error trying to complete write");
        }
    }
}

if((need_to_disconnect_from_app == TRUE) && (writeInProgress == FALSE))
{
    outgoing_packet.command = COMMAND_APP_DISCONNECT;
    outgoing_packet.data = 0;
    writeInProgress = TRUE;

    errorCode = AndroidAppWrite(device_handle,(BYTE*)&outgoing_packet, 2);
    if( errorCode != USB_SUCCESS )
    {
        DEBUG_ERROR("Error trying to send button update");
    }
}

if(connected_to_app == FALSE)
{
    //If the app hasn't told us to start sending data, let's not do anything else.
    continue;
}

//If we need up update the button status on the Android device and we aren't
// already busy in a write, then we can send the new button data.
if((buttonsNeedUpdate == TRUE) && (writeInProgress == FALSE))
{
    outgoing_packet.command = COMMAND_UPDATE_PUSHBUTTONS;
    outgoing_packet.data = pushButtonValues;

    errorCode = AndroidAppWrite(device_handle,(BYTE*)&outgoing_packet, 2);
    if( errorCode != USB_SUCCESS )
    {
        DEBUG_ERROR("Error trying to send button update");
    }

    buttonsNeedUpdate = FALSE;
    writeInProgress = TRUE;
}

//If we need up update the pot status on the Android device and we aren't
// already busy in a write, then we can send the new pot data.
if((potNeedsUpdate == TRUE) && (writeInProgress == FALSE))
{
    outgoing_packet.command = COMMAND_UPDATE_POT;
    outgoing_packet.data = potPercentage;

    errorCode = AndroidAppWrite(device_handle,(BYTE*)&outgoing_packet, 2);
    if( errorCode != USB_SUCCESS )
    {
        DEBUG_ERROR("Error trying to send pot update");
    }
}

```

```

        potNeedsUpdate = FALSE;
        writeInProgress = TRUE;
    }
} //while(1) main loop
}

```

Function:

BOOL USB_ApplicationDataEventHandler(BYTE address, USB_EVENT event, void *data, DWORD size)

Summary:

Handles USB data application events

Description:

Handles USB data application events

Precondition:

None

Parameters:

BYTE address - address of the device causing the event
 USB_EVENT event - the event that has occurred
 void* data - data associated with the event
 DWORD size - the size of the data in the data field

Return Values:

BOOL - Return TRUE of the event was processed. Return FALSE if the event wasn't handled.

Remarks:

None

```

BOOL USB_ApplicationDataEventHandler( BYTE address, USB_EVENT event, void *data, DWORD size )
{
    return FALSE;
}

```

Function:

BOOL USB_ApplicationEventHandler(BYTE address, USB_EVENT event, void *data, DWORD size)

Summary:

Handles USB application events

Description:

Handles USB application events

Precondition:

None

Parameters:

BYTE address - address of the device causing the event
 USB_EVENT event - the event that has occurred
 void* data - data associated with the event
 DWORD size - the size of the data in the data field

Return Values:

BOOL - Return TRUE of the event was processed. Return FALSE if the event wasn't handled.

Remarks:

None

```

BOOL USB_ApplicationEventHandler( BYTE address, USB_EVENT event, void *data, DWORD size )
{
    switch( event )
    {

```

```

case EVENT_VBUS_REQUEST_POWER:
    // The data pointer points to a byte that represents the amount of power
    // requested in mA, divided by two. If the device wants too much power,
    // we reject it.
    if (((USB_VBUS_POWER_EVENT_DATA*)data)->current <= (MAX_ALLOWED_CURRENT / 2))
    {
        return TRUE;
    }
    else
    {
        DEBUG_ERROR( "\r\n***** USB Error - device requires too much current *****\r\n" );
    }
    break;

case EVENT_VBUS_RELEASE_POWER:
case EVENT_HUB_ATTACH:
case EVENT_UNSUPPORTED_DEVICE:
case EVENT_CANNOT_ENUMERATE:
case EVENT_CLIENT_INIT_ERROR:
case EVENT_OUT_OF_MEMORY:
case EVENT_UNSPECIFIED_ERROR: // This should never be generated.
case EVENT_DETACH:           // USB cable has been detached (data: BYTE, address of device)
case EVENT_ANDROID_DETACH:
    device_attached = FALSE;
    return TRUE;
    break;

// Android Specific events
case EVENT_ANDROID_ATTACH:
    device_attached = TRUE;
    device_handle = data;
    return TRUE;

default :
    break;
}
return FALSE;
}

```

/*****

Function:
 void InitPIC32(void)

Summary:
 Initialize the PIC32 core to the correct modes and clock speeds

Description:
 Initialize the PIC32 core to the correct modes and clock speeds

Precondition:
 Only runs on PIC32

Parameters:
 None

Return Values:
 None

Remarks:
 None

*****/

```

#if defined(__PIC32MX__)
static void InitPIC32(void)
{
    int value;

    #if defined(RUN_AT_60MHZ)
        // Use OSCCON default
    #else

```

```

    OSCCONCLR = 0x38000000; //PLLODIV
    #if defined(RUN_AT_48MHZ)
        OSCCONSET = 0x08000000; //PLLODIV /2
    #elif defined(RUN_AT_24MHZ)
        OSCCONSET = 0x10000000; //PLLODIV /4
    #else
        #error Cannot set OSCCON
    #endif
#endif

value = SYSTEMConfigWaitStatesAndPB( GetSystemClock() );

// Enable the cache for the best performance
CheKseg0CacheOn();

INTEnableSystemMultiVectoredInt();

DDPCONbits.JTAGEN = 0;

value = OSCCON;
while (!(value & 0x00000020))
{
    value = OSCCON; // Wait for PLL lock to stabilize
}

INTEnableInterrupts();
}
#endif

/*****
Function:
void SetLEDs(BYTE setting)

Summary:
change the LED settings of the boards

Description:
change the LED settings of the boards

Precondition:
None

Parameters:
BYTE setting - bitmap for desired LED setting (1 = On, 0 = Off)
    bit 0 = LED 0
    bit 1 = LED 1
    bit 2 = LED 2
    ...
    bit 7 = LED 7

Return Values:
None

Remarks:
None
*****/
static void SetLEDs(BYTE setting)
{
    if((setting & 0x01) == 0x01)
    {
        LED0_On();
        MiApp_FlushTx();
        MiApp_WriteData(0x01);
        MiApp_BroadcastPacket(FALSE);
    }
    else { LED0_Off(); }
    if((setting & 0x02) == 0x02)
    {
        LED1_On();
        MiApp_FlushTx();
        MiApp_WriteData(0x02);
    }
}

```

```

        MiApp_BroadcastPacket(FALSE);
    }
    else { LED1_Off(); }
    if((setting & 0x04) == 0x04) { LED2_On(); } else { LED2_Off(); }
    if((setting & 0x08) == 0x08) { LED3_On(); } else { LED3_Off(); }
    if((setting & 0x10) == 0x10) { LED4_On(); } else { LED4_Off(); }
    if((setting & 0x20) == 0x20) { LED5_On(); } else { LED5_Off(); }
    if((setting & 0x40) == 0x40) { LED6_On(); } else { LED6_Off(); }
    if((setting & 0x80) == 0x80) { LED7_On(); } else { LED7_Off(); }
}

/*****
Function:
    BYTE GetPushbuttons(void)

Summary:
    Reads the current push button status.

Description:
    Reads the current push button status.

