ECE 4600 Group Design Project

Design and Implementation of a Framework for Remote Automated Environmental Monitoring and Control

by
Group 07

Jonathan-F. Baril
Shawn Koop
Paul White

Valerie Beynon
Sean Rohringer

Final report submitted in partial satisfaction of the requirements for the degree of Bachelor of Science in Electrical and Computer Engineering in the Faculty of Engineering of the University of Manitoba

Academic Supervisor(s)
Dr. Ken Ferens
Department of Electrical and Computer Engineering
University of Manitoba

Date of Submission
March 10, 2014

Abstract

Intelligent systems for monitoring and controlling environments are becoming increasingly popular. These systems allow users to easily handle repetitive tasks and remotely control devices scattered throughout an area. As a result, they are often used to increase user comfort or to reduce energy consumption. However, these intelligent systems are often very specialized and difficult to adapt to new environments. Our solution to this problem is to design a software framework that allows users to easily set up an intelligent system for personal use as well as allow developers to easily adapt the system for use in custom applications or environments. The prototype showcases the framework's ability to remotely monitor and control a simulated home environment. The prototype uses hardware that is energy efficient, inexpensive, and easy to interface with standard transducers and actuators in order to monitor and control an environment. The wireless peripheral nodes are programmed with the popular Arduino software, and the custom framework and development tools which we have designed make it easy for developers to create custom peripherals. Users control the system with a simple web accessible graphical user interface by associating events with actions that the system must execute (e.g. a switch causes a lamp's state to toggle). Users can also provide direct commands to the system (e.g. manually turn a lamp on or off). The demonstration system uses a combination of code written by group members that were experienced and familiar with the framework, as well as group members that had no previous experience working with the framework. Developing the demonstration system in this way tested and illustrated the robustness and ease of use of our software framework and development tools.

Contributions

	JFB	VB	SK	SR	PW
Software Architecture	●	●	●	●	●
Base Station Selection/Testing	●		●	●	●
Peripheral Core Selection	●	○	○	○	●
Peripheral Core Testing	●	○			○
Demonstration System Hardware Design/Impl.	●				
Demonstration System Embedded System Code	●	●			
Demonstration System Testing	●	○	○	○	○
Wireless Protocol/Packet Structure Design/Impl.		●			
Peripheral Core Embedded System Code		●			○
Base Station Transceiver Embedded System Code		●			
Base Station to Peripheral Network Interfacing		●			●
Developer Workflow Design		○	○		●
Developer Workflow Impl. (Peripheral Descriptors)	○		○		●
Developer Workflow Impl. (Peripheral Cores)	○	●			○
Plugin Framework Design/Implement/Test	○	○	●	○	●
Attach Peripheral Interface to Executive Software			●	○	●
Service Engine Design			●		●
Service Engine Implementation			●		
Service Engine Testing			●	○	○
Web-server to User Interface Communication			●	●	
Web-server to Main Program Communication			●		
Web-server Testing			●		
User Authentication			●	●	
User Interface Method Selection			○	●	
User Interface Design/Impl.				●	
User Interface Testing			○	●	
Project Management	●		○		○

Legend: ● Major Contributor ○ Minor Contributor

Acknowledgements

The most significant contributions to our collective experience has come from the large number of people who have played a part, either directly or indirectly, in making this design project a reality.

First we would like to thank our advisor Dr. Ken Ferens, for playing the role of the client, and for granting us the freedom, independence and opportunity to jump into the void and soar on our own wings.

Dr. Robert McLeod, Dr. Marcia Friesen, and the graduate students of the Internet Innovation Center, thank you for graciously allowing us the use of your lab (and more critically, your whiteboards) in order to host design meetings, sketch out software, and perform the day-to-day work of this project in a quiet place. In no way could we repay you for your patience, generosity, and ability to withstand our disruptions, for the past year.

Dr. Ahmad Byagowi, thank you not only for your work and support with the ArduIMU v4 which you more than willingly shared with us, but also for introducing us to the beauty and joy of homebrew electronic design which inspired this project. We owe you thanks for your enthusiasm, technical advice and guidance over the past few years.

Zoran Trajkoski, we owe you many thanks for taking time out of your busy schedule to hand-assemble prototype ArduIMUs for us to use and fix the ones we managed to break. Without your help this project would have been little more than lines of code on a computer screen.

Dr. Zahra Moussavi, many thanks for taking many of us on as NSERC undergraduate research students after 2nd year and as a result providing us with one of the most wonderful and formative experiences of this entire degree. You and the graduate students in your lab have had an immeasurable impact on how this project turned out.

Daniel Card, thank you for the insight and experience you were able to provide when we were way out of our technical and engineering depth.

Aidan Topping, thank you for entertaining our various questions and concerns about technical communications, but mostly for willingly enduring the drafts of this and the other reports submitted throughout the course.

Dr. Behzad Kordi, thank you for your encouragement and good humour in conducting and organizing the course.

Sinisa Janjic, Ken Biegun, Allan McKay and Mount First Ng, we are incredibly grateful for your assistance in helping us order and acquire many of the parts required for this project.

Cory Wiebe, thank you for providing us with advice and tools to resolve some of the mechanical issues that cropped up during the design.

Benjamin Baril, many thanks for your assistance in the machining and mechanical design of our system enclosures. You are the reason our stuff looks so nice.

Kyle Nemez, and many of the folks in Dr. Joe LoVetri's Biomedical Imaging and Biosensor Research Facility, thank you so much for your help in antenna design, and in letting us access various bits of your testing equipment and knowledge so we could improve this project.

James White, thank you for lending us your experience and suggestions and intentionally breaking our project and final report so we could rebuild it and make it better.

Hack A Day, thank you for inspiration.

Our fellow ECE 4600 classmates, thank you for being willing to share your design experience and knowledge, even though you were all busy enough with your own projects. Also, thank you for not holding a grudge following the impromptu lecture on design specifications changes. Yes, that was our doing.

But most of all, we owe an immense debt of gratitude to our friends and families, in particular those who patiently endured and suffered with us through the past 5 years of this degree despite our complaints, frustration, technical ramblings, neglect, and absence as a result of mid-terms, finals, and the endless stream of engineering labs, assignments, and projects. Thank you for your love, encouragement, and patience. It seems only fitting that we dedicate this culmination of 5 years worth of blood, sweat, and tears, to you.

Table of Contents

List of Figures

List of Tables

Nomenclature

Symbol	**Equation(s)**	**Description**
c	2.5	Speed of light [m/s]
d	4.1	Distance [m]
D_{out}	2.9 & 2.10 & 2.11	Analog-to-digital converter output value [unitless]
E	2.6	Battery energy capacity [mAH]
f	2.5	Frequency [Hz]
f_c	2.1 & 2.4 & 2.5	Center frequency in [Hz]
I_{avg}	2.6 & 2.7	Average current of [A]
I_s	2.7	Sleep current consumption [A]
I_t	2.7	Transmission current consumption [A]
k	2.1	Desired RF channel [unitless]
l	2.3 & 2.4 & 2.5	Length of a quarter-wavelength monopole [m]
L	2.7	Length of packet [bits]
P	2.7	Packet transmission period [s]
R	2.7	Transmission rate of [bps]
R_{div}	2.8	Voltage divider series resistor resistance [Ω]
R_{pot}	2.8 & 2.9 & 2.10 & 2.11	Current potentiometer resistance [Ω]
R_{potmax}	2.8 & 2.10	Maximum potentiometer resistance [Ω]
T	2.6	Battery life [years]
t_p	4.1	Propagation time [m/s]
V_{in}	2.8 & 2.10	Input voltage [V]
v_p	2.2 & 2.4 & 4.1	Propagation speed [m/s]
V_{ref}	2.9 & 2.10	Analog-to-digital converter reference voltage [V]
λ	2.2	Wavelength [m]
λ_c	2.3	Center wavelength [m]

Glossary

Term	**Definition**
Action	Task that can be executed by a peripheral
Base Station	Computer that directs the activities of peripherals
MessagePack	Algorithm for efficiently serializing and de-serializing data
Peripheral	Node in a controlled environment that can monitor or control some aspect of the environment
Peripheral Descriptor	Program that describes to the base station how to communicate with a specific type of peripheral, including the triggers and actions it supports
Service	Relationship between a set of triggers and a set of actions that defines a behaviour (e.g. if Switch A turns ON, activate Lamp B)
Trigger	Flag that can be set by a peripheral to indicate that a specific event has occurred (e.g. set the position of a light switch)
Type Code	Unique number used to identify the type of peripheral (e.g. light switch, coffee maker, etc.)

Chapter 1

Introduction

1.1 Motivation

Home automation systems are systems which allow users to remotely interact with a variety of devices scattered through the home. Currently these systems are well-established products in the market and they are becoming increasingly popular. In addition to being able to directly monitor and control various environmental conditions through the use of devices such as thermometers and light switches, some home automation systems even have the capability to coordinate the activities of multiple devices, resulting in actions occurring in reaction to certain conditions. Unfortunately, these systems are typically designed to accommodate only a limited selection of devices and behaviours. This project was initiated to address this shortcoming by creating a system which provides users with all of the regular features of a home automation system, as well as the ability to create their own specialized devices and behaviours which would better suit their needs. Thus, the proposed system would cease to be just a home automation system, and instead be a generalized environmental automation system, one which is not limited to use within a home.

1.2 Project Scope

The primary objective of this project was to create a software framework for a generalized automation system. A necessary component in realizing this goal was the selection and integration of hardware with which to develop the framework, as well as demonstrate the framework's functionality. From a hardware perspective, the system consists of a number of wirelessly networked devices and the user's personal device, such as a smartphone or a laptop, which can be used to interact with the rest of the system. At the centre of the network is the base station, which is responsible for managing the system's activities and communicating with the user's personal device. The other devices are the peripherals, which are responsible for carrying out the basic functions of the system, i.e. monitoring and controlling environmental conditions. The two software components of the project are the peripheral software and the base station software, where the base station software is further divided into the following areas: the peripheral interface, which coordinates communications between devices; the service engine, which manages pre- or user-defined system behaviours; the user interface; and the web server.

1.3 Possible Applications

Since our system is designed to be highly modular, the hardware can easily be swapped out for applications with different hardware requirements such as range, frequency band, etc. The hardware for this project was chosen to satisfy the context of a home automation system. Possible alternative contexts include:

- Monitoring environmental factors in a beehive
- Monitoring the state of a sandbag dam
- Geospatial temperature monitoring to detect when underground pipes might burst

Chapter 2

System Hardware

2.1 Hardware Overview

While the innovative component of our project is the framework, this massive software structure must run on hardware in order to monitor and control a physical environment. In this chapter of the report we detail the design, implementation, and testing of the system hardware.

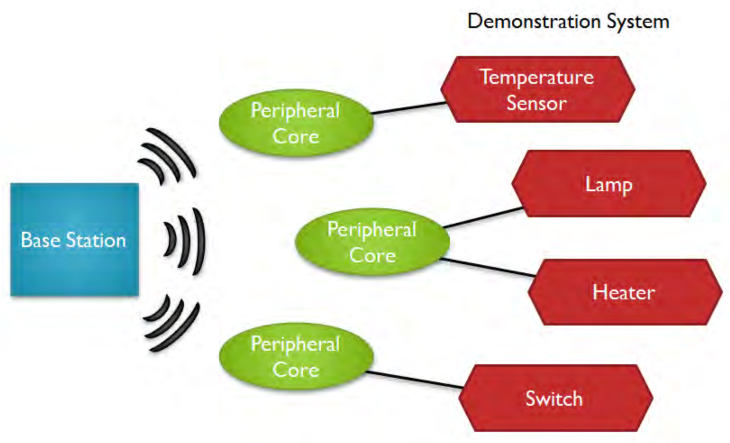


Figure 2.1: System hardware overview

As can be seen in Figure 2.1, the system hardware can be divided into three subsystems: the base station hardware, the peripheral core hardware, and the demonstration system.

The base station hardware represents the physical components on which the base station software (peripheral interface, service engine, etc.) executes.

The peripheral core hardware represents the physical system on which the peripheral software executes. These devices communicate wirelessly with the base station and via wired connections to the demonstration system hardware.

The demonstration system consists of all the extra components that are attached to the peripheral cores to form useful peripherals. These components allow our system to physically interact with its surroundings, i.e. control and monitor the local environment.

2.2 Base Station Hardware

The base station hardware consists of two essential components. The first is the central processing unit (the base station computer) and the second is the transceiver, which allows the base station to communicate with the peripheral network. This is illustrated in Figure 2.2.

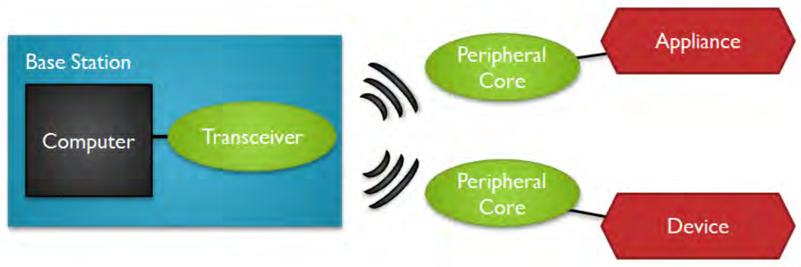


Figure 2.2: Base station overview

We selected a commercially available product to use as the base station computer: the

Raspberry Pi Model B [1]. The base station transceiver uses the same hardware that is used for the peripheral cores, the ArduIMU v4 (see Section 2.3). It is connected via USB cable to the base station computer. Through testing we verified that the combination of a Raspberry Pi with a peripheral core acting as a transceiver satisfied the requirements for the base station. While the system latency between the user interface and the peripherals borders on being slow, it is acceptable based on the requirements established in the proposal [2]. It does however present an area that could be optimized in future work done on this project.

2.2.1 Base Station Design

As mentioned, the base station hardware consists of two essential components. The first being the base station computer that does the majority of the processing and the second being the base station transceiver that allows the base station to communicate with the peripheral network.

Base Station Computer Design

Since one of the goals of the framework is to be expandable and customizable, the base station, in particular the base station computer, should be generic and flexible enough so that it can easily be upgraded or replaced if developers require something more powerful.

The only rigid requirements for the base station were derived from the planned high-level software architecture (see Section 3) which mandated that the hardware would need to support the Java programming language, multi-threaded applications, and be able to host a web server to service user interface requests. These requirements are most reasonably met by a conventional personal computer (PC) running an operating system (OS) such as Windows, Unix or Mac OS X.

We chose to use the Raspberry Pi Model B (shown in Figure 2.3) as our PC. This single-board computer can run various UNIX OS derivatives, therefore it satisfies the software requirements previously mentioned [1]. The Raspberry Pi is preferable to a regular full-sized PC for 3 main reasons:

1. It provides many of the form-factor benefits of an embedded microcontroller such as small size and power efficiency. The Raspberry Pi consumes at maximum 3.5 [W] or 1.5 to 5% of the 80 to 250 [W] used by a typical home computer, and is not much larger than a credit card [1, 3].

2. The Raspberry Pi is equipped with a 700 [MHz] processor and 512 [MB] of RAM, which means it very likely represents the bottom end of hardware on which a typical consumer might run our framework [1]. In other words, we would be designing the system to work in what should be considered to be a worst-case hardware scenario.

3. At the time of writing this report, the Raspberry Pi cost about $40 CAD making it a very cost-effective solution for both a consumer and our design team, especially since we already had access to 3 units.

Figure 2.3: Raspberry Pi

Base Station Transceiver Design

In reference to Figure 2.2, the Raspberry Pi acts as the computer subsection of the base station. However, in order for the base station to be able to communicate with the peripheral network the base station requires a suitable transceiver. While the Raspberry Pi allows programmers to have direct access to hardware GPIO as well as I^2C, SPI, and UART buses, we made the decision of using a transceiver that could communicate with the base station computer over USB since typical PCs do not have accessible GPIO or I^2C, SPI, and UART buses, but do have many USB ports.

Since the device we used as our peripheral core hardware (as discussed in Section 2.3) has an FTDI chip that allows it to communicate over USB we used an ArduIMUv4 as our base station transceiver. To do this, we simply reprogrammed a peripheral core to create a dedicated base station transceiver, connecting it directly via USB port to the Raspberry Pi. This was a logical solution since it simplified the design by making use of already existing hardware and software.

In summary, we selected two components to make up the base station hardware: the Raspberry Pi Model B combined with an ArduIMUv4 to act as a transceiver. This was the best choice of the options considered since it met our internal requirements including: supporting multi-threaded applications, support for selected programming languages, and the ability to host a web server, all the while being cost-effective and flexible enough to upgrade if more processing power was required.

2.2.2 Base Station Testing

In order to test the base station hardware, we simply verified that the Raspberry Pi was able to run the required development tools and applications by installing and using them on the device. However, it was only after developing the base station computer and transceiver software that we were able to fully test the performance of the Raspberry Pi running our software framework.

The Raspberry Pi is capable of running the required system software, albeit with considerable latency. Here, latency is defined as the delay between requesting a command via the user interface and the actual execution of the command. This latency is on the order of tens of seconds. We note however that the base station software has not yet been performance optimized, and that the Raspberry Pi Model B represents a “worst-case hardware” scenario. As a result, while tens of seconds is nowhere near an instantaneous response, it represents an acceptable latency since our proposal did not specify a requirement for system latency [2]. While our team deemed it outside of the scope of this project to pursue the issue much further, system latency is certainly an aspect of the design that could be improved as part of future work on this project.

In summary, the combination of a Raspberry Pi with a peripheral core to act as a transceiver proved to be successful. While the system latency borders on being slow it is still acceptable within the requirements established in the proposal.

2.3 Peripheral Core Hardware

The peripheral core hardware consists of a microcontroller, a radio-frequency (RF) transceiver and supporting circuitry. These components allow a peripheral to communicate with the base station as well as provide control signals to the external circuitry that makes up the demonstration system.

After investigating various potential options, we selected a soon-to-be commercially available product, the ArduIMU v4 (ArduIMU) [4]. In testing we found that the ArduIMU was able to meet the proposed hardware requirements for the peripheral core hardware when equipped with a simple quarter-wave monopole antenna.

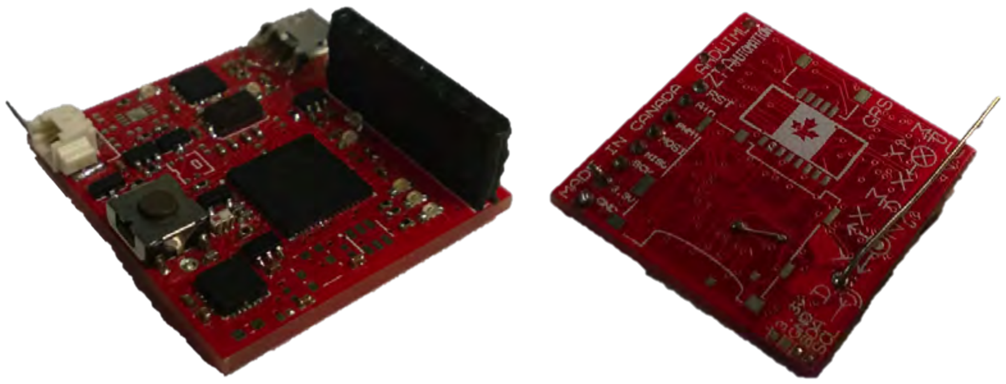
2.3.1 Peripheral Core Design

We first set about designing custom peripheral core hardware according to the set of specifications in our project proposal [2]. However, we quickly realized that we could produce a product that was a better fit for our client's wishes by modifying the originally proposed specifications. Our client and design team agreed on a set of revised specifications which are listed in Appendix A.

The ArduIMU v4 (shown in Figure 2.4) is a commercial solution that meets our revised requirements. It combines the required microcontroller, transceiver, and supporting circuitry into a compact form factor (about 30 [mm] x 30 [mm]) perfectly suited for use as a peripheral core[4]. Although we explored the idea of designing a custom solution using embedded microcontrollers[1] it became clear to us that using the already available commercial solution was preferable from the perspective of our client. A custom solution would only achieve a marginal improvement in performance characteristics, at the cost of significant design and testing time. Furthermore, the ArduIMU uses an Arduino-compatible microcontroller, and can leverage the simplicity of the Arduino framework. This can significantly accelerate software development time for developers.

