

# **A MOBILE DEVICE-CONTROLLED BLOOD PRESSURE MONITOR**

---

A Thesis

Presented to the

Faculty of

San Diego State University

---

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Andrew Joseph Luxner

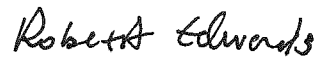
Spring 2013

**SAN DIEGO STATE UNIVERSITY**

The Undersigned Faculty Committee Approves the

Thesis of Andrew Joseph Luxner:

A Mobile Device-Controlled Blood Pressure Monitor



---

Robert A. Edwards, Chair  
Department of Computer Science



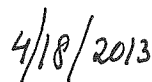
---

Roger Whitney  
Department of Computer Science



---

Mahasweta Sarkar  
Department of Electrical and Computer Engineering



---

Approval Date

Copyright © 2013

by

Andrew Joseph Luxner

All Rights Reserved

## **DEDICATION**

This thesis is dedicated to the open source projects and their communities that provide the building blocks and jump-starts to countless solutions around the world: Amarino, Arduino, Apache, Android, Eclipse, Libomron... and many more.

## **ABSTRACT OF THE THESIS**

A Mobile Device-Controlled Blood Pressure Monitor

by

Andrew Joseph Luxner

Master of Science in Computer Science

San Diego State University, 2013

High blood pressure, or hypertension, is a serious condition that can cause damage to the heart and other organs and increase the risk of heart attack and stroke. The question is when should hypertension be treated with medication? Most people have the condition at least occasionally, such as while at the doctor's office. It would be helpful to have a convenient way to automatically take many blood pressure readings throughout the day and over time to see how often one's blood pressure is high. Such a device, called an Ambulatory Blood Pressure Monitor (ABPM), could also be valuable to researchers looking to correlate the instances of heart disease, stroke, heart attack, and other ailments with sufferers' blood pressure readings over time. Such devices exist, but often cost thousands of dollars and are too bulky to use conveniently. Individual blood pressure monitoring and new research studies would be aided by ABPMs whose technology makes them cheaper and easier to use.

Mobile devices such as smart phones are ideal candidates to control such devices. They have ample processing power and wireless capability. Applications are easily installed on them, and their use is widespread. People can install an application that could wirelessly control a blood pressure monitor, display the readings on the mobile device, and transmit the readings to a central server for further use. For example, the data could be used as part of a study in which many users' readings are analyzed to detect patterns in blood pressure fluctuations and determine the significance of such fluctuations. Also, the data, once transmitted, could be accessed via a web page so that doctors could conveniently check their patients' readings.

This thesis documents the creation of an ABPM solution, including an investigation into some alternative technologies. An off-the-shelf blood pressure monitor was purchased and hacked into in order to allow control by an Arduino board (an open-source electronics device). A Bluetooth module, attached to the Arduino, facilitates two-way communication between the Arduino and the mobile device (an Android phone) which displays the readings. Finally, the phone sends the blood pressure readings to a RESTful web service running on a remote computer where the data are stored and can be accessed via a web page

## TABLE OF CONTENTS

	PAGE
ABSTRACT.....	v
LIST OF FIGURES.....	viii
CHAPTER	
1 INTRODUCTION .....	1
1.1 Mobile Devices as Part of a Solution .....	2
1.2 Current Availability of a Mobile Solution.....	3
1.3 Security and Patient Confidentiality .....	3
2 TECHNOLOGY REVIEW AND RESEARCH.....	4
2.1 Android.....	4
2.2 iOS.....	4
2.2 Arduino.....	5
2.3 Wireless Technology: ZigBee vs. Bluetooth, WiFi .....	5
2.4 Blood Pressure Monitors .....	6
2.5 Web Technologies.....	7
2.6 Android's ADK.....	8
3 METHODOLOGY .....	12
3.2 Non-ADK Bluetooth .....	12
3.3 Preparing the Bluetooth module .....	14
3.4 Problems uploading code to the Arduino board .....	15
3.5 User Interface.....	15
3.6 Web Service for data storage and collation .....	16
3.7 A Simple way to Store and Access the data .....	19
3.8 Interacting with a Blood Pressure Monitor.....	19
3.9 Integrating Arduino Programs .....	23
4 SECURITY .....	26
4.1 Securing Web Traffic .....	26
4.2 Bluetooth Issues .....	26

4.2 Patient Confidentiality.....	27
5 SUMMARY AND CONCLUSIONS .....	28
REFERENCES.....	31
APPENDIX	
A SOLUTION SOURCE CODE .....	36

## LIST OF FIGURES

	PAGE
Figure 1. Solderless breadboard.....	14
Figure 2. Bluetooth module with headers soldered.....	15
Figure 3. Basic Android user interface running in an emulator.....	16
Figure 4. Convert reading object to a list of name/value pairs. ....	18
Figure 5. Sending data via RESTful web service using libraries in Android SDK. ....	18
Figure 6. JSP page displaying saved BP readings. ....	19
Figure 7. Test pads on back of circuit boards. ....	21
Figure 8. Test pads after soldering wires.....	21
Figure 9. Arduino UNO board connected to BP monitor.....	22
Figure 10. Close-up of Arduino ADK with power connected at right. ....	23
Figure 11. Solution overview.....	25



## CHAPTER 1

### INTRODUCTION

Many types of blood pressure monitors exist, including both wrist monitors and the more common arm band monitors. The arm band monitors are more cumbersome because they typically are larger, require a separate module tethered to the arm band, and are less convenient than the wrist monitors [1]. The wrist monitors are self-contained – they have no separate module. The arm band monitors are still more prevalent because they've been around longer and have the reputation of being more accurate than the newer wrist monitors. According to the American Heart Association, "The American Heart Association recommends an automatic, cuff-style, bicep (upper-arm) monitor. Wrist and finger monitors are not recommended because they yield less reliable readings" [2]. Both types of monitors can be purchased for under \$45.

Ambulatory Blood Pressure Monitors (ABPMs) take readings continuously to collect many readings from patients to see how their blood pressure varies throughout the day. This is important because a doctor may unnecessarily prescribe costly medication to a patient who only has high blood pressure when readings are taken in the doctor's office (a condition known as White Coat Syndrome). Britain's National Health Service has begun using ABPMs to help prevent misdiagnosis [3]. It's well-known that people's blood pressure (BP) varies a lot during the day but it's debatable what to do about it. If someone's blood pressure is normal when taken at home in the morning but then reads high when taken later the same day in a doctor's office – what does this show? If the patient's BP is mostly high then perhaps it should be treated, but if it only occasionally spikes up during moments of stress then perhaps nothing should be done, especially since every prescription medication used to treat the condition has side effects. Unfortunately many patients with high BP are misdiagnosed and prescribed unnecessary, even harmful medication [4]. Doctors do not want to over or under-treat. An ambulatory tool that provides many reading throughout the day would give a truer picture of patients' blood pressure. Such a device could also be a research tool to help better understand the effects of blood pressure on the body. ABPMs typically cost hundreds or even

thousands of dollars [5] and are often loaned to patients by clinics or hospitals [6]. Patients would likely benefit from an off-the-shelf ABPM that is easy to use and not so costly, as they wouldn't need to depend on their health care providers to loan them expensive units. Additionally, cheap, reliable ABPMs would allow researchers to more easily conduct studies that help determine the effects of blood pressure changes. For example, do wild swings in blood pressure make one more susceptible to stroke or heart attack than one with a consistently slightly-elevated blood pressure? How much do substances like alcohol or caffeine elevate one's blood pressure throughout the day?

### **1.1 MOBILE DEVICES AS PART OF A SOLUTION**

The objective of the thesis is to show the potential for mobile devices to interact with devices external to them for the purpose of providing biomedical-related feedback – in this case blood pressure readings. There are two reasons why this mobile device/blood pressure monitor interaction is desirable:

1. The blood pressure device could be smaller because the user interface and control logic could be pushed off to the mobile device, which many people carry with them at all times anyway. An advantage of smaller blood pressure devices is that they support an ambulatory monitor. Readings could be taken continuously during the day with minimum inconvenience to the user.
2. Nearly all mobile devices have internet access, meaning that the devices could – immediately after taking blood pressure readings – send the readings to a central server as part of a larger study or to provide reports to a doctor.

Mobile devices, especially smart phones and tablet computers, are ubiquitous in society and come with many built-in sensors such as accelerometers, GPS devices, compasses, and light sensors. However, the potential for applications will not be realized if mobile application developers limit themselves to only the built-in sensors. Imagine applications such as Geiger counters, RFID readers, or even thermometers; all are well-suited for use with mobile devices but mobile devices will never ship with the sensor hardware necessary to support these specialized applications. A blood pressure application is another good use for a mobile device/external hardware solution because no mobile device is going to come with the hardware necessary to take blood pressure readings. Such hardware requires a means to restrict the blood flow to a user's arm or wrist, a procedure that typically involves a pump that inflates a cuff wrapped around a person's arm. The cuff device is external to the mobile device but must communicate with the mobile device: sending readings and receiving

commands to start and stop the procedure. Finally, the mobile device could use a web service to transmit the data to a central server. The data collected by a combination of the mobile device and external device can be saved at a central server for the purpose of collation and analysis. For example, many users could participate in a study that could advance the understanding of the “normal” range of blood pressure fluctuation during the day. The study could correlate blood pressure variations with the incidence of heart attacks and strokes among the participants.

The challenge for this project is determining how best to achieve the mobile device/external device communication, and developing a sample application. Any solution will have several components; the development of each requires considering alternative approaches. A working prototype has been developed that uses recent mobile technology, hacked off-the-shelf blood pressure monitors, and trial-and-error integration techniques in an effort to determine a feasible approach to the development of a mobile device-controlled ABPM.

## **1.2 CURRENT AVAILABILITY OF A MOBILE SOLUTION**

Very few blood pressure monitors are directly compatible with common mobile devices such as iPods and Android devices. Some can store their blood pressure readings, which can then be uploaded to mobile devices via a docking station [7]. One can provide direct communication with an iPod, but only when attached by a cable [8]. And these existing devices do not take continuous readings automatically as do ABPMs.

## **1.3 SECURITY AND PATIENT CONFIDENTIALITY**

The solution involves transmitting health data wirelessly and over the internet. The choice of software, hardware, and several design decisions affect how securely data can be transmitted. Storing users’ data on a server also touches on security and confidentiality issues. The issues become more sensitive for solutions intended for end consumers rather than for research purposes. In any case, these issues were examined and are discussed in Chapter 4.

## **CHAPTER 2**

### **TECHNOLOGY REVIEW AND RESEARCH**

This chapter identifies the main technologies that were considered for use in a solution and describes any research done to evaluate the technologies.

#### **2.1 ANDROID**

Mobile devices with an Android operating system accounted for over 72% of worldwide mobile device sales towards the end of 2012 [9]. Besides its popularity, Android seemed a promising platform for the project in part because of its Accessory Development Kit (ADK) [10-11]. According to Google, “The Accessory Development Kit (ADK) is a reference implementation for hardware manufacturers and hobbyists to use as a starting point for building accessories for Android.” Google’s website encourages development using this protocol.

The protocol was released at the Google IO conference in May of 2011, and updated at the Google IO conference in June 2012. The Android protocol is compelling because the necessary libraries are open-source and easy to integrate into an existing Android Software Development Kit (SDK) (a fundamental component in Android development). Additionally, the ADK is a standard part of the Android operating system which means that, at least in theory, all Android devices created today are compatible with this protocol, which allows for rapid acceptance. The ADK technology is discussed further at the end of this chapter.

#### **2.2 IOS**

Apple and devices running the iOS operating system shares dominance of new sales in the mobile device market with Android [9]. The platform generally offers the same features as Android; in fact iOS was the first to offer a framework for accessory development with its External Accessory Framework. But the standard was not adopted quickly because in order to develop external accessory apps developers had to conform to Apple’s MFi program which could be difficult and costly [12]. When Android made it easy to address external

development in 2011 with the ADK (as described above) Apple responded later that same year with a similar framework that involved a Serial Cable for iOS from Redpark [13-14].

## **2.2 ARDUINO**

Arduino [15] is an open-source electronics prototyping platform [16]. It's designed to support rapid-prototyping of electronics devices. It also can act as a bridge between a mobile device such as an Android phone and a sensor such as a Geiger counter or a thermometer. Anything that can be attached to an Arduino board can potentially communicate with an Android phone. Arduino boards provide a microcontroller, EEPROM memory to store data between uses, connections for powering the board, and many digital and analog input "pins" that make it easy to connect wires to the board with no soldering. Arduino provides a free IDE that allows developers to write programs and upload them to the board.

The downside of Arduino boards is that, because they're meant to be easily used in many projects, they contain a lot of hardware that a single project doesn't need. For example, an Arduino board may provide twenty input pins and an EEPROM chip, but many projects need only a few digital pins and no EEPROM. The boards are therefore much bigger than they have to be for a finished project. Additionally, while a basic board usually costs around \$25 or \$30 the components actually needed for the finished product (such as a microcontroller and a circuit board) may only cost \$5, so the boards are not practical for a mass-produced product. The intent of Arduino is that the boards are for prototyping. Therefore, when it's time to transition the prototype to a finished product more work will need to be done.

## **2.3 WIRELESS TECHNOLOGY: ZIGBEE VS. BLUETOOTH, WIFI**

For wireless solutions many Arduino developers turn to the ZigBee protocol with XBee radios because of the low cost and the ease of creating a wireless network [17]. ZigBee is also well-suited for situations involving small amounts of data, which fits this project [18]. Neither Android nor iOS devices support the protocol out-of-the-box although workarounds and adapters are available. But both mobile operating systems do support Bluetooth [19], [20]. Because of this and the heavy use of Bluetooth technology in wireless smart phone

project tutorials it was assumed that any wireless solution for the project would involve Bluetooth.