Precondition:
    None

Parameters:
    None

Return Values:
    BYTE - bitmap for button representations (1 = pressed, 0 = not pressed)
        bit 0 = button 1
        bit 1 = button 2
        bit 2 = button 3
        bit 3 = button 4

Remarks:
    None
*****/
static BYTE GetPushbuttons(void)
{
    BYTE toReturn;

    InitAllSwitches();

    toReturn = 0;

    if(Switch1Pressed()){toReturn |= 0x1;}
    if(Switch2Pressed()){toReturn |= 0x2;}
    if(Switch3Pressed()){toReturn |= 0x4;}
    if(Switch4Pressed()){toReturn |= 0x8;}

    return toReturn;
}

/*****
Function:
    BYTE ReadPOT(void)

Summary:
    Reads the pot value and returns the percentage of full scale (0-100)

Description:
    Reads the pot value and returns the percentage of full scale (0-100)

Precondition:
    A/D is initialized properly

Parameters:
    None
*****/

```

Return Values:

None

Remarks:

None

*****/

```
static BYTE ReadPOT(void)
{
    WORD_VAL w;
    DWORD temp;

    #if defined(__18CXX)
        mInitPOT();

        ADCON0bits.GO = 1;          // Start AD conversion
        while(ADCON0bits.NOT_DONE); // Wait for conversion

        w.v[0] = ADRESL;
        w.v[1] = ADRESH;

    #elif defined(__C30__) || defined(__C32__)
        #if defined(PIC24FJ256GB110_PIM) || \
            defined(PIC24FJ256DA210_DEV_BOARD) || \
            defined(PIC24FJ256GB210_PIM)

            AD1CHS = 0x5;          //MUXA uses AN5

            // Get an ADC sample
            AD1CON1bits.SAMP = 1;    //Start sampling
            for(w.Val=0;w.Val<1000;w.Val++); //Sample delay, conversion start automatically
            AD1CON1bits.SAMP = 0;    //Start sampling
            for(w.Val=0;w.Val<1000;w.Val++); //Sample delay, conversion start automatically
            while(!AD1CON1bits.DONE); //Wait for conversion to complete

            temp = (DWORD)ADC1BUF0;
            temp = temp * 100;
            temp = temp/1023;

        #elif defined(PIC24F_ADK_FOR_ANDROID)
            AD1CHS = 0x9;          //MUXA uses AN9

            // Get an ADC sample
            AD1CON1bits.SAMP = 1;    //Start sampling
            for(w.Val=0;w.Val<1000;w.Val++); //Sample delay, conversion start automatically
            AD1CON1bits.SAMP = 0;    //Start sampling
            for(w.Val=0;w.Val<1000;w.Val++); //Sample delay, conversion start automatically
            while(!AD1CON1bits.DONE); //Wait for conversion to complete

            temp = (DWORD)ADC1BUF0;
            temp = temp * 100;
            temp = temp/1023;

        #elif defined(PIC24FJ64GB004_PIM)
            AD1CHS = 0x7;          //MUXA uses AN7

            // Get an ADC sample
            AD1CON1bits.SAMP = 1;    //Start sampling
            for(w.Val=0;w.Val<1000;w.Val++); //Sample delay, conversion start automatically
            AD1CON1bits.SAMP = 0;    //Start sampling
            for(w.Val=0;w.Val<1000;w.Val++); //Sample delay, conversion start automatically
            while(!AD1CON1bits.DONE); //Wait for conversion to complete

            temp = (DWORD)ADC1BUF0;
            temp = temp * 100;
            temp = temp/1023;

        #elif defined(PIC24F_STARTER_KIT)
            AD1CHS = 0x0;          //MUXA uses AN0

            // Get an ADC sample
```

```

    AD1CON1bits.SAMP = 1;          //Start sampling
    for(w.Val=0;w.Val<1000;w.Val++); //Sample delay, conversion start automatically
    AD1CON1bits.SAMP = 0;          //Start sampling
    for(w.Val=0;w.Val<1000;w.Val++); //Sample delay, conversion start automatically
    while(!AD1CON1bits.DONE);      //Wait for conversion to complete

    temp = (DWORD)ADC1BUF0;
    temp = temp * 100;
    temp = temp/1023;

#elif defined(PIC32MX460F512L_PIM) || defined(PIC32MX795F512L_PIM)
    AD1PCFG = 0xFFFFB; // PORTB = Digital; RB2 = analog
    AD1CON1 = 0x0000; // SAMP bit = 0 ends sampling ...
    // and starts converting
    AD1CHS = 0x00020000; // Connect RB2/AN2 as CH0 input ..
    // in this example RB2/AN2 is the input
    AD1CSSL = 0;
    AD1CON3 = 0x0002; // Manual Sample, Tad = internal 6 TPB
    AD1CON2 = 0;
    AD1CON1SET = 0x8000; // turn ADC ON

    AD1CON1SET = 0x0002; // start sampling ...
    for(w.Val=0;w.Val<1000;w.Val++); //Sample delay, conversion start automatically
    AD1CON1CLR = 0x0002; // start Converting
    while (!(AD1CON1 & 0x0001)); // conversion done?

    temp = (DWORD)ADC1BUF0;
    temp = temp * 100;
    temp = temp/1023;

#elif defined(PIC32_USB_STARTER_KIT) || \
    defined(DSPIC33E_USB_STARTER_KIT) || \
    defined(PIC32_ETHERNET_STARTER_KIT) || \
    defined(DSPIC33EP512MU810_PIM) || \
    defined(PIC24EP512GU810_PIM) || \
    defined(DSPIC33E_USB_STARTER_KIT)
    //Doesn't have a Pot
    w.Val = 50;
    temp = 50;
#else
    #error
#endif

#endif

return (BYTE)temp;
} //end ReadPOT

```

Figure 32: HardwareProfile.c

```

/*****
// Function - Initializes the board's clock and I/O ports.
//      Sets the SPI for proper functionality
*****/
#if defined(EXPLORER16)
    ANSELA = 0x0000;
    ANSELB = 0x0000;
    ANSELC = 0x0000;
    ANSELD = 0x0000;
    ANSELE = 0x0000;
    ANSELG = 0x0000;
/*****
* Look into PPS to set SDI2, SDO2, and SCK2. It is on page 192

```

```

*****/
    RPINR21 = 19;                //Mapping SDI2 to RD2
    RPOR23 = 9;                  //Mapping SDO2 to RD6
    RPOR21 = 10;                 //Mapping SCK2 to RD4
    RPOR9bits.RP101R = 0x3;

    PLLFBD = 38;                 /* M = 60 */
    CLKDIVbits.PLLPOST = 0;      /* N1 = 2 */
    CLKDIVbits.PLLPRE = 0;      /* N2 = 2 */
    OSCUN = 0;

/*    Initiate Clock Switch to Primary
 *    Oscillator with PLL (NOSC= 0x3)*/

__builtin_write_OSCCONH(0x03);
__builtin_write_OSCCONL(0x01);

    while (OSCCONbits.COSC != 0x3);
    ACLKCON3 = 0x24C1;
    ACLKDIV3 = 0x7;

    ACLKCON3bits.ENAPLL = 1;
    while(ACLKCON3bits.APLLCK != 1);
    // Make RB0 as Digital input
    //ms AD1PCFGbits.PCFG2 = 1;
    ANSELBbits.ANSB8 = 1; // Added for guessed reasons
        ANSELBbits.ANSB9 = 1;
    //    ANSELEbits.ANSE8 = 0;
    //    ANSELEbits.ANSE9 = 0;
    //    ANSELBbits.ANSB2 = 0;
    //    ANSELBbits.ANSB1 = 0;

    // A14
    // set I/O ports
    // BUTTON_1_TRIS = 1;
    // BUTTON_2_TRIS = 1;
    //BUTTON_3_TRIS = 1;
    LED_1_TRIS = 0;
    LED_2_TRIS = 0;
    LED_3_TRIS = 0;

    #if defined(MRF24J40) || defined(MRF49XA)
    PHY_CS_TRIS = 0;
    PHY_CS = 1;
    PHY_RESETEn_TRIS = 0;
    PHY_RESETEn = 1;
    #endif

    RF_INT_TRIS = 1;

    SDI_TRIS = 1;
    SDO_TRIS = 0;
    SCK_TRIS = 0;
    SPI_SDO = 0;
    SPI_SCK = 0;

    #if defined(MRF49XA)
    nFSEL_TRIS = 0;
    FINT_TRIS = 1;

    nFSEL = 1;    // nFSEL inactive
    #elif defined(MRF24J40)
    PHY_WAKE_TRIS = 0;
    PHY_WAKE = 1;
    #else
    Data_nCS_TRIS = 0;
    Config_nCS_TRIS = 0;
    Data_nCS = 1;
    //MRF89XA

```

```

Config_nCS = 1;
PHY_IRQ1_TRIS = 1;

#endif

```

Figure 33: HardwareProfile.h

```

/*****
// Function - Sets the board for proper clock frequency and pin
// definitions
*****/
#define EXPLORER16
#if defined(EXPLORER16)

    #if defined(__PIC32MX__)
        #define CLOCK_FREQ 64000000
        #define RFIF IFS0bits.INT1IF
        #define RFIE IEC0bits.INT1IE
    #else
        #define CLOCK_FREQ 8000000
        #define RFIF IFS1bits.INT1IF
        #define RFIE IEC1bits.INT1IE
    #endif