As discussed in Section 2.3.2, the stock ArduIMU v4 was verified to meet all of our requirements except the range requirement. In order to meet this requirement we found that we needed to add an antenna to the PCB since the stock device did not have one supplied. After discussing with the ArduIMU development team and some colleagues at the university, we found that adding a simple quarter-wave monopole antenna to the antenna output on the PCB would likely improve our transmission range.

[1]We considered the ATmega128RFA1, which is used by the ArduIMU, as well as the CC430F6135, a Texas Instruments system on a chip that combines an MSP430 with an RF transceiver core [5, 6]. (The latter was familiar to our team).



(a) **(b)**

Figure 2.4: ArduIMU v4: (a) top, (b) bottom

Peripheral Core Antenna Design

The design of a quarter-wave monopole antenna is fairly straightforward and simply required cutting a length of wire (in our case 22 AWG solid core wire that we had available in our lab) to be as long as a quarter of the wavelength associated with the desired center frequency of the antenna. The Atmega128RFA1 user guide gives us Equation 2.1 which allows us to determine the center frequency f_c according to the desired RF channel k. Since we do not and cannot guarantee that a specific channel will be used, we choose to place our antenna at the center of all channels, i.e. $k = 18.5$ for which $f_c = 2.442$ [GHz].

$$f_c = 2405 + 5(k - 11)\ [MHz]\ for\ k = 11, 12, ..., 26 \tag{2.1}$$

Knowing the frequency f of a wave, its wavelength λ and propagation speed v_p are related by Equation 2.2. The approximate length L of a quarter-wavelength monopole with center wavelength λ_c is given by Equation 2.3. Combining the two yields Equation 2.4, which allows us to directly determine the required length of our peripheral core antennas [7].

$$v_p = \lambda f \tag{2.2}$$

$$l \approx \frac{\lambda_c}{4} \tag{2.3}$$

$$l \approx \frac{v_p}{4 f_c} \tag{2.4}$$

In order to determine the propagation speed of the EM waves in our system, we first realize that in the context of our demonstration system our peripherals will be used in a dry air-filled environment, i.e. freespace. Since we know that in free-space the propagation speed v_p of an EM wave is just shy of the speed of light c, we can approximate Equation 2.4 using Equation 2.5 [7].

$$l \approx \frac{c}{4 f_c} \tag{2.5}$$

Finally, using Equation 2.5, and given the desired antenna center frequency of $f_c =$ 2.442 [GHz], we determine that the length of our quarter-wavelength monopole antenna should be about 3.07 [cm], or given our manufacturing tolerances: just longer than 3 [cm].

In summary, we selected the ArduIMU v4 to be the peripheral core, and designed a quarter-wavelength monopole antenna to improve the communication range of the device so it could meet our range requirement.

2.3.2 Peripheral Core Testing

Since most of the hardware specifications, namely the speed, A/D resolution, and the number of available I/O, are guaranteed by the datasheet of the ATmega128RFA1, testing the peripheral core hardware came down to verifying that the range and battery life requirements were met.

Peripheral Core Range Requirement Testing

In order to verify that the ArduIMU met the range requirement of our specification, we configured two ArduIMUs to communicate with each other. The first device was connected to a computer and acted as a dedicated receiver. It printed received messages via serial port to the computer terminal. The second ArduIMU acted as a dedicated transmitter, sending out messages containing test data and a counter that was incremented after each transmission. This allowed us to detect missed or corrupted packets. The transmitter ArduIMU was moved progressively further away from the receiver ArduIMU while we monitored the message received on the computer terminal. Both ArduIMUs were held in the same orientation to reduce polarization loss[2]. This test setup is summarized in Figure 2.5.

We performed this test in two locations:

1. Outside one of the University buildings, where we were able to achieve totally unimpeded line of sight with minimal noise and RF interference.

2. In a hallway on the sixth floor of the EITC which more accurately represented a noisy environment filled with RF interference.

Without an antenna, we found that the ArduIMU achieved only about half of our range requirement (about 20 [m]). After designing the antenna described in Section 2.3.1 and installing an antenna to both the transmitter and receiver, we again tested the system and and achieved a range of 45 [m] or slightly above our range requirement of 40 [m]. At this range our system was able to successfully transmit 44-50% of packets. While this number is not particularly high, we note that this test was done without a network layer to manage retransmissions and guarantee packet delivery (See Section 4.1.3). Since our device is able

[2]Monopole antennas are polarized along the direction of the antenna, i.e. vertical antennas are vertically polarized and as a result do not receive horizontally polarized electromagnetic waves very well [7].

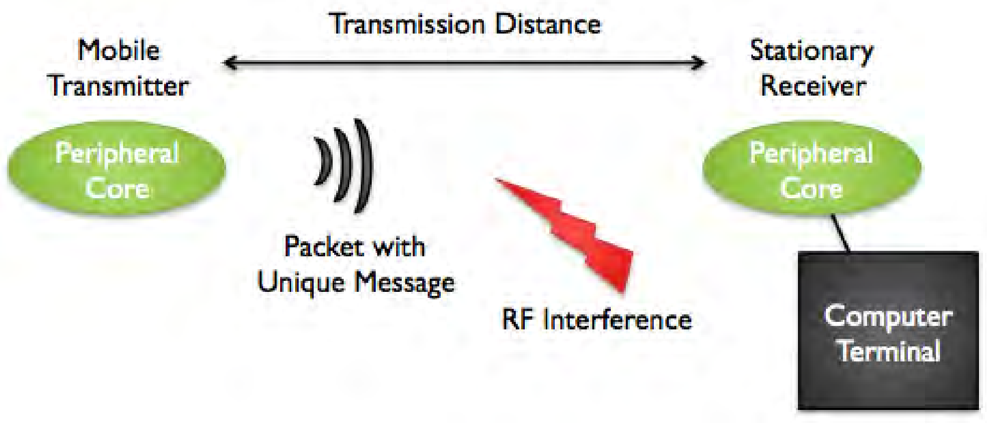


Figure 2.5: Transmission range test setup

to meet the requirement of communicating at 40 [m] or above, our software team developed algorithms to compensate for the losses in the channel.

Peripheral Core Battery Life Testing

Confirming that the device met the battery life requirement was more difficult than the range requirement since it was not practical (or possible) for us to simply connect an ArduIMU to a battery and wait the 6 - 7 months required to verify if the ArduIMU met the specification. Instead, we came up with a method of extrapolating and estimating the battery life based on current consumption.

Assuming that an ArduIMU is connected to a battery with a capacity of E in [mAH] and consumes an average current of I_{avg} in [A] we can establish the relation defined by Equation 2.6 which gives us the battery life T in years.

$$T = (\frac{E\ [mAH]}{I_{avg}\ [A]})(\frac{1\ [day]}{24\ [hours]})(\frac{1\ [year]}{365\ [days]}) \tag{2.6}$$

Calculating the battery life for a sleeping ArduIMU is simple since the ArduIMU current consumption does not change. We empirically determined that the sleeping current consumption for the ArduIMU board with processor in deep sleep mode but with active sensors is 200 [μA]. Plugging this into Equation 2.6, we get a battery life estimate of 8.2 months. Since this exceeds the typical sleeping battery life requirement by over a month, we are confident that the ArduIMU meets the specification.

Calculating the battery life for an ArduIMU in normal use is more involved since it requires that we estimate the average current consumption of the ArduIMU. We can do this by assuming that the ArduIMU is transmitting a packet of length L [bits] at a rate of R [bps] once every transmission period P [s]. The ArduIMU is assumed to return to deep sleep between transmissions and consumes I_s [A] of current while deep sleeping. On the other hand the ArduIMU consumes I_t [A] of current while transmitting (i.e. at the highest power consumption level). The relationship shown in Equation 2.7 gives us the effective average current, I_{avg} [A], based on the parameters defined above.

$$I_{avg} = \frac{L}{PR} I_t + (1 - \frac{L}{PR}) I_s \tag{2.7}$$

According to the test parameters outlined by the hardware specification, the battery life normal use test is done when the ArduIMU transmits a 10 byte (80 bit) packet every second. We assume that that transmission is done at 250 [kbps] (the slowest data rate supported by the ATmega128RFA1) since this would give us the worst case battery consumption since the transmitter spends more time transmitting per second [5].

Since it is impractical to empirically determine the ArduIMU board transmitting current consumption we instead add up the maximum current consumption of the ATmega128RFA1 specified in its datasheet as well as the maximum current consumption of all the sensors and peripheral core circuitry on the board as specified in their respective datasheets. This gives us a worst case estimated current consumption of 60 [mA] while transmitting. To this value we added 40 [mA] as a safety factor giving us an I_t of 100 [mA] along with our empirically determined I_s of 200 [μA].

Running all of the above values though Equation 2.7 we get an estimated worst case I_{avg} while transmitting of 232 [μA]. Using Equation 2.6 with our I_{avg} value, we get a battery life estimate of 7.0 months. Since we used conservative worst-case assumptions and because the calculated value of 7 months still exceeds our typical normal use battery life requirement by a month we feel confident that the ArduIMU meets the normal use battery life specification.

In summary, based on our calculations and the measured current consumption of the device, we extrapolated that the battery life of the ArduIMU will theoretically exceed the requirements and after the addition of the quarter-wavelength monopole antenna discussed in Section 2.3.1, our peripheral core hardware is able to meet our range requirement. Since the speed, A/D resolution, and number of GPI/O pins are guaranteed and verified working, this means that the ArduIMU v4 fully meets our hardware specifications.

2.4 Demonstration System Hardware

The demonstration system serves the purpose of demonstrating the capabilities of the entire software framework. This is done in a variety of ways. One way is to simply use the temperature and light sensors which happen to be included in our peripheral core hardware. A second way, which demonstrates the flexibility of our system, is to use various custom peripheral circuits. These include an LED light strip, an indicator box, a control box, and a compound peripheral. These are described in detail below.

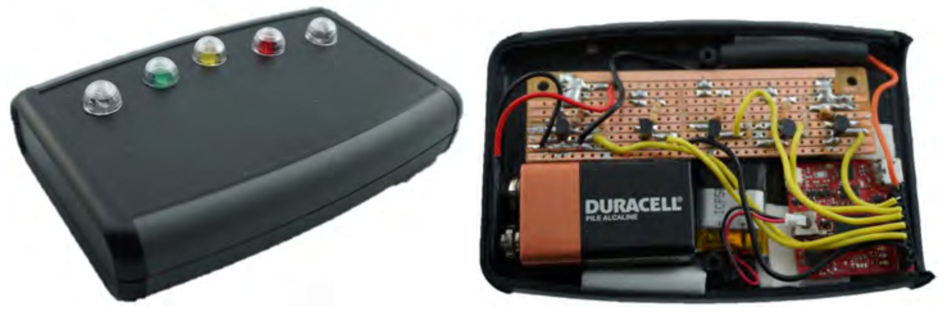
2.4.1 Demonstration System Design/Implementation

The combination of all of the software and hardware layers that lie above the demonstration system allow a user to remotely monitor and control the set of inputs, outputs, and buses that connect the peripheral cores to the demonstration system. So, for the demonstration system we simply had to design circuits that allowed us to take the different types of signals produced by the peripheral cores and use them to control standard home devices[3].

We identified the following set of signals produced by the peripheral cores and controllable by our system software: digital input & output, analog input, PWM output, and an SPI bus.

Indicator Box Peripheral

In order to demonstrate the use of the digital output and PWM output signals, we designed an indicator box peripheral. The schematic for this peripheral appears in Figure B.2 in Appendix B. The bill of materials appears in Table C.IV in Appendix C. Interior and exterior views can be seen in Figure 2.6.



(a) **(b)**

Figure 2.6: Indicator box peripheral: (a) exterior, (b) interior

The indicator box takes the peripheral core's PWM output as well as the 4 digital outputs and connects each of them through a gate resistor[4] to an NMOS transistor. Each

[3]In designing the demonstration system we first made the grave mistake of not formally codifying its purpose. As a result, our first attempt at its design resulted in the creation of designs for several expansion boards that would allow the peripheral core to control various transducers and actuators. Unfortunately, these did not serve to advance the purpose of the demonstration system. Since implementing and testing these boards would have represented a significant time-investment for little benefit, we decided to archive the designs and did not use them in our final demonstration system. As a result, these designs are not discussed in this report, however the schematics for the proposed designs can be found in Appendix B.

[4]MOSFETS are highly susceptible to electrostatic discharge (ESD). Since several of the transistors were damaged during assembly by the tape used to secure them, we added the series gate resistors to help protect the MOSFETS.

transistor acts as a switch that turns on the corresponding LED that is fed by a 9 [V] battery. As a result the LEDs connected to the digital outputs can be turned either on or off, while the LED connected to the PWM output can be dimmed by varying the PWM duty cycle.

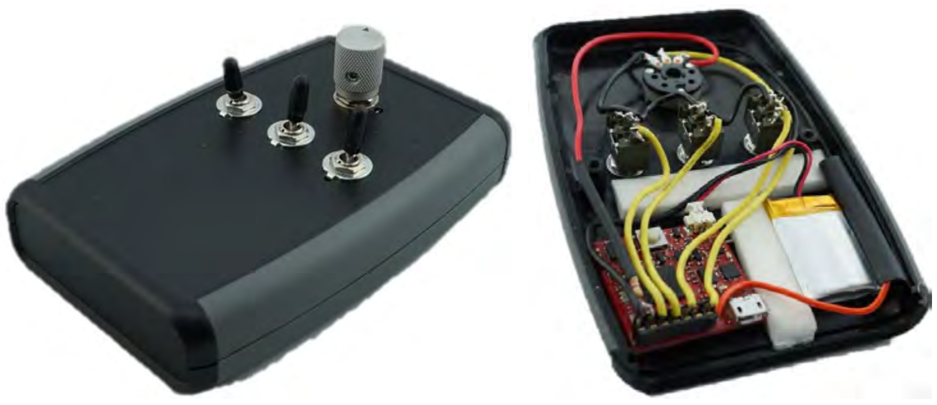
Since the entire circuit is placed in a small closed enclosure, a magnetic reed switch is connected to the peripheral core reset pin inside the enclosure. By placing a magnet on the outside of the enclosure near the location of the reed switch, the whole peripheral can be externally reset without opening the enclosure.

Since each LED consumes up to 20 [mA], the indicator box peripheral needs a higher capacity battery than simply the small 150 [mAH] lithium polymer battery that powers the peripheral core. Since 9 [V] batteries are more readily available to consumers and the two batteries fit very nicely within the enclosure (as can be seen in Figure 2.6b), we opted to have the LEDs powered by a 9 [V] battery and the ArduIMU powered by a separate small 150 [mAH] lithium polymer battery.

Control Box Peripheral

We designed a control box peripheral to demonstrate the use of the digital input and analog input signals. The schematic for this peripheral is shown in Figure B.3 in Appendix B. The bill of materials appears in Table C.V in Appendix C, and a photograph of the control box can be seen in Figure 2.7.

The control box produces digital and analog signals to be read and monitored by the peripheral core. The digital input signals are produced by 3 single-pole single-throw (SPST) switches. These switches can be toggled to connect the digital inputs to logic low, the circuit ground, i.e. 0 [V]. When the switches are disconnected, the peripheral core internal pull-up resistors are configured to pull the digital input back up to the high logic level of 3.3 [V]. The analog input is generated by a combination of the potentiometer and voltage divider.



(a) **(b)**

Figure 2.7: Control box peripheral: (a) exterior, (b) interior

The peripheral core internal analog reference, i.e. the maximum analog input voltage, of the peripheral core is set to 1.8 [V], and the potentiometer is fed with a 3.3 [V] source [5]. In order to scale down the potentiometer input voltage so that it is always within 0 to 1.8 [V] we use a 47 [kΩ] resistor in series with the 50 [kΩ] potentiometer. Equation 2.8 gives the output voltage of the potentiometer and voltage divider combination where: V_{in} is the input voltage of the circuit (3.3 [V]), R_{potmax} is the maximum resistance of the potentiometer (50 [kΩ]), R_{pot} is the current resistance of the potentiometer as determined by location of the wiper, and R_{div} is the value of the series resistor in the voltage divider circuit (47 [kΩ]).

$$V_{pot} = V_{in} \frac{R_{pot}}{R_{potmax} + R_{div}} \tag{2.8}$$

Equation 2.9, gives the digital output value D_{out} produced by the peripheral core analog-to-digital (A/D) converter given an input voltage V_{pot} with a A/D reference of V_{ref} (1.8 [V]) [5].

$$D_{out} = 1023 \frac{V_{pot}}{V_{ref}} \tag{2.9}$$

Combining equations 2.8 and 2.9 results in equation 2.10.

$$D_{out} = 1023 \frac{R_{pot}}{R_{potmax} + R_{div}} \frac{V_{in}}{V_{ref}} \tag{2.10}$$

Evaluating Equation 2.10 using the values previously specified for each parameter produces equation 2.11.

$$D_{out} = 1023 \frac{R_{pot}}{50\ [k\Omega] + 47\ [k\Omega]} \frac{3.3\ [V]}{1.8\ [V]} \tag{2.11}$$

Equation 2.11 clearly demonstrates the linear dependence of the digital output value D_{out} on the the value of R_{pot}. Furthermore, the equation makes clear that since we have constrained V_{pot} to remain below V_{ref} through the use of a voltage divider, the D_{out} can never exceed 1023. The maximum output value in fact is 966, and is a result of R_{div} not being equal to R_{potmax}. However, since the selected value of R_{div} is the closest 5% resistor value available and we are making use of more than 94%, or essentially all, of our available A/D converter range we judged that this was good enough for our application and did not need to be improved use the full range of 1023.

As with the indicator box, since the entire circuit is placed in a closed enclosure the magnetic reed switch reset circuit is also included in the control box to allow the whole peripheral to be externally reset without opening the enclosure.

The control box only uses one battery, the small 150 [mAH] lithium polymer battery that powers the peripheral core.

Light Strip Peripheral

In order to demonstrate the SPI bus, we designed a light strip peripheral that could be controlled over SPI. The schematic for this peripheral is shown in Figure B.1 in Appendix B. The bill of materials appears in table C.III in Appendix C. Figure 2.8 shows the light strip peripheral with the blue logic level shifter exposed.

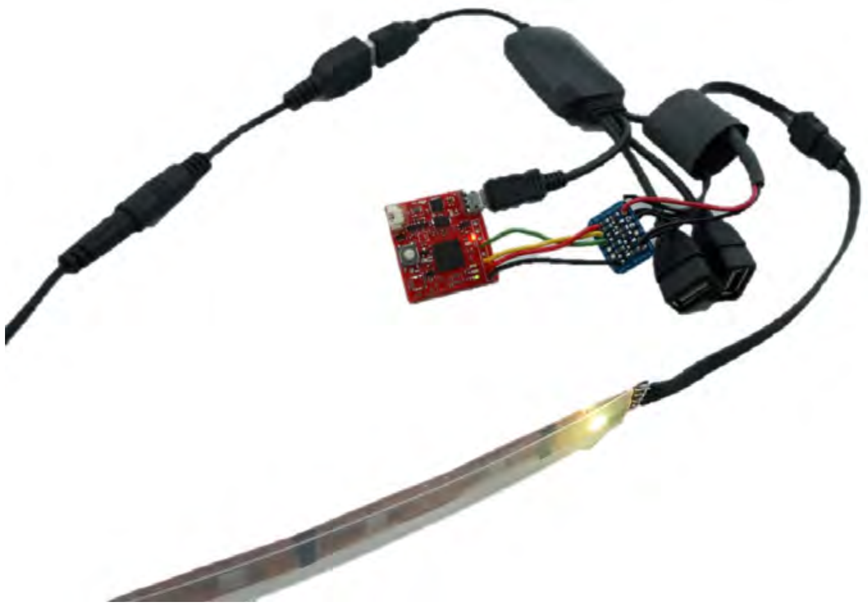


Figure 2.8: Light strip peripheral

The light strip peripheral is designed around a commercial light strip containing 32 digitally addressable RGB LEDs. Each individual light strip segment has connections for a 5[V] power supply input, clock input (CI) and output(CO) pins, and data input (DI) and output (DO) pins. The CI and DI pins of the first light strip segment are connected to the peripheral core serial clock (SCK) and master-in-slave-out (MOSI) pins of the SPI bus through a bi-directional logic level shifter. The logic level shifter is required since the light strip segments operate on 5 [V] whereas the peripheral core only produces 3.3 [V] signals. The CO and DO pins are then wired directly into the next light strip segment's CI and DI pins. In this way, many light strips can be daisy chained together. For our demonstration system we only used 1 [m] of light strip containing 16 segments.