The vision from the start of this research was to have an external device communicating with a mobile device, and have the mobile device controlling the external device, displaying the data and sending it along to a central server. It should be possible to eliminate the mobile device altogether, however, by using an Arduino WiFi Shield [21]. This device is an add-on to Arduino for the purpose of connecting the Arduino board to the internet. Data from the blood pressure monitor could be directly sent to a server. This solution would either push all the display responsibility to the blood pressure monitor or make the display completely web-based, with no app to install on any mobile device and no display on the blood pressure device. Users could retrieve data via a web interface. The mobile phone is either eliminated from the solution or it could interact with the monitor indirectly through web service calls. This would work, but not without problems. First, eliminating the mobile device and leaving all logic and display on the blood pressure monitor loses the flexibility of quick updates to the user interface that can be done with a downloaded phone app. It would also force the blood pressure monitor to be more complex and probably larger. Second, using web methods from the mobile app forces each side of the solution (monitor side and phone side) to constantly invoke web service methods to check for updates and to determine whether to stop or start the device. This constant use of a web service would likely be too hard on battery usage.

There are other issues that need to be considered such as the requirement of a WiFi connection and the use of specific wireless routers that means users would have to enter an encryption key or at least a password for every new WiFi environment; this is not practical unless the monitor is only to be used in one location.

## **2.4 BLOOD PRESSURE MONITORS**

The only feasible, non-invasive means of monitoring blood pressure outside of a hospital setting involves using an inflatable cuff placed around a person's arm to stop the flow of blood in the artery [22]. There are emerging techniques to take blood pressure readings without the use of artery constriction [23] but such methods are not commercially available. Inflating the cuff automatically requires a pump and the development of

algorithms and circuitry to control the pump. Logic must be written to determine readings based on the release of pressure in the cuff. For these and several other problems involved building a blood pressure monitor from scratch is beyond the scope of the project. The hope was that one or more existing off-the-shelf blood pressure monitors would have a simple interface that would allow programmatic control and access to the unit's memory, perhaps via a USB cable. Unfortunately there are no such monitors on the market.

The most feasible alternative for building a prototype is to find an existing blood pressure monitor and open it up, or “hack” into it, for the purpose of controlling it and reading its data. None of the off-the-shelf models are intended to be opened by their users, and none were found to easily allow programmatic control by another application. But some are able to be controlled by grounding certain points on the circuit board or snooping in on certain points when data is written to the monitor's memory. These techniques will be discussed.

There is also a project called Libomron that provides a partial solution for interacting with some blood pressure monitors [24]. The Libomron library allows USB access to the readings saved on certain Omron monitors (Omron is a manufacturer of blood pressure monitors). The project is written in the C programming language and provides a driver to access a monitor's data. Unfortunately, the library is not able to control the monitor – such as turning it on or off – it is just able to access its data. However, simply stopping and starting monitors seems to be much easier than accessing their memory, so Libomron was of potential use to the project. However, the library was not written to run on an Arduino board and the prospect of adapting or re-writing the library for Arduino was not promising enough to pursue.

## 2.5 WEB TECHNOLOGIES

The project requires a way to easily send data from a mobile device or desktop computer to a location where they can be processed and saved. It also requires a way to access the data. There are many web technologies that could be used for this. Most would involve a web server such as Tomcat, Jetty, or IIS. These servers are able to both accept and serve data in a variety of formats. This area of the project has a lot of flexibility for third-party tools to use since the end user experience would be nearly the same in most cases. In this case good technologies for a prototype are ones that facilitate quick development.

## 2.6 ANDROID'S ADK

The Arduino platform was chosen because it is compatible with the Android ADK, and thus an Arduino board [11] and introductory “Getting Started” book [16] were purchased to prototype development. The book is structured like a tutorial that gets the reader started developing programs that run on a chip built into a palm-sized board. The programs interact with sensors such as light sensors and switches. The projects in this book do not involve mobile devices, but the Arduino board does interact, via a USB cable, with other programs running on a computer. Completing the exercises in the book require readers to get their feet wet building basic circuits and using electronic equipment such as capacitors, switches, LEDs, light sensors, etc.

Next an attempt was made to get a program running on an Arduino board to communicate with an Android phone. Knowing that the ADK was introduced for Android 2.3.4 the assumption was that any new phone would be ready for ADK development. In June of 2012 a Samsung Galaxy SII Skyrocket with operating system 2.3.6 was purchased. The carrier was AT&T. A “Hello World” application was rapidly built, trying to get the Arduino ADK board to recognize the new phone. The simple application was based on a project in a book on working with Android ADK and Arduino [25]. The Arduino program did indeed recognize the phone, but the phone failed to supply its protocol (a requirement of the ADK). The error persisted: “Data packet error: 5could not read device protocol version”. Using the built-in Android debugging tools, including DDMS, and after much digging, mainly in the developer forums, it became obvious that the then-new phone, that was assumed to support the ADK when purchased, did not have one of the required Java libraries (Jar files) called `com.android.future.usb.accessory.jar`. The jar file does not come with Android's Software Development Kit (SDK). Google-approved Android devices (such as Samsung's Nexus line) have the software already on the phone, so there is no need to install it from elsewhere. However, many phones (including the Skyrocket used) do not have this jar file included. The program was also tested with a slightly older Samsung Galaxy SII, model SGH-T989, running operating system 2.3.5; the carrier was T Mobile, Build number GINGERBREAD.UVKID.

As a side note, when reading about procedures for a mobile device, or when describing interactions with a device to a third-party, it is essential to be clear on the model,



Android version, and often the Kernel version and Build number. These things can be found on most Android phones by accessing Applications → Settings → About phone . Like the Samsung Skyrocket used, the Galaxy SII did not have the `com.android.future.usb.accessory.jar` file installed. The file was found on a developer forum, and also through the Cyanogenmod 7.2.0 release [26].

After downloading this library the Android Debug Bridge was used to push the file onto the phone. On both phones the location of this file should be in the phone's `system/framework` directory. Pushing these files is different from installing an application (.apk file) on the phone. Placing a java library onto the phone seemed to be required in this case yet very few users of phones have the experience of performing such an operation. In fact, the copy initially could not be done because in order to copy non-application files (non-.apk files) onto a phone in the appropriate directory one must be a “root” user. The Android OS is a version of Unix but the owner of the phone does not have the root password to the operating system.

The process of gaining root access is called “rooting” or “unlocking” your phone (on the iOS this is called “Jailbreaking”). The procedure is risky because, for the most part, the phone manufacturers do not want users to unlock their phones and therefore offer no information on how to do so. Instead, developers must scour online forums [27-28] for tips on how to unlock the phones. There used to be nothing illegal about doing this, but it may be illegal currently. Rooting the phone, however, will likely void any warranty and could “brick” the phone (damage it to the point of inoperability). Following the steps to root a phone in one of these forums means trusting a hacker that one doesn't know, hoping that his or her advice will provide root access and not damage one's phone.

To further complicate things, consider the fact that the unlock procedure is different for nearly every phone, even for phones from the same manufacturer. Sometimes even the same phone from the same manufacturer will have a different unlock procedure because the manufacturers update their hardware and software frequently. This problem was encountered while following a guide on rooting the Skyrocket. Scanning replies to the post revealed that, although the guide seemed to work for many users with operating system 2.3.4, the same steps did not work for those that had the slightly different operating system 2.3.6.

An attempt was made to root the T.Mobile Galaxy following advice in a developer forum/ YouTube video. The content has since been removed from YouTube, but the eight minute video, in conjunction with further details at an xda-developers web forum [29] explained how to unlock the phone. The steps involved installing some software called Odin that helps “flash” files to the phone. There were a couple of other files to download from sites such as goo.im and mediafire.com. The file, named su-3.0.5-efgh-signed.zip, has since been removed but can be found elsewhere. The YouTube video steps were followed and were risky since they involved installing unknown software from an unknown source and allowing it to fundamentally alter the phone. However the procedure completed as described.

Even after that procedure the jar file still could not be pushed to the phone even while running as root user. Even with this SuperUser application installed attempts to figure out how to change file permissions were unsuccessful (the normal unix/linux means did not work). An application called Root Explorer was purchased for \$3.99 from the Google Play app store. The application allows one to browse the entire directory structure of the phone and change permissions; it’s like a file manager app that runs directly on the phone. The jar file had to be added to the /system/framework directory and a file called android.hardware.usb.accessory.xml had to be added to the /etc/permissions directory. Both of these files were included in the cyanogenmod download. After transferring these files the error “Package helloworld.adk requires unavailable shared library com.android.future.usb.accessory; failing!” was still encountered until the phone was rebooted. Then the Hello World ADK application could be run on the phone. And yet the “Hello World” Arduino application was still unable to determine the phone’s protocol.

It’s assumed that buying a Google-approved phone will work with the ADK, so if using the ADK were a hard requirement for the project that would have been the next avenue to pursue. However, because of frustration with the ADK it was decided to evaluate some non-ADK options, especially Bluetooth. ADK technology does not seem to be a viable option for developers since there doesn’t seem to be much push from carriers or phone manufacturers to make sure their phones are compatible with this “standard”.

The latest ADK specification has support for Bluetooth [11]. However, only ADK 2012 supports Bluetooth, and ADK 2012 only works with devices running Android 4.1, that very few devices were running during the time of this development. Furthermore, due to

failed attempts to get a phone to work with Arduino via the ADK, other ways of using Bluetooth were pursued.

## **CHAPTER 3**

### **METHODOLOGY**

This chapter details what was actually done to achieve a working prototype. The development effort addressed the problem by breaking an overall solution into functional components, including:

- phone/BP monitor communication
- phone/server communication
- phone user interface
- web user interface
- controlling the BP monitor, and
- parsing the BP monitor results

Android was chosen as the platform for mobile development, but research into the ADK yielded disappointing results, so the chapter begins with attempts to use a non-ADK means of sending messages between a phone and an external device. The chapter concludes with a diagram showing an overview of the finished solution.

### **3.2 NON-ADK BLUETOOTH**

Amarino [30] is a promising Bluetooth-related technology. According to Amarino's homepage "Amarino is a toolkit to connect Android-driven mobile devices with Arduino microcontrollers via Bluetooth". The toolkit provides easy access to internal phone events that can be further processed on the Arduino open-source prototyping platform. The Amarino project began as a master's thesis by Bonifaz Kaufmann [31]. The project includes an Android application that is used to detect Bluetooth applications that are in range of the phone. The software helps "pair" the Bluetooth module with the Android device. Amarino's code, in part, builds on the Android SDK's `android.bluetooth` package. The result of the pairing is that an ID is generated for the Bluetooth device [32]. Incorporating this ID in your Android application allows your Android app to communicate with the Bluetooth module.

This means that two Android apps are needed: (1) the downloaded Amarino app that takes care of pairing the Bluetooth module with your phone and (2) your own Android application for your specific project. This Amarino app seems to be a convenience for getting started with Bluetooth projects, and using this app seems like a good initial step. However, a finished, commercial project would want to eliminate a dependency on another application.

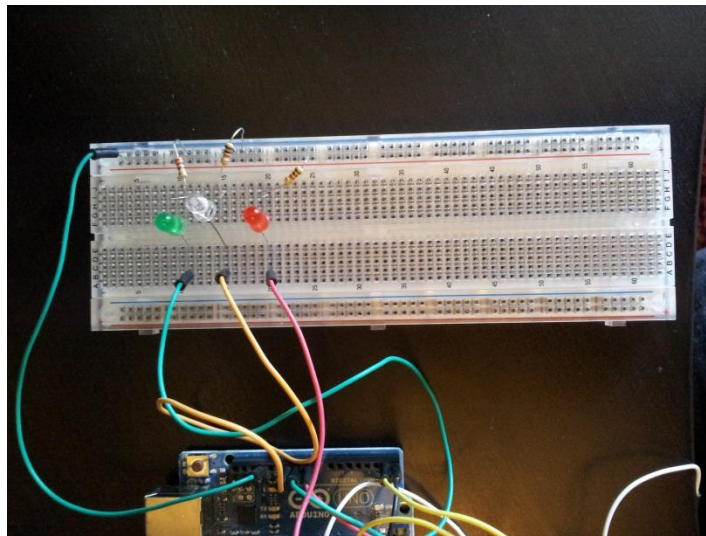
A Bluetooth Module was purchased from Sparkfun Electronics, a leading provider of Arduino and other supplies for electronics projects [33]. The Amarino app was installed from a download page hosted at Google Code [34] and a package that includes the Arduino portion of the solution along with tutorial examples was downloaded from Amarino [35]. A second Amarino Android app, called `AmarinoPluginBundle.apk` was also necessary to install on the phone [34]. The app does not include an icon; there is no indication that it has been installed and does not appear in the phone's list of "Applications". But it serves as a startup for Amarino projects and allows developers to add custom events needed for communication. Finally, Amarino includes a java library that must be included in an Android project so that the app can use Amarino functionality[36]. To summarize, the main components needed for Amarino are:

1. Arduino library (referenced when the Arduino project is compiled)
2. Android app with an interface that explicitly allows users to pair Bluetooth devices
3. a second Android app (the `AmarinoPluginBundle`) to install directly on the phone
4. Amarino jar file to include in each Android app developed with Amarino as part of the solution.

As a proof-of-concept test of the basic Amarino technology three tutorials were followed: a "Hello World"-like application [37], a Sensor Graph application that continuously transmits the level of light falling on a sensor from an Arduino board to a phone [38], and a third that allows an Android phone to control lights attached to an Arduino Uno [39]. The technology worked as hoped that led to the decision to pursue a solution involving Bluetooth technology. Later some of this logic from the Amarino tutorials was adapted and integrated with other logic responsible for working with the BP monitor.