    #if defined(__PIC24F__) || defined(__PIC24FK__) || defined(__PIC32MX__)
        #define RF_INT_PIN PORTEbits.RE8
        #define RF_INT_TRIS TRISEbits.TRISE8
    #elif defined(__dsPIC33E__) || defined(__PIC24H__) || defined(__dsPIC33E__)
        #define RF_INT_PIN PORTAbits.RA14 //ms was #define RF_INT_PIN PORTAbits.RA12
        #define RF_INT_TRIS TRISAbits.TRISA14 //ms was #define RF_INT_TRIS TRISAbits.TRISA12
    #endif

    // #define USE_EXTERNAL_EEPROM

    // Transceiver Configuration
    #if (defined(MRF24J40) || defined(MRF49XA))
        #define PHY_CS LATGbits.LATG9 //was B2
        #define PHY_CS_TRIS TRISGbits.TRISG9 //was B2
        #define PHY_RESETh LATAbits.LATA2//ms was #define PHY_RESETh LATGbits.LATG2
        #define PHY_RESETh_TRIS TRISAbits.TRISA2 //ms was #define PHY_RESETh_TRIS TRISGbits.TRISG2
    #endif

    #define SPI_SDI PORTGbits.RG7 //ms was #define SPI_SDI PORTFbits.RF7
    #define SDI_TRIS TRISGbits.TRISG7 //ms was #define SDI_TRIS TRISFbits.TRISF7
    #define SPI_SDO LATGbits.LATG8 //ms was #define SPI_SDO LATFbits.LATF8
    #define SDO_TRIS TRISGbits.TRISG8 //ms was #define SDO_TRIS TRISFbits.TRISF8
    #define SPI_SCK LATGbits.LATG6 //ms was #define SPI_SCK LATFbits.LATF6
    #define SCK_TRIS TRISGbits.TRISG6 //ms was #define SCK_TRIS TRISFbits.TRISF6

    #if defined(MRF49XA)
        #define nFSEL LATBbits.LATB9
        #define nFSEL_TRIS TRISBbits.TRISB9
        #define FINT PORTAbits.RA15
        #define FINT_TRIS TRISAbits.TRISA15
        #define INT PORTAbits.RA9 // Created ourselves.
    #endif
Additional.
        #define INT_TRIS TRISAbits.TRISA9 // Created ourselves. Additional.
    #endif
    #elif defined(MRF24J40)
        #define PHY_WAKE LATAbits.LATA3
        #define PHY_WAKE_TRIS TRISAbits.TRISA3
        #else //Does not enter this loop
            #define PHY_IRQ1 IFS1bits.INT2IF
            #define PHY_IRQ1_En IEC1bits.INT2IE
            #define PHY_IRQ1_TRIS TRISEbits.TRISE9

```

```

        #define Config_nCS    LATBbits.LATB9
        #define Config_nCS_TRIS TRISBbits.TRISB9
        #define Data_nCS     LATBbits.LATB2
        #define Data_nCS_TRIS TRISBbits.TRISB2
    #endif

    // #define PUSH_BUTTON_1    PORTDbits.RD6
    // #define PUSH_BUTTON_2    PORTDbits.RD7
    //          #define PUSH_BUTTON_3          PORTDbits.RD13    //added on 11/28

    #define LED_1            LATDbits.LATD3          //changed location
    #define LED_2            LATDbits.LATD1          //changed location
    #define LED_3            LATDbits.LATD2          //Added on 11/28

    // #define BUTTON_1_TRIS    TRISDbits.TRISD6
    // #define BUTTON_2_TRIS    TRISDbits.TRISD7
    //          #define BUTTON_3_TRIS          TRISDbits.TRISD13 //Added on 11/28

    #define LED_1_TRIS        TRISDbits.TRISD3
    #define LED_2_TRIS        TRISDbits.TRISD1
    #define LED_3_TRIS        TRISDbits.TRISD2      //added on 11/28

    // Define SUPPORT_TWO_SPI if external EEPROM use the second SPI
    // port alone, not sharing SPI port with the transceiver
    // #define SUPPORT_TWO_SPI

    // External EEPROM SPI chip select pin definition
    #define EE_nCS_TRIS        TRISGbits.TRISG1 //ms was #define EE_nCS_TRIS    TRISDbits.TRISD12
    #define EE_nCS            LATGbits.LATG1 //ms was #define EE_nCS            LATDbits.LATD12

    #define TMRL TMR2
#endif

```

Figure 34: ECE480Sensor.java

```

/*****
Software License Agreement:

The software supplied herewith by Microchip Technology Incorporated
(the "Company") for its PIC(R) Microcontroller is intended and
supplied to you, the Company's customer, for use solely and
exclusively on Microchip PIC Microcontroller products. The
software is owned by the Company and/or its supplier, and is
protected under applicable copyright laws. All rights are reserved.
Any use in violation of the foregoing restrictions may subject the
user to criminal sanctions under applicable laws, as well as to
civil liability for the breach of the terms and conditions of this
license.

THIS SOFTWARE IS PROVIDED IN AN "AS IS" CONDITION. NO WARRANTIES,
WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED
TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE COMPANY SHALL NOT,
IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL OR
CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
*****/

package com.microchip.android.BasicAccessoryDemo;

import android.app.Activity;
import android.app.Notification;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.content.Context;
import android.content.Intent;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.os.Bundle;

```

```

import android.os.Handler;
import android.os.Message;
import android.util.Log;
import android.view.View;
import android.widget.TextView;
import android.widget.Toast;
import android.widget.ToggleButton;

public class ECE480Sensor extends Activity{
    private final static int USBAccessoryWhat = 0;
    private static final int DANGER_ID = 1;

    public Notification notifyDetails;

    public static final int BUTTON1_STATUS          = 1;
    public static final int BUTTON2_STATUS          = 1;

    public static final int UPDATE_LED_SETTING      = 1;
    public static final int PUSHBUTTON_STATUS_CHANGE = 2;
    public static final int POT_STATUS_CHANGE       = 3;
    public static final int APP_CONNECT              = (int)0xFE;
    public static final int APP_DISCONNECT          = (int)0xFF;

    public static final int LED_0_ON                = 0x01;
    public static final int LED_1_ON                = 0x02;
    public static final int LED_2_ON                = 0x04;
    public static final int LED_3_ON                = 0x08;
    public static final int LED_4_ON                = 0x10;
    public static final int LED_5_ON                = 0x20;
    public static final int LED_6_ON                = 0x40;
    public static final int LED_7_ON                = 0x80;

    public static final int BUTTON_1_PRESSED        = 0x01;
    public static final int BUTTON_2_PRESSED        = 0x02;
    public static final int BUTTON_3_PRESSED        = 0x04;
    public static final int BUTTON_4_PRESSED        = 0x08;

    public static final int POT_UPPER_LIMIT          = 100;
    public static final int POT_LOWER_LIMIT          = 0;

    private boolean deviceAttached = false;

    private int firmwareProtocol = 0;

    private String TAG = "MICROCHIP";

    private enum ErrorMessageCode {
        ERROR_OPEN_ACCESSORY_FRAMEWORK_MISSING,
        ERROR_FIRMWARE_PROTOCOL
    };

    private USBAccessoryManager accessoryManager;

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        try {
            PackageManager manager = this.getPackageManager();
            PackageInfo info = manager.getPackageInfo(this.getPackageName(), 0);
            Log.d(TAG, "Info:" + info.packageName + "\n" + info.versionCode + "\n" + info.versionName);
        } catch (NameNotFoundException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }

        setContentView(R.layout.main);
    }

```

```

accessoryManager = new USBAccessoryManager(handler, USBAccessoryWhat);

try {
    //Set the link to the message handler for this class
    LEDControl ledControl;

    ledControl = ((LEDControl)findViewById(R.id.led_0));
    ledControl.setHandler(handler);

} catch (Exception e) {
}

//Restore UI state from the savedInstanceState
// If the savedInstanceState Bundle exists, then there is saved data to
// restore.
if (savedInstanceState != null) {
    try {
        //Restore the saved data for each of the LEDs.
        LEDControl ledControl;

        //Buttons
        updateButton(R.id.button3,savedInstanceState.getBoolean("BUTTON3"));
        updateButton(R.id.button2,savedInstanceState.getBoolean("BUTTON2"));
        updateButton(R.id.button1,savedInstanceState.getBoolean("BUTTON1"));