Since the light strip peripheral does not have an enclosure, the peripheral can be reset using the hardware push button on the peripheral core.

The light strip peripheral does not use any batteries and is instead supplied by a 5 [V] wall supply. According to the light strip specifications the supply must be able to supply 60 [mA] per RGB LED or just below 2 [A] of current for every meter of light strip [8]. In our case we used a power supply rated for 2 [A] as the power consumption of the peripheral core is negligible when added to power consumption of the LEDs.

Compound Peripheral

Since the previously discussed peripherals demonstrate that our system can use all the various signals produced by the peripheral core, the purpose of the compound peripheral is to demonstrate that our system is capable of controlling devices found in a standard home. The schematic for this peripheral is shown in Figure B.4 in Appendix B. The bill of materials appears in Table C.VII in Appendix C. The compound peripheral can be seen in Figure 2.9.

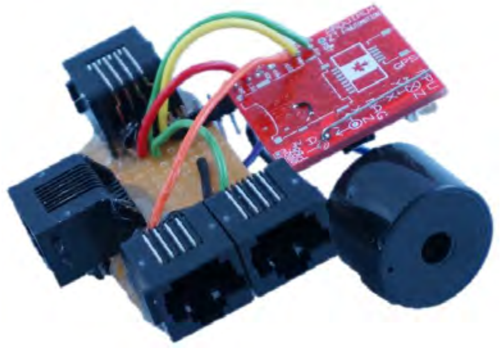


Figure 2.9: Compound peripheral

Some of the most common electrical devices found in homes are lamps and lights of various types. Other common devices include heaters, fans, and tea kettles. These devices can easily be switched on and off by connecting them or disconnecting them to the mains supply. While designing a device that can switch mains voltage devices is relatively straightforward, inexpensive commercial devices already exist that allow mains voltage to be switched using a low voltage microcontroller. Two such devices are the PowerSwitch Tail 2 and PowerSSR Tail which use an electromechanical relay and solid state relay as mains voltage switching elements. [9, 10]. The PowerSSR Tail is shown in Figure 2.10.

Figure 2.10: PowerSSR Tail

Using these PowerSwitch Tails simply involves applying 3.3 [V] across the positive and negative input terminals of the devices to switch the relay on, or appling 0 [V] to switch the relay off. Since the PowerSwitch Tail 2 uses an electromechanical relay and can switch up to 1500 [W] it can be used to control high power devices such as tea kettles and heaters, or inductive devices such as fans and motors [9]. The PowerSSR tail only supports non-inductive devices up to 300 [W] but can switch devices more quickly, quietly, and can endure more switching cycles since it uses a solid-state relay [10]. As a result, the PowerSSR tail is perfectly suited to turning on or even dimming lamps.

Dimming lamps connected to mains voltage requires the use of a zero-crossing detection circuit to synchronize the switching of the lamp. We both designed and successfully tested such a circuit on a breadboard. However the final demonstration system does not include this circuit due to manufacturing issues[5]. This report does not discuss the circuit in further detail since it is an optional and non-essential component. The interested reader can however

[5]Due to an overzealous attempt to squeeze the zero-crossing detection circuit into too small a space, the prototype did not operate correctly.

view the proposed schematic in Figure B.5 in Appendix B.

Along with switching home appliances, we identified that another possible use for our system was to secure a home. To replicate how one might use our system to secure a home we attached a magnetic contact switch to the compound peripheral along with a buzzer. The magnetic contact switch is connected to the circuit ground reference of 0 [V] so that when the contacts are together, the switch is closed and the peripheral core sees a low logic level. Conversely, when the contacts are apart, the switch is open and the internal pull-up resistors in the peripheral core pull the value up to 3.3 [V] so the the peripheral core sees a high logic level. If one part of the contact switch is placed on the frame of a door or window and the second part of the switch is placed on the door or window itself it is then possible for our system to detect if the door or window is open or closed. The buzzer attached to this peripheral can then be activated by applying 3.3 [V] to the input pins.

Since all of these appliances, door or window may not be in close proximity, the PowerSwitch Tails and magnetic contact sensor are individually wired to RJ-45 connectors allowing the consumer to use an appropriate length of CAT-5 cable to span the gaps between the appliance, door or window and the compound peripheral.

The compound peripheral does not have an enclosure, therefore the peripheral can easily be reset using the hardware push button on the peripheral core.

The compound peripheral in our demonstration system is powered by a commercial, portable USB solar charger with integrated battery (see Table C.VII in Appendix C).

Summary

In summary, the demonstration system is composed of 4 different types of peripherals:

1. The indicator box peripheral that demonstrates our system's ability to produce and control digital and PWM outputs.
2. The control box peripheral that provides a physical and interactive way to demonstrate

our system's ability to monitor digital and analog inputs.

3. The light strip peripheral that demonstrates our system's ability to control devices over an SPI bus.

4. The compound peripheral that demonstrates our system's ability to control standard home appliances like lamps, fans, heaters and tea kettles, as well as provide basic home security.

2.4.2 Demonstration System Testing

We first tested and debugged each of the demonstration system peripheral hardware components using an Arduino UNO since we had insufficient ArduIMUv4 peripheral cores available at the time. However, once we received more peripheral cores we were able to verify that the individual peripherals functioned as intended when connected to a peripheral core.

A full test of the demonstration system was completed after the system software was implemented. During this test we were able to verify that all the components of the demonstration system were able to send and receive signals to and from the peripheral cores, which in turn passed them up to the base station for processing. Through this, we were able to confirm that the demonstration system accomplishes its purpose of demonstrating the capabilities of the software framework we developed.

Chapter 3

Software Architecture

The software architecture is the most critical design element in the development of any complex software. Many design goals need to be taken into account, including simplicity, flexibility and efficiency. This chapter discusses the base station and how it interacts with peripherals and users, and some major design decisions that were made in order to make everything work well.

3.1 System Software Summary

Figure 3.1 illustrates a simplified version of the software architecture. There are two types of entities in the system: a base station and the peripherals. Each peripheral provides a set of functions that it is capable of performing, such as reading from sensors, communicating with attached devices, writing to digital I/O pins, etc. These functions can be remotely called from software on the base station called a **peripheral descriptor**. The peripheral descriptor represents the peripheral within the software space of the base station.

The user is able to control the peripherals in their system by using entities called **services**. A service is created through the graphical user interface, and has one or more inputs and one or more outputs. Each output corresponds to a task that must be performed by

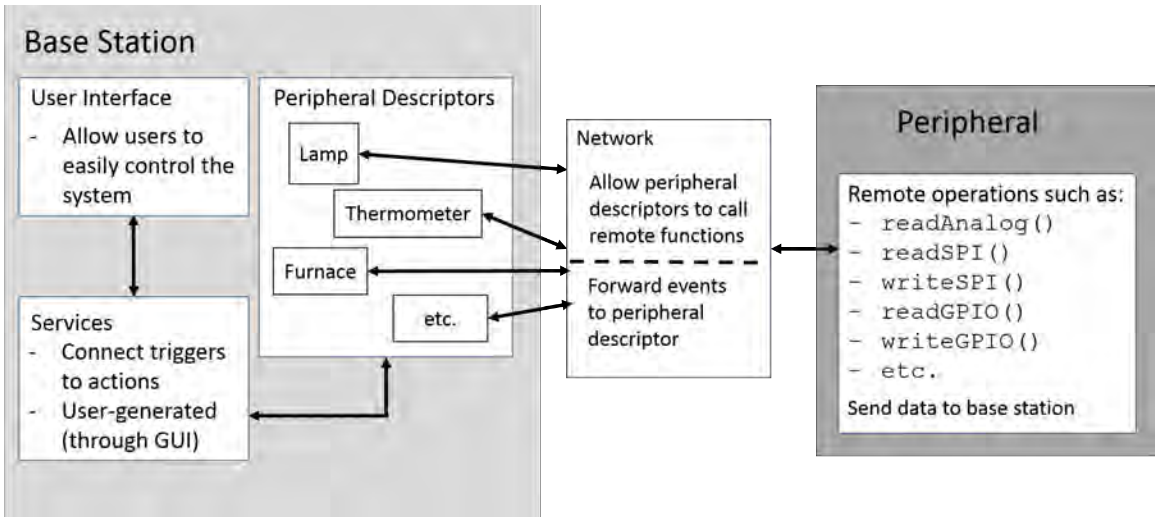

Base Station
User Interface
- Allow users to easily control the system
Services
- Connect triggers to actions
- User-generated (through GUI)
Peripheral Descriptors
Lamp
Thermometer
Furnace
etc.
Network
Allow peripheral descriptors to call remote functions
Forward events to peripheral descriptor
Peripheral
Remote operations such as:
- readAnalog()
- readSPI()
- writeSPI()
- readGPIO()
- writeGPIO()
- etc.
Send data to base station

Figure 3.1: A simplified block diagram of the software architecture

some peripheral (such as turning on the furnace) while each input corresponds to some event that a peripheral is capable of detecting (such as an uncomfortably low room temperature). The inputs to services are called triggers, and the outputs are called actions. Together, triggers, actions and services allow a user to define the behaviour of the environment being controlled.

The primary purpose of the peripheral descriptor is to translate actions into calls to peripheral functions, and to translate peripheral events into triggers. Since each type of peripheral supports different functions and events (and by extension different actions and triggers), each type of peripheral needs a unique peripheral descriptor. The peripheral descriptor must be provided by the developer of the peripheral. In additions to the peripheral descriptor, developers are expected to provide custom remote operations for their peripherals. Offloading work to the peripheral like this can reduce network traffic and conserve energy.

As can be seen in Figure 3.1, the most complex software component is the base station. The base station software architecture was split into three layers: the user interface, the executive software and the peripheral interface.

The user interface is responsible for displaying the system status to a user and handling requests from the user. It is implemented as a website, and can be used by means of a web browser on a computer, tablet, or even a smartphone. This is the highest level layer because it interfaces directly with the user. Since it is a web interface, it does not run on the base station but rather runs in a web browser and is instead served by the base station web server. See Chapter 8 for further details.

The executive software is a collective label given to describe the lower-level code that is responsible for accepting inputs from the peripheral network and delegating tasks to appropriate peripherals. It also handles configuration changes from the user, as sent through the user interface. This layer includes the service engine and the web server components. See Chapter 7 and Chapter 9 for more information.

The lowest layer is the peripheral interface. This layer is responsible for interfacing the various peripherals in the network with the higher level layers. It provides a consistent way to control peripherals and receive events from them, even if the peripherals are very different from one another.

In summary, this architecture is centred around the idea of services representing the various behaviours of the environment being controlled. A service has a collection of triggers, and when all the triggers are satisfied, a collection of one or more actions will be performed.

3.2 System Software Design Decisions

At this high a level, all design decisions had to be made as a group, since all the various components would be affected by them in some way. The two most important of these decisions were system workload distribution and platform choice. The decisions we reached were to offload as much work as possible to the peripherals, and to use Java 7 as a platform [11].

3.2.1 Workload Distribution

The system workload distribution refers to the distribution of processing tasks between the peripheral cores and the base station computer. The main factors that were taken into consideration while designing the distribution were:

- Peripheral energy consumption
- Peripheral and base station complexity
- Flexibility for developers

We originally considered using a centralized approach in order to keep peripherals as simple as possible by only programming them with basic functions. These functions would include commonly used functions such as reading from an analog port or switching a digital I/O pin. The main reasons behind this design were to reduce the amount of computation performed by the battery powered peripherals, to keep the peripheral cores as uniform as possible, and to avoid requiring developers to write custom software for the peripheral cores.

In this centralized design, all peripherals are micromanaged by the base station. If a sensor is to be read every second, the base station would have to send the peripheral a message to instruct it to read the sensor every second. While this approach reduces the amount of computational energy consumed, sending and receiving wireless transmissions consumes almost four times the amount of energy, thus unnecessarily reducing battery life [5].

This problem was solved by increasing the complexity of the peripherals by making them less uniform and instead customized to the actions they are capable of performing. This also allows for actions to be created that allow the peripherals to behave more autonomously to potentially reduce energy consumption. Consider a temperature application where a thermometer must be read every minute. Using the centralized design, the base station must query the peripheral every minute for a temperature measurement, meaning that each measurement requires at least two transmissions from the peripheral, an acknowledgement and the results of the sensor reading. A custom design would allow a developer to program the peripheral to send a measurement every minute, without needing to first receive the command transmissions. This would reduce the amount of network communication by half.

3.2.2 Platform Selection

A sophisticated software architecture such as this imposes rather high requirements of its platform:

- Peripheral descriptors should be able to run on separate threads in order to reduce the chance of them negatively interfering with each other
- The base station software needs to be as platform-agnostic as possible in order to maximize flexibility
- The high level of complexity means that we should choose a platform that we are familiar with to limit the amount of time spent trying to learn new tools

We decided to build the base station software using Java, since it has a number of benefits over other alternatives.

- Java is a very popular language to teach new programmers; we are already quite comfortable with it, as are most young programmers
- Java will run on any platform that has a Java Virtual Machine (JVM), so we can run the base station software on any PC running Windows, Mac OS or even Linux.
- The web UI can be easily controlled by Java's server classes, and there are several libraries that allow Java to interface with hardware peripherals. This means that all the software on the base station can be programmed using the same language, which will streamline development.

The chief drawback of using Java is that it runs in a virtual machine, so it adds more overhead than a language such as C++. However, low level languages like these are subject to many other pitfalls such as poorer portability and our own lack of experience.[1] It also requires a sizeable amount of memory as compared to lower-level languages like C or C++.

The higher hardware requirements are not a concern, since the base station will be run on a PC in order to take advantage of networking hardware and a more powerful CPU to serve the web UI. Given the benefits, we feel that Java is a good platform to build our base station on.

Since the selected peripheral core is Arduino-compliant, (Section 2.3.1) we decided to use the Arduino platform to program the peripehral cores.

In summary, we elected to distribute processing workload where possible to limit wireless transmissions in the network, and we selected Java as a platform for the base station software. The peripheral cores will be programmed using the Arduino IDE.

[1]This is a fault of ours rather than C++, but it is an issue nonetheless.

Chapter 4

Peripheral Software

The peripheral software executes on the peripheral cores and on the transceiver as shown in Figure 2.2 (reproduced below).

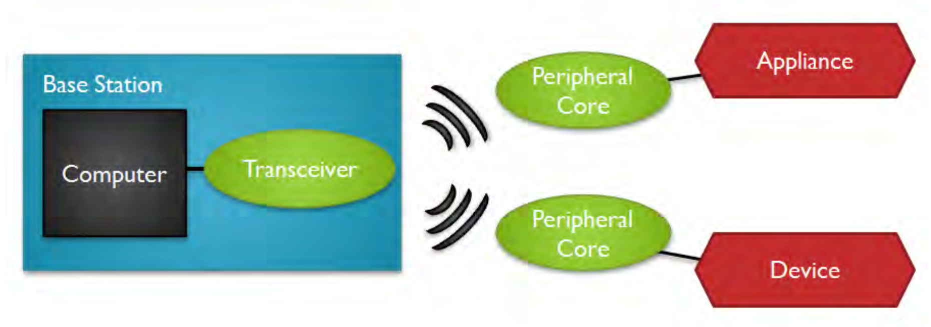


Figure 4.1: System Hardware Overview (Figure 2.2)

4.1 Wireless Network

A wireless network was designed to allow peripherals to wirelessly communicate with the base station. The ArduIMU board that was selected as the hardware for the peripheral core uses an ATmega128RFA1 microcontroller which has a built in IEEE 802.15.4 transceiver [5] that was used for performing our wireless communications. The open source “Zigduino-Radio” library was used for transmitting RF messages using the onboard transceiver [12]. As a result of peripheral core testing (see Section 2.3.2), it was found that the ArduIMU boards were able to send and receive messages with an average success rate of 45% at a distance of 45 [m]. This met the range requirement of 40 [m], but a networking protocol was needed to compensate for the losses through acknowledgements and retransmissions.

Initially, we attempted to implement a ZigBee protocol network layer. Unfortunately, we encountered a number of problems, the majority of which were due to the fact that the ZigBee stack provided for the ATmega128RF microcontroller was closed source and designed for a particular development board making porting it to the custom board very difficult. After consulting the development team for the ArduIMU board we learned that they too had been unsuccessful in the past with the ZigBee stack and that it would potentially require much more time than we had originally planned for. Therefore, we made the decision to instead implement our own custom networking protocol. This is made reasonable by the fact that we are using a star topology and that our network is independent. We decided to use a star topology for our system because by having peripherals interact only with the base station, routing is made very simple. It requires that peripherals only learn and store information for the base station, effectively ignoring other peripherals in the network. By independent network, we mean that the boards are only interacting with one another and are not required to communicate with other devices using the ZigBee protocol. In the following subsections the custom networking packet structure, packet encryption, the protocol for sending and receiving packets, and the process for adding new peripherals to the network are discussed.

4.1.1 Custom Networking Packet Structure

The 802.15.4 transceiver provides the physical and data layer for communications and an automatic Frame Sequence Check based on cyclic redundancy check for error detection is used so that only error free packets are placed in the transceiver's receive buffer [5]. Therefore, error detection does not need to be included in the networking packet structure. The designed packet structure is shown in Figure 4.2.

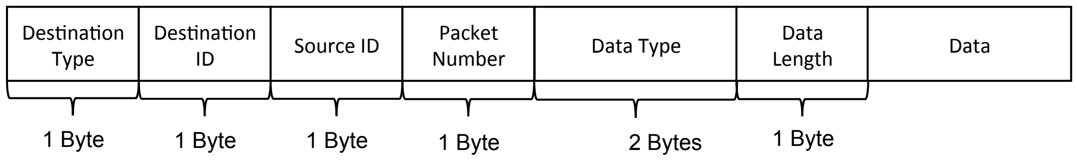


Figure 4.2: Wireless Packet Structure

This structure results in a seven byte header and a description of each of the fields is summarized in Table 4.I. It should be noted that the Destination Type field provides functionality similar to short addressing modes by allowing for both the peripherals and base stations to use the same ID numbers and still only receive packets addressed to them. This also allows for base stations to have unique IDs, such that multiple base stations can exist in the same area. The header does not include a Source Type field because base station only send packets to peripherals and peripherals only send packets to the base station, therefore any packets for a peripheral must be from a base station and similarly any packets for the base station must be from a peripheral. The peripheral IDs and base station IDs were both chosen to be one byte long because it is only required that a maximum of 64 peripherals are able to join a network, therefore more than one byte is not needed to uniquely identify each peripheral. Also, the situation should not arise that more than 255 base stations exist within 40 [m] of each other. The packet number is also only one byte long because, since only a single packet is sent for each transmission, this number will increment slowly. In other words, because packets are not sent in bursts (see Section 4.1.3), this packet number will not loop back around before acknowledgements are received. Each time a packet is sent from a source to destination the packet number is incremented by one and once it reaches its maximum value of 255 it will roll over to 0. These packet numbers are associated with unidirectional communication, meaning that two separate counters are used for communication with each device, one for the last received packet number from device A, and a second for the last sent packet number to device A. Correspondingly, each peripheral in the network needs to only store two counters because it can only communicate with the base station. The contents of the Packet Type field are explained in Table 4.II. The data length field is variable but is limited to be a maximum of eight bytes due to the current encryption process as is discussed in Section 4.1.2.