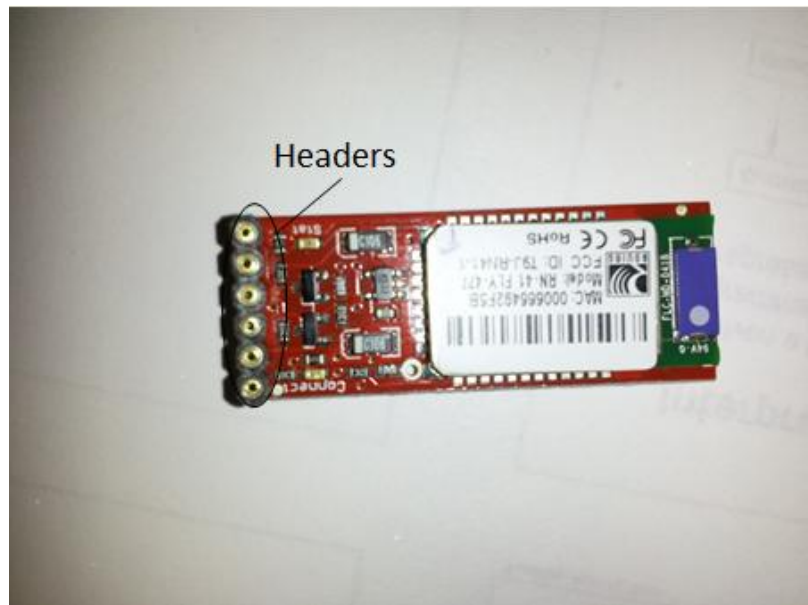
### 3.3 PREPARING THE BLUETOOTH MODULE

A frustrating fact to one with little experience with electronics is that any non-trivial electronics project is eventually going to require soldering (connecting wires to surfaces to form circuits by melting metal in between surfaces so that electrical current can flow). The project's initial work with Arduino did not require any soldering because of the use of a solderless breadboard of the type shown in Figure 1. These breadboards make sense for prototyping because they allow one to quickly connect components without soldering.



**Figure 1. Solderless breadboard.**

Most Bluetooth modules require one to solder the module to the wires that connect it to the circuit board. For more versatility the six connectors of the module were soldered to headers, as shown in Figure 2. The holes in the headers allow connecting wires to be plugged and unplugged easily without additional soldering. The module used in the project and shown in Figure 2 is smaller than a typical stick of gum. The Bluetooth modem's RX (receive) and TX (transmit) connectors were attached to the Arduino's TX and RX pins, respectively, so that the wireless data sent to and from the modem can reach the Arduino circuitry. The Bluetooth's GND (ground) was connected to one of the few Arduino GND pins; the modem's VCC (voltage) was connected to the Arduino's 5 volt pin so that the Arduino can power the Bluetooth modem. The two remaining connectors, labeled RTS (Request To Send) and CTS (Clear To Send) support hardware flow control [17], but were not needed for the



**Figure 2. Bluetooth module with headers soldered.**

application. The Amarino tutorials recommended simply joining the two connectors (the top and bottom holes in Figure 2's Headers) with a single wire.

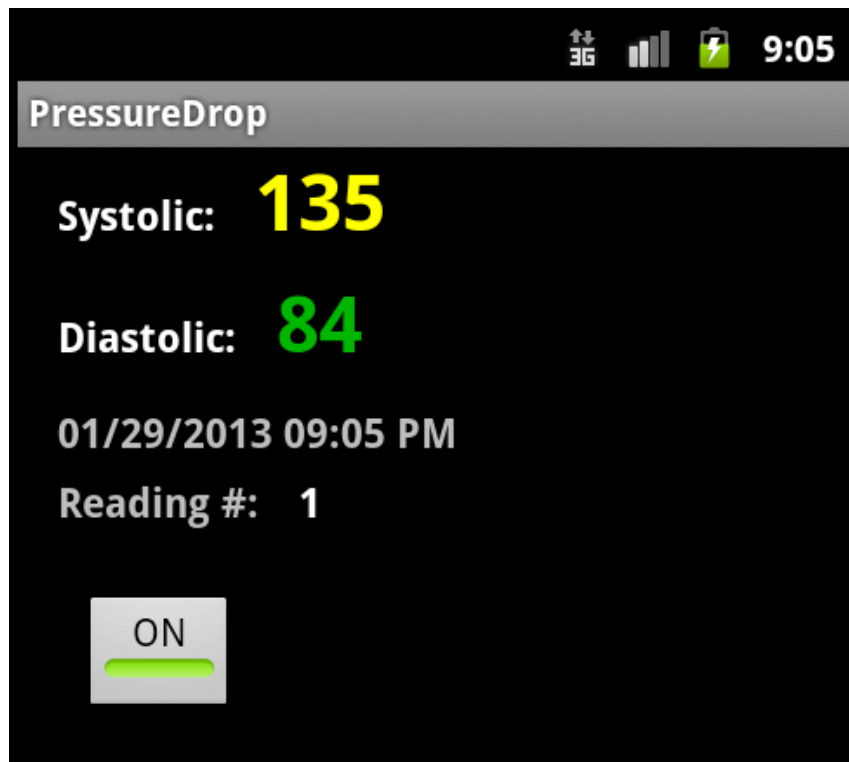
### **3.4 PROBLEMS UPLOADING CODE TO THE ARDUINO BOARD**

During testing with the Bluetooth module a frustrating roadblock kept occurring. Attempts to upload programs to the Arduino board kept failing for no obvious reason. Even shutting down and restarting the Arduino software did not correct the problem. Finally it was determined that it's impossible to upload an Arduino program while a Bluetooth module is attached to the Arduino board. Uploading the Bluetooth device requires first disconnecting the Bluetooth module by unplugging the wire going to Arduino's voltage pin. After doing this the program loads immediately. The Bluetooth module does something to the Arduino board that prevents programs from being uploaded while it is active.

### **3.5 USER INTERFACE**

A basic user interface for the Android App was developed using Eclipse as an IDE. Logic was added to color-code the systolic and diastolic readings based on their values: for systolic, any value up to 125 is coded green, between 126 and 135 (inclusive) will show as yellow, and above 135 shows as red. For diastolic, values up to 85 are green, between 85 and 90 (inclusive) display as yellow; values above 90 are red. These thresholds were based on the

National Heart, Lung, and Blood Institute’s “Categories for Blood Pressure Levels in Adults” [40]; perhaps an added feature would make the thresholds configurable. Figure 3 below is a screen-shot that shows the user interface running in an Android emulator.



**Figure 3. Basic Android user interface running in an emulator.**

### **3.6 WEB SERVICE FOR DATA STORAGE AND COLLATION**

Besides creating useful interaction between a mobile device and a medical device, another goal of the project is to transmit the data to a central location so it can be viewed/tracked/collated/processed by researchers, doctors, or analysts. The format of the collected data will change based on the application, but the basic needs are constant:

1. Server-side means of accepting data from the mobile devices
2. Client-side means of sending the data efficiently

The obvious solution to support both 1 and 2 above is to use some form of web service so that the data can be transmitted from device to server in a standardized way via the internet. Traditional web services use SOAP, which relies on XML. These kinds of services have been used reliably for many years and there are a variety of toolkits available to ease development. However, a couple key factors led to the decision to not use traditional SOAP-based web services. One is that, while there are plenty of tools that exist to consume web



services, most notably ws-import (that ships with the standard Java Development Kit), research in Android developer forums revealed that these libraries do not work well with Android apps. Google Cloud Messaging for Android [41] is a service that provides a means of updating a number of mobile devices from the server; the solution does not fit this project's situation.

JSON tends to be a more efficient technology for communication between Android client and server than XML [42]. The standard Android SDK has a built-in library for sending and receiving JSON-formatted objects against a RESTful web service. Using a SOAP-based service would require additional libraries and configuration on the Android client side. Therefore, unless a solution requires the use of a legacy SOAP-based web service, there is no reason to use the less efficient technology. Though either type of web services solution will likely work, it was decided to use the JSON format for communication.

Besides the Arduino code, which is in a C-like language called Processing, it was desirable to keep all other solution code in Java. This is simple on the client side since Android development is primarily done in Java, but on the server side research was needed to setup a Java RESTful web service. Many examples with JSON use PHP, as the platform is especially well-suited to quickly get RESTful services up and running. A REST web service was encoded using Eclipse Java EE IDE for Web Developers [43-44], which supports using JSON and RESTful services in a Java servlet container. The project leveraged the IDE's feature for automatically generating a WAR file and deploying to a Tomcat web server, which was also downloaded and installed [45].

Upon receiving and displaying the parsed blood pressure values the Android application creates an instance of a class created for this project called ServiceProxy, which is an implementation of the AsyncTask interface. Implementing this interface allows the execution of web service access code in a separate thread, which is a recommended approach to prevent the application from hanging, or locking up. The web service access happens in the background. The ServiceProxy class is constructed with an object that represents a single blood pressure reading. No imports or special libraries for dealing with web services needed to be included in the Android project.

The logic necessary to parse an application-specific BpReading object to one that can be sent easily over a RESTful service is shown in Figure 4.

```

/**
 * convert from an application object to webservice friendly ones
 * @param reading
 */
private void parseReading(Reading reading) {
    params.add(new BasicNameValuePair("systolic",
        String.valueOf(reading.getSystolic())));
    params.add(new BasicNameValuePair("diastolic",
        String.valueOf(reading.getDiastolic())));
    params.add(new BasicNameValuePair("id",
        String.valueOf(reading.getId())));
}

```

**Figure 4. Convert reading object to a list of name/value pairs.**

The “params” object that the code in Figure 5 refers to is a list of NameValuePair objects. This list is then used in the logic to send the data via the web service in the next code sample:

```

private HttpResponse doResponse(String url) {
    // Use our connection and data timeouts as parameters for our
    // DefaultHttpClient
    HttpClient httpclient = new DefaultHttpClient(getHttpParams());
    HttpResponse response = null;

    try{
        HttpPost httppost = new HttpPost(url);
        // Add parameters
        httppost.setEntity(new UrlEncodedFormEntity(params));
        response = httpclient.execute(httppost);
    } catch (Exception e) {
        Log.e(TAG, e.getLocalizedMessage(), e);
    }
    return response;
}

```

**Figure 5. Sending data via RESTful web service using libraries in Android SDK.**

### 3.7 A SIMPLE WAY TO STORE AND ACCESS THE DATA

The REST web service receives and parses the data. For this prototype the readings are stored in an in-memory data structure. Users can access the stored results by visiting a web page from any web browser or smart phone. JavaServer Pages (JSP) technology is a way of executing Java code on the server and mixing the processing output with html for the purpose of serving the results as a web page. The project uses a JSP page to pull the results of the in-memory data structure, iterate through it, and display readings in a table. Some simple Javascript logic was added to the JSP page (Figure 6) to cause an automatic periodic refresh.

## Readings for Andrew Luxner

**Most recent reading (2013.01.29 08:52:02 PM): 135 / 84**

Systolic	Diastolic	Date/Time
135	84	2013.01.29 08:52:02 PM
126	88	2013.01.29 08:49:38 PM
123	89	2013.01.29 08:47:17 PM
140	85	2013.01.29 08:44:57 PM

The count is: 4

**Figure 6. JSP page displaying saved BP readings.**

### 3.8 INTERACTING WITH A BLOOD PRESSURE MONITOR

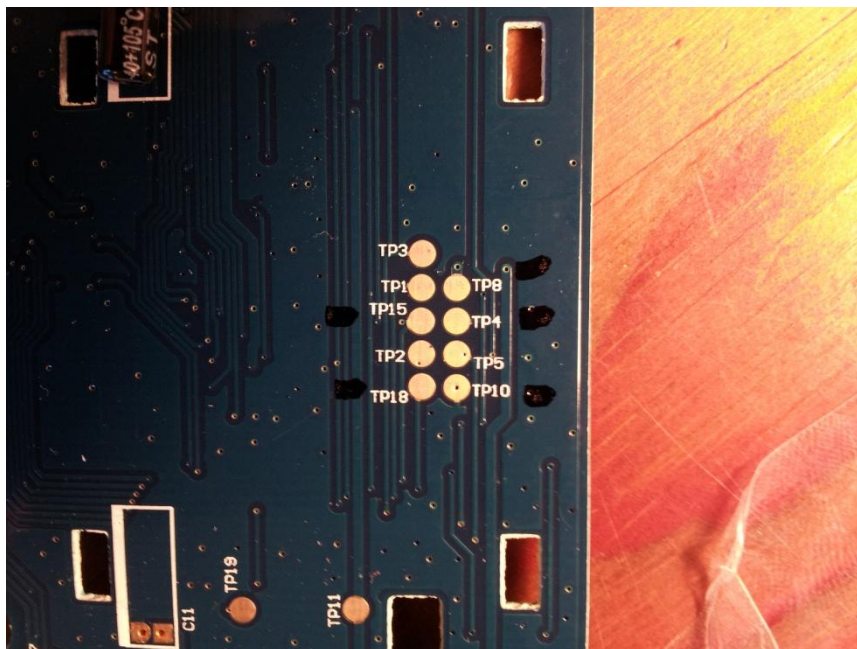
The most challenging part of the project was interfacing with a blood pressure monitor. Time constraints did not allow the development of a monitor from scratch but a project requirement is that the mobile app needs to be able to control a blood pressure monitor and access its readings for use in the application. Ideally one would be able to buy a monitor that provides the hardware and software interface to allow other devices to control it and extract data from it. Unfortunately such devices don't exist off-the-shelf.