        //LEDs
        ledControl = (LEDControl)findViewById(R.id.led_0);
        ledControl.setState(savedInstanceState.getBoolean("LED0"));

    } catch (Exception e) {
        //Just in case there is some way for the savedInstanceState to exist but for a single
        // item not to exist, lets catch any exceptions that might come.
    }
}

}

public void OnClickPower(View v)
{
    ToggleButton tg=(ToggleButton)v;
    if(tg.isChecked())
        DisplayToast("Checked");
    else
        DisplayToast("Not Checked");
}

@Override
public void onStart() {
    super.onStart();

    if(checkForOpenAccessoryFramework() == false){
        showErrorPage(ErrorMessageCode.ERROR_OPEN_ACCESSORY_FRAMEWORK_MISSING);
        return;
    }

    this.setTitle("Sensor Control App: Device not connected.");
}

@Override
public void onResume() {
    super.onResume();

    if(checkForOpenAccessoryFramework() == false){
        showErrorPage(ErrorMessageCode.ERROR_OPEN_ACCESSORY_FRAMEWORK_MISSING);
        return;
    }
}

```

```

        accessoryManager.enable(this, getIntent());

    }

    @Override
    public void onSaveInstanceState(Bundle savedInstanceState) {
        if(deviceAttached == false){
            return;
        }

        if(checkForOpenAccessoryFramework() == false){
            showErrorPage(ErrorMessageCode.ERROR_OPEN_ACCESSORY_FRAMEWORK_MISSING);
            //Call the super function that we are over writing now that we have saved our data.
            super.onSaveInstanceState(savedInstanceState);
            return;
        }

        //Save the UI state into the savedInstanceState Bundle.
        // We only need to save the state of the LEDs since they are the only control.
        // The state of the potentiometer and push buttons can be read and restored
        // from their current hardware state.

        savedInstanceState.putBoolean("LED0", ((LEDControl)findViewById(R.id.led_0)).getState());
        savedInstanceState.putBoolean("BUTTON1", isButtonPressed(R.id.button1));
        savedInstanceState.putBoolean("BUTTON2", isButtonPressed(R.id.button2));
        savedInstanceState.putBoolean("BUTTON3", isButtonPressed(R.id.button3));

        //Call the super function that we are over writing now that we have saved our data.
        super.onSaveInstanceState(savedInstanceState);
    }

    private boolean checkForOpenAccessoryFramework(){
        try {
            @SuppressWarnings({ "unused", "rawtypes" })
            Class s = Class.forName("com.android.future.usb.UsbManager");
            s = Class.forName("com.android.future.usb.UsbAccessory");
        } catch (ClassNotFoundException e) {
            Log.d("ClassNotFound",e.toString());
            return false;
        }
        return true;
    }

    @Override
    public void onPause() {
        if(checkForOpenAccessoryFramework() == false){
            showErrorPage(ErrorMessageCode.ERROR_OPEN_ACCESSORY_FRAMEWORK_MISSING);
            super.onPause();
            return;
        }

        switch(firmwareProtocol) {
        case 2:
            byte[] commandPacket = new byte[2];
            commandPacket[0] = (byte) APP_DISCONNECT;
            commandPacket[1] = 0;
            accessoryManager.write(commandPacket);
            break;
        }

        try {
            while(accessoryManager.isClosed() == false) {
                Thread.sleep(2000);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

```

```

    }

    super.onPause();
    accessoryManager.disable(this);
    disconnectAccessory();
}

/** Resets the demo application when a device detaches
 */
public void disconnectAccessory() {
    if(deviceAttached == false) {
        return;
    }

    Log.d(TAG,"disconnectAccessory()");

    this.setTitle("Sensor Control App: Device not connected.");

    LEDControl ledControl;

    updateButton(R.id.button3,false);
    updateButton(R.id.button2,false);
    updateButton(R.id.button1,false);

    ledControl = (LEDControl)findViewById(R.id.led_0);
    ledControl.setState(false);

    LEDButtonEnable(false);
}

/**
 * Handler for receiving messages from the USB Manager thread or
 * the LED control modules
 */
private Handler handler = new Handler() {
    @Override
    public void handleMessage(Message msg) {
        byte[] commandPacket = new byte[2];

        switch(msg.what)
        {
            case UPDATE_LED_SETTING:
                if(accessoryManager.isConnected() == false) {
                    return;
                }

                commandPacket[0] = UPDATE_LED_SETTING;
                commandPacket[1] = 0;

                if(((LEDControl)findViewById(R.id.led_0)).getState()) {
                    commandPacket[1] |= LED_0_ON;
                }

                accessoryManager.write(commandPacket);
                break;

            case USBAccessoryWhat:
                switch(((USBAccessoryManagerMessage)msg.obj).type) {
                    case READ:
                        if(accessoryManager.isConnected() == false) {
                            return;
                        }

                        while(true) {
                            if(accessoryManager.available() < 2) {

```

there are less

```
//All of our commands in this example are 2 bytes. If
// than 2 bytes left, it is a partial command
break;
}

accessoryManager.read(commandPacket);

switch(commandPacket[0]) {

    case PUSHBUTTON_STATUS_CHANGE:
        updateButton(R.id.button1, ((commandPacket[1] &
BUTTON_1_PRESSED) == BUTTON_1_PRESSED)?true:false);
        updateButton(R.id.button2, ((commandPacket[1] &
BUTTON_2_PRESSED) == BUTTON_2_PRESSED)?true:false);
        updateButton(R.id.button3, ((commandPacket[1] &
BUTTON_3_PRESSED) == BUTTON_3_PRESSED)?true:false);
        break;
    }
    break;
case CONNECTED:
    break;
case READY:
    setTitle("Sensor Control App: Device connected.");

    Log.d(TAG, "Sensor Control App :Handler:READY");

    LEDButtonEnable(true);

    String version =
((USBAccessoryManagerMessage)msg.obj).accessory.getVersion();
    firmwareProtocol = getFirmwareProtocol(version);

    switch(firmwareProtocol){
    case 1:
        deviceAttached = true;
        break;
    case 2:
        deviceAttached = true;
        commandPacket[0] = (byte) APP_CONNECT;
        commandPacket[1] = 0;
        Log.d(TAG,"sending connect message.");
        accessoryManager.write(commandPacket);
        Log.d(TAG,"connect message sent.");
        break;
    default:

        showErrorPage(ErrorMessageCode.ERROR_FIRMWARE_PROTOCOL);
        break;
    }
    break;
case DISCONNECTED:
    disconnectAccessory();
    break;
}

break;
default:
    break;
}
} //handleMessage
}; //handler

private int getFirmwareProtocol(String version) {

    String major = "0";

    int positionOfDot;
```

```

        positionOfDot = version.indexOf('.');
        if(positionOfDot != -1) {
            major = version.substring(0, positionOfDot);
        }

        return new Integer(major).intValue();
    }

    private void updateButton(int id, boolean pressed) {
        TextView textviewToUpdate;

        textviewToUpdate = (TextView)findViewById(id);

        //If pressing the button for functioning
        if(id==2131165189)
        {
            if(pressed)
            {
                textviewToUpdate.setBackgroundResource(R.color.notfunctioning);
                textviewToUpdate.setText(R.string.pressed);
            } else {
                textviewToUpdate.setBackgroundResource(R.color.functioning);
                textviewToUpdate.setText(R.string.not_pressed);
            }
        }

        //Else button for battery
        else if (id==2131165190)
        {
            if(pressed)
            {
                textviewToUpdate.setBackgroundResource(R.color.notfunctioning);
                textviewToUpdate.setText(R.string.low_battery);
            } else {
                textviewToUpdate.setBackgroundResource(R.color.functioning);
                textviewToUpdate.setText(R.string.good_battery);
            }
        }

        //Push button 3 for Checmical levels
        if(id==2131165191)
        {
            //Dangerous Chemical Levels
            if(pressed)
            {
                //Give reference to notification
                String ns = Context.NOTIFICATION_SERVICE;
                NotificationManager mNotificationManager = (NotificationManager)
getSystemService(ns);

                //Instantiate the notification
                int icon = R.drawable.spartan_logo;
                CharSequence tickerText = "CHEMICAL LEVELS WARNING";
                long when = System.currentTimeMillis();

                Notification notification = new Notification(icon, tickerText, when);