4.1.2 Wireless Packet Encryption

The ATmega128RFA1 microcontroller offers an AES security module with hardware accelerated encryption and decryption. The Electronic Code Book method is used and performed independently of the transceiver using the encryption and decryption process found on page 95 of the microcontroller's data sheet [5]. It uses a 128-bit encryption key and the decryption key is determined by first encrypting a dummy block of zeros and then reading 16 bytes from the `AES_KEY` register [5]. Because it uses 128-bit blocks, all packets that are to be encrypted must be 16 bytes long and therefore our code currently limits the maximum packet size to 16 bytes. If packets are less than 16 bytes long they are padded at the end with zeros. This padding does not affect the processing of the packets because the Data Length field in the header indicates where the data ends. The encryption and decryption functions perform any necessary padding and, as can be seen in the flow charts in Figure 4.3 of functions contained on both the peripheral and base station transceiver, encryption is the last step before sending a packet and decryption is the first step after receiving a

TABLE 4.I: WIRELESS PACKET STRUCTURE DETAILS

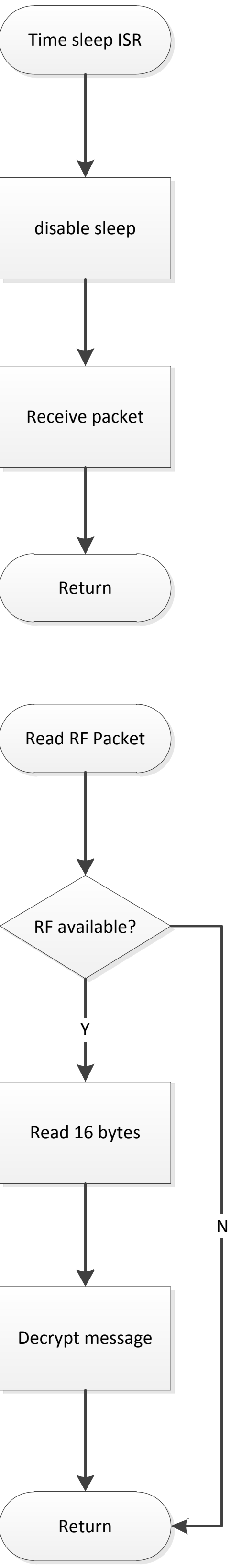
Field	Length	Explanation
Destination Type	1 byte	Should contain either the char 'N' if the packet is addressed to a base station or the char 'P' if the packet is addressed to a peripheral
Destination ID	1 byte	Contains the ID of the peripheral or base station the packet is intended for
Source ID	1 byte	Contains the ID of the peripheral or base station sending the packet
Packet Number	1 byte	Contains the packet number of the unidirectional communication from the source to the destination. If the packet is an acknowledgement packet as determined by the Packet Type field then the packet number is the packet number of the packet it is acknowledging.
Packet Type	2 bytes	Consists of two chars as determined by table 4.II and indicates what type of information is in the packet
Data Length	1 byte	Contains the number of bytes contained in the data field
Data	Given by Data Length field value	The information contained in this field depends on the packet type as shown in Table 4.II.

packet.

Limiting the packet size to 16 bytes is suitable for our application because the header is seven bytes long, leaving nine bytes for data. This should be plenty as most functions take very few parameters, if any. This could be changed in future versions of the code by first breaking long packets into 16-byte blocks before encryption and then during decryption, the Data Length field in the header found in the first block could be used to determine how many blocks should be read.

TABLE 4.II: WIRELESS PACKET TYPES

Packet Type (char values)	Explanation of Packet Type	Data Field Content
‘NP’	New peripheral broadcast sent from the base station transceiver	No data
‘TC’	Type code packet sent from a peripheral to the base station transceiver	The 4 byte type code of the peripheral
‘ID’	New ID packet sent from the base station transceiver to a new peripheral joining the network	The 4 byte type code of the new peripheral
‘AK’	Acknowledgement packet	No data
‘IN’	Instructions packet sent from the base station transceiver to a peripheral	The MessagePack packet to be forwarded from the base station computer
‘RT’	A results packet sent from a peripheral to the base station transceiver	The MessagePack packet sent from the peripheral containing the results



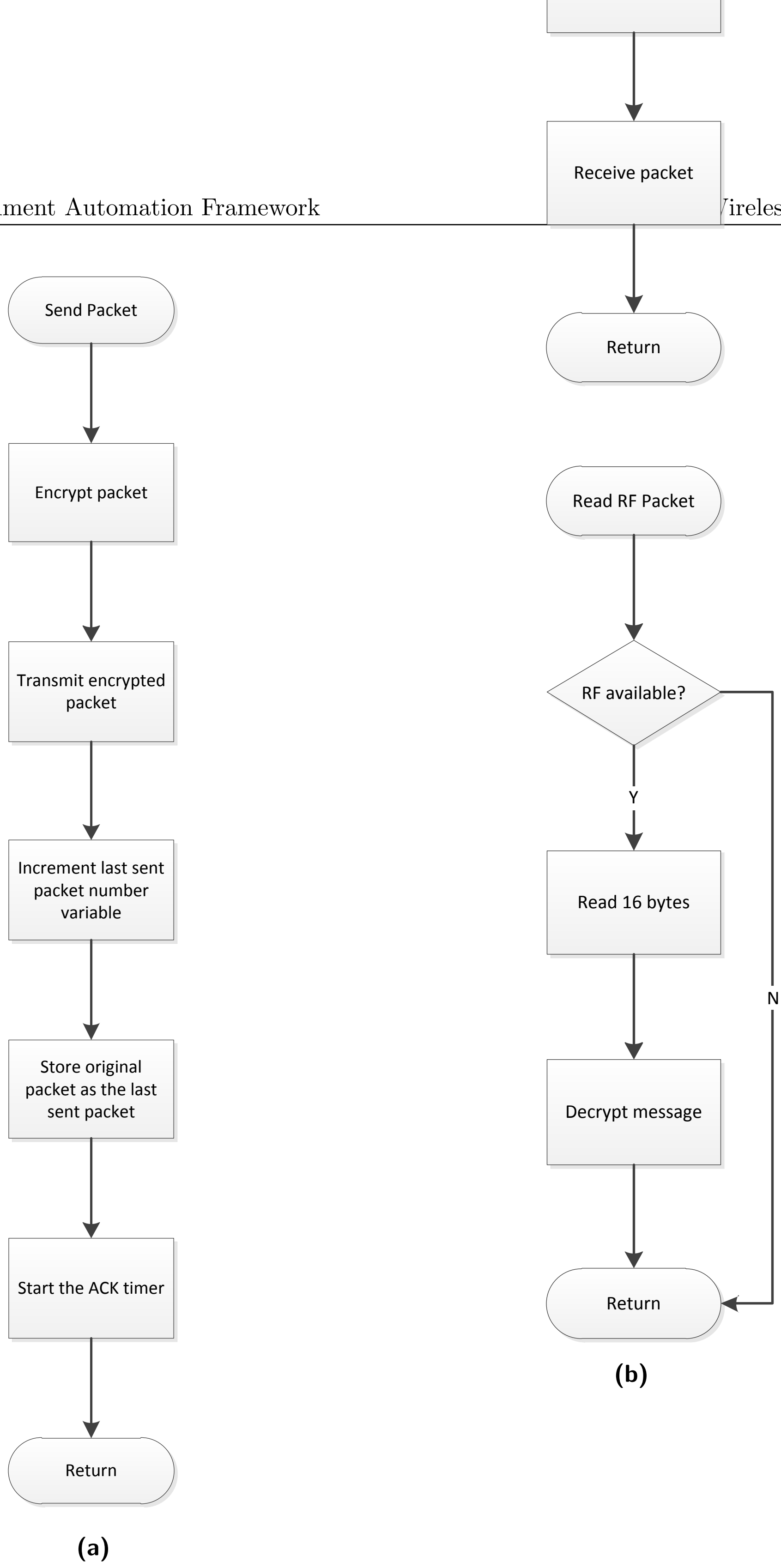


Figure 4.3: Functions Used by Both Base Station Transceiver and Peripherals: (a) Transmitting and (b) Receiving a Wireless Packet

4.1.3 Protocol for Sending and Receiving Wireless Packets

The processes for sending and receiving packets differ slightly between peripherals and the base station due to the limitations of using a single processor, which requires the use of cooperative multitasking. Explanations of the protocols and the differences between them are outlined in the following section.

Base Station

When the base station transceiver sends a message to a peripheral, a timer is started. If the timer expires before an acknowledgement has been received, the packet is retransmitted and the process repeats until the maximum number of retransmissions is reached. Once the maximum is reached, the base station transceiver informs the base station computer that the peripheral is not responding. At this point, instead of removing the peripheral from the network, the variable storing the last sent packet number for that peripheral is decremented by one and the acknowledgement timer is set to zero to indicate that the base station is no longer waiting on an acknowledgement. The base station then behaves as if the unacknowledged packet was never sent. This allows for a peripheral that becomes responsive at a later time to continue interacting with the network without needing to be reset and re-added to the network.

The only issue that may arise is if the base station times out prematurely when in fact the previous packet was received. In this case the next packet sent from the base station will be given the same packet number. When received by the peripheral, the packet will be acknowledged but the instructions will be ignored because the peripheral will think it received a duplicate packet. The user would then have to send the command again, however implementation decisions have been made to help prevent this from occurring. The timeout period has been chosen carefully, as discussed below, and an option is provided for developers who require extensive use of the CPU to allow for tasks to be interrupted periodically for the purpose of checking the transceiver and acknowledging any received packets (see Section 6.3).

Originally the base station transceiver was designed so that it could send packets to many peripherals but could not send the next packet to a particular peripheral until that peripheral had acknowledged the previous packet. However, because acknowledgements should be received relatively quickly, instructions will not be received on the serial port very often, and the system is not required to be real-time sensitive, to make the base station less complex, we decided it should wait for any required acknowledgements before checking the serial port again, which controls when another packet needs to be transmitted. This is less complex because originally if the base station transceiver is required to transmit instructions to a peripheral that is currently waiting for an acknowledgement, it would need buffer the new packet until the acknowledgement has been received. This method also simplifies the implementation of the retransmission protocol and reduces the memory usage because it does not require each peripheral to maintain a separate timer. Instead, a single timer is maintained for the only outstanding packet. As well, the previously sent message for each of the peripherals does not need to be stored separately. This does however cause the system to behave in a serial manner, but the period of time for which the base station is waiting for an acknowledgement, maximum one second as discussed below, is much shorter than the expected period between instruction packets sent from the base station computer.

The acknowledgement timeout value is set to be 200 [ms] and the system retransmits a maximum of four times before informing the base station that the peripheral is not responding. The propagation delay, t_p, can be calculated using Equation 4.1, where d is the distance and v_p is the propagation speed.

$$t_p = \frac{d}{v_p} \tag{4.1}$$

Therefore by using the maximum distance of 40 [m] each way for a total of 80 [m] and assuming the messages travel at the speed of light, 3×10^8 [ms], the total propagation delay for sending a message and receiving an acknowledgement is only 266 [ns]. Therefore, 200 [ms] is sufficient time to cover the propagation delay, and is quite a bit larger because both packets will also be subject to queuing and processing delays as well. Both messages and acknowledgements will be buffered and will need to wait for the processor to finish its current task before being attended to. Originally the timeout value was set to 100 [ms] but it was found during testing that sometimes an acknowledgement packet would be received after the base station timer had already expired, resulting in some unnecessary retransmissions. By increasing the timeout to 200 [ms], retransmissions were not prematurely sent using unnecessary energy. This still allows for the base station transceiver to inform the base station within approximately one second if a peripheral is not responding. We chose the maximum number of retransmissions to be four, for a total of five attempts, due to the results from our original range tests. Over four trials, each sending 100 packets from one ArduIMU to another at a distance of 45 [m], the average success rate was approximately 45%, indicating that at that distance it might require three transmissions before a packet is successfully received. The worst trial only had a 29% success rate indicating that it might require four transmissions before a packet was received correctly. Because the range was tested in a building with high wireless traffic, the success rate may have been lower than expected, however the system is being designed for homes and small buildings which may also have similar interference, especially since the transceiver runs on the same 2.4 GHz ISM band as Wi-Fi [5]. Therefore, if after five attempted transmissions the base station does not receive an acknowledgement, the probability that the peripheral is not responding or is out of range is much higher than the probability that the packet was lost. Also, with a maximum of four retransmissions, the base station is able to determine within a second if a peripheral is not responding, so other tasks will only be blocked for an acceptable amount time.

Peripheral

Similar to the base station, the peripheral has been designed to wait for an acknowledgement before sending the next packet. The flow chart for sending a packet to the base station can be found in Figure 4.3 a. It shows that once a packet is sent to the base station, the variable containing the previously sent packet number is increased, the packet is stored in case retransmissions are required, and a timer is started. When the acknowledgement is received, the timer is reset. If the timer expires before an acknowledgement is received, the packet is retransmitted.

At 500 [ms], the acknowledgement timeout period is slightly longer than for the base station transceiver because the base station is more complex and therefore it might take longer for it to send an acknowledgement. This is due to the fact that it needs to perform other tasks, such as checking the serial port, as well as the fact that it may receive packets from multiple peripherals. Another difference is that when the peripheral sends a packet, it waits for an acknowledgement before performing any other tasks and does not have a retransmission limit. We decided to implement this difference because the peripheral is only controlled via the base station and therefore, until the peripheral establishes communication with the base station, it should not be controlling elements of the environment. More details regarding peripheral operations can be found in Section 4.5.1.

4.1.4 Adding New Peripherals to the Network

The process of adding new peripherals to the network is shown in Figure 4.4 for the base station and Figure 4.5 for the peripheral. Adding a new device requires a slightly different message exchange protocol because the new peripheral does not yet have a unique ID. As such, the base station broadcasts messages by setting the destination ID to 0xFF. Similarly the source ID for messages sent by peripherals trying to join the network is always 0x00. Both processes begin with the base station broadcasting a search message. Any peripherals listening for this broadcast will then wait for external authentication once this broadcast is received. External authentication is obtained by the peripheral when the user presses the push button located on the ArduIMU. This method of authentication ensures that the user has possession of both the base station and the peripheral to prevent unauthorized users from adding peripherals to the network. Once external authentication has been obtained, the peripheral sends its type code to the base station. After receiving the peripheral's type code, the base station transceiver determines and assigns the peripheral the next available ID, which it then forwards along with the peripheral's type code to the base station computer over the serial port. The base station then transmits a packet containing the assigned ID as well as the type code of the peripheral it is adding, to ensure that the correct peripheral receives the packet. The ID packet is sent to the destination address of the assigned peripheral as packet number 1 so that the regular acknowledgement and retransmission protocol proceeds between the base station and the peripheral. The peripheral verifies that the type code is correct, sends an acknowledgement, and sets its own ID to match that of the packet's destination ID. Once this process has finished, the peripheral is considered to have joined the network and is now able to receive packets addressed specifically to it. While the base station is waiting to receive a reply to its search broadcast, it periodically checks the serial port to see if it has received a stop searching instruction from the base station computer.

4.2 Network Testing

To test that the communications network had properly been established, one ArduIMU was programmed with the base station transceiver code and another was programmed with the peripheral code. The peripheral was able to successfully join the network and send and receive packets. Packets were sent by each of the boards approximately every second and printed to the serial port by the base station transceiver, verifying that the correct packets

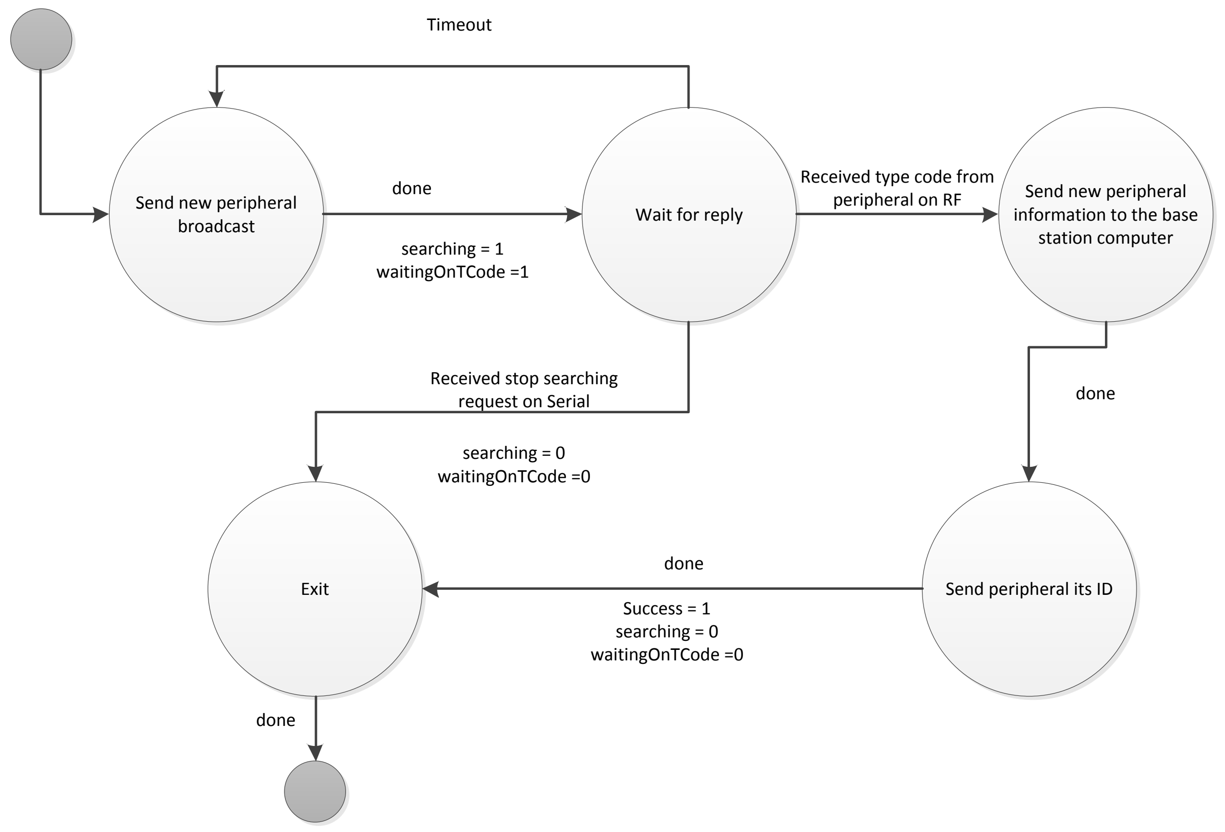

Timeout
Send new peripheral broadcast
done
searching = 1
waitingOnTCode =1
Wait for reply
Received type code from peripheral on RF
Send new peripheral information to the base station computer
Received stop searching request on Serial
searching = 0
waitingOnTCode =0
done
Exit
done
Success = 1
searching = 0
waitingOnTCode =0
Send peripheral its ID
done

Figure 4.4: State Diagram of Base Station Transceiver when Adding a New Peripheral

were being sent and that the correct acknowledgements were being received. This was left to run for about 20 minutes within which the packet number looped back to zero three times. Thus, no problems were encountered.