Researching this issue, however, led to a blog in which the author details his experience on hacking into an existing monitor [46–48]. The blood pressure monitor in the

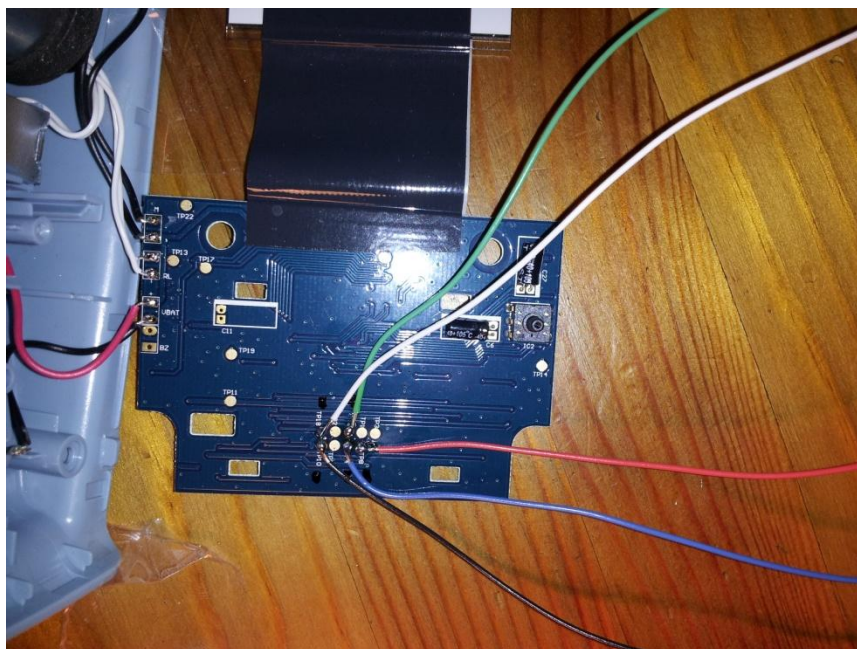
blog posts uses an EEPROM memory chip. This kind of memory allows data to be stored on a device and retrieved later, even after the device shuts down. Many other electronic devices use such chips to store device data. The EEPROM in the blood pressure monitor used conforms to a protocol called I<sup>2</sup>C (often written on the web as I2C). The protocol allows serial communication between the chip and the logic controlling the chip (known as the “master”) by using only two indicators, known as SCL (or “clock”) and SDA (or “data”) [49]. The protocol is common, and Arduino has a library to help interface with it [50]. Truly understanding the I2C protocol takes some time, but the basic idea is that the SDA line provides the data values, bit by bit, and the SCL helps the program decide when the data bits should be sampled.

A nearly identical device (model BM35) to the one described in the blog posts was purchased at CVS after contacting the European reseller for how to purchase the device in the United States. The device was disassembled to gain access to a number of test pads located on one side of the blood pressure monitor’s printed circuit board. By tying into the test pads, such as by soldering wires to them, one can electronically and programmatically control the device. Two of the test pads were tied to the SCL and SDA pins of the chip and thus were accessed to read the EEPROM chip. Two more pads (the ground and voltage) were accessed to provide power to the Arduino board. A fifth pad was used to control the device’s start button. Figure 7 and Figure 8 below are photos that show the test pads on the back of the board before and after wires were soldered to them

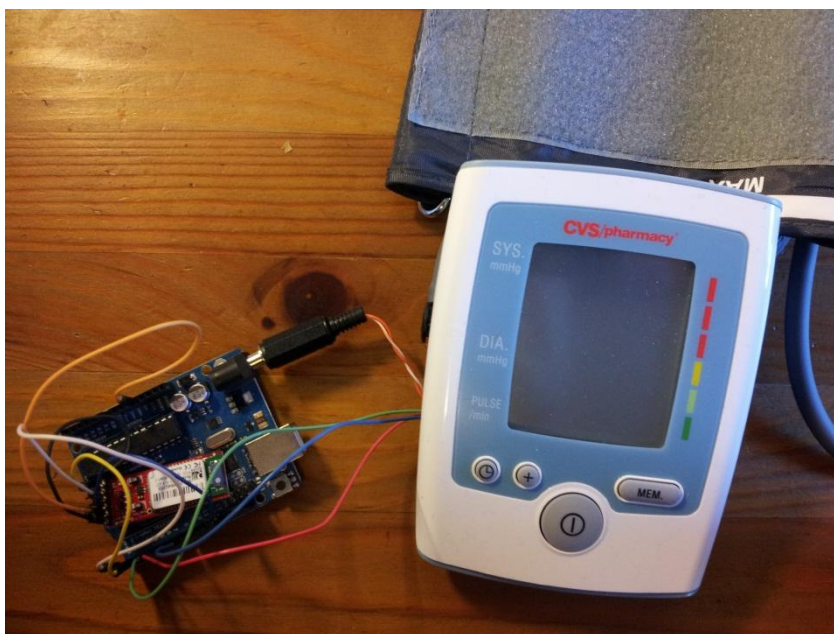
A hole was drilled on one side of the device so the wires soldered to the test pads could be accessed after the unit was closed up. Figure 9 below shows that the ground and voltage wires were connected to a 5.5x2.1mm barrel jack connector that plugs into the Arduino. In another iteration the barrel jack connector was not used; the voltage and ground lines were instead plugged into the Arduino’s voltage (Vin) and ground (GND) pins. No additional power supply was needed as the six volts that the monitor’s four AAA batteries supplied were sufficient to power both the monitor and the Arduino board.



**Figure 7. Test pads on back of circuit boards.**



**Figure 8. Test pads after soldering wires.**

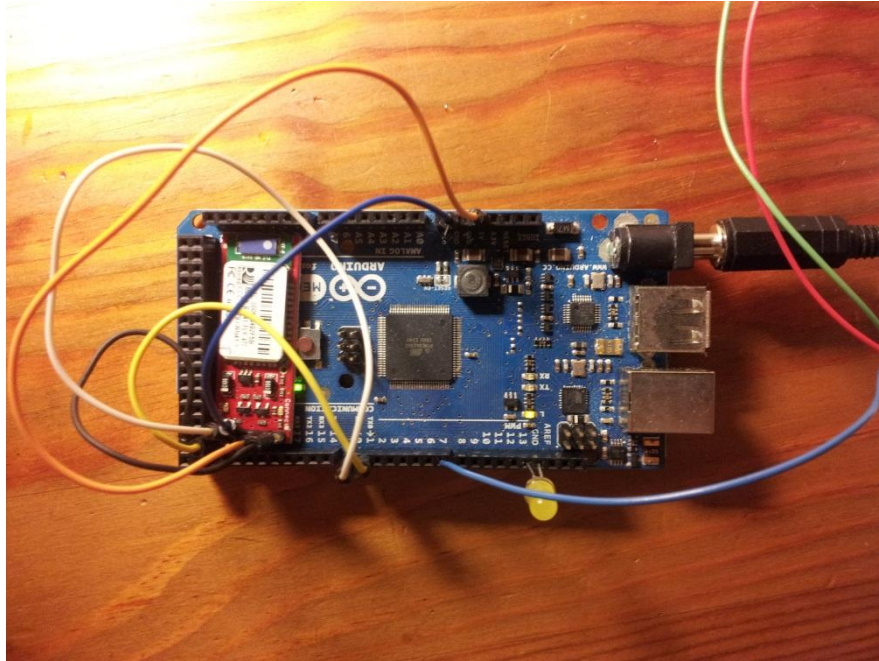


**Figure 9. Arduino UNO board connected to BP monitor.**

The test pads on the back of a circuit board made it unnecessary to use the Arduino's "Wire" library, which ships with Arduino and can be used to work with I2C data. The two wires connected to the SCL and SDA pads were inserted into two digital pins of the Arduino. The part of the program running in the Arduino's "loop" method constantly listens for any messages to and from the EEPROM chip. The Arduino therefore has no need to invoke a read or write directly on the chip; it just passively listens in on the normal activity and sets aside the pieces of data pertaining to blood pressure readings so the data can be sent to the phone.

Two Arduino boards were used in development. The Arduino ADK R3, shown in Figure 10, was originally intended for the project because the original assumption was that the project would be done in conformance with the Android ADK protocol as described above. However, once some basic functionality in stopping and starting the device and Bluetooth transmission was achieved it was desirable to use another Arduino board so that the different functional areas could stay isolated for debugging purposes. A second board, an Arduino UNO R3, was employed to help debug the I2C snooping logic. The ADK board cost more than double the price of a \$30 UNO, and once the original plan to use the Android ADK was abandoned there was no real need for the ADK board. However, most Arduino boards have many basic features in common and could be used interchangeably.





**Figure 10. Close-up of Arduino ADK with power connected at right.**

### **3.9 INTEGRATING ARDUINO PROGRAMS**

During development of the prototype two Arduino programs were written. Both programs interacted with wires soldered to the blood pressure device's test pads. The first program simply waited for signals coming from the Bluetooth device to tell the Arduino to start or stop taking readings. The code was written using examples from the Arduino tutorials and code samples. Upon a "start" signal, the logic had the Arduino ground one of the wires once to wake up the device from a dormant state and a second time to cause the device to start. The program repeated this logic every 30 seconds. This program made no attempt to get the results of the reading. Instead, the Arduino code generated random values for the reading and immediately sent them back to the phone via the Bluetooth device. Thus the first program accomplished both control of the blood pressure device and ability to send readings back to the phone.

The second version of the program concentrated on the data itself, and its functionality was much more difficult to achieve. The core logic was copied directly from the code provided on Joe Desbonnet's web site [47] but had to be debugged. In fact the strategy of snooping on the SDA and SCL data lines was nearly abandoned because the logic seemed to be yielding inconsistent results. During testing and development the data traffic written to

and read from the EEPROM chip was output to the Arduino serial console. There are eight pieces of data that make up a single blood pressure reading:

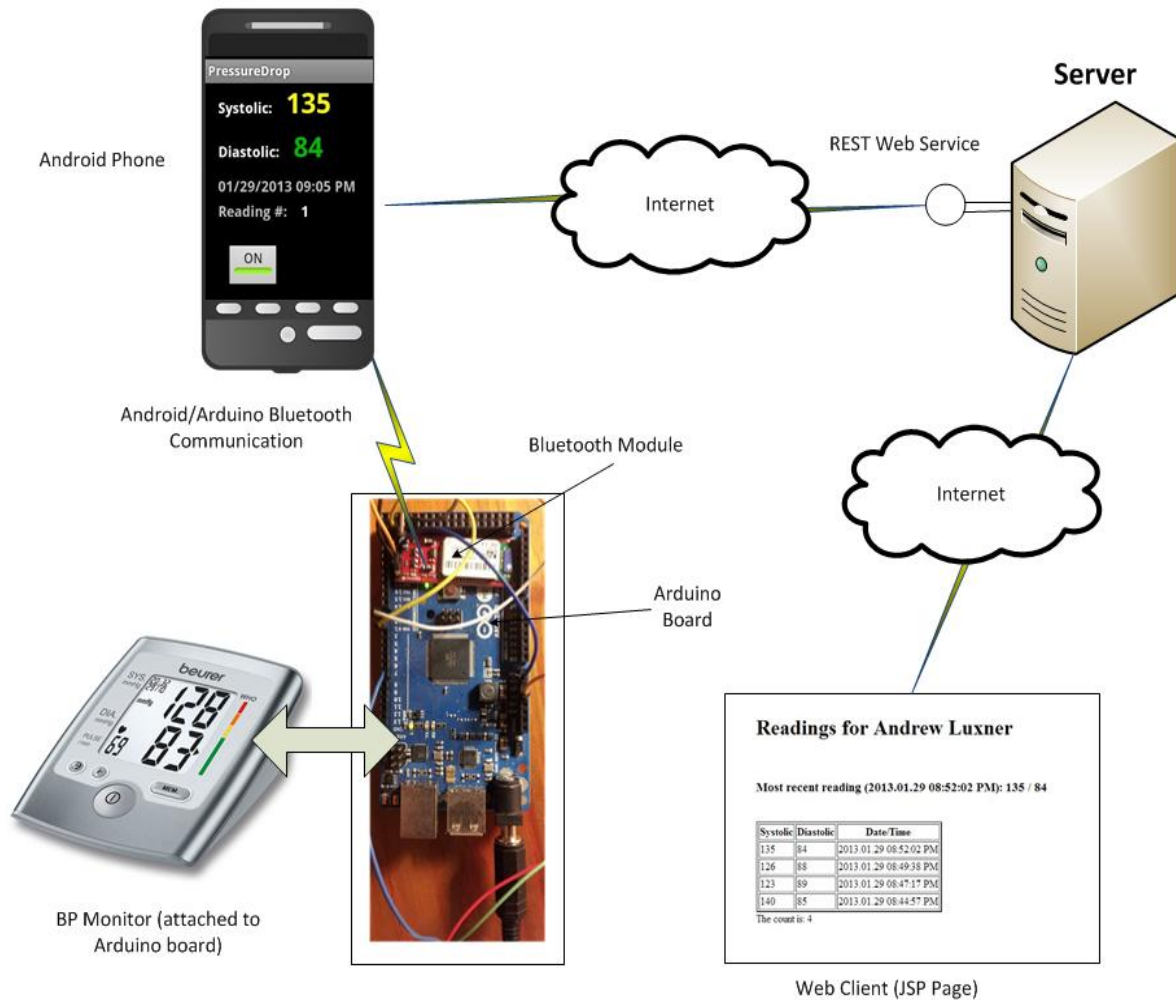
1. month
2. day of month
3. hour
4. minute
5. hundreds digit of the systolic and diastolic
6. systolic ones and tens digits,
7. diastolic ones and tens digits, and
8. heart rate

Code to parse this data had to be written and the code depends on a predictable format. But during testing sometimes some of the data elements were missing. Even though the device seemed to operate normally the SCL and SDA data seemed to inconsistently skip some of the data elements. During debugging it was discovered that printing out the data elements sometimes interfered with the snooping logic. The logic in the original program had to be altered to extract only the necessary data elements: the hundreds digit of the systolic and diastolic values was provided in one byte that had to be evaluated. The systolic ones and tens digits are sent in another byte, followed by a byte for the diastolic ones and tens. The rest of the data elements were ignored. Additional changes were made to ignore all the values that are read from the chip and concentrate only on the values *written to* the chip. Also, the updated logic ignores certain memory addresses that don't pertain to individual readings.