                //Define notification message
                Context context = getApplicationContext();
                CharSequence contentTitle = "Dangerous Chemical Levels";
                CharSequence contentText = "Dangerous Checmical Levels";
                Intent notificationIntent = new Intent(this, ECE480Sensor.class);
                PendingIntent contentIntent = PendingIntent.getActivity(this, 0, notificationIntent, 0);

                notification.setLatestEventInfo(context, contentTitle, contentText, contentIntent);
            }
        }
    }

```

```

        //Set sound
        notification.defaults |= Notification.DEFAULT_SOUND;
        //Make sure sound does not stop until notification is cleared.
        notification.flags = Notification.FLAG_INSISTENT;

        mNotificationManager.notify(DANGER_ID, notification);

        //Display dangerous levels text and change background to red
        textViewToUpdate.setBackgroundResource(R.color.notfunctioning);
        textViewToUpdate.setText(R.string.danger);
    } else {
        //Remove text and set background to default
        textViewToUpdate.setBackgroundResource(R.color.background);
        textViewToUpdate.setText("");
    }
}

//Check if button is pressed
private boolean isButtonPressed(int id)
{
    //Create variables
    TextView buttonTextView;
    String buttonText;

    buttonTextView = ((TextView)findViewById(id));
    buttonText = buttonTextView.getText().toString();
    return buttonText.equals(getString(R.string.pressed));
}

private void LEDButtonEnable(boolean enabled) {
    // Set the link to the message handler for this class
    LEDControl ledControl;

    ledControl = ((LEDControl) findViewById(R.id.led_0));
    ledControl.setEnabled(enabled);
}

//Error pages if something goes wrong
private void showErrorPage(ErrorMessageCode error){
    setContentView(R.layout.error);

    TextView errorMessage = (TextView)findViewById(R.id.error_message);

    switch(error){
    case ERROR_OPEN_ACCESSORY_FRAMEWORK_MISSING:
        errorMessage.setText(getResources().getText(R.string.error_missing_open_accessory_framework));
        break;
    case ERROR_FIRMWARE_PROTOCOL:
        errorMessage.setText(getResources().getText(R.string.error_firmware_protocol));
        break;
    default:
        errorMessage.setText(getResources().getText(R.string.error_default));
        break;
    }
}

//Easy function to display messages.
private void DisplayToast(String msg)
{
    Toast.makeText(getBaseContext(), msg,
        Toast.LENGTH_SHORT).show();
}

} //Class definitions

```

LEDcontrol.java

Software License Agreement:

The software supplied herewith by Microchip Technology Incorporated (the "Company") for its PIC(R) Microcontroller is intended and supplied to you, the Company's customer, for use solely and exclusively on Microchip PIC Microcontroller products. The software is owned by the Company and/or its supplier, and is protected under applicable copyright laws. All rights are reserved. Any use in violation of the foregoing restrictions may subject the user to criminal sanctions under applicable laws, as well as to civil liability for the breach of the terms and conditions of this license.

THIS SOFTWARE IS PROVIDED IN AN "AS IS" CONDITION. NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE COMPANY SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.

```
package com.microchip.android.BasicAccessoryDemo;
```

```
import android.content.Context;
import android.graphics.Canvas;
import android.os.Handler;
import android.os.Message;
import android.util.AttributeSet;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.ImageButton;
```

```
/** Class to control a custom button representing an LED
```

*

* @author Microchip Technology Inc.

*

 $\ast/$

```
public class LEDControl extends ImageButton implements OnClickListener{
```

```
/* Constant value sent to the GUI handler in the "what" feild */
```

```
public static final int UPDATE_LED_SETTING = 1;
```

```
/* Boolean indicating the current LED status */
```

```
private boolean on;
```

```
/* Boolean that indicates if the LED button should be active */
```

```
private boolean enabled;
```

```
/* Reference to the user interface handler */
```

```
private Handler uiHandler;
```

```
/** Creates instance of the LED button
```

*

* **@param** context The associated context

*

```
public LEDControl(Context context) {
```

```
super(context);
```

```
init();
```

}

```
/** Creates instance of the LED button
```

*

* **@param** context The associated context

* **@param** attrs The associated AttributeSet for this item

*

```
public LEDControl(Context context, AttributeSet attrs) {
```

```
super(context, attrs);
```

```

        init();
    }

    /** Initialized the internal state variables and registers the
     * click listener for this button
     */
    private void init() {
        on = false;
        enabled = false;
        this.setOnClickListener(this);
        uiHandler = null;
    }

    /** Set the user interface handler where this class should
     * send messages to when a press event occurs
     * @param handler If a GUI wants to be notified of this event, it can
     * send it's handler here (only supports one handler).
     */
    public void setHandler(Handler handler) {
        uiHandler = handler;
    }

    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
    }

    public void onClick(View view) {
        if(enabled == true) {
            setState(!on);
        }
    }

    public void setEnabled(boolean state) {
        enabled = state;
    }

    /** Gets the current state of the button
     *
     * @return boolean that indicates the current button state, true = on
     */
    public boolean getState() {
        return on;
    }

    /** Sets the current state of the button
     *
     * @param state true = on, false = off
     */
    public void setState(boolean state) {

        //save the new value for the LED from the parameter passed in.
        on = state;

        if(state == false) {
            //If the user is turning off the LED, then
            // set the image resource for the button to the "led_off" image
            this.setImageResource(R.drawable.power_off);
            //ToggleButton led_0 = (ToggleButton) findViewById(R.id.led_0);

        } else {
            //If the user is turning on the LED, then
            // set the image resource for the button to the "led_on" image
            this.setImageResource(R.drawable.power_on);
        }
    }

```

```

        Message ledUpdate = Message.obtain(uiHandler, UPDATE_LED_SETTING);

        if(uiHandler != null) {
            uiHandler.sendMessage(ledUpdate);
        }
    }
}

```

USBAccessoryManager.java

/*

 Software License Agreement:
 The software supplied herewith by Microchip Technology Incorporated
 (the "Company") for its PIC(R) Microcontroller is intended and
 supplied to you, the Company's customer, for use solely and
 exclusively on Microchip PIC Microcontroller products. The
 software is owned by the Company and/or its supplier, and is
 protected under applicable copyright laws. All rights are reserved.
 Any use in violation of the foregoing restrictions may subject the
 user to criminal sanctions under applicable laws, as well as to
 civil liability for the breach of the terms and conditions of this
 license.
 THIS SOFTWARE IS PROVIDED IN AN "AS IS" CONDITION. NO WARRANTIES,
 WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED
 TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE COMPANY SHALL NOT,
 IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL OR
 CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
 *****/
 package com.microchip.android.BasicAccessoryDemo;
 import java.io.FileInputStream;
 import java.io.FileOutputStream;
 import java.io.IOException;
 import java.util.ArrayList;
 import android.app.PendingIntent;
 import android.content.BroadcastReceiver;
 import android.content.Context;
 import android.content.Intent;
 import android.content.IntentFilter;
 import android.os.Handler;
 import android.os.ParcelFileDescriptor;
 import android.util.Log;
 import com.android.future.usb.UsbAccessory;
 import com.android.future.usb.UsbManager;
 /**
 * A class created to assist in making accessing a USB accessory easier for
 * those that are less familiar with programming in Java, working with
 * threads/handlers/synchronization, and those that are not familiar with the
 * Open Accessory framework interface
 *
 * @author Microchip Technology Inc.
 */
 public class USBAccessoryManager {
 private String actionString = null;
 private Handler handler;
 private int what;
 private boolean enabled = false;
 private boolean permissionRequested = false;
 private boolean open = false;
 private FileOutputStream outputStream = null;

```

private ParcelFileDescriptor parcelFileDescriptor = null;
private ReadThread readThread = null;

private ArrayList<byte[]> readData = new ArrayList<byte[]>();

private String TAG = "MICROCHIP";

/*****
/** Public API **/
*****/

/**
 * Creates new USB Accessory Manager
 *
 * @param handler
 *      The handler where to send USB accessory event messages
 * @param what
 *      The "what" value to use for USB accessory event messages
 */
public USBAccessoryManager(Handler handler, int what) {
    this.handler = handler;
    this.what = what;
}

/**
 * Enumeration of possible return values for the enable function
 */
public enum RETURN_CODES {
    DEVICE_MANAGER_IS_NULL, ACCESSORIES_LIST_IS_EMPTY,
    FILE_DESCRIPTOR_WOULD_NOT_OPEN, PERMISSION_PENDING, SUCCESS
}