Retransmissions and timeouts were also tested. When the peripheral was turned off the base station transceiver would retransmit for a total of five transmissions and then print a message indicating that the peripheral was not responding. The next time a packet was to be sent, it had the same number as the one that was not acknowledged, which was the correct behaviour as discussed in Section 4.1.3. A third ArduIMU was then programmed with the peripheral code and both peripherals were added to the network and the base

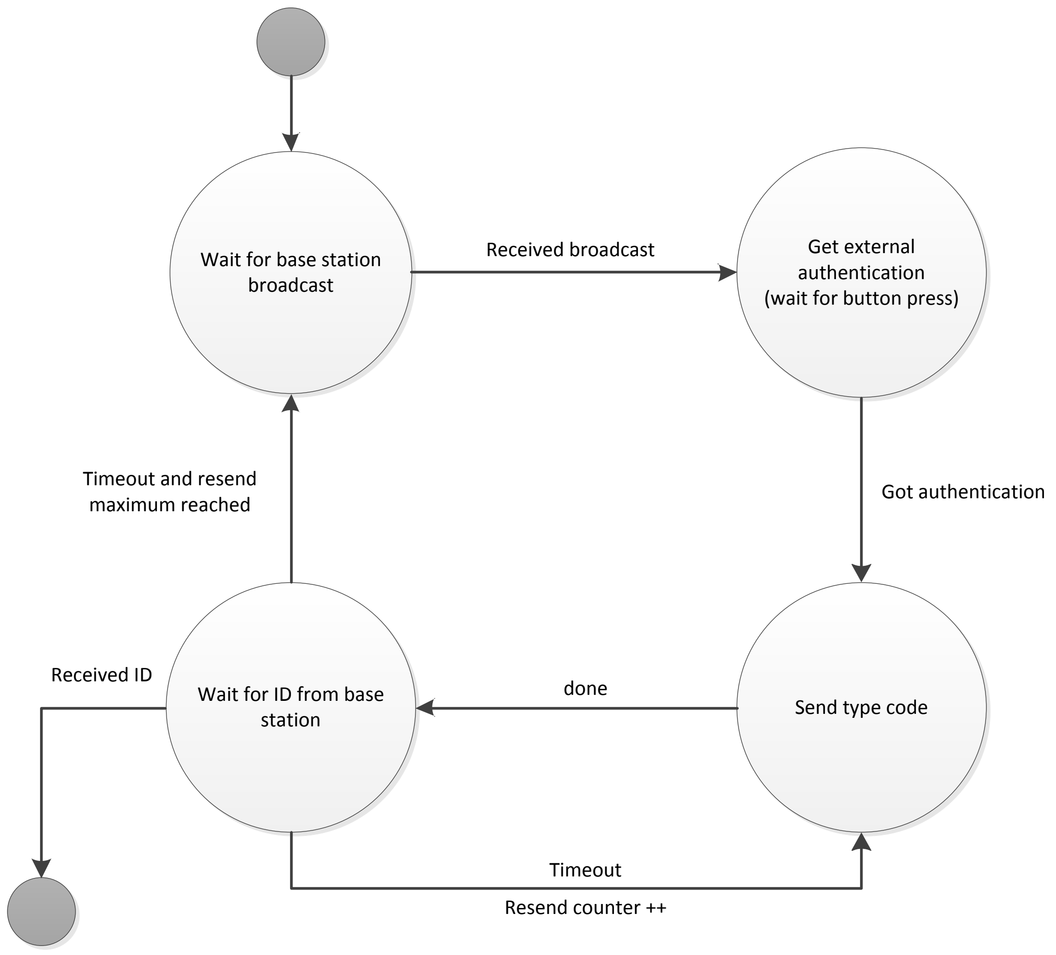

Wait for base station broadcast
Received broadcast
Get external authentication (wait for button press)
Got authentication
Send type code
done
Wait for ID from base station
Received ID
Timeout and resend maximum reached
Timeout
Resend counter ++

Figure 4.5: State Diagram of Peripheral when Joining a Network

station transceiver was able to send separate messages to each peripheral and only the appropriate peripheral would respond with an acknowledgement.

4.3 Base Station to Peripheral Network Interfacing

Communications between the base station transceiver and the base station computer occur over the serial port. The port speed is set to a baud rate of 19200 and the packet structure for these communications is discussed below. It was found that the serial connection is reliable and no packets are lost, therefore no acknowledgement or retransmission protocol is used. The packet structure for these communications is shown in Figure 4.6. The header is three bytes long and an explanation of the fields can be found in Table 4.III.

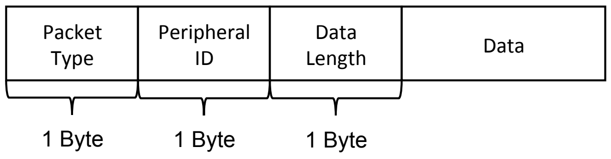


Figure 4.6: Packet Structure for Communications between Base Station Transceiver and Base station Computer

TABLE 4.III: BASE STATION COMPUTER/TRANSCEIVER COMMUNICATION PACKET DETAILS

Field	Length	Explanation
Packet Type	1 byte	Contains the value of the characters defined in Table 4.IV
Peripheral ID	1 byte	Contains the ID of the peripheral the packet is intended for when sent from the base station computer or the ID of the peripheral the packet is coming from when sent by the base station transceiver
Data Length	1 byte	Contains the number of bytes contained in the Data field
Data	Given by Data Length field value	The information contained in this field depends on the Packet Type field and can be found in Table 4.IV

TABLE 4.IV: BASE STATION COMPUTER/TRANSCEIVER COMMUNICATION PACKET TYPES

Packet Type (char value)	Explanation of Packet Type	Data Field Content
'N'	Sent from the base station computer to the base station transceiver instructing it to search for new peripherals	No data
'S'	Sent from the base station computer to the base station transceiver instructing it to stop searching for new peripherals	No data
'T'	Sent from the base station computer to the base station transceiver instructing it to transmit the information found in the data field	The MessagePack packet to be transmitted to a peripheral
'R'	Sent from the base station computer to the base station transceiver instructing it to remove a peripheral	No data
'P'	Sent from the base station transceiver to the base station computer containing the information of a newly added peripheral	The 4 byte type code
'E'	Sent from the base station transceiver to the base station computer containing data returned from a peripheral	The MessagePack packet sent from the peripheral containing the results
'X'	Sent from the base station transceiver to the base station computer when a device is not responding	No data

4.4 Base Station Transceiver Embedded System Code

The main functions of the base station transceiver are to add peripherals to the network, forward instructions from the base station computer to the peripheral, and forward the data sent from peripherals to the base station computer. Communication between the peripherals and the base station transceiver is performed wirelessly and communication between the base station transceiver and base station computer is performed over the serial port.

The main loop running on the base station is described in the flow chart shown in Figure 4.7. From that flow chart, it can be seen that the base station transceiver must alternate between checking for messages from the base station computer and checking for messages from the peripherals. While waiting for an acknowledgement packet, we decided that the base station should check the transceiver and follow the retransmission process before checking the serial port again. This reduces the need for additional buffering because if the base station is already waiting on an acknowledgement it would not be able to send the next packet (see Section 4.1.3). Instead messages are left in the serial port buffer until they can be dealt with. As discussed in Section 4.1.3, the base station transceiver's timer will expire after after one second, and because commands from the base station will not be sent very often, serial port buffer overflow should not be a problem.

The flow chart in Figure 4.8 describes the check serial function in which a packet is read and the corresponding action is performed. The message protocol between the base station transceiver and base station computer is described in Section 4.3. The flow chart in Figure 4.9 describes the steps taken after receiving a wireless packet. As shown in table 4.II, there are three types of packets that the base station transceiver can receive. The first of which is an acknowledgement packet that, when received, if it is the correct acknowledgement, the timer is stopped and the variable signalling that the base station is waiting on an acknowledgement is set to zero. The second type of packet is a results packet, the data from which is forwarded to the base station computer. The third type of packet is only received when adding new peripherals to the network and is a packet containing a new peripheral's type code. A single variable is used to indicate whether or not the base station is waiting on a type code and is set to one when the receive packet function is called from within the add new peripheral function. If this variable is set to one and a type code is received, this function returns to the join network function immediately. In the case where this function is called from within the main loop, i.e. the base station transceiver is only waiting for acknowledgements or results, packets are read and dealt with until there are no longer any packets in the transceiver's receive buffer.

4.5 Peripheral Embedded System Code

4.5.1 Peripheral Core

The following section contains a description of the peripheral core code and a description of the customized code running on the peripheral can be found in Section 6.3. The peripheral core code is the same across all peripherals, regardless of the peripheral's type. The peripheral core's primary function is to receive commands from the base station and perform the corresponding task. The types of tasks that can be performed are controlling tasks, sensing tasks, and monitoring tasks. Controlling tasks, such as turning on a light, output data to adjust the environment and do not usually involve sending results back to the base station. Sensing tasks gather information from the environment and send it to the base station, and are only performed upon request. Monitoring tasks are similar to sensing tasks, but differ in that they are more autonomous; they are performed without instruction and can be configured to send data at regular intervals or when some condition changes. The peripheral core has been designed so that it can accommodate all three types of tasks and only the functions defined in the peripheral descriptor (see Section 3.1) need to be written by developers using the template files discussed in Section 6.3. As per our requirements, the peripheral core is able to perform digital I/O, read analog input, write analog output in the form of PWM, and communicate with attached hardware via SPI.

The flowchart for the main loop running on the peripheral is shown in Figure 4.10. Depending on decisions made by the peripheral developer (see Section 6.3), the green and blue sections joined by a dashed lines may or may not be included. The reasoning behind this design was to allow for a main loop that could be used by all types of peripherals performing the different types of tasks, without the main file needing to be changed. The majority of the time, the microcontroller on the peripheral will be sleeping, however the transceiver clock will remain on so that the peripheral can still receive messages. An interrupt is used to waken the microcontroller from sleep, and the type of interrupt depends on which sleep function is called. Two sleep functions have been written and are available to peripheral developers. If `TIME_ SLEEP` is defined then an interrupt will wake the microcontroller after half a second and automatically check the transceiver as shown in Figure 4.11. If this variable is not defined then the microcontroller will only wake when the transceiver receives a packet.

If an instruction packet has been stored by the time sleep ISR the task is performed else the receive packet function is called to get the next instruction packet, and this repeats until no more packets are received. As mentioned in Section 4.1.3 the peripheral will wait for an acknowledgement if needed until it is received. The variable `task` is used by developers when they have peripherals that require extensive use of the cpu and is further explained in Section 6.3. If this variable is set to one and `TASK` is defined then the peripheral's current task is resumed. Then if `TIME_SLEEP` is defined the monitoring function is called (see Section 6.3) and if results need to be sent back to the base station the peripheral will wait for the acknowledgement before returning to the start of the loop, however if `task` is set to one the peripheral will not go back to sleep because it has a task to perform.

The receive packet function flow chart is shown in Figure 4.12. It shows that while waiting on an acknowledgement the peripheral only takes packets from the transceiver. If an incorrect acknowledgement is received, the previous packet is resent and if the received packet is an instruction packet instead it is stored in a buffer to be dealt with once the acknowledgement has been received (see Section 4.1.3). If the peripheral is not waiting for an acknowledgement, it takes the first instruction packet out of the buffer. If the buffer is empty, the peripheral instead reads the next packet from the transceiver. If the packet is an instruction packet, an acknowledgement is sent and the packet is stored in a variable available to the main loop, otherwise the packet is ignored and the next packet is read. The receive packet is exited once an instruction packet has been received or there are no more packets to attend to.

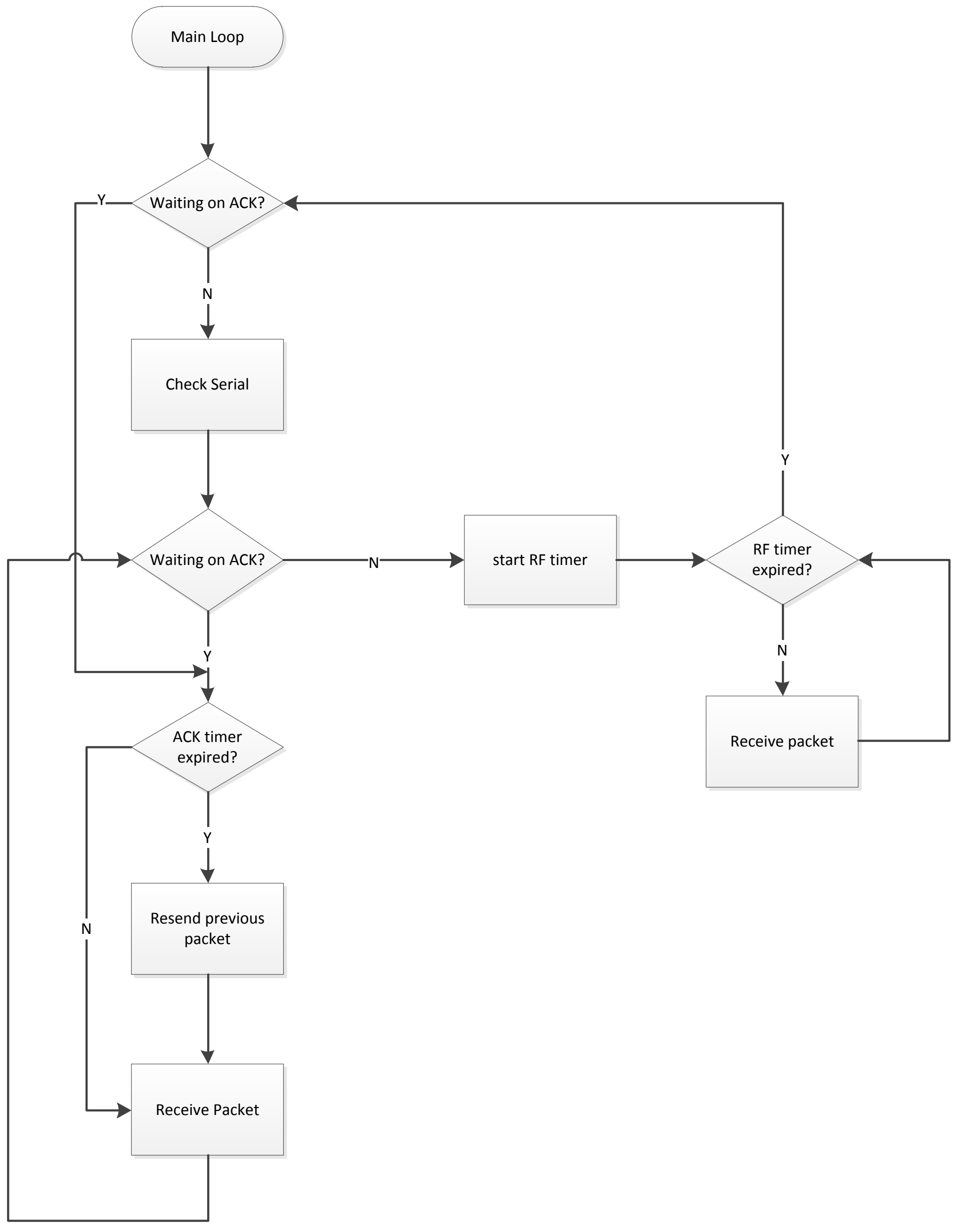


Figure 4.7: Flowchart of the Base Station Tranceiver's Main Loop

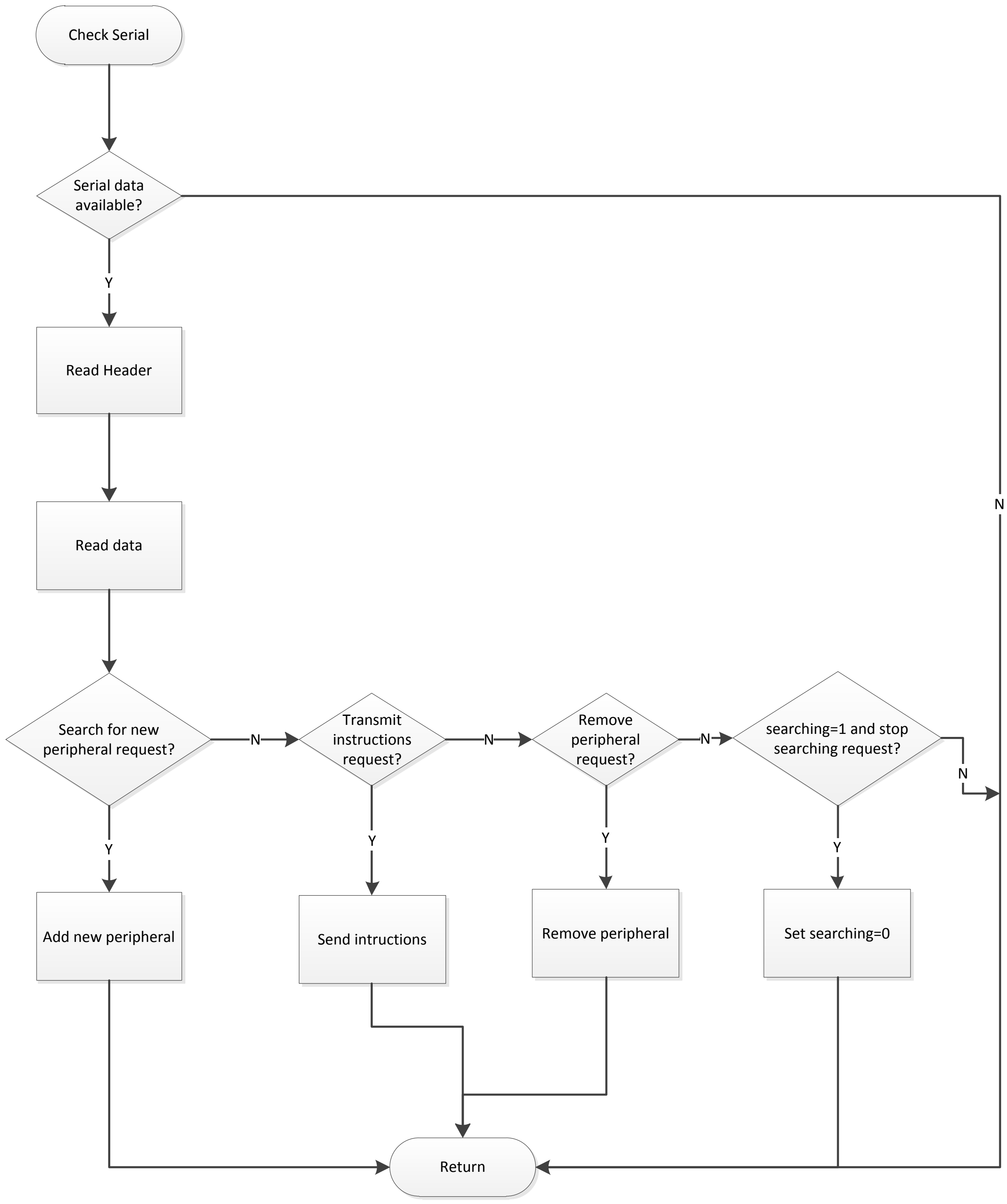


Figure 4.8: Flowchart of Base Station Transceiver's Check Serial Function

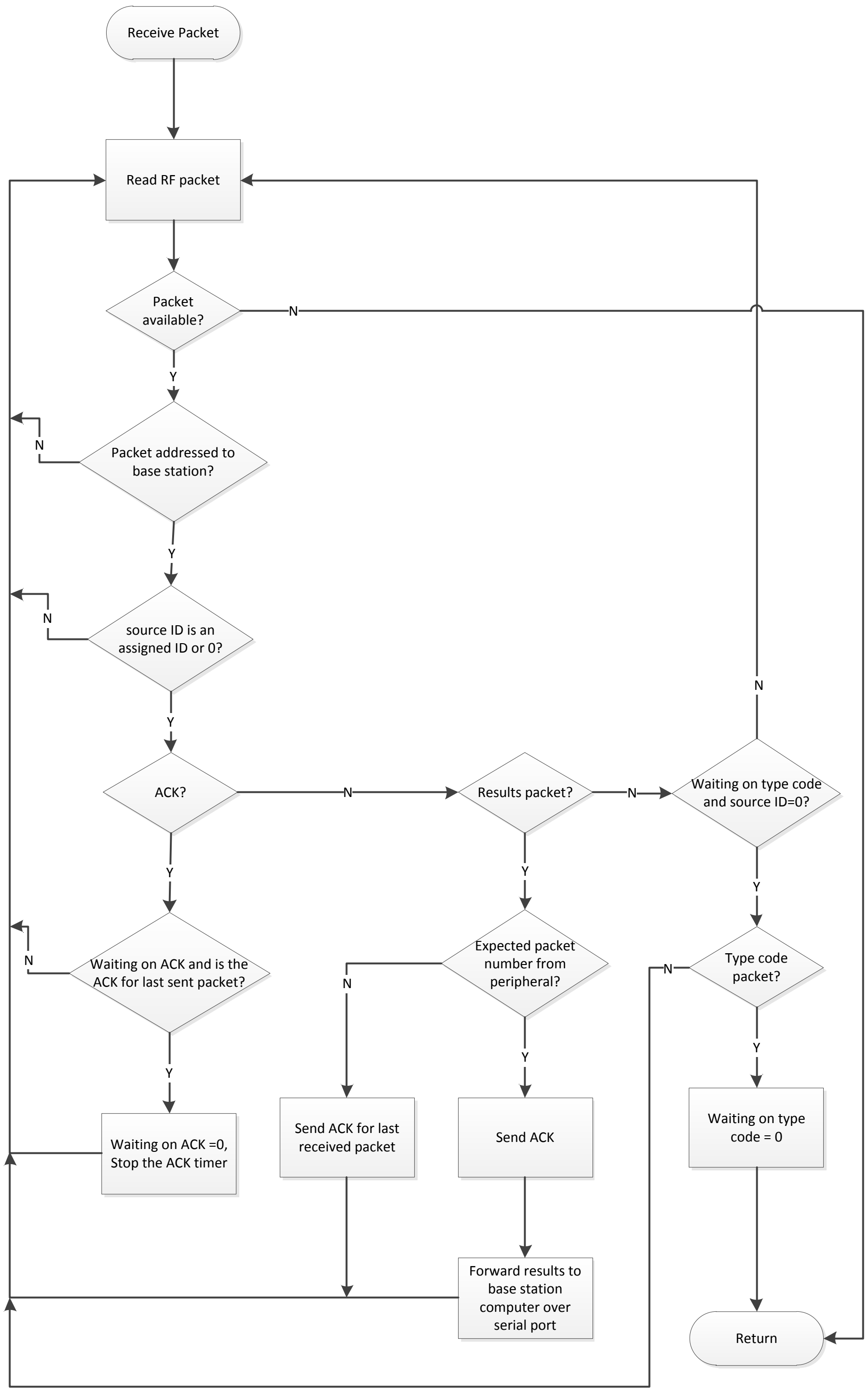

Receive Packet
Read RF packet
Packet available?
N
Y
Packet addressed to base station?
N
Y
source ID is an assigned ID or 0?
N
Y
ACK?
N
Results packet?
N
Waiting on type code and source ID=0?
N
Y
Y
Waiting on ACK and is the ACK for last sent packet?
N
Y
Waiting on ACK =0, Stop the ACK timer
Expected packet number from peripheral?
N
Y
Send ACK for last received packet
Send ACK
Forward results to base station computer over serial port
Type code packet?
N
Y
Waiting on type code = 0
Return