Together both of the programs provide all the basic functionality for the application, but once both programs were working reliably they then had to be integrated. Most of the logic for snooping on the EEPROM writes was encapsulated into a method called `listenForReadingData`. The method is invoked by the `takeReading` method, which also handles grounding the necessary test pad, thus starting the BP monitor. When `listenForReading` data completes, `takeReading` invokes code in the Amarino-supplied library to perform the Bluetooth communication. The resulting program is approximately 260 lines long (including comments) and compiles to a 7108 byte binary file that is uploaded to the Arduino. The integrated program, called "PressureDrop.ino", successfully starts and stops the blood pressure device at signals from the phone, and also listens for data written to the



EEPROM chip. Upon detecting a “write” operation, the necessary elements are parsed and sent back to phone. Figure 11 gives a summary overview of the entire solution.



**Figure 11. Solution overview.**

## **CHAPTER 4**

### **SECURITY**

Security must be handled both on the client side and on the server side. Securing data transmitted over the internet must be addressed, as well securing data sent between the mobile device and the Bluetooth device. Finally, because the solution could be used by actual patients, a discussion of security should include patient confidentiality.

#### **4.1 SECURING WEB TRAFFIC**

A common way of securing data sent over the internet is to use HTTPS. HTTPS provides a way to secure data using Transport Layer Security (TLS) or Secure Sockets Layer (SSL, the predecessor of TLS) [51]. The protocol is popular and widely supported by web browsers. Java and the Android SDK both provide libraries for working with the protocol, and most of the details are handled behind the scenes. But Android applications are often not adequately secured, even when SSL is used. It's easy to code a sloppy SSL implementation that leaves many holes for "man-in-the-middle" and other attacks [52]. SSL, when properly implemented, however, can provide sufficient security for this type of application [53]. Using HTTPS for this application would involve installing a TLS/SSL certificate on the server. Then some minor changes would need to be made in the Android code to use HTTPS instead of HTTP.

#### **4.2 BLUETOOTH ISSUES**

Bluetooth is a wireless standard for exchanging data over short distances. The module used in this project (a Sparkfun BlueSMiRF Gold modem) claims to be effective up to 106 meters [54], a distance considered long-range for Bluetooth devices. Blood pressure readings are transmitted from Bluetooth device to phone and therefore security must be considered since other devices within the Bluetooth modem's range could potentially intercept the data.

Bluetooth communication typically requires two devices to explicitly pair together. In this solution the Amarino software reports on available Bluetooth devices and requires the user to choose which devices to pair with. Pairing is a one-time operation; information about the other device is stored in the application so the next connection is automatic. During this

pairing process the devices exchange a key that allows data to be encrypted [55]. The BlueSMiRF Gold Bluetooth device boasts “Secure communications, 128 bit encryption” [56]. Bluetooth has vulnerabilities [55] but newer versions of the standard attempt to improve on security.

## **4.2 PATIENT CONFIDENTIALITY**

A potential use of this solution is for academic research where anonymous volunteers would wear a device and have their blood pressure readings sent to a server in an attempt to learn more about the nature of blood pressure. In this case there would be little need for patient confidentiality. But the solution could also be used by doctors in an attempt to monitor patients whose data are tied to insurance records. In this case, special care must be taken to secure the server itself by the use of passwords or ssh keys and to limit access to the data exposed in web applications or other applications. The access and use of medical data, including blood pressure readings, are subject to local, state, and federal laws in the United States, including the Health Insurance Portability and Accountability Act (HIPAA) [57]. The complex nature of these and other legal requirements causes this and any solution involving medical data to expend considerable time and money in an effort to handle medical data ethically, legally, and competently, thus avoiding lawsuits. Patients must generally give consent before medical records can legally be disclosed [58].

## **CHAPTER 5**

### **SUMMARY AND CONCLUSIONS**

Using a smart phone as a controller and display component of an Ambulatory Blood Pressure Monitor seems like a good solution on the surface. The application does not require any cutting edge features of mobile phones so there would be few compatibility problems. Using the phone as a display may eliminate the need for a redundant LCD display on the monitor itself. The On, Off, and other buttons on the monitor can also go away and be replaced with controls on the mobile app. Also the readings can be stored on the phone or on the central server instead of on the monitor. All these things would allow the monitor to be smaller, simpler and possibly cheaper. Manufacturers are free to get their product out the door with a basic app, and then provide app updates including more features such as graphs of the BP readings and advice tailored to the user. The Bluetooth module can be a fairly costly component, but the prices vary dramatically and a single unit can be found for under \$10 on ebay.

The Arduino platform is mainly a prototyping tool never meant as a production component. Though it fits in the palm of a hand it has many input pins, connectors, and other hardware that are not of use to this project. A goal for a future phase of the project, or for manufacturers, would be to take the logic that is running on this project's Arduino board and incorporate that into a chip within the monitor's case. Services exist to translate a design into a finished printed circuit board (PCB). One such service is called "Fritzing Fab" from Fritzing.org, a maker of electronics documentation software [59]. Fritzing software is popular for documenting electronics designs (including those made with Arduino boards). The software is free, but for a fee users can take their designs and order finished, professional PCBs that are potentially much smaller than an Arduino board and contain only the necessary components.

Another area for improvement is the blood pressure monitor itself. For this proof of concept an existing, off-the-shelf device was hacked so that logic running on an external board could snoop on data written to the blood pressure monitor. This worked, but it required

tedious logic to sift through data never intended for external use. A better plan would be to ease the extraction of data from the monitor by providing some kind of interface, or API, so that external devices could access the BP readings more easily. Ideally, data written to the monitor's memory would also be formatted in a way to facilitate this API. This kind of a solution could involve building a blood pressure monitor from component parts, a much more involved undertaking but one that would provide much better integration in the end solution. A company called Freescale Semiconductor provides an "Application Note" document that suggests a design for building a BP monitor using some of its components [60].

When the project began it seemed the Android ADK protocol, which was released in 2011, was a suitable solution for part of the project's requirements, as it was intended to allow developers to start developing for external devices. Unfortunately, nearly two years later, it doesn't seem that the standard is catching on and phone manufacturers do not seem interested in making sure their devices are compatible with the protocol. Furthermore, for this type of project there does not seem to be a real need for the ADK, as basic wireless access can be done without it.

The choice of software and basic design for the web application makes sense because of the wide range of supporting software projects available in Java. For example, there are plenty of open-source Java application and web servers available to choose from. Also, both the client and server were done in Java so, going forward, it will likely be possible to use some of the same components on client and server. And using a common language makes it easier for developers to switch between client and server development.

Amarino helped jump-start the project's phone-to-Bluetooth communication. However, the current solution requires multiple apps to be installed on a user's phone, which is not desirable. A better solution would be to extract the necessary Bluetooth pairing logic and put it into the main "PressureDrop" app so that only one app needs to be installed. Doing so may also help resolve the occasionally-occurring situation where the phone fails to pair with the Bluetooth module on the first attempt; sometimes the app must be restarted. Amarino provides a slick interface that helps pair phones and test basic communication, but such an interface is not really necessary for a finished product.

The web application was developed as a proof of concept for storing and retrieving data. Right now accessing the JSP page simply serves a list of blood pressure readings. Undoubtedly users will want more functionality, such as a graph of their blood pressure readings over time, or a summary of when their readings were in the “high” range. If used as part of a study the server-side application can provide some aggregation of multiple users’ data for processing and display in an attempt to draw conclusions about what should be considered “normal” blood pressure fluctuation and what seems to be abnormal or dangerous. The mobile app could also be enhanced such that the web page would be embedded within the app. In effect the mobile app would wrap the web page, so users could access the web results without leaving the application.

## REFERENCES

- [1] A. Cloe. Wrist Blood Pressure Cuff Vs. Arm Blood Pressure Cuff, 2011.  
<http://www.livestrong.com/article/226183-wrist-blood-pressure-cuff-vs-arm-blood-pressure-cuff/>, accessed Jan. 2013.
- [2] American Heart Association. Choosing a Home Blood Pressure Monitor, 2012.  
[http://www.heart.org/HEARTORG/Conditions/HighBloodPressure/SymptomsDiagnosisMonitoringofHighBloodPressure/Choosing-a-Home-Blood-Pressure-Monitor\\_UCM\\_303322\\_Article.jsp.](http://www.heart.org/HEARTORG/Conditions/HighBloodPressure/SymptomsDiagnosisMonitoringofHighBloodPressure/Choosing-a-Home-Blood-Pressure-Monitor_UCM_303322_Article.jsp.), accessed Jan. 2013.
- [3] R. Ramesh. Blood Pressure Test Changes Expected to Cut Misdiagnoses, 2011.  
<http://www.guardian.co.uk/society/2011/aug/24/blood-pressure-test-changes-misdiagnoses>, accessed Jan. 2013.
- [4] S. Adams. Millions of High Blood Pressure Patients are Wrongly Diagnosed, 2011.  
<http://www.telegraph.co.uk/health/healthnews/8339545/Millions-of-high-blood-pressure-patients-are-wrongly-diagnosed.html>, accessed Feb. 2013.
- [5] Google. Ambulatory Blood Pressure Monitor - Google Search, 2013.  
[https://www.google.com/search?q=ambulatory+blood+pressure+monitor&aq=f&oq=ambulatory+blood+pressure+monitor&aqs=chrome.0.57j65j5j0l2j62.8950&sourceid=chrome&ie=UTF8#q=ambulatory+blood+pressure+monitor&hl=en&sa=N&tbs=cat:495,p\\_ord:pd&tbm=shop&ei=iLgOUei9Cse62gWoDADw&ved=0CFkQuw0oAjgU&fp=1&biw=1342&bih=927&bav=on.2,or.r\\_gc.r\\_pw.r\\_cp.r\\_qf.&cad=b&sei=s7gOUcuhNKrM2AW-z4DoBA](https://www.google.com/search?q=ambulatory+blood+pressure+monitor&aq=f&oq=ambulatory+blood+pressure+monitor&aqs=chrome.0.57j65j5j0l2j62.8950&sourceid=chrome&ie=UTF8#q=ambulatory+blood+pressure+monitor&hl=en&sa=N&tbs=cat:495,p_ord:pd&tbm=shop&ei=iLgOUei9Cse62gWoDADw&ved=0CFkQuw0oAjgU&fp=1&biw=1342&bih=927&bav=on.2,or.r_gc.r_pw.r_cp.r_qf.&cad=b&sei=s7gOUcuhNKrM2AW-z4DoBA), accessed Feb. 2013.
- [6] Healthwise Incorporated. Home Blood Pressure Test: Types of Blood Pressure Monitors, 2011. <http://www.webmd.com/hypertension-high-blood-pressure/home-blood-pressure-test>, accessed Jan. 2013.
- [7] Apple. iHealth Blood Pressure Dock, n.d.  
<http://store.apple.com/us/product/H4659LL/A/ihealth-blood-pressure-dock>, accessed Feb. 2013.
- [8] Withings. Blood Pressure Monitor – Features, n.d.  
<http://www.withings.com/en/bloodpressuremonitor/features#anchor3>, accessed Feb. 2013.
- [9] C. Pettey and R. van der Meulen. Gartner Says Worldwide Sales of Mobile Phones Declined 3 Percent in Third Quarter of 2012; Smartphone Sales Increased 47 Percent, 2012. <http://www.gartner.com/newsroom/id/2237315>, accessed Feb. 2013.
- [10] Google. Accessory Development Kit, Android Developers, n.d.  
<http://developer.android.com/tools/adk/index.html>, accessed Jan. 2013.
- [11] Google. Accessory Development Kit 2012 Guide, n.d.  
<http://developer.android.com/tools/adk/adk2.html>, accessed Jan. 2013.

- [12] A. Allan. Fighting the Next Mobile War, 2011.  
<http://radar.oreilly.com/2011/09/next-mobile-war-external-accessory.html>, accessed Jan. 2013.
- [13] A. Allan. The Daily ACK: Connect your iPhone to the Real World, 2011.  
<http://www.dailyack.com/2011/07/connect-your-iphone-to-real-world.html>, accessed Jan. 2013.
- [14] B. Jepson. \$59 Cable Lets You Connect iPhone to Arduino — No Jailbreaking!, 2011.  
<http://blog.makezine.com/2011/07/18/59-cable-lets-you-connect-iphone-to-arduino-no-jailbreaking/>, accessed Jan. 2013.
- [15] Arduino. Arduino - HomePage, 2013.  
<http://www.arduino.cc/>, accessed Jan. 2013.
- [16] M. Banzi. *Getting Started with Arduino*. Make, 2 edition, 2011.
- [17] T. Igoe. *Making Things Talk: Using Sensors, Networks, and Arduino to See, Hear, and Feel Your World*. Make, 2 edition, 2011.
- [18] Sena blogger. The Comparison of Wi-Fi, Bluetooth and ZigBee, 2010.  
<http://www.sena.com/blog/?p=359>, accessed Jan. 2013.
- [19] Google. Bluetooth, 2013.  
<http://developer.android.com/guide/topics/connectivity/bluetooth.html>, accessed Jan. 2013.
- [20] Apple. Bluetooth Device Access Guide: Developing Bluetooth Applications, 2012.  
[https://developer.apple.com/library/mac/#documentation/devicedrivers/conceptual/bluetooth/BT\\_Develop\\_BT\\_Apps/BT\\_Develop\\_BT\\_Apps.html](https://developer.apple.com/library/mac/#documentation/devicedrivers/conceptual/bluetooth/BT_Develop_BT_Apps/BT_Develop_BT_Apps.html), accessed Jan. 2013.
- [21] Arduino. Arduino, n.d. <http://arduino.cc/en/Main/ArduinoWiFiShield>, accessed Feb. 2013.
- [22] Wikipedia. Blood Pressure, 2013.  
[http://en.wikipedia.org/wiki/Blood\\_pressure#Measurement](http://en.wikipedia.org/wiki/Blood_pressure#Measurement), accessed Jan. 2013.
- [23] C. T. Phua and G. Lissorgues. Measurement of blood pressure using magnetic method of blood pulse acquisition. In *Nano/Molecular Medicine and Engineering (NANOMED), 2009 IEEE International Conference*, pages 112–115. IEEE, 2009.
- [24] K. Machulis. Libomron - Open source Driver for Omron USB Products, 2011.  
<http://qdot.github.com/libomron/index.html>, accessed Jan. 2013.
- [25] M. Böhmer. *Beginning Android ADK with Arduino*. Apress, 1 ed, 2012.
- [26] CyanogenMod. CyanogenMod Downloads, 2012.  
<http://download.cyanogenmod.com/?type=stable&device=galaxys2>, accessed Jan. 2013.
- [27] Android Forums. Galaxy S2 Skyrocket - All Things Root, n.d.  
<http://androidforums.com/galaxy-s2-skyrocket-all-things-root/>, accessed Feb. 2013.
- [28] P. Nunal. Root Samsung Galaxy S2 Skyrocket SGH-I727 running Android 4.0.3 firmware UCALC4, 2012.  
<http://www.androidauthority.com/galaxy-s2-skyrocket-att-sgh-i727-android-4-0-3-root-72082/>, accessed Feb. 2013.