/**
 * Enables the
 *
 * @param context
 *      The context that the USB manager should register to
 * @return RETURN_CODES - the status of the enable request
 */
public RETURN_CODES enable(Context context, Intent intent) {
    // Grab the packageName to use for an attach Intent
    actionString = context.getPackageName() + ".action.USB_PERMISSION";

    PendingIntent permissionIntent = PendingIntent.getBroadcast(context, 0,
        new Intent(actionString), 0);

    // If the USB manager isn't already enabled
    if (enabled == false) {
        // Create a new filter with the package name (for the accessory
        // attach)

        try {
            Thread.sleep(500);
        } catch (InterruptedException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        }

        IntentFilter filter = new IntentFilter(actionString);
        // Also add a few other actions to the intent...
        filter.addAction(UsbManager.ACTION_USB_ACCESSORY_ATTACHED);
        filter.addAction(UsbManager.ACTION_USB_ACCESSORY_DETACHED);
        // and register the intent with the specified context
        context.registerReceiver(receiver, filter);

        UsbManager deviceManager = null;
        UsbAccessory[] accessories = null;
        UsbAccessory accessory = null;

        // Get a UsbManager object from the specified intent (only works for

```

```

// v2.3.4)
deviceManager = UsbManager.getInstance(context);

// If we were unable to get a UsbManager, return an error
if (deviceManager == null) {
    return RETURN_CODES.DEVICE_MANAGER_IS_NULL;
}

// Get a list of all of the accessories from the UsbManager
accessories = deviceManager.getAccessoryList();

// If the list of accessories is empty, then exit
if (accessories == null) {
    return RETURN_CODES.ACCESSORIES_LIST_IS_EMPTY;
}

// Get the first accessory in the list (currently the Android OS
// only
// supports one accessory, so this is it)
accessory = accessories[0];

// If the accessory isn't null, then let's try to attach to it.
if (accessory != null) {
    // If we have permission to access the accessory,
    if (deviceManager.hasPermission(accessory)) {
        // Try to open a ParcelFileDescriptor by opening the
        // accessory
        parcelFileDescriptor = deviceManager
            .openAccessory(accessory);

        if (parcelFileDescriptor != null) {
            // Create a new read thread to handle reading data from
            // the accessory
            readThread = new ReadThread(parcelFileDescriptor);
            readThread.start();

            deviceManager.requestPermission(accessory,
                permissionIntent);

            if(parcelFileDescriptor == null) {
                Log.d(TAG, "USBAccessoryManager:enable()

parcelFileDescriptor == null");

                return
                RETURN_CODES.FILE_DESCRIPTOR_WOULD_NOT_OPEN;
            }

            // Open the output file stream for writing data out to
            // the accessory
            outputStream = new FileOutputStream(
                parcelFileDescriptor.getFileDescriptor());

            if(outputStream == null) {
                Log.d(TAG, "USBAccessoryManager:enable()

outputStream == null");

                try {
                    parcelFileDescriptor.close();
                } catch (IOException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                return
                RETURN_CODES.FILE_DESCRIPTOR_WOULD_NOT_OPEN;
            }

            Log.d(TAG,
                "USBAccessoryManager:enable()

outputStream open");

            // If the ParcelFileDescriptor was successfully opened,

```

```

        // mark the accessory as enabled and open
        enabled = true;
        open = true;

        handler.obtainMessage(
            what,
            new USBAccessoryManagerMessage(

USBAccessoryManagerMessage.MessageType.READY,

accessory)).sendToTarget();

        Log.d(TAG,
            "USBAccessoryManager:enable() device

ready");

        return RETURN_CODES.SUCCESS;
    } else {
        /*
         * If we weren't able to open the ParcelFileDescriptor,
         * then we will not be able to talk to the device. Due
         * to a bug in the Android v2.3.4 OS this situation may
         * occur if a user presses the "home" or "back" buttons
         * while an accessory is still attached. In this case
         * the attempt to close the ReadThread will fail if a
         * read() is in progress on the FileInputStream. This
         * results in the ParcelFileDescriptor not being freed
         * up for later access. A future attempt to connect to
         * the accessory (via reopening the app) will end up
         * having the openAccessory() request return null,
         * ending up in this section of code.
         */
        return
RETURN_CODES.FILE_DESCRIPTOR_WOULD_NOT_OPEN;
    }
    } else {
        /*
         * If we don't currently have permission to access the
         * accessory, then we need to request it. If we haven't
         * requested it already...
         */
        if (permissionRequested == false) {
            // Then go ahead and request it...
            deviceManager.requestPermission(accessory,
                permissionIntent);

            permissionRequested = true;
            return RETURN_CODES.PERMISSION_PENDING;
        }
    }

    return RETURN_CODES.ACCESSORIES_LIST_IS_EMPTY;
}

return RETURN_CODES.SUCCESS;
}

/**
 * Disables the USB manager and releases all resources
 *
 * @param context
 *     The context that the manager was enabled with
 */
public void disable(Context context) {
    // Free up all of the required resources
    closeAccessory();

    // Unregister the broadcast receiver
    try {

```

```

        context.unregisterReceiver(receiver);
    } catch (Exception e) {
    }
}

/**
 * Describes if an accessory is attached or not
 *
 * @return boolean - true if one is attached, false otherwise
 */
public boolean isConnected() {
    return open;
}

/** I/O API *****/

/**
 * discards the specified number of bytes from the internal read buffer
 *
 * @param num
 *     the number of bytes to discard
 */
void ignore(int num) {
    int amountRead = 0;

    // If the accessory is not connected, then we can't really do anything
    if (isConnected() == false) {
        // throw new USBAccessoryManagerException(
        //     USB_ACCESSORY_NOT_CONNECTED );
        return;
    }

    // Must request to ignore 1 or more bytes, otherwise do nothing
    if (num <= 0) {
        return;
    }

    // synchronizing to the readData object so that the ReadThread and this
    // don't try to access the readData object at the same time. This will
    // block whoever tries to access it second until it is available.
    synchronized (readData) {
        // Keep reading through the data objects until one of the many
        // internal
        // checks decides to exit. This each entry in this loop either
        // returns
        // or removed objects from the list (which ultimately causes a
        // return),
        // this loop should always exit.
        while (true) {
            // If there are no more data objects in the list, exit
            if (readData.size() == 0) {
                return;
            }

            // If we have ignored the requested amount of data from the
            // buffer, then exit
            if (amountRead == num) {
                return;
            }

            if ((num - amountRead) >= readData.get(0).length) {
                // We need to ignore equal to or greater than the size of
                // data in this entry

                // then ignore the whole size of the entry
                amountRead += readData.get(0).length;

                // and remove it from the list.
                readData.remove(0);
            } else {

```

```

        // We need to ignore only a portion of the data in this
        // entry

        // only remove a portion of the entry, leaving part of the
        // data
        int amountRemoved = num - amountRead;
        byte[] newData = new byte[readData.get(0).length
            - amountRemoved];

        // Copy the remaining data in to a new buffer
        for (int i = 0; i < newData.length; i++) {
            newData[i] = readData.get(0)[i + amountRemoved];
        }

        // remove the old entry
        readData.remove(0);

        // add a new entry to the front of the buffer with the
        // remaining data
        readData.add(0, newData);

        // since we have now read all of the data that we need, exit
        return;
    }
}

/**
 * fills the array with data from the read buffer without discarding it.
 *
 * @param array
 *         the buffer to fill
 * @return the number of bytes copied from the buffer
 */
int peek(byte[] array) {
    int amountRead = 0;
    int currentNode = 0;

    // If an accessory is not connected, this request is invalid
    if (isConnected() == false) {
        // throw new USBAccessoryManagerException(
        //     USB_ACCESSORY_NOT_CONNECTED );
        return 0;
    }

    // If there is no data available, then return 0 bytes read
    if (array.length == 0) {
        return 0;
    }

    /**
     * Synchronize to the readData object so that the ReadThread doesn't
     * attempt to add more data to the readData list while we are accessing
     * it.
     */
    synchronized (readData) {
        /** While we still have data to read */
        while (true) {
            /**
             * if we have run out of list objects to peek from... (no data
             * left to peek)
             */
            if (currentNode >= readData.size()) {
                /** then return the amount that we have read */
                return amountRead;
            }

            /**
             * if the amount that we have read exactly matches the amount

```