Figure 4.9: Flowchart of Base Station Transceiver Receive Packet Function

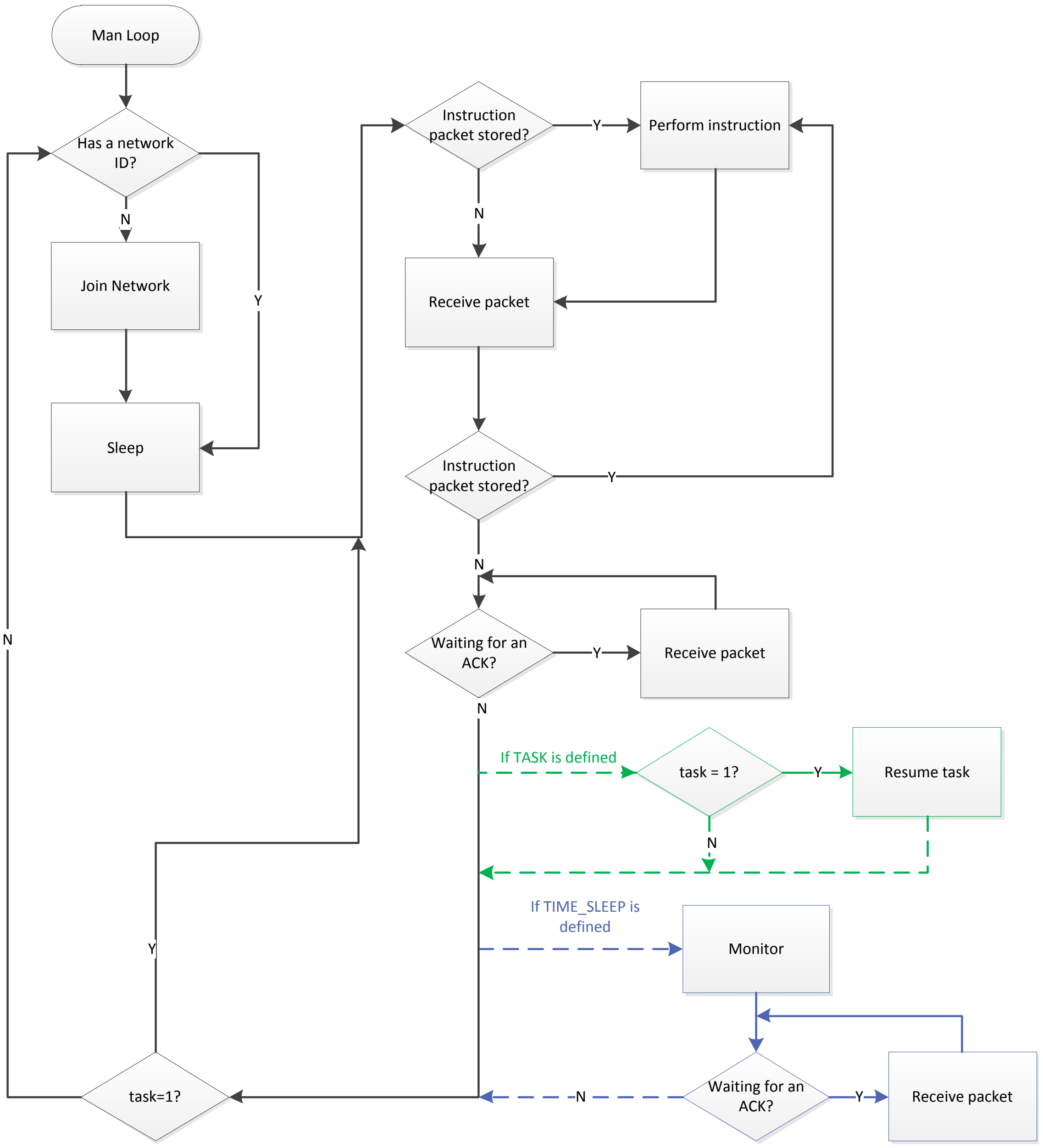
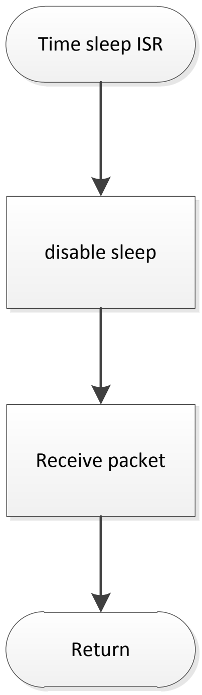


Figure 4.10: Flowchart of the Peripheral's Main Loop



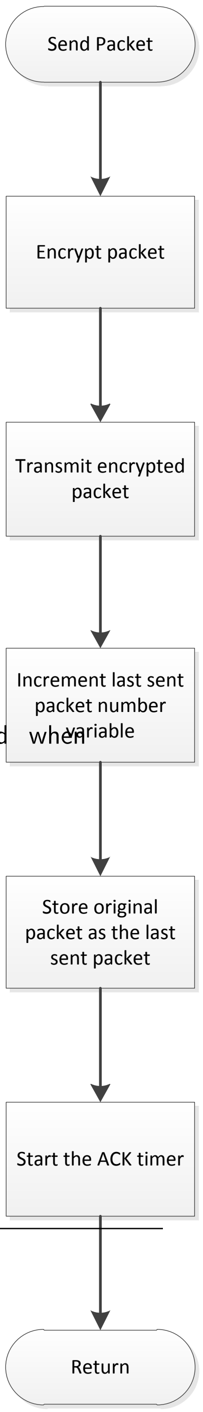
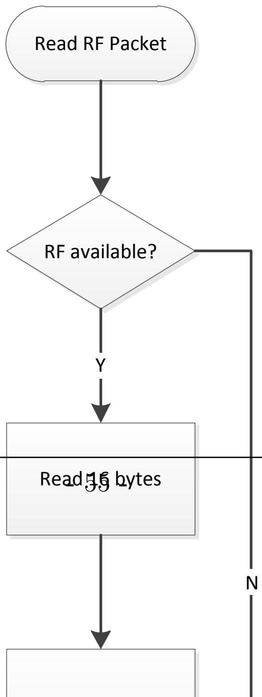

Send Packet
Encrypt packet
Transmit encrypted packet
Increment last sent packet number variable
Store original packet as the last sent packet
Start the ACK timer
Return

Figure 4.11: Flowchart of Peripheral's Interrupt Service Routine Enabled when TIME_SLEEP is Defined



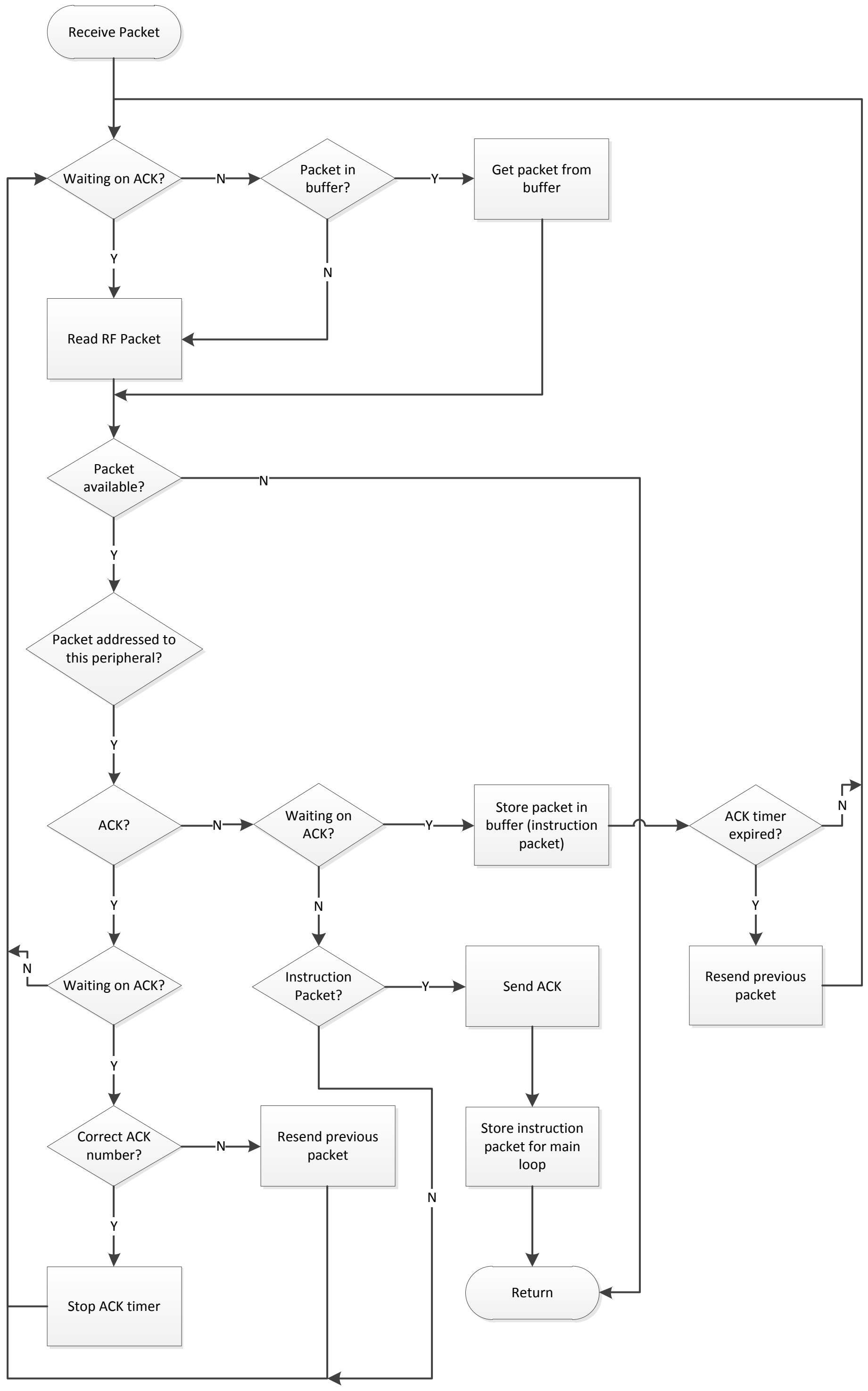


Figure 4.12: Flowchart of Peripheral Receive Packet Function

Chapter 5

Peripheral Interface

The peripheral interface was designed to accomplish two main goals: 1) generalize communication with peripherals of different types and 2) provide a means for developers to build custom peripherals.

5.1 Interfacing with Executive Software

Recall from Chapter 3 that triggers and actions are used to interface between services and peripheral descriptors. This concept will now be further discussed.

5.1.1 Design of Triggers and Actions

Consider the two peripherals shown in Figure 5.1. The switch peripheral has a trigger to indicate when it is switched on and a trigger to indicate when it is switched off. The lamp peripheral has actions to turn it on or turn it off. Unfortunately, static triggers and actions like these are not very flexible. Consider the situation where a user has a temperature sensor, and they want to create a service where a particular action executes when the temperature sensor detects a particular temperature. With static triggers, one of two situations would arise:

1. The available triggers may not have sufficient granularity; the user would be forced to choose between a temperature that is either too high or too low

2. There would be a large number of available triggers enumerating all possible temperatures. This is time consuming for a developer to implement and confusing for the user

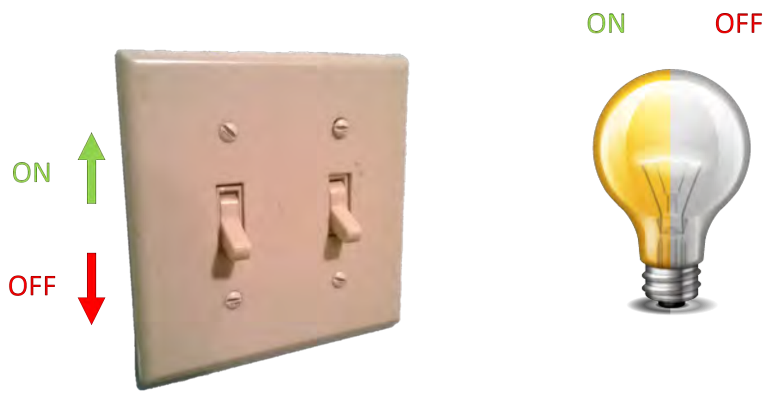


Figure 5.1: Light switch and light bulb peripherals [13]

Triggers and actions can be made more flexible by associating them with *parameters*. Instead of two static triggers, the light switch would support a single parameterized trigger: ‘Switch State Changed,’ whose parameter would be a true/false value specifying the new state. Likewise, lamp would support a single parameterized action called ‘Set State’ whose argument specifies whether to turn the light on or off. If the user wants to use a temperature sensor, a trigger with an integer parameter can be used. This would allow the user to easily set their exact desired temperature.

5.1.2 Implementation of Triggers and Actions

Each peripheral includes a function called `doAction()` that is called by the service engine when the peripheral is supposed to execute an action. This function accepts a `String` containing the desired action's identifier and an `array` of `String`s to denote the action parameters. Note that while the parameters themselves do have types (such as Boolean, integer, etc.) as described above, they are passed to and come from the service engine as `String`s. We decided that since the service engine processed triggers and actions as `String`s anyway, it would be best to rely on developers to encode their arguments as `String`s in the case that they chose to use parameter types that we could not anticipate.

Each peripheral has a reference to the service engine in order to send triggers. The service engine implements a function called `triggerOccurred()` that may be called by a peripheral when it wants to send a trigger. This function accepts `Boolean` indicating the new trigger state, a `Peripheral` object indicating which peripheral caused the trigger, a `String` containing the trigger's identifier, and an array of `String`s to denote the trigger parameters. Once again, these actions do have types, but they are represented as `String`s for flexibility and convenience.

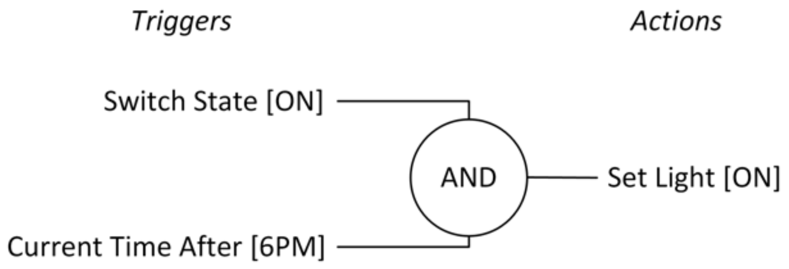


Figure 5.2: Simple service involving two triggers and a single action

The `Boolean` in `triggerOccurred()` accomodates services involving more than one trigger. Consider the service in Figure 5.2. This service turns on a set of lights if both the light switch is set to the ON position, and the current time is later than 6pm. Clearly, the service engine needs to remember that the Current Time After [6PM] trigger occurred, even after it happened. In the interest of remembering that a trigger has occurred, triggers are stored in the service manager as `Boolean`s, and when a trigger occurs, its `Boolean` value is set to true.[1]

5.2 Interfacing with Peripheral Hardware

Interfacing with peripheral hardware provides a different set of challenges than interfacing with the executive software. Wireless transmissions use lots of power, so interfacing with the peripheral hardware should send as little data as possible.

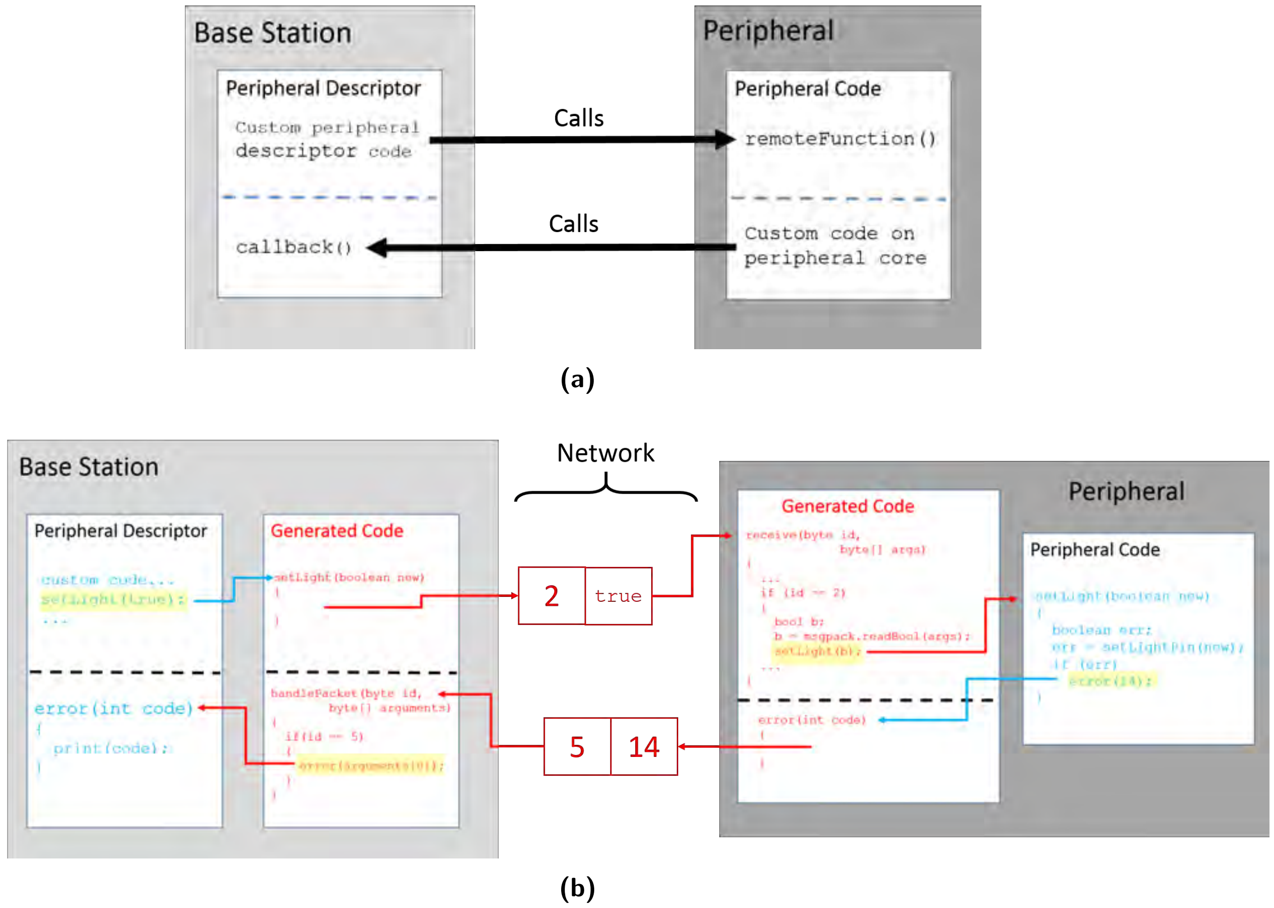
We decided to use a Remote Procedure Call (RPC) model to implement this communication. This model is illustrated in Figure 5.3a. If a value (such as a `float`) is to be sent to a peripheral from the base station, a remote function may be defined on the peripheral that accepts a `float` as a parameter. Conversely, if a value (such as an `int16_t`) is to be sent from the peripheral to the base station, a callback may be defined on the base station that accepts an `int16_t` as a parameter. We chose to use this system because it spares developers from having to create their own packet structure, meaning that transmissions can be as efficient as possible.