- [29] Melvin. [HOWTO] ROOT T-Mobile SGSII via ODIN— Updated Recovery, 2011.  
<http://forum.xda-developers.com/showthread.php?t=1311194>, accessed Jan. 2013.
- [30] B. Kaufmann. Marino – “Android meets Arduino” - Home, n.d.  
<http://www.amarino-toolkit.net/>, accessed Jan. 2013.
- [31] B. Kaufmann. Design and implementation of a toolkit for the rapid prototyping of mobile ubiquitous computing. Master’s thesis, University of Klagenfurt, Klagenfurt, Austria, 2010.
- [32] S. Monk. *Arduino + Android Projects for the Evil Genius: Control Arduino with Your Smartphone or Tablet*. McGraw-Hill/TAB Electronics, 1 edition, 2011.
- [33] Sparkfun. SparkFun Electronics, n.d.  
<http://www.sparkfun.com/>, accessed Jan. 2013.
- [34] B. Kaufmann. AmarinoPluginBundle.apk - amarino - Amarino Plug-in Bundle (v\_0\_3) - Android meets Arduino, 2010.  
<http://code.google.com/p/amarino/downloads/detail?name=AmarinoPluginBundle.apk&can=2&q=m>, accessed Jan. 2013.
- [35] B. Kaufmann. MeetAndroid\_4.zip - amarino - MeetAndroid (Arduino library) v\_4 - Android meets Arduino, 2011.  
[http://code.google.com/p/amarino/downloads/detail?name=MeetAndroid\\_4.zip&can=2&q=](http://code.google.com/p/amarino/downloads/detail?name=MeetAndroid_4.zip&can=2&q=), accessed Jan. 2013.
- [36] B. Kaufmann. AmarinoLibrary\_v0\_55.jar - amarino - Amarino Library v\_0\_55 - Android meets Arduino, 2011.  
[http://code.google.com/p/amarino/downloads/detail?name=AmarinoLibrary\\_v0\\_55.jar&can=2&q=](http://code.google.com/p/amarino/downloads/detail?name=AmarinoLibrary_v0_55.jar&can=2&q=), accessed Jan. 2013.
- [37] B. Kaufmann. Amarino – “Android meets Arduino” - Getting Started, n.d.  
<http://www.amarino-toolkit.net/index.php/getting-started.html>, accessed Jan. 2013.
- [38] B. Kaufmann. SensorGraph\_02.zip - amarino - SensorGraph Tutorial - Android meets Arduino, 2011.  
[http://code.google.com/p/amarino/downloads/detail?name=SensorGraph\\_02.zip&can=2&q=](http://code.google.com/p/amarino/downloads/detail?name=SensorGraph_02.zip&can=2&q=), accessed Jan. 2013.
- [39] B. Kaufmann. MultiColorLamp.zip - amarino - Mutlicolor Lamp example - Android meets Arduino, 2010.  
<http://code.google.com/p/amarino/downloads/detail?name=MultiColorLamp.zip&can=2&q=>, accessed Jan. 2013.
- [40] National Heart Lung and Blood Institute. What Is High Blood Pressure? - NHLBI, NIH, 2012. <http://www.nhlbi.nih.gov/health/health-topics/topics/hbp/>, accessed Jan. 2013.
- [41] Google. Google Cloud Messaging for Android, n.d.  
<http://developer.android.com/google/gcm/index.html>, accessed Jan. 2013.
- [42] R. Whitney. Android11FontsWeb.pdf, 2011.  
<http://www.eli.sdsu.edu/courses/spring11/cs696/notes/Android11FontsWeb.pdf>, accessed Jan. 2013.

- [43] Avilyne Technologies. An Android REST Client and Tomcat REST Webservice, 2012. <http://avilyne.com/?p=105>, accessed Jan. 2013.
- [44] L. Vogel. REST with Java (JAX-RS) using Jersey – Tutorial, 2012. <http://www.vogella.com/articles/REST/article.html>, accessed Jan. 2013.
- [45] The Apache Software Foundation. Apache Tomcat - Welcome!, 2013. <http://tomcat.apache.org/>, accessed Jan. 2013.
- [46] J. Desbonnet. How to Make an Ambulatory Blood Pressure Monitor (ABPM) for just €20 (\$25), 2010. <http://jdesbonnet.blogspot.com/2010/05/how-to-make-ambulatory-blood-pressure.html>, accessed Jan. 2013.
- [47] J. Desbonnet. Using an Arduino as a simple logic analyzer (part 3), 2010. <http://jdesbonnet.blogspot.com/2010/05/using-arduino-as-simple-logic-analyzer.html>, accessed Jan. 2013.
- [48] J. Desbonnet. Sanitas SBM30 (aka HL868BA) teardown, 2011. <http://jdesbonnet.blogspot.com/2011/01/sanitas-smb30-aka-hl868ba-teardown.html>, accessed Jan. 2013.
- [49] Wikipedia. I<sup>2</sup>C, 2013. <http://en.wikipedia.org/wiki/I%C2%B2C>, accessed Jan. 2013.
- [50] humanHardDrive. Arduino Tutorial #8: I2C Communication, 2012. <http://www.youtube.com/watch?v=J3nuIL2dBak>, accessed Jan. 2013.
- [51] E. Rescorla. RFC 2818 - HTTP Over TLS, 2000. <http://tools.ietf.org/html/rfc2818>, accessed Feb. 2013.
- [52] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben. Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In L. Baumgärtner and B. Freisleben, editors, *ACM CCS 2012*, pages 50–61. ACM, North Carolina, 2012.
- [53] S. Gilbertson. HTTPS Is More Secure, So Why Isn't the Web Using It?, 2011. <http://www.webmonkey.com/2011/03/https-is-more-secure-why-isnt-the-web-using-it-today/>, accessed Feb. 2013.
- [54] SparkFun Electronics. Bluetooth Modem - BlueSMiRF Gold, n.d. <https://www.sparkfun.com/products/10268>, accessed Feb. 2013.
- [55] J. T. Vainio. Bluetooth, 2000. <http://www.yuuhaw.com/bluesec.pdf>, accessed Feb. 2013.
- [56] Roving Networks. *Bluetooth-RN-41-DS*, 2011. <http://dlnmh9ip6v2uc.cloudfront.net/datasheets/Wireless/Bluetooth/Bluetooth-RN-41-DS.pdf>, accessed Feb. 2013.
- [57] 104th Congress. Health Insurance Portability and Accountability Act of 1996, 1996. <http://www.gpo.gov/fdsys/pkg/PLAW-104publ191/html/PLAW-104publ191.htm>, accessed Feb. 2013.
- [58] The American Medical Association. Patient Confidentiality, n.d. <http://www.ama-assn.org/ama/pub/physician-resources/legal-topics/patient-physician-relationship-topics/patient-confidentiality.page>, accessed Feb. 2013.
- [59] Fritzing. Fritzing Fab, n.d. <http://fab.fritzing.org/fritzing-fab>, accessed Mar. 2013.

- [60] S. Lopez. *FreescaleAN4328*. Freescale Semiconductor Inc., Tempe, Arizona, 2012.

**APPENDIX**

**SOLUTION SOURCE CODE**

## ANDROID CODE

This section of the appendix includes the core code that, using the Eclipse plugin for Android development, forms an .apk file that can be executed on an Android phone.

### PressureDropActivity.java

```
package luxner.pressureDrop;

import java.text.SimpleDateFormat;
import java.util.Date;

import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.graphics.Color;
import android.os.Bundle;
import android.util.Log;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;

import android.widget.TextView;
import android.widget.ToggleButton;
import at.abraxas.amarino.Amarino;
import at.abraxas.amarino.AmarinoIntent;

public class PressureDropActivity extends Activity {

    private static int readingCount = 0;
    private static SimpleDateFormat dateFormat
        = new SimpleDateFormat("MM/dd/yyyy hh:mm
a");

    private static final String DEVICE_ADDRESS = "00:06:66:49:2F:5B";
    private static final String TAG = "PressureDrop";
    private ArduinoReceiver arduinoReceiver = new ArduinoReceiver();
    private static final String PROD_URL
        = "http://goldenhillbooks.com/pressureDrop/rest/readings";

    private static final String LOCAL_URL
        = "http://68.8.148.182:8080/pressureDrop/rest/readings";
    // = "http://192.168.1.101:8080/pressureDrop/rest/readings";
    private String url = PROD_URL;

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.d(TAG, "onCreate!");
        setContentView(R.layout.main);
        registerReceiver(arduinoReceiver,
            new IntentFilter(AmarinoIntent.ACTION_RECEIVED));
        Amarino.connect(this, DEVICE_ADDRESS);
    }

    /**
     * Handle the toggle button click - user is either starting or stopping
     * device
     * @param v
     */
}
```

```

public void toggle(View v){
    ToggleButton button = (ToggleButton)findViewById(R.id.startBtn);
    //TextView msgLabel = (TextView)findViewById(R.id.msgLabel);
    if(button.isChecked()){
        stopOrStart(true);
    }else{
        stopOrStart(false);
    }
}

private void send(Reading reading){

    ServiceProxy proxy = new ServiceProxy(this, "Sending reading to server",
        reading);
    proxy.execute(new String[]{ url } );
}

/**
 * Stop or start the remote blood pressure device
 * @param start send true to start, false to stop
 */
private void stopOrStart(boolean start){
    int val = (start) ? 1 : 0;
    // The character must be matched to a character registered in a function
    // in the Arduino sketch
    Log.d(TAG, "stopOrStart!: sending value:" + val);
    Amarino.sendDataToArduino(this, DEVICE_ADDRESS, 'o', val);
}

public void setBp(Reading reading){
    TextView sysTv = (TextView)findViewById(R.id.systolicVal);
    if(sysTv != null){

        sysTv.setText(String.valueOf(reading.getSystolic()));
        colorCodeValue(reading.getSystolic(), sysTv, 125, 135);
    }

    TextView diaTv = (TextView)findViewById(R.id.diastolicVal);
    if(diaTv != null){
        diaTv.setText(String.valueOf(reading.getDiastolic()));
        colorCodeValue(reading.getDiastolic(), diaTv, 85, 90);
    }
    incrementReadingNum();
    setDate();
}

private static void colorCodeValue(int value, TextView tv,
    int warnThreshold, int seriousThreshold){

    tv.setTextColor(Color.parseColor("#00B700"));
    if(value > warnThreshold){
        tv.setTextColor(Color.parseColor("#FFFF00"));
    }
    if(value > seriousThreshold){
        tv.setTextColor(Color.parseColor("#FF0000"));
    }
}

public void incrementReadingNum(){
    readingCount++;
    TextView countTv = (TextView)findViewById(R.id.readingVal);
    if(countTv != null){
        countTv.setText(String.valueOf(readingCount));
    }
}

/**
 * Format and display the current local date
 */
public void setDate(){

```

```

        TextView dateView = (TextView)findViewById(R.id.timeLabel);
        if(dateView != null){
            String text = dateFormat.format(new Date());
            dateView.setText(text);
        }
    }

    @Override
    public void onResume() {
        super.onResume();
        Log.d(TAG, "onResume!");
    }

    @Override
    public void onRestart() {
        super.onRestart();
        // Gets called before onResume. But not called on initial startup.
        // Need to reconnect because we disconnected when stopping
        Log.d(TAG, "onRESTART");
        Amarino.connect(this, DEVICE_ADDRESS);
    }

    @Override
    protected void onStop() {
        super.onStop();
        Log.d(TAG, "onStop 1: disconnecting");
        Amarino.disconnect(this, DEVICE_ADDRESS);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.main_menu, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case R.id.prod:
                Log.d(TAG, "onOptionsItemSelected: using production url");
                url = PROD_URL;
                return true;
            case R.id.local:
                Log.d(TAG, "onOptionsItemSelected: using LOCAL url");
                url = LOCAL_URL;
                return true;
            case R.id.exit:
                Log.d(TAG, "onOptionsItemSelected: exiting via menu item");
                Amarino.disconnect(this, DEVICE_ADDRESS);
                int pid = android.os.Process.myPid();
                android.os.Process.killProcess(pid);
                return true;

            default:
                return super.onOptionsItemSelected(item);
        }
    }

    /**
     * parse out a reading from raw data received from arduino
     * @param rawData
     * @return
     */
    private static Reading parseBpData(String rawData) {
        String[] parts = rawData.split("--");

        if(parts.length == 2) {
            Log.d(TAG, "part 0:" + parts[0] + " part 1:" + parts[1]);
            int sys = parseValue(parts[0]);
            int dia = parseValue(parts[1]);
        }
    }