```

        * that we need
        */
        if (amountRead == array.length) {
            return amountRead;
        }

        if ((array.length - amountRead) >= readData.get(currentNode).length) {
            // If the amount we need to read is larger than or equal to
            // the data buffer for this node
            for (int i = 0; i < readData.get(currentNode).length; i++) {
                array[amountRead++] = readData.get(currentNode)[i];
            }
            currentNode++;
        } else {
            // If the amount we need to read is less than the data
            // buffer for this node
            for (int i = 0; i < (array.length - amountRead); i++) {
                array[amountRead++] = readData.get(currentNode)[i];
            }
        }

        return amountRead;
    }
}

/**
 * Indicates the number of bytes that are currently in the read buffer.
 * There will be at least this many bytes to read from the buffer (as long
 * as the accessory has not detach or been closed since the call to this
 * function.
 *
 * @return the number of bytes available in the read queue
 */
int available() {
    int amount = 0;

    // If the accessory is not connected, then this request is invalid
    if (isConnected() == false) {
        // throw new USBAccessoryManagerException(
        // USB_ACCESSORY_NOT_CONNECTED );
        return 0;
    }

    /*
     * Synchronize to the readData object so that the ReadThread doesn't try
     * to add data the list while we are accessing it.
     */
    synchronized (readData) {
        for (byte[] b : readData) {
            amount += b.length;
        }
    }

    return amount;
}

/**
 * Reads bytes from the read buffer, removing them from the buffer once read
 *
 * @param array
 *         where to copy the data
 * @return the number of bytes copied (maximum will be the length of the
 *         array param
 */
int read(byte[] array) {
    int amountRead = 0;

    /* If an accessory is not connected, this request is not valid */
    if (isConnected() == false) {

```

```

        // throw new USBAccessoryManagerException(
        // USB_ACCESSORY_NOT_CONNECTED );
        return 0;
    }

    /* if there is no data to read, return 0 instead of blocking */
    if (array.length == 0) {
        return 0;
    }

    /*
     * Synchronize to the readData object so that the ReadThread doesn't try
     * to add data to the list while we are accessing it.
     */
    synchronized (readData) {
        /* while we still have data to read */
        while (true) {
            /*
             * if there is no data left in the list, exit with the amount
             * read already
             */
            if (readData.size() == 0) {
                return amountRead;
            }

            /*
             * if we have read all of the data that we can store in the
             * buffer provided, then exit.
             */
            if (amountRead == array.length) {
                return amountRead;
            }

            if ((array.length - amountRead) >= readData.get(0).length) {
                /*
                 * If the amount we need to read is larger than or equal to
                 * the data buffer for this node, then copy what is here and
                 * remove it from the list
                 */
                for (int i = 0; i < readData.get(0).length; i++) {
                    array[amountRead++] = readData.get(0)[i];
                }
                readData.remove(0);
            } else {
                // If the amount we need to read is less than the data
                // buffer for this node
                int amountRemoved = 0;

                /* then copy what we need */
                for (int i = 0; i < (array.length - amountRead); i++) {
                    array[amountRead++] = readData.get(0)[i];
                    amountRemoved++;
                }

                /* create a new buffer the size of the remaining data */
                byte[] newData = new byte[readData.get(0).length
                    - amountRemoved];

                /* copy the remaining data to that buffer */
                for (int i = 0; i < newData.length; i++) {
                    newData[i] = readData.get(0)[i + amountRemoved];
                }

                /*
                 * remove the old object and add a new object with the
                 * remaining data
                 */
                readData.remove(0);
                readData.add(0, newData);
            }
        }
    }

```

```

        return amountRead;
    }
}

/**
 * Writes data to the accessory
 *
 * @param data
 *      the data to write
 * @throws InterruptedException
 */
public void write(byte[] data) {
    int tries = 2;

    if (isConnected() == true) {
        if (outputStream != null) {
            /*
             * we try here a few times because on first attachment the
             * Android pipe doesn't appear to be completely open yet and
             * calling write causes an exception in the first ~1 second
             * after the devices attachment. Data sent in that first second
             * may not get sent without retrying.
             */
            while (tries-- > 0) {
                try {
                    outputStream.write(data);
                } catch (IOException e) {
                    Log.d(TAG,

"USBAccessoryManager:write():IOException: "

                                + e.toString());

                    try {
                        Thread.sleep(2000);
                    } catch (InterruptedException e1) {
                        // TODO Auto-generated catch block
                        e1.printStackTrace();
                    }
                }
            }
        }
    }
}

/***** Private section *****/
/*****
// Create a BroadcastReceiver for Bluetooth events
private final BroadcastReceiver receiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        /* get the action for this event */
        String action = intent.getAction();

        /*
         * if it corresponds to the packageName, then it was a permissions
         * grant request
         */
        if (actionString.equals(action)) {
            /* see if we got permission */
            if (intent.getBooleanExtra(UsbManager.EXTRA_PERMISSION_GRANTED,
                false)) {
                /* if so, then try to open the accessory */
                UsbManager deviceManager = null;
                UsbAccessory[] accessories = null;
                UsbAccessory accessory = null;

                deviceManager = UsbManager.getInstance(context);

```

```

        if (deviceManager == null) {
            // TODO: error. report to user?
            return;
        }

        accessories = deviceManager.getAccessoryList();

        if (accessories == null) {
            // TODO: error. report to user?
            return;
        }

        accessory = accessories[0];

        parcelFileDescriptor = deviceManager
            .openAccessory(accessory);

        if (parcelFileDescriptor != null) {
            enabled = true;
            open = true;

            outputStream = new FileOutputStream(
                parcelFileDescriptor.getFileDescriptor());

            readThread = new ReadThread(parcelFileDescriptor);
            readThread.start();

            Log.d(TAG,

"USBAccessoryManager:BroadcastReceiver()-1");

            handler.obtainMessage(
                what,
                new USBAccessoryManagerMessage(

USBAccessoryManagerMessage.MessageType.READY,

accessory)).sendToTarget();

                } else {
                    // TODO: error. report to user?
                    return;
                }
            }
        }

        if (UsbManager.ACTION_USB_ACCESSORY_ATTACHED.equals(action)) {
            /*
             * if it was a device attach notice, then try to open the
             * accessory
             */
            UsbManager deviceManager = null;
            UsbAccessory[] accessories = null;
            UsbAccessory accessory = null;

            deviceManager = UsbManager.getInstance(context);

            if (deviceManager == null) {
                // TODO: error. report to user?
                return;
            }

            accessories = deviceManager.getAccessoryList();

            if (accessories == null) {
                // TODO: error. report to user?
                return;
            }

            accessory = accessories[0];

            parcelFileDescriptor = deviceManager.openAccessory(accessory);

```

```

        if (parcelFileDescriptor != null) {
            enabled = true;
            open = true;

            outputStream = new FileOutputStream(
                parcelFileDescriptor.getFileDescriptor());

            readThread = new ReadThread(parcelFileDescriptor);
            readThread.start();

            Log.d(TAG, "USBAccessoryManager:BroadcastReceiver()-2");
            handler.obtainMessage(
                what,
                new USBAccessoryManagerMessage(
                    USBAccessoryManagerMessage.MessageType.READY,
                                                                accessory)).sendToTarget();
        } else {
            // TODO: error. report to user?
            return;
        }
    } else if (UsbManager.ACTION_USB_ACCESSORY_DETACHED.equals(action)) {
        /*
         * if it was a detach notice, then close the accessory and
         * notify the user
         */
        closeAccessory();
        handler.obtainMessage(
            what,
            new USBAccessoryManagerMessage(
                USBAccessoryManagerMessage.MessageType.DISCONNECTED))
            .sendToTarget();
    } else if (UsbManager.EXTRA_PERMISSION_GRANTED.equals(action)) {
    }
}

};

/**
 * Closes the accessory and cleans up all loose ends
 */
private void closeAccessory() {
    if (open == false) {
        return;
    }

    open = false;
    enabled = false;
    permissionRequested = false;

    if (readThread != null) {
        readThread.cancel();
    }

    if (outputStream != null) {
        try {
            outputStream.close();
        } catch (IOException e) {
        }
    }

    if (parcelFileDescriptor != null) {
        try {
            parcelFileDescriptor.close();
        } catch (IOException e) {

```