Each remote function and callback has an 8-bit function ID associated to it, allowing up to 255 remote functions (Function 0 is reserved to allow the network to ping the peripheral). When a function or callback is called, a packet is generated that contains the function ID and any applicable arguments. This is illustrated in Figure 5.3b.

In order to maximize transmission efficiency, the MessagePack serialization algorithm was used to serialize arguments before transmitting them [14]. Messagepack provides libraries for both C++[2] and Java. This means that primitives such as Booleans, integers, Strings, floats, etc. can be sent between the base station and a peripheral, and decoded on

[1] Of course, it is expected that the peripheral that sends the Current Time After trigger will set the trigger Current Time After[6PM] to false once it becomes appropriate. The conditions under which the peripheral does this are the responsibility of the developer.

[2] The official C++ library was not used due to difficulties running it on the Arduino platform. The msgpack-alt library was used instead to provide peripheral-side MessagePack support [15].



```
Custom peripheral
descriptor code

callback()
```

```
remoteFunction()

Custom code on
peripheral core
```

```
custom code...
setLight(true);
...

error(int code)
{
  print(code);
}
```

```
setLight(boolean new)
{
}

handlePacket(byte id,
        byte[] arguments)
{
  if(id == 5)
  {
    error(arguments[0]);
  }
}
```

```
receive(byte id,
          byte[] args)
{
  ...
  if (id == 2)
  {
    bool b;
    b = msgpack.readBool(args);
    setLight(b);
  ...
}

error(int code)
{

}
```

```
setLight(boolean new)
{
  boolean err;
  err = setLightPin(new);
  if (err)
    error(14);
}
```

Figure 5.3: Remote functions and callbacks: (a) basic, (b), detailed

either end as the corresponding type on the destination platform.

Since the base station hardware does not have an integrated transceiver to communicate with the various peripherals, a peripheral core unit is attached to the base station by means of a USB serial port, and functions as a transceiver. Communication with the transceiver is accomplished using the Java Simple Serial Connector (jSSC) serial port library [16]. This library was chosen because it can be used to communicate with USB serial ports on Windows , Mac OS X and Linux, so developers can use their OS of choice when developing and testing custom peripherals.

Chapter 6

Developer Workflow

In most cases, developers will have a custom hardware peripheral. They will need to provide code to run on both the base station and the peripheral core of their hardware peripheral. Before writing code for either platform, developers create an XML (eXtensible Markup Language) file that defines the triggers and actions of their custom peripheral, as well as any status variables [17]. In this file, they must also define any remote functions that their hardware peripheral offers to the base station, such as turning on lights or reading sensors. The XML file may also define callback functions that the peripheral core may call on the base station. Recall that these remote functions/callbacks are different from triggers and actions. Triggers and actions only exist on the base station, while the remote functions define communication between the base station and a remote peripheral. This is illustrated in Figure 6.1.

The XML file is parsed by our software and used to generate source code to make the remote link transparent to the developer, as shown in Figure 5.3. This allows developers to call a remote function like `turnOnLight(5)` from within their java-based peripheral descriptor in order to invoke the C function `turnOnLight(int brightness)` on the peripheral hardware, without needing to understand the lower-level communication details.

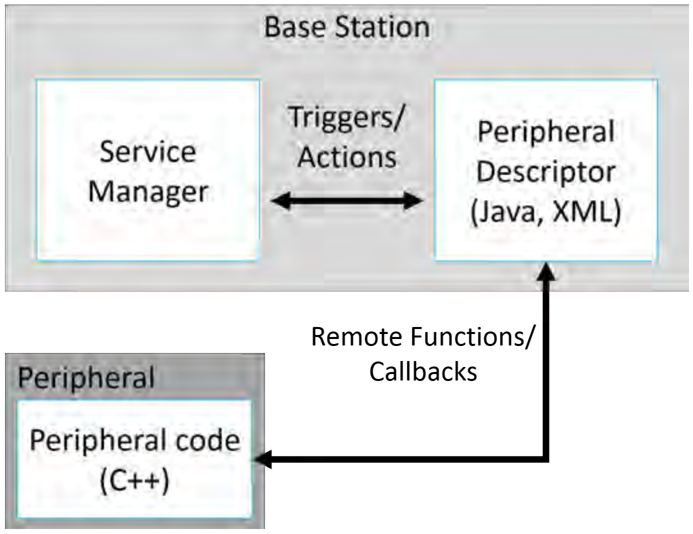


Figure 6.1: Service manager interacting with a peripheral

The developer will be responsible for writing a Java class to implement callbacks to handle both the actions and the remote callbacks defined in the XML. The Java class is also expected to generate triggers at appropriate times and pass them up to the service engine. Developers will also be responsible for providing C++ code to implement the remote functions on the peripheral core as defined in their XML. The developer manual in Appendix D provides a detailed tutorial of how to do this.

In some cases, developers might not have a physical peripheral, and the peripheral will only manifest itself on the base station. A good example of this is the timer peripheral that allows users to schedule triggers to occur at certain times. These types of peripherals are called **virtual peripherals**. Virtual peripherals only require the XML file and a Java-based peripheral descriptor. These peripherals do not have any remote functions or remote callbacks because they do not correspond to any remote peripheral (See Figure 6.2).

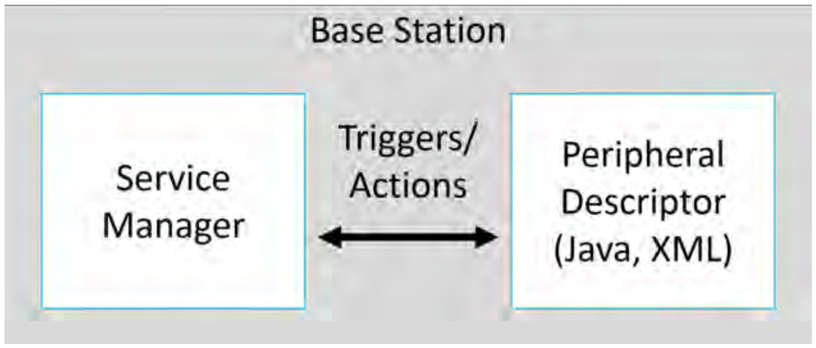


Figure 6.2: Virtual peripheral with no physical counterpart

6.1 Developer Workflow Design

The chief goals of any workflow design is to minimize the amount of repeated work, and limit the complexity of what must be done. Figure 6.3 illustrates the architecture of the peripheral in detail. The original design of the developer workflow assumed that developers would provide a full implementation of the `PeripheralDescriptor` interface, which could be rather simply done in a single Java file. However, as test peripherals were developed, it became apparent that lots of similar code had to be written for each peripheral. In order to reduce the amount of repeated work, we decided that this code could be automatically generated for the developer. This could be done by having the developer declare the functionality of their peripheral in a generic, simplified syntax such as XML or Java annotations, and then process that functionality and generate the necessary hooks and callbacks. Generating code like this would free the developer to focus on implementing the custom functionality of their peripheral, such as reading from a sensor or controlling a particular actuator, without forcing them to worry about the lower-level communications and processing.

6.2 Workflow Implementation (Base Station)

The original design goal for code generation was to allow a developer to specify their entire peripheral descriptor inside a single file. Details such as triggers, actions and status variables would be declared using Java Annotations. Annotations are a feature of the Java language that allows developers to specify meta-data about their programs and can even be used to generate code, saving them time [18]. From a developer's point of view, annotations are very convenient. Unfortunately, implementing an annotation processor turned out to be rather complicated, and certain features[1] that would have been necessary for the intended workflow will not be available until the Java JDK 8 is released in March 2014 [19]. A developer preview is available, but it is not advisable to rely so heavily on a feature of unreleased software since there is no guarantee that feature will make it into the release version, especially since this feature has already been delayed or dropped in the past.

Since annotations were not an option, the single file design goal was not achievable. The next best generic, simplified syntax is XML. XML files are hierarchically structured documents that can be read and processed by many different systems. They are a widely recognized standard for presenting structured information such as a list of movies in a person's home collection, or the capabilities of a peripheral in an environmental automation system. XML files can be transformed using XSL (eXtensible Stylesheet Language) into other documents to make them easier to understand. Using an XSL transform (XSLT) program, the developer's XML file can be parsed and used to generate a Java class that handles the necessary communication with higher layers and with remote peripherals. The Java class will also include the necessary hooks and callbacks for the developer to implement their desired functionality.

The Saxon-B XSLT processor was chosen because it implements XSLT 2.0 [20, 21]. XSLT 2.0 provides support for many advanced transformation features such as generating

[1]Namely the "repeats" keyword

multiple documents. These features were necessary because in addition to the Java files, code would need to be generated for the peripheral core to provide hooks for any remote functions.

6.2.1 Plugin Framework

One key requirement of this system was that it have the capability to be extended by developers [2]. We decided that the best way to do this would be to use plugins. Plugins are a way of extending a software product without having to recompile it or in many cases, even restart it. The Java Simple Plugin Framework (JSPF) was chosen to provide support for plugin functionality because it is very easy to use, and effective. The alternative frameworks such as OSGI and JPF are more complicated, although they are better supported [22, 23].

Developers export their descriptors to a JAR file (See developer manual for more details), which can then be distributed to users. Users can upload the JAR file containing the peripheral descriptor to their base station for JSPF to discover.

6.3 Workflow Implementation (Peripheral Core)

The code generation discussed in Section 6.2 is also capable of generating code to run on the peripheral cores. Since the peripheral cores are not multi-threaded, we had to design the developer interface such that the developer's code could be interrupted in order to deal with network events. If this were not done, the peripheral might not be able to respond to packets and the base station would register it as 'not responding.' This means that we could not allow developers to use the Arduino `setup()` and `loop()` functions. Instead, we provide them with two files in which to write their custom code: "custom.h" and "custom.ino", which can be found in Appendix E. "custom.h" allows developers to enable or disable certain functionality of the peripheral core by defining compiler constants, while "custom.ino" is where the actual functions are to be written.

The compiler constants that can be defined are:

TIME_SLEEP This should be defined if the peripheral is going to be used for a monitoring application in which it will be required to perform a task periodically, such as checking the state of a switch, without being told to do so each time by the base station. The `monitor()` function will periodically be called (see below).

AUTO_JOIN This option disables the requirement for the user to press the push-button to connect to the network. We do not generally recommended using this option for the security reasons that were discussed in Section 4.1.4, but it may be necessary in peripherals where users do not have access to the push button (such as the indicator box in Section 2.4.1).

TASK This option allows for long-running tasks to be interrupted to check the transceiver for packets following the interrupt service routine shown in Figure 4.11. This option should be used along with `resumeTask()`, which is described below.

In "custom.ino" there are three possible functions depending on which statements were defined in "custom.h":

periphInit() This function gets called once during peripheral startup. In this function the developer must set the variable `typeCode` to match the type code in the peripheral's peripheral descriptor. This function can also be used to perform any other initialization tasks specific to the attached custom hardware.

resumeTask() This function allows long-running operations to be broken up into smaller pieces, to give the peripheral a chance to receive wireless messages and handle received instructions that could potentially instruct it to abort a long-running operation. `resumeTask()` is called from within the main loop. These tasks are encouraged to save their state and `return` frequently because the more frequently these tasks return, the more quickly the peripheral will be able to handle received instructions.

monitor() This function is periodically called, allowing a peripheral to check its inputs and call a callback on the base station if appropriate.[2]

Once these functions have been written, the developer must add any remote peripheral functions that they defined in their XML file, as described at the beginning of the chapter.

6.3.1 Testing

The development tools described above, including the plugin framework, automatic code generation and interfacing systems were tested by actually developing custom peripherals, and giving the system to other team members to work with. This testing was crucial to identify consistency issues within the code generation, because it gave a good idea of the sorts of unexpected things developers might try to do. Particular combinations of arguments or trigger/action declarations resulted in generated code with compile errors, and these issues were promptly fixed.

To test the performance of the generated code and the hardware interface, a network simulator was developed using the Arduino tools and implemented on an Arduino Uno equipped with a character LCD as shown in Figure 6.4a. This simulator receives data packets directed to the network transceiver and extracts the payload destined for a peripheral. It can also send data packets back to the PC running the peripheral descriptors.

The simulator can also run on the peripheral cores and can be used to test that the remote functions developed for a peripheral work correctly, before adding that peripheral to a real network. This tool is very helpful at detecting programming errors in developer code without needing the overhead of an entire network, as illustrated by the simple development setup in Figure 6.4b. All that is required is a PC running the development tools and a peripheral core. The custom hardware for the peripheral (in this case an AdaFruit LED

[2]The only limitation with `monitor()` is that the developer needs to ensure that only one data packet is sent back to the base station due to the protocol in Section 4.1.3. This can be accomplished by ensuring that only a single callback is called during the execution of this function.

strip) can be tested without compromising a running network.

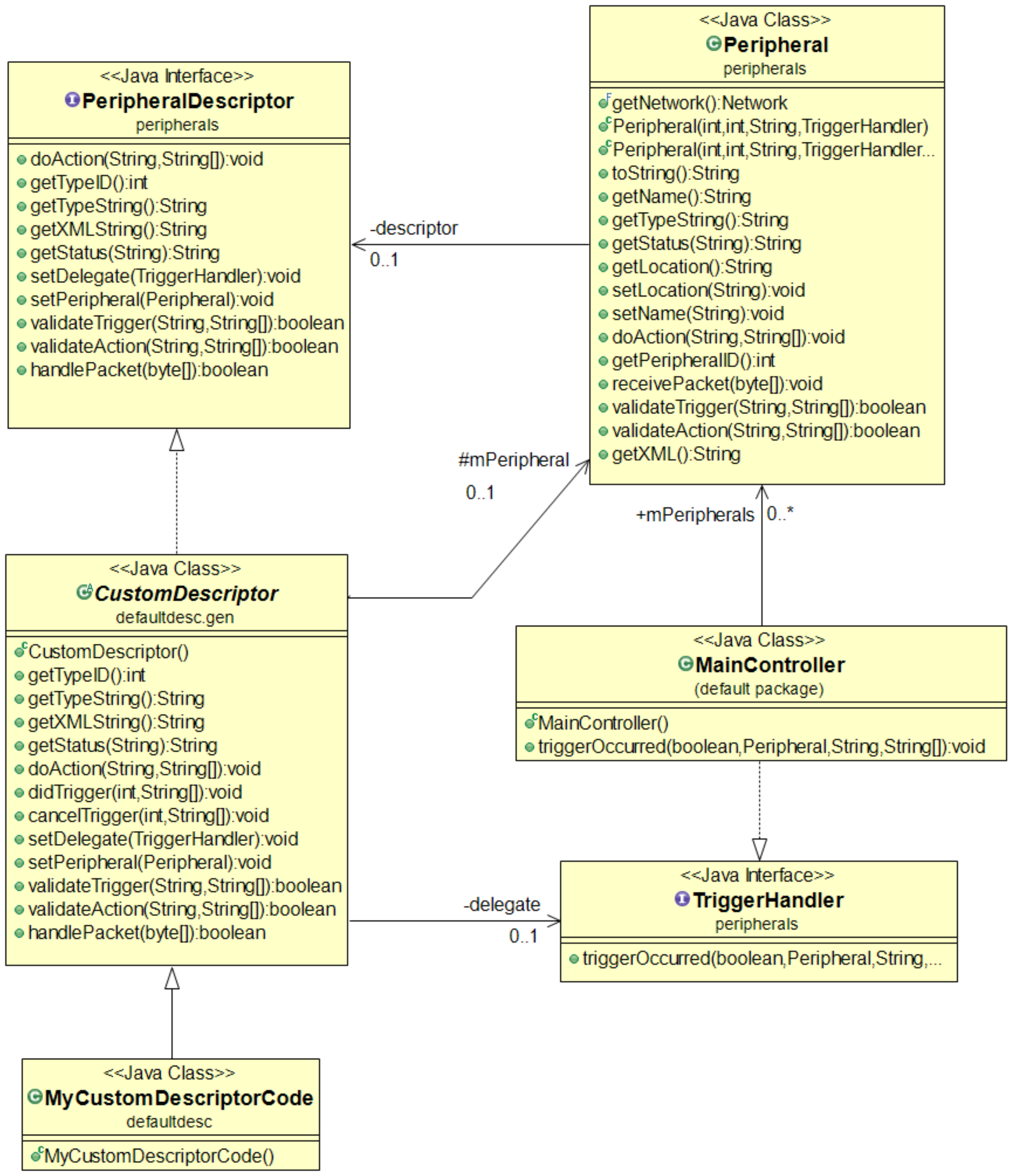
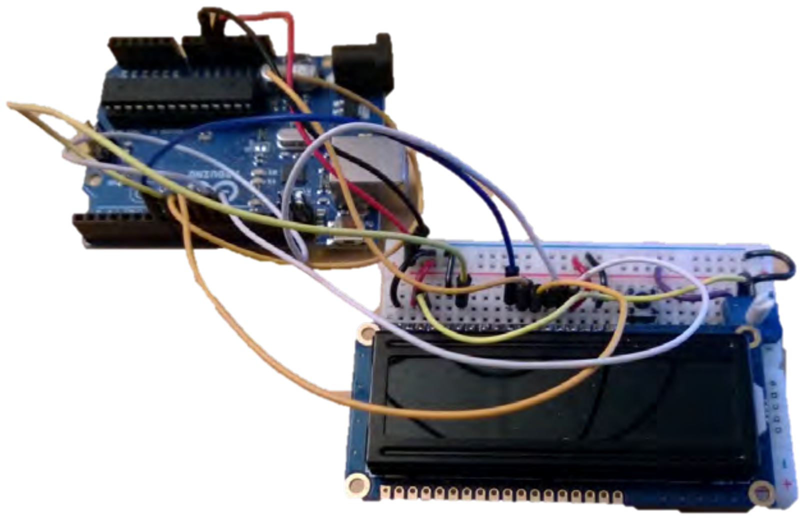
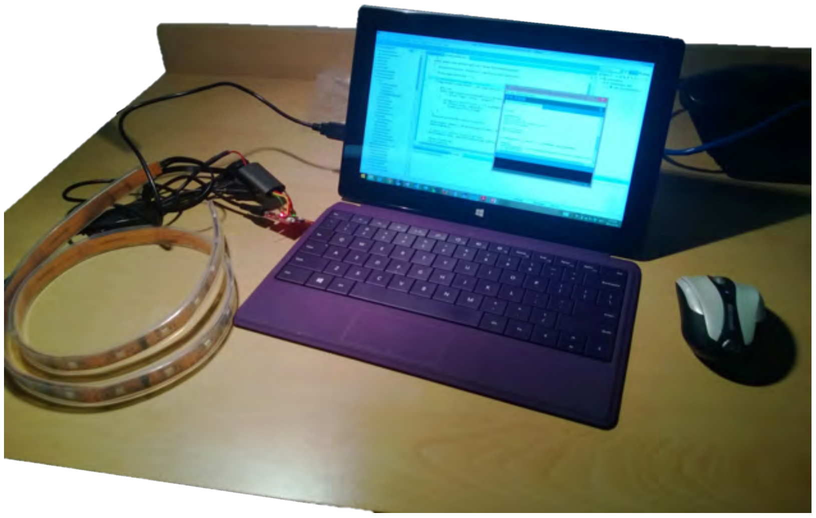

<<Java Interface>>
PeripheralDescriptor
peripherals
doAction(String,String[]):void
getTypeID():int
getTypeString():String
getXMLString():String
getStatus(String):String
setDelegate(TriggerHandler):void
setPeripheral(Peripheral):void
validateTrigger(String,String[]):boolean
validateAction(String,String[]):boolean
handlePacket(byte[]):boolean
<<Java Class>>
Peripheral
peripherals
getNetwork():Network
Peripheral(int,int,String,TriggerHandler)
Peripheral(int,int,String,TriggerHandler...
toString():String
getName():String
getTypeString():String
getStatus(String):String
getLocation():String
setLocation(String):void
setName(String):void
doAction(String,String[]):void
getPeripheralID():int
receivePacket(byte[]):void
validateTrigger(String,String[]):boolean
validateAction(String,String[]):boolean
getXML():String
-descriptor
0..1
#mPeripheral
0..1
+mPeripherals
0..*
<<Java Class>>
CustomDescriptor
defaultdesc.gen
CustomDescriptor()
getTypeID():int
getTypeString():String
getXMLString():String
getStatus(String):String
doAction(String,String[]):void
didTrigger(int,String[]):void
cancelTrigger(int,String[]):void
setDelegate(TriggerHandler):void
setPeripheral(Peripheral):void
validateTrigger(String,String[]):boolean
validateAction(String,String[]):boolean
handlePacket(byte[]):boolean
<<Java Class>>
MainController
(default package)
MainController()
triggerOccurred(boolean,Peripheral,String,String[]):void
-delegate
0..1
<<Java Interface>>
TriggerHandler
peripherals
triggerOccurred(boolean,Peripheral,String,...
<<Java Class>>
MyCustomDescriptorCode
defaultdesc
MyCustomDescriptorCode()

Figure 6.3: Classes that make up the peripheral interface



(a)



(b)

Figure 6.4: Development workspace: (a) simulator with character LCD, (b) complete workspace

Chapter 7

Service Engine

The purpose of the service engine is to provide the ability to connect triggers and actions of one or more peripherals in a way that is flexible and intuitive enough for a user with no programming experience to make use of through the user interface. To allow a user to keep track of the connections between triggers and actions in their system, the concept of a service was developed. A service is a relationship between one or more triggers and one or more actions in the system. The service engine is capable of managing multiple services so that a user can sort and manage multiple small simple relationships instead of one large complex relationship for the entire system.