```

```

        return new Reading(sys, dia, DEVICE_ADDRESS);
    }else Log.d(TAG, "parseBP: parts has:" + parts.length
        + " parts (NOT TWO)");
    return new Reading();
}

private static int parseValue(String rawData){
    Log.d(TAG, "parseValue rawData:" + rawData);
    String[] partArr = rawData.split(":");

    if(partArr.length == 2){
        Log.d(TAG, "partArr 0:" + partArr[0] + " partArr 1: " + partArr[1]);
    }
    try{
        return Integer.parseInt(partArr[1]);
    }catch (NumberFormatException nfe){
        Log.d(TAG, "parseValue: number format exception for " + partArr[1]);
        return -1;
    }
}

public class ArduinoReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        String data = null;

        // the type of data which is added to the intent
        final int dataType
            = intent.getIntExtra(AmarinoIntent.EXTRA_DATA_TYPE, -1);
        if(dataType != AmarinoIntent.STRING_EXTRA){
            Log.d(TAG, "onReceive: Intent is not String extra type");
            return;
        }
        // we only expect String data though, but it is better to check
        // if really string was sent later Amarino will support
        // different data types, so far data comes always as string and
        // you have to parse the data to the type you have sent from
        // Arduino, like it is shown below
        data = intent.getStringExtra(AmarinoIntent.EXTRA_DATA);
        Log.d(TAG, "onReceive: data:" + data);
        if (data != null && dataType == AmarinoIntent.STRING_EXTRA
            && data.contains("SYS")){
            Reading reading = parseBpData(data);
            if(reading.isValid()){
                setBp(reading);
                // send reading to server
                send(reading);
            }else{
                Log.d(TAG, "onReceive: reading is invalid");
            }
        }
    }
}

```

## Reading.java

```

package luxner.pressureDrop;
/**
 * Represents a Blood Pressure reading
 * @author andrew
 * TODO: use the BpReading class in server-side project instead of this. Need
 * to create a separate library for reuse
 */
public class Reading {

    private int systolic = -1;
    private int diastolic = -1;
    private String id;

```



```

/**
 * Follows the null object pattern. Construct an invalid Reading.
 */
public Reading() {}

public Reading(int systolic, int diastolic, String id) {
    this.systolic = systolic;
    this.diastolic = diastolic;
    this.id = id;
}

public int getSystolic() {
    return systolic;
}

public void setSystolic(int systolic) {
    this.systolic = systolic;
}

public int getDiastolic() {
    return diastolic;
}

public void setDiastolic(int diastolic) {
    this.diastolic = diastolic;
}

public String getId() {
    return id;
}

public void setId(String id) {
    this.id = id;
}

@Override
public String toString() {
    return "Reading: id:" + id + " systolic:" + systolic + " diastolic:"
        + diastolic;
}

public boolean isValid() {
    return systolic > 0 && diastolic > 0;
}
}

```

## ServiceProxy.java

```

package luxner.pressureDrop;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;

import org.apache.http.HttpResponse;
import org.apache.http.NameValuePair;
import org.apache.http.client.HttpClient;
import org.apache.http.client.entity.UrlEncodedFormEntity;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.impl.client.DefaultHttpClient;
import org.apache.http.message.BasicNameValuePair;
import org.apache.http.params.BasicHttpParams;
import org.apache.http.params.HttpConnectionParams;
import org.apache.http.params.HttpParams;

import android.app.ProgressDialog;
import android.content.Context;

```

```

import android.os.AsyncTask;
import android.util.Log;

/**
 * Component for sending web service data
 * @author andrew
 */
public class ServiceProxy extends AsyncTask<String, Integer, String> {

    private static final String TAG = "PressureDrop";
    private static final int TIMEOUT_MILLISEC = 10000;

    private Context mContext = null;
    private String processMessage = "Processing...";
    private ArrayList<NameValuePair> params = new ArrayList<NameValuePair>();
    private ProgressDialog progDialog = null;

    public ServiceProxy(Context context, String processMessage,
        Reading reading) {
        this.mContext = context;
        this.processMessage = processMessage;
        parseReading(reading);
    }

    /**
     * convert from an application object to webservice friendly ones
     * @param reading
     */
    private void parseReading(Reading reading) {
        params.add(new BasicNameValuePair("systolic",
            String.valueOf(reading.getSystolic())));
        params.add(new BasicNameValuePair("diastolic",
            String.valueOf(reading.getDiastolic())));
        params.add(new BasicNameValuePair("id",
            String.valueOf(reading.getId())));
    }

    @Override
    protected String doInBackground(String... urls) {
        String url = urls[0];
        String result = "";
        Log.d(TAG, "doInBackground: url:" + url);

        HttpResponse response = doResponse(url);

        if (response == null) {
            return result;
        } else {
            try {
                result = inputStreamToString(response.getEntity().getContent());
            } catch (IllegalStateException e) {
                Log.e(TAG, e.getLocalizedMessage(), e);
            } catch (IOException e) {
                Log.e(TAG, e.getLocalizedMessage(), e);
            }
        }

        return result;
    }

    // Establish connection and socket (data retrieval) timeouts
    private HttpParams getHttpParams() {
        HttpParams httpParams = new BasicHttpParams();

        HttpConnectionParams.setConnectionTimeout(httpParams, TIMEOUT_MILLISEC);
        HttpConnectionParams.setSoTimeout(httpParams, TIMEOUT_MILLISEC);
    }
}

```

```

        return httpParams;
    }

    private HttpResponse doResponse(String url) {
        // Use our connection and data timeouts as parameters for our
        // DefaultHttpClient
        HttpClient httpClient = new DefaultHttpClient(getHttpParams());
        HttpResponse response = null;

        try {
            HttpPost httpPost = new HttpPost(url);
            // Add parameters
            httpPost.setEntity(new UrlEncodedFormEntity(params));
            response = httpClient.execute(httpPost);
        } catch (Exception e) {
            Log.e(TAG, e.getLocalizedMessage(), e);
        }
        return response;
    }

    @Override
    protected void onPreExecute() {
        showProgressDialog();
    }

    private void showProgressDialog() {
        progressDialog = new ProgressDialog(mContext);
        progressDialog.setMessage(processMessage);
        progressDialog.setProgressDrawable(mContext.getWallpaper());
        progressDialog.setProgressStyle(ProgressDialog.STYLE_SPINNER);
        progressDialog.setCancelable(false);
        progressDialog.show();
    }

    @Override
    protected void onPostExecute(String response) {
        progressDialog.dismiss();
    }

    private static String inputStreamToString(InputStream is) {
        /*
         * To convert the InputStream to String we use the BufferedReader.readLine()
         * method. We iterate until the BufferedReader return null which means
         * there's no more data to read. Each line will be appended to a StringBuilder
         * and returned as String.
         */
        BufferedReader reader = new BufferedReader(new InputStreamReader(is));
        StringBuilder sb = new StringBuilder();

        String line = null;
        try {
            while ((line = reader.readLine()) != null) {
                sb.append(line + "\n");
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                reader.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return sb.toString();
    }
}

```

**main.xml**

```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
    <!-- Systolic data -->
    <TextView
        android:layout_width="wrap_content"
        android:textColor="#FFFFFF"
        android:textSize="15sp"
        android:textStyle="bold"
        android:layout_marginLeft="15dip"
        android:layout_marginTop="15dip"
        android:id="@+id/systolicLabel"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
        android:text="Systolic:" />
    <TextView
        android:id="@+id/systolicVal"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="15dip"
        android:textSize="30sp"
        android:text=""
        android:textColor="#00B700"
        android:layout_toRightOf="@+id/systolicLabel"
        android:layout_alignBaseline="@+id/systolicLabel"
        android:textStyle="bold" />

    <!-- diastolic data -->
    <TextView
        android:layout_width="wrap_content"
        android:id="@+id/diastolicLabel"
        android:text="Diastolic:"
        android:textColor="#FFFFFF"
        android:textSize="15sp"
        android:textStyle="bold"
        android:layout_marginLeft="15dip"
        android:layout_marginTop="25dip"
        android:layout_below="@+id/systolicLabel"
        android:layout_height="wrap_content" />

    <TextView
        android:layout_width="wrap_content"
        android:id="@+id/diastolicVal"
        android:text=""
        android:textColor="#FF0000"
        android:layout_toRightOf="@+id/diastolicLabel"
        android:textSize="30sp"
        android:textStyle="bold"
        android:layout_marginLeft="15dip"
        android:layout_alignBaseline="@+id/diastolicLabel"
        android:layout_height="wrap_content" />
    <!-- date display -->

    <TextView
        android:layout_width="wrap_content"
        android:id="@+id/timeLabel"
        android:text=""
        android:textSize="15sp"
        android:textStyle="bold"
        android:layout_marginLeft="15dip"
        android:layout_marginTop="15dip"
        android:layout_below="@+id/diastolicLabel"
        android:layout_height="wrap_content" />

    <!-- reading display -->
    <TextView

```

```

        android:layout_width="wrap_content"
        android:id="@+id/readingLabel"
        android:text="Reading #:"
        android:textSize="15sp"
        android:textStyle="bold"
        android:layout_marginLeft="15dip"
        android:layout_marginTop="5dip"
        android:layout_below="@+id/timeLabel"
        android:layout_height="wrap_content" />

<TextView
    android:layout_width="wrap_content"
    android:id="@+id/readingVal"
    android:text="0"
    android:textColor="#FFFFFF"
    android:layout_toRightOf="@+id/readingLabel"
    android:textSize="15sp"
    android:textStyle="bold"
    android:layout_marginLeft="15dip"
    android:layout_alignTop="@+id/readingLabel"
    android:layout_height="wrap_content" />

<ToggleButton
    android:id="@+id/startBtn"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@+id/readingVal"
    android:layout_marginTop="25dp"
    android:layout_marginLeft="25dip"
    android:onClick="toggle"
    />
<!-- use this label to display any messages -->
<TextView
    android:layout_width="wrap_content"
    android:id="@+id/msgLabel"
    android:textColor="#FFFFFF"
    android:textSize="15sp"
    android:textStyle="bold"
    android:layout_marginLeft="15dip"
    android:layout_marginTop="15dip"
    android:layout_below="@+id/startBtn"
    android:layout_height="wrap_content" />

</RelativeLayout>

```

## AndroidManifest.xml

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="luxner.pressureDrop"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="10" />
    <uses-permission android:name="android.permission.INTERNET"/>
    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:name=".PressureDropActivity"
            android:label="@string/app_name"
            android:screenOrientation="portrait" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

```

```
</manifest>
```

## ARDUINO CODE

This section includes the code, written in the C-like language called Processing, that runs on the Arduino UNO board. Two Arduino boards were used in testing a system prototype. The code can be used on either board; the only modification necessary is that for the ADK board “PIND” in the code below must be replaced with “PINA”.

### PressureDrop.ino

```

/*
  Adapted from a blog post by Joe Desbonnet:
  http://jdesbonnet.blogspot.com/2010/05/using-arduino-as-simple-logic-analyzer.html
*/
#include <MeetAndroid.h>

MeetAndroid meetAndroid;
int onboardLed = 13;
boolean isOn = false;
long lastReading = 0; // last time the reading was taken
long currentTime = 0;
const long READING_INTERVAL = 90000; // how often to take reading: 90 seconds
const int BP_MON = 7; // pin from the monitor
char hexval[16] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'};
#define LOG_SIZE 25
byte datalog[LOG_SIZE];
byte logptr;
boolean sysHundred = false;
boolean diaHundred = false;
long last = 0;
long lastDataChunk = 0;
String sysStr = "SYS:";
String diaStr = "--DIA:";
// after we start getting write data wait this long before
// writing
static long WRITE_PAUSE = 2000;

void setup(){
  // baud rate must be compatible with the bluetooth module
  Serial.begin(115200); //57600
  // register callback functions, which will be called when an associated event occurs.
  // The flag 'o' should match an event code in the Android app.
  meetAndroid.registerFunction(testEvent, 'o');

```

```

    pinMode(onboardLed, OUTPUT);
    digitalWrite(onboardLed, HIGH);
    pinMode(BP_MON, OUTPUT);
    digitalWrite(BP_MON, HIGH);
}

void loop(){
    meetAndroid.receive(); // must keep this in the loop() to receive events
    if(shouldTakeReading()){
        // the takeReading logic will take a minute to execute. During this time the
        // program will not be responsive to the stop/start button. COULD probably put
        // the meetAndroid.receive call in the takeReading logic but I'm afraid that
        // would burn cycles needed to parse the reading results.
        takeReading();
    }
}

boolean shouldTakeReading(){
    if(isOn){
        currentTime = millis();
        if((currentTime - lastReading) > READING_INTERVAL){
            lastReading = currentTime;
            return true;
        }
    }
    return false;
}

/*
 * This method is called constantly. Check for turning the device on and off
 * note: flag is in this case 'o' and numOfValues is 0 (since test event doesn't send any
 data)
 */
void testEvent(byte flag, byte numOfValues){
    int val = meetAndroid.getInt();

    flashLed(300);
    flashLed(300);
    // logic on the Android side will send either a 0 or a 1 to turn the device
    // off or on.
    if(val == 0){
        isOn = false;
    }else if(val == 1){
        isOn = true;
        takeReading();
    }
}
}

```



```

void takeReading(){
    digitalWrite(onboardLed, HIGH); // turn LED on
    // ground once to turn device on
    digitalWrite(BP_MON, LOW);
    delay(128);
    digitalWrite(BP_MON, HIGH);
    // wait two seconds...
    delay(2000);
    // ground the thing again to take a reading
    digitalWrite(BP_MON, LOW);
    delay(128);
    digitalWrite(BP_MON, HIGH); //reset the pin to high
    digitalWrite(onboardLed, LOW); // turn LED off
    // now that the thing has started, need to listen for reading data
    String reading = listenForReadingData();
    char resArr[16];
    reading.toCharArray(resArr, 16);
    meetAndroid.send(resArr);
    lastReading = millis();
}