```

        }
    }

    outputStream = null;
    // readThread = null;
    parcelFileDescriptor = null;
}

public boolean isClosed() {
    boolean isAlive;

    if (readThread != null) {
        isAlive = readThread.isAlive();

        if (isAlive == false) {
            readThread = null;
        }

        return isAlive;
    }

    return true;
}

/**
 * The thread for reading data from the FileInputStream so the main thread
 * doesn't get blocked.
 */
private class ReadThread extends Thread {
    private boolean continueRunning = true;

    private FileInputStream inputStream;
    private ParcelFileDescriptor myparcelFileDescriptor;

    public ReadThread(ParcelFileDescriptor p) {
        myparcelFileDescriptor = p;
        inputStream = new FileInputStream(p.getFileDescriptor());
    }

    @Override
    public void run() {
        byte[] buffer = new byte[1024]; // buffer store for the stream
        int bytes; // bytes returned from read()

        while (continueRunning) {
            try {
                // Read from the InputStream
                bytes = inputStream.read(buffer);

                // Send the obtained bytes to the UI Activity
                byte[] data = new byte[bytes];
                System.arraycopy(buffer, 0, data, 0, bytes);

                /*
                 * Synchronize to the readData object to make sure that no
                 * user API is accessing it at the moment
                 */
                synchronized (readData) {
                    readData.add(data);
                }

                handler.obtainMessage(
                    what,
                    bytes,
                    -1,
                    new USBAccessoryManagerMessage(
                        USBAccessoryManagerMessage.MessageType.READ,
                        data)).sendToTarget();
            } catch (IOException e) {

```

```

// Exiting read thread
break;
    }
}

public void cancel() {
    continueRunning = false;
    try {
        inputStream.close();
    } catch (IOException e) {
    }

    try {
        myparcelFileDescriptor.close();
    } catch (IOException e) {
    }
}
}

/***** Exception definition section *****/
/***** Exception definition section *****/
// private final static String USB_ACCESSORY_NOT_CONNECTED =
// "USB Accessory is not attached.";

/**
 * Exception that can be thrown by the manager - currently not used
 */
class USBAccessoryManagerException extends RuntimeException {
    private static final long serialVersionUID = 2329617898883120248L;
    String errorMessage;

    public USBAccessoryManagerException(String message) {
        errorMessage = message;
    }

    public String toString() {
        return errorMessage;
    }
}
}

```

USBAccessoryManagerMessage.java

```

/*****

```

Software License Agreement:

The software supplied herewith by Microchip Technology Incorporated (the "Company") for its PIC(R) Microcontroller is intended and supplied to you, the Company's customer, for use solely and exclusively on Microchip PIC Microcontroller products. The software is owned by the Company and/or its supplier, and is protected under applicable copyright laws. All rights are reserved. Any use in violation of the foregoing restrictions may subject the user to criminal sanctions under applicable laws, as well as to civil liability for the breach of the terms and conditions of this license.

THIS SOFTWARE IS PROVIDED IN AN "AS IS" CONDITION. NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE COMPANY SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.

```

*****/

```

```

package com.microchip.android.BasicAccessoryDemo;

```

```

import com.android.future.usb.UsbAccessory;

/** Basic Message class for the USBAccessoryManager. This is used
 * to send messages from the USB Accessory's read thread to the
 * GUI thread to notify the GUI thread of various USB Accessory
 * events (like data available or device attachment).
 *
 * @author Microchip Technologies Inc.
 */
public class USBAccessoryManagerMessage {
    /** Types of messages that can be sent */
    public enum MessageType {
        READ,
        ERROR,
        CONNECTED,
        DISCONNECTED,
        READY
    };

    /** The MessageType for this message instance */
    public MessageType type;
    /** Any text information that needs to be sent with data */
    public String text = null;
    /** Data send in the read MessageType */
    public byte[] data = null;
    /** A USB accessory that attached */
    public UsbAccessory accessory = null;

    /** Creates new message of specified type
     *
     * @param type The type of this message
     */
    public USBAccessoryManagerMessage(MessageType type) {
        this.type = type;
    }

    /** Creates a new message of specified type with specified data
     *
     * @param type The type of this message
     * @param data The data associated with this message
     */
    public USBAccessoryManagerMessage(MessageType type, byte[] data) {
        this.type = type;
        this.data = data;
    }

    /** Creates a new message of specified type with specified data
     *
     * @param type The type of this message
     * @param data The data associated with this message
     * @param accessory The accessory associated with this message
     */
    public USBAccessoryManagerMessage(MessageType type, byte[] data, UsbAccessory accessory) {
        this.type = type;
        this.data = data;
        this.accessory = accessory;
    }

    /** Creates a new message of specified type with specified data
     *
     * @param type The type of this message
     * @param accessory The accessory associated with this message
     */
    public USBAccessoryManagerMessage(MessageType type, UsbAccessory accessory) {
        this.type = type;
        this.accessory = accessory;
    }
}

```

Main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <!-- Title bar for Functioning Yes/No -->

    <LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:background="@color/title_bar"
        android:orientation="horizontal" >

        <TextView
            android:id="@+id/text_view"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="@string/functioning_title"
            android:textSize="20sp"
            android:textStyle="bold" />
    </LinearLayout>

    <!-- Status of functioning Yes/No -->

    <LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >

        <TextView
            android:id="@+id/button1"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:layout_marginBottom="10sp"
            android:background="@color/functioning"
            android:gravity="center_vertical|center_horizontal"
            android:text="@string/not_pressed"
            android:textColor="#000"
            android:textSize="30sp" />
    </LinearLayout>

    <!-- Title bar for Low Battery -->

    <LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:background="@color/title_bar"
        android:orientation="horizontal" >

        <TextView
            android:id="@+id/text_view"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="@string/Battery_title"
            android:textSize="20sp"
            android:textStyle="bold" />
    </LinearLayout>

    <!-- Status of Battery -->

    <LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >
```

```

<TextView
    android:id="@+id/button2"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="10sp"
    android:background="@color/functioning"
    android:gravity="center_vertical|center_horizontal"
    android:text="@string/good_battery"
    android:textColor="#000"
    android:textSize="30sp" />
</LinearLayout>

<!-- Title bar for potentiometer -->

<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:background="@color/title_bar"
    android:orientation="horizontal" >

    <TextView
        android:id="@+id/text_view"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/chemical_level"
        android:textSize="20sp"
        android:textStyle="bold" />
</LinearLayout>

<!-- Chemical Level Status -->

<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal" >

    <TextView
        android:id="@+id/button3"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="10sp"
        android:gravity="center_vertical|center_horizontal"
        android:textColor="#000"
        android:textSize="30sp" />
</LinearLayout>

<!-- Title bar for Power -->

<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:background="@color/title_bar"
    android:orientation="horizontal" >

    <TextView
        android:id="@+id/TextView01"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/PowerHeader"
        android:textSize="20sp"
        android:textStyle="bold" />
</LinearLayout>

<!-- Power On/Off Button -->

<LinearLayout

```

```

xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:gravity="center"
android:orientation="horizontal"
android:paddingTop="10sp" >

<view
    android:id="@+id/led_0"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:background="@null"
    class="com.microchip.android.BasicAccessoryDemo.LEDControl"
    android:src="@drawable/power_off" />
</LinearLayout>

</LinearLayout>

Strings.xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name"> ECE480 Team 4: Sensor Control Application</string>
    <string name="led_header">LED Controls 1</string>
    <string name="functioning_title">Sensor Functioning Properly?</string>
    <string name="Battery_title">Battery Indicator</string>
    <string name="chemical_level">Chemical Levels</string>
    <string name="button_1">Functioning:</string>
    <string name="button_2">Battery:</string>
    <string name="not_pressed">Yes</string>
    <string name="pressed">No</string>
    <string name="good_battery">Good Battery</string>
    <string name="low_battery">Battery Low</string>
    <string name="PowerHeader">Turn Power On/Off</string>
    <string name="danger">Dangerous</string>
    <string name="error_default">An unknown error has occurred. Please contact ECE480t4@gmail.com so that we can
determine the source of the issue and attempt to fix it.</string>
    <string name="error_missing_open_accessory_framework">The Android OS version that you are running is missing the
com.android.future.usb library that is required to run this demo. The OS version that you are running should contain these
libraries. Please contact ECE480t4@gmail.com so that we can work to correct this issue for this device. </string>
    <string name="error_firmware_protocol">The firmware version on the attached device requires a protocol that is not
supported by this version of the application. Please check the Android Marketplace to see if there is an update available for
this application.</string>
</resources>

```