// logic to listen to the bp device
String listenForReadingData(){
    byte sdaSample, dataByte, byteCounter, bitCounter, rwFlag;
    byte addr_hi, addr_lo;
    logptr = 0;

    waitForStart:
    // Expect both SLC and SDA to be high
    while ( (PIND & 0b00001100) != 0b00001100) ;
    // both SLC and SDA high at this point
    // Looking for START condition. Ie SDA transitioning from
    // high to low while SLC is high. SCL is 2. SDA is 3
    while ( (PIND & 0b00001100) != 0b00000100) { // while sda not low
        if (shouldWrite()){
            return parseData();
        }
    }

    firstBit:

    byteCounter = 0;
    bitCounter = 0;

    nextBit:

```

```

// If SCL high, wait for SCL low
while ( (PIND & 0b00000100) != 0) ;
// Wait for SCL to transition high. Nothing of interest happens when SCL is low.
while ( (PIND & 0b00000100) == 0) ;
// Sample SDA at the SCL low->high transition point. Don't know yet if this is a
// data bit or a STOP or START condition.
sdaSample = PIND & 0b00001000;

// Wait for SCL to transition low while monitoring SDA for a transition.
// No transition means we have data or ACK bit (sample in 'sdaSample'). A high to
// low SDA = START, a low to high SDA transition = STOP.
if (sdaSample == 0) {
    // loop while SCL high and SDA low
    while ( (PIND & 0b00001100) == 0b00000100) ;
    // know that it's not the case that SDA low AND SCL high
    if ( (PIND & 0b00001100) == 0b00001100) {
        // STOP condition detected - SCL and SDA both high
        goto waitForStart;
    }
} else {
    // loop while SCL high and SDA high
    while ( (PIND & 0b00001100) == 0b00001100) ;
    if ( (PIND & 0b00001100) == 0b00000100) {
        // START condition.
        goto firstBit;
    }
}

// This is a data bit.
bitCounter++;

if (bitCounter < 9) {
    dataByte <<= 1; // shift existing bits over one to the left to make room for new bit
    // if data bit is '1' set it in LSB position (will default to 0 after the shift op)
    if (sdaSample != 0) {
        dataByte |= 0b00000001;
    }
    goto nextBit;
}

// The 9th bit is the ack/noack part of the I2C protocol and can be ignored.
// All data should be stored in dataByte

bitCounter = 0;
byteCounter++;

switch (byteCounter) {
    case 1: // 1010AAAW where AAA upper 3 bits of address, W = 0 for write, 1 for read

```

```

    if ( (dataByte & 0b11110000) != 0b10100000) {
        // expected format not found ?
        goto waitForStart;
    }
    // Set A9,A8 bits of address
    // ditch the r/w flag and just use first 2 bits of addresss (?)
    addr_hi = (dataByte>>1) & 0b00000011;
    rwFlag = dataByte & 0b00000001;
    break;

    case 2: // data if rwFlag==1 else lower 8 bits of address
    if (rwFlag != 1) {
        addr_lo = dataByte;
        // don't need to save the Read values, only write
    }
    break;

    case 3: // only have 3rd byte if rwFlag==0. This will be the data.
    if (rwFlag == 0 && (addr_hi > 0 || addr_lo > 16)) {
        datalog[logptr++] = dataByte;
        lastDataChunk = millis();
        break;
    }
} // end switch

goto nextBit;
}

// Deal with the hundreds digits of the systolic and diastolic
void setHundreds(char data){
    sysHundred = false;
    diaHundred = false;
    if(data>>4 == 1){
        sysHundred = true;
    }
    if((data & 0b00001111) == 1){
        diaHundred = true;
    }
}

boolean shouldWrite(){
    long now = millis();
    if((logptr > 0) && ((now - lastDataChunk) > WRITE_PAUSE)){
        return true;
    }
    return false;
}

```

```

String parseData(){
    // index 4 is hundreds
    // index 5 is sys
    // index 6 is dia
    setHundreds(datalog[4]);
    String sys = parseSysDia(datalog[5]);
    if(sysHundred){
        sys = "1" + sys;
    }
    String dia = parseSysDia(datalog[6]);
    if(diaHundred){
        dia = "1" + dia;
    }

    String reading = sysStr + sys + diaStr + dia;
    logptr=0;
    return reading;
}

// parse systolic or diastolic into a string to send out
String parseSysDia(byte b){
    char digit1 = hexval[b>>4];
    char digit2 = hexval[b & 0b00001111];
    char str[3] = {digit1, digit2, '\0'};
    return str;
}

void flashLed(int time){
    digitalWrite(onboardLed, LOW);
    delay(time);
    digitalWrite(onboardLed, HIGH);
    delay(time);
}

```

## WEB SERVER CODE

The following code, after packaged up into a standard .war file, executes in a java servlet container. Apache Tomcat was used for development; the same war file was then deployed on a Resin server run by an internet service provider.

### BloodPressureService.java

```
package pressureDrop;

import java.util.ArrayList;
import java.util.List;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.MultivaluedMap;
import javax.ws.rs.core.Request;
import javax.ws.rs.core.UriInfo;

@Path("/readings")
public class BloodPressureService {
    // Allows to insert contextual objects into the class,
    // e.g. ServletContext, Request, Response, UriInfo
    @Context
    UriInfo uriInfo;

    @Context
    Request request;

    //Return the list of readings to the user in a browser
    @GET
    @Produces(MediaType.TEXT_XML)
    public List<BpReading> getReadingsBrowser(){
        System.out.println("getReadingsBrowser");

        List<BpReading> allReadings = new ArrayList<BpReading>();
        for(List<BpReading> userReadings :
            ReadingsDao.instance.getReadings().values()){
            allReadings.addAll(userReadings);
        }

        return allReadings;
    }

    // Return the list of readings for applications
    @GET
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public List<BpReading> getReadings() {
        System.out.println("getReadings");
        List<BpReading> allReadings = new ArrayList<BpReading>();
        for(List<BpReading> userReadings :
            ReadingsDao.instance.getReadings().values()){
            allReadings.addAll(userReadings);
        }
    }
}
```

```

        return allReadings;
    }

    // returns the number of readings
    // Use http://localhost:8080/pressureDrop/rest/readings/count
    // to get the total number of records
    @GET
    @Path("count")
    @Produces(MediaType.TEXT_PLAIN)
    public String getCount(){
        int count = ReadingsDao.instance.getReadings().size();
        return String.valueOf(count);
    }

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    public BpReading postReading(MultivaluedMap<String, String> readingParams){
        int systolic = Integer.parseInt(readingParams.getFirst("systolic"));
        int diastolic = Integer.parseInt(readingParams.getFirst("diastolic"));
        String id = readingParams.getFirst("id");

        BpReading reading = new BpReading(systolic, diastolic, id);
        System.out.println("postReading: received:" + reading);
        ReadingsDao.instance.addReading(reading);
        return reading;
    }

    // Defines that the next path parameter after readings is treated as a
    // parameter and passed to the BloodPressureService
    // Allows to type http://localhost:8080/pressureDrop/rest/readings/1
    // "1" will be treaded as parameter reading and passed to ReadingResource
    @Path("/{reading}")
    public BpReadingResource getReading(@PathParam("reading") String id) {
        System.out.println("getReading 0: id:" + id);
        return new BpReadingResource(uriInfo, request, id);
    }
}

```

## BpReading.java

```

package pressureDrop;

import java.util.Date;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
//JAX-RS supports an automatic mapping from JAXB annotated class to XML and JSON
public class BpReading {

    private int systolic;
    private int diastolic;
    private String id; // uniquely identifies a user associated with reading
    // represents the date when the reading taken
    private long timestamp;

    // assume the empty constructor is needed by the framework
    public BpReading(){}

    public BpReading(int systolic, int diastolic, String id){
        this.systolic = systolic;
        this.diastolic = diastolic;
        this.id = id;
        // default to NOW for the reading date as a convenience
    }
}

```

```

        this.timestamp = new Date().getTime();
    }

    public int getSystolic() {
        return systolic;
    }

    public void setSystolic(int systolic) {
        this.systolic = systolic;
    }

    public int getDiastolic() {
        return diastolic;
    }

    public void setDiastolic(int diastolic) {
        this.diastolic = diastolic;
    }

    public long getTimestamp() {
        return timestamp;
    }

    public void setTimestamp(long timestamp) {
        this.timestamp = timestamp;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    @Override
    public String toString(){
        return "BpReading: id:" + id + " systolic:" + systolic + " diastolic:"
            + diastolic;
    }
}

```

## BpReadingResource.java

```

package pressureDrop;

import java.util.List;

import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Request;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;
import javax.xml.bind.JAXBElement;

public class BpReadingResource {
    @Context
    UriInfo uriInfo;
    @Context
    Request request;
}

```

```

String id;
public BpReadingResource(UriInfo uriInfo, Request request, String id) {
    this.uriInfo = uriInfo;
    this.request = request;
    this.id = id;
}

//Application integration
@GET
@Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public List<BpReading> getReadings() {
    List<BpReading> readings = ReadingsDao.instance.getReadings().get(id);
    if(readings == null){
        throw new RuntimeException("Get: BpReading with " + id
                                   + " not found");
    }
    return readings;
}

@PUT
@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public Response putReading(JAXBElement<BpReading> readingElement) {

    BpReading reading = readingElement.getValue();
    System.out.println("BpreadingResource.putReading:" + readingElement);
    return putAndGetResponse(reading);
}

@DELETE
public void deleteReadings() {
    System.out.println("deleteReading: about to delete:" + id);
    List<BpReading> readings
        = ReadingsDao.instance.getReadings().remove(id);
    if(readings == null){
        throw new RuntimeException("Delete: BpReading with " + id
                                   + " not found");
    }
}

private Response putAndGetResponse(BpReading reading) {
    Response res = Response.created(uriInfo.getAbsolutePath()).build();

    ReadingsDao.instance.addReading(reading);
    return res;
}
}

```

## ReadingsDao.java

```

package pressureDrop;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * temporary data access object for bp readings
 */
public enum ReadingsDao {
    instance;

    private Map<String, List<BpReading>> contentProvider
        = new HashMap<String, List<BpReading>>();
}

```



```

    public Map<String, List<BpReading>> getReadings(){
        return contentProvider;
    }

    public void addReading(BpReading reading){
        if(reading == null){
            return;
        }
        if(contentProvider.containsKey(reading.getId())){
            List<BpReading> readings = contentProvider.get(reading.getId());
            readings.add(reading);
        }else{
            ArrayList<BpReading> readings = new ArrayList<BpReading>();
            readings.add(reading);
            contentProvider.put(reading.getId(), readings);
        }
    }

    /**
     * get number of readings for for an id
     * @param id
     * @return count of readings for for an id that represents a user
     */
    public int getCount(String id){
        if(id == null){
            return 0;
        }
        List<BpReading> readings = contentProvider.get(id);
        if(readings == null){
            return 0;
        }
        return readings.size();
    }

    /**
     *
     * @param id user id
     * @return the last reading for a user
     */
    public BpReading getLastReadingForUser(String id){
        if(id == null || contentProvider.get(id) == null){
            return null;
        }
        int size = contentProvider.get(id).size();
        if(size == 0){
            return null;
        }
        // use the index to return the last reading
        return contentProvider.get(id).get(size - 1);
    }
}

```

## web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-
app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_2_5.xsd"
    id="WebApp_ID" version="2.5">
    <display-name>pressureDrop</display-name>
    <servlet>
        <servlet-name>Jersey REST Service</servlet-name>

```

```

<servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
<init-param>
  <param-name>com.sun.jersey.config.property.packages</param-name>
  <param-value>pressureDrop</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Jersey REST Service</servlet-name>
  <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
</web-app>

```

## readings.jsp

```

<%@page import="java.util.Date"%>
<%@page import="java.text.SimpleDateFormat"%>
<%@page import="java.util.List"%>
<%@page import="java.util.Map"%>
<%@ page import="pressureDrop.*" %>
<head>
<script type="text/JavaScript">
<!--
function timedRefresh(timeoutPeriod) {
    setTimeout("location.reload(true);",timeoutPeriod);
}
// -->
</script>
</head>
<body onload="JavaScript:timedRefresh(20000);">
<h1>Readings for Andrew Luxner</h1>
<%
SimpleDateFormat formatter = new SimpleDateFormat("yyyy.MM.dd hh:mm:ss a");
String userId = "00:06:66:49:2F:5B";

Map<String, List<BpReading>> map = ReadingsDao.instance.getReadings();
BpReading recent = ReadingsDao.instance.getLastReadingForUser(userId);
if(recent != null){
    String dateStr = formatter.format(new Date(recent.getTimestamp()));
%>
<h3>
Most recent reading <div id="recentDate" style="display:inline">(<%= dateStr %>)</div>:
<div id="recentSys" style="display:inline"><%= recent.getSystolic() %></div> /
<div id="recentDia" style="display:inline"><%= recent.getDiastolic() %></div>

</h3>
</br>

<TABLE BORDER="2">
<tr>
    <th>Systolic</th>
    <th>Diastolic</th>
    <th>Date/Time</th>
</tr>
<%
    List<BpReading> readings = map.get(userId);
    if(readings != null && (readings.size() != 0)){
        BpReading reading = null;
        // print the readings in reverse order that they were recorded
        for(int index = (readings.size() - 1); index >= 0; index--){
            reading = readings.get(index);
            dateStr = formatter.format(new Date(reading.getTimestamp()));

            <tr><td><%= reading.getSystolic() %></td>
                <td><%= reading.getDiastolic() %></td>
                <td><%= dateStr %></td></tr>

            <%
        }
    }

```

```
        }  
    %>  
</table>  
<%  
    int count = ReadingsDao.instance.getCount(userId);  
    out.println("The count is: " + count);  
}else{  
    out.println("<h3>No Readings found for user.</h3>");  
}  
%>  
</body>
```