7.1 Service Representation

Several options were considered for the method of representing a service, namely a separate user written program, the observer pattern, a string representation, and a custom approach called logic blocks. The options and their advantages and disadvantages are outlined in the following subsections.

7.1.1 Separate Program

The first method of representing a service that we considered was to allow a user to write their own Java program and have it running in its own thread on the base station. This method provides the ability for a flexible service but requires a user to write a program for each service. This method was not chosen since the average user is not expected to have the knowledge required to write a program and because writing a whole program for a basic service is inefficient for users who capable of writing such a program.

7.1.2 Observer Pattern

The second method of representing a service that we considered made use of the observer pattern, with actions being observers and triggers being observables. This method would be understandable for the user who would be able to select an action and the trigger it observes. One disadvantage with this method is the potential for a large number of threads since the number of threads is proportional to the number of actions. Another disadvantage with this method was that a user may wish to require a combination of triggers to occur before an action is called, something that could be difficult to implement under the observer pattern. This method was not chosen because of the challenges in implementing trigger combinations, which are necessary for a flexible service.

7.1.3 String Representation

The third method of representing a service that we considered made use of a String to represent a service. The syntax of the String would be designed so that combinations of triggers and actions could be specified, using AND and OR operations that are intuitive to the average user. Past occurrences of triggers could be stored so as to allow an action to occur only if the logic of the specified trigger combination had been previously satisfied. The disadvantage of this method is that every time a trigger in the system occurs, the

String representation of every service in the system has to be parsed in order to determine if it used the trigger. Then, the representation of each service that did use that trigger has to be checked to see if all of the other trigger conditions have been satisfied. This method was not chosen since it requires too much computation each time a trigger occurs.

7.1.4 Logic Block Approach

The fourth method of representing a service that we considered was named the logic block approach. The logic block approach makes the use of individual nodes called *logic blocks* in a directed graph. There are four different kinds of logic blocks: trigger logic blocks, action logic blocks, AND logic blocks, and OR logic blocks. By representing the system as a collection of logic blocks, the efficiency problem encountered using the third method is resolved by keeping a collection of the trigger blocks. When a trigger occurs, the trigger blocks are searched for the appropriate block. When the trigger block has been found, the logic blocks it is connected to represent every service in the system that uses that trigger. The trigger can propagate through the logic blocks that represent trigger combinations until one or more action blocks are reached at which point the appropriate action is executed. Each service is represented as a collection of logic blocks within the larger collection of logic blocks that represent the entire system so as to eliminate the potential for redundant logic blocks in the system. The disadvantage of this method is that creating a service and adding the appropriate logic blocks to the system can be time consuming and difficult to implement. This option was selected since services are expected to be added to the system much less often than a service will be executed. The details of the implementation of the logic block approach are discussed in the next section.

7.2 Service Engine Implementation

The implementation of the service engine included four primary tasks: creating, removing, and editing a service as well as operation of a service. Each task is discussed in the following subsections.

7.2.1 Service Operation

The operation of a service is dependant upon its logic blocks, which are each represented by `LogicBlock` objects. Each `LogicBlock` object contains a list of `LogicBlock` objects connected as inputs (also called previous logic blocks) and a list of `LogicBlock` objects connected to the output (also called next logic blocks) as well as a boolean `output` value that can be updated using the `update` method. The `LogicBlock` class is defined as an abstract class and leaves the abstract update method for the subclasses to implement. The update method accepts a boolean parameter whose use is specific to the subclass. Four subclasses of the `LogicBlock` class were implemented: `TriggerLogicBlock`, `ActionLogicBlock`, `AndLogicBlock`, and `OrLogicBlock`.

The `TriggerLogicBlock` class overrides the method for adding an input logic block to the list of input logic blocks so the list is never populated. The update method of a `TriggerLogicBlock` is called by a trigger manager thread that removes trigger requests from a queue that is populated by the peripherals. The update method of the `TriggerLogicBlock` class sets the output of the `TriggerLogicBlock` to the value provided to the method and then calls the update method of all `LogicBlock` objects in the list of next `LogicBlock` objects. Each `TriggerLogicBlock` object contains a reference to a `Peripheral` object, a `String` that contains the name of the trigger, and an array of String arguments for the trigger. These objects are used to identify the trigger that is represented by a `TriggerLogicBlock`.

The `ActionLogicBlock` class overrides the method for adding a next logic block to the list of next logic blocks so the list is never populated. Similar to a `TriggerLogicBlock`, an

`ActionLogicBlock` object contains a reference to a Peripheral object, a String that contains the name of the action, and an array of String arguments for the action. When the update method of the `ActionLogicBlock` class is called, the `doAction` method of the `Peripheral` object is called if the boolean input is true and the action name and arguments are passed as inputs. If the boolean input is false nothing is done.

The `AndLogicBlock` object provides the ability to combine triggers using the AND operation. When the update method of an `AndLogicBlock` object is called, the output values of all logic blocks in the list of previous `LogicBlock` objects are checked. If the output values are all true, then the output value of the `AndLogicBlock` is set to true. Otherwise the output is false. The update methods of all logic blocks in the list of next `LogicBlock` objects are then called if the output is true or if the output changed from true to false. If the output was false before the update method was called the next blocks are not updated since it should not cause an action to occur and the next blocks are all still up to date.

Similar to the `AndLogicBlock`, the `OrLogicBlock` provides the ability to combine triggers using the OR operation. When the update method of an `OrLogicBlock` object is called, the output value of the `LogicBlock` that called the update method is passed as the boolean input parameter. If the caller output is true, there must be at least one logic block in the list of previous logic blocks with a true output, so the output is set to true and the next logic blocks are updated. If the caller output is false, all of the previous logic blocks are checked to see if any of them have a true output. If not, the output of the `OrLogicBlock` object is set to false. If the output value of the `OrLogicBlock` has changed as a result of calling the update method, the next blocks are updated.

As can be seen in the example service in Fig. 7.1, the different logic block types can be combined to produce a service with a combination of triggers causing different actions to occur. In this example, if trigger T1 is updated to true then action A1 is performed. If trigger T3 is updated to true and T1 has not been updated to be false, then action A2 is performed. If trigger T3 is then updated to true both A1 and A2 are performed.

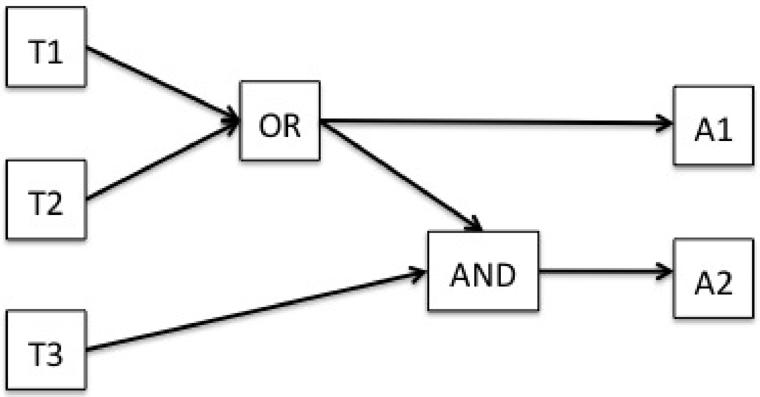


Figure 7.1: An example of a service using logic blocks with triggers T1, T2, and T3 and actions A1 and A2

7.2.2 Creating a Service

In order to create a service there needed to be a way of determining the associated triggers and actions as well as the relationship between them. The information also needs to be in a format that can be passed from the user interface. After receiving the necessary information, the service creation algorithm needs to generate or find the appropriate logic blocks.

Service Representation for Service Creation

Two options were considered for representing a service to the user interface. The first option was to use Javascript Object Notation (JSON) to represent logic block objects and have the user interface send a list of logic blocks represented in JSON. The first option was not chosen because it would require the user interface to manipulate logic block objects, which could be difficult to implement. Conversely, the second option represents a service as a String that contains the necessary information to reproduce the service. The second option was chosen to allow easier implementation of the user interface and is detailed in the remainder of this section.

Triggers and actions are both represented by a name associated with the peripheral whose trigger or action is being represented, followed by a ':::', followed by the name of the trigger or action itself, followed by zero or more arguments, all separated by a ":::". For example, a trigger named "trig" occurring on peripheral "ed" is represented as "ed:::trig". Another example would be a trigger named "tempGreaterThan" occurring on peripheral "thermometer" with one argument, the threshold temperature, is represented as "thermometer:::tempGreaterThan:::21". We chose to use three colons as the delimiter because a peripheral, action, or trigger name or an argument may contain a single colon and a triple colon is much less likely to be used.

All services codified in Strings follow the format "A::->B" where B is an action and A is a combination of various triggers. B follows the format described in the previous paragraph. For example, an action named "doStuff" with no arguments to be done on peripheral "c" is represented as "c:::doStuff". The combination of triggers that precedes the implication sign can make use of none, one, or both of the logical operators OR and AND. The logical operator OR is represented as a '+' and the logical operator AND is represented as a '&'. Each layer of operations is to be enclosed in parentheses preceded by a double colon, including the case with no operations (a trigger). For example, the combination of the triggers "1:::a" and "2:::b" using the OR operator is denoted as "::(::(1:::a::)+::(2:::b::)::)". It is important that no spaces are used in the representation, except in the actual names of peripherals, triggers, and arguments. For another example, consider "::(::(::(1:::a::)+::(2:::b::)::)&::(3:::c::)::)", taking the AND of the result of the previous example and the trigger "3:::c". Both the AND and OR operations support more than two terms. For example, "::(::(1:::a::)+::(2:::b::)+::(3:::c::)::)" is a valid combination. Lastly, for clarity, a service whose combination consists of only one trigger, say "1:::a", has a trigger combination "::(1:::a::)".

A service may be composed of multiple parts with the same or different action. Each part follows the "A::->B" format mentioned earlier and is separated from other parts by a "::,". For example, "::(1:::a::)::->4:::doStuff::,::(::(1:::a::)&::(2:::b::)::)::->3:::doOtherStuff" is a valid service representation.

Service Creation from Service Representation

To create a service from the String representation used by the user interface, the `String` is first split around occurrences of "::,". Each part is then processed separately. Next, each part is split with "::->" as the delimiter. The sub-string before the "::->" is the combination of triggers and the sub-string after the "::->" is the action. Next, the algorithm finds the logic block whose output is logically equivalent to the trigger combination through the use of a recursive function, creating new logic blocks where necessary.

The recursive function used to find the logic block whose output is logically equivalent to the combination of triggers first removes the outer pair of parentheses (and the double colon delimiters) from the trigger combination passed to it. If after removing the outer parentheses there are no more parentheses in the trigger combination, then the trigger combination is a single trigger. If the trigger combination is a single trigger, the algorithm checks for a corresponding `TriggerLogicBlock`. If one is found it is returned. If there is no corresponding `TriggerLogicBlock` in the system one is created and returned. If the trigger combination is not a single trigger the algorithm iterates the String, incrementing a count when "::(" is encountered and decrementing the count when "::)" is encountered. Whenever the count becomes zero the end of a term of either an AND or OR operation has been reached. The operation symbol is checked to ensure that only one operation is being used. The sub-string representing each term is then passed to the recursive function. The algorithm stores the logic blocks returned from calling the recursive function on each term and then checks to see if a logic block for the corresponding operation exists that uses these stored blocks as inputs. If such a block exists it is returned. If not, the appropriate block is created, connected to the blocks returned from calling the recursive function on each term, and then returned.

After calling the recursive function on the trigger combination of the service, a logic block representing the trigger combination has been either created or found. The substring representing the action is parsed and an appropriate `ActionLogicBlock` is found, or created if one does not exist. The logic block representing the trigger combination is then connected to the `ActionLogicBlock` and the service has been successfully created. An `Exception` is thrown if there was an error in service creation such as invalid syntax, mismatched operations, or a request for a peripheral that no longer exists in the system.

7.2.3 Removing a Service

Each `Service` object contains a method called `deleteFromSystem` that removes the service from the system. Since multiple services can use the same logic block, one cannot simply remove all of the logic blocks associated with the service from the system. In order to allow removal of the appropriate logic blocks, each logic block contains a counter called `numServices` that keeps track of the number of services that use said logic block. The number of services is incremented every time a new service uses the logic block and is decremented when a service that uses the logic block is removed from the system. If the number of services equals zero when after decrementing it, the logic block is removed from the system.

7.2.4 Editing a Service

Two methods were considered for editing a service. The first method was for the user interface to provide a list of changes to make to a service. The second method was for the user interface to simply provide a representation of what the newly edited service should look like. The difficulty involved in implementing the first method in the user interface as well as in the service software makes the first option impractical. The advantage of the second option is that the method used for creating a service can be adapted for the purposes of editing a service. Therefore the second option was selected because it was easier to implement.

Instead of adapting the algorithm for service creation to editing a service, we devised a solution that was much simpler to implement. First, the representation of what the edited service should look like is taken and passed to the method for adding a service to the system. Then the logic blocks corresponding to the previous representation of the service are removed using the method discussed in Section 7.2.3. After the removal is completed the only logic blocks left in the Service object are the ones that represent the edited service and the logic blocks that are no longer used by any services have been removed from the system.

Chapter 8

User Interface

The user interface acts as the control panel of the system. Its primary purpose is to provide users with an intuitive platform for monitoring or controlling peripherals, and configuring services. Additional features of the user interface include a message board which can display notifications posted by services, a user authentication procedure which provides a measure of security to the system, and a method for developers to upload customized peripheral descriptors. The creation of the user interface involved four primary tasks: selecting and becoming familiar with a platform, researching and designing the appearance and functionality, implementing both the front-end and the back-end of the interface, and testing it with and without the web server located on the base station.

8.1 Method Selection

Early in the project timeline, our group made the decision to implement the user interface as a web page. The primary alternative considered at that time was an app. The web page was the better choice for two main reasons. First, web browsing is a function provided by most modern computing devices, including phones, tablets, and laptop or desktop computers, whereas apps are most often usable only by the platform for which they are developed.

Second, web pages inherently make use of well-established networking communications protocols. Even in the early stages of the project we recognized that the user interface would be required to communicate with at least one other device in order to relay commands from the user to the rest of the system. Creating an app which is capable of communicating with other devices requires more work than simply creating a web page.

Once the decision was made to implement the user interface as a web page, the most immediate task was to become familiar with the tools which are most commonly used to create web pages. This was a fairly sizeable task since it involved learning how to use a number of previously unfamiliar programming languages, such as HyperText Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript. Thankfully, this process was aided by many tutorials and examples which are available online [24]. Using the knowledge gleaned from these resources, a number of common design techniques were practiced through the creation of a dummy web page. Some of these techniques included how to neatly array multiple items within a single web page, how to dynamically size items with respect to the size of the browser window, how to link buttons to dynamic content, and how to generate non-intrusive popup windows.

8.2 Research and Design

The second primary task in creating the user interface was researching and designing its appearance and functionality. The research took two forms: viewing and analyzing interfaces for existing systems which were similar to the one proposed for the project, and gathering sketches of imagined interfaces from volunteers to whom the project system was described.

Analyzing Existing Systems

The first research method served as a means to gather many ideas quickly, and by comparing similar features between multiple sample interfaces, certain ideas were seen to be better than others. For example, in one of the sample interfaces, all of the information regarding multiple devices were displayed in a single screen, whereas another sample interface listed the names of multiple devices, and displayed more detailed information only if further queried by the user. Between the two interfaces, the second did a much better job of presenting the same information since it not only made reading the information easier for the user, it also more easily accommodated the possibility that different devices might need to display different types of information. As a result, we made use of the idea of initially displaying only the names of peripherals was noted, and later on made an appearance in the design phase.

Acquiring Sketches

The second research method, gathering sketches, was done to ensure that the design phase would be at least partly influenced by potential users. Ideas gathered using this method were meant to strengthen and in some cases redirect the ideas that were gathered from the first research method. Perhaps the most important idea that was expressed in one of the sketches was the concept that users be provided with the ability to explore the system from the perspective of individualized services. This concept extended itself through the design phase and is now an inherent part of the user interface.

Paper Prototypes

The design phase primarily involved the creation of sketched proposals of the final design, or "paper prototypes". By drawing all of the various screens and features which the research phase suggested be present, the entirety of the user interface was explored prior to implementation. In situations where several different ideas seemed plausible, multiple designs of a particular feature were drawn and then compared. For process-driven features, drawings were presented in a manner similar to flow charts, indicating which stages followed which user actions. For example, adding a new peripheral to the system involves a number of steps, one of which involves the user pressing a button on the peripheral to send a confirmation signal to the base station. Therefore, the drawing for this complete process, which can be seen in Figure 8.1, shows the appearance of and the connections between several stages, including:

1. An “add” button
2. A timer indicating the number of seconds remaining for the user to press the button
3. A message indicating that a signal was not received (in case the timer expired)
4. A message indicating that a signal was received (if the user pressed the button in time) and a request that the user name the newly attached peripheral.

The design phase was concluded with a formal design review in which we analyzed and discussed all of the drawings. For each drawing, a group decision regarding the best design was made. The most important result of this meeting was the first version of the web page hierarchy, which is embodied in the diagram seen in Figure 8.2. This hierarchy outlines how it was thought that the web page should flow following the design phase of the user interface, and it includes the three main pages as well as which features which are accessible from those pages. Items in rectangular boxes represent web pages while items in oval boxes represent features.

8.3 Front-End Programming

The front-end of the user interface is that which the user sees; it encompasses all of the formatting and features which contribute to the general appearance of the user interface. Because the user interface is a web page, its front-end is defined by code written in HTML and CSS. The majority of this code was written ahead of the back-end programming, but some sections were added or adjusted at a later time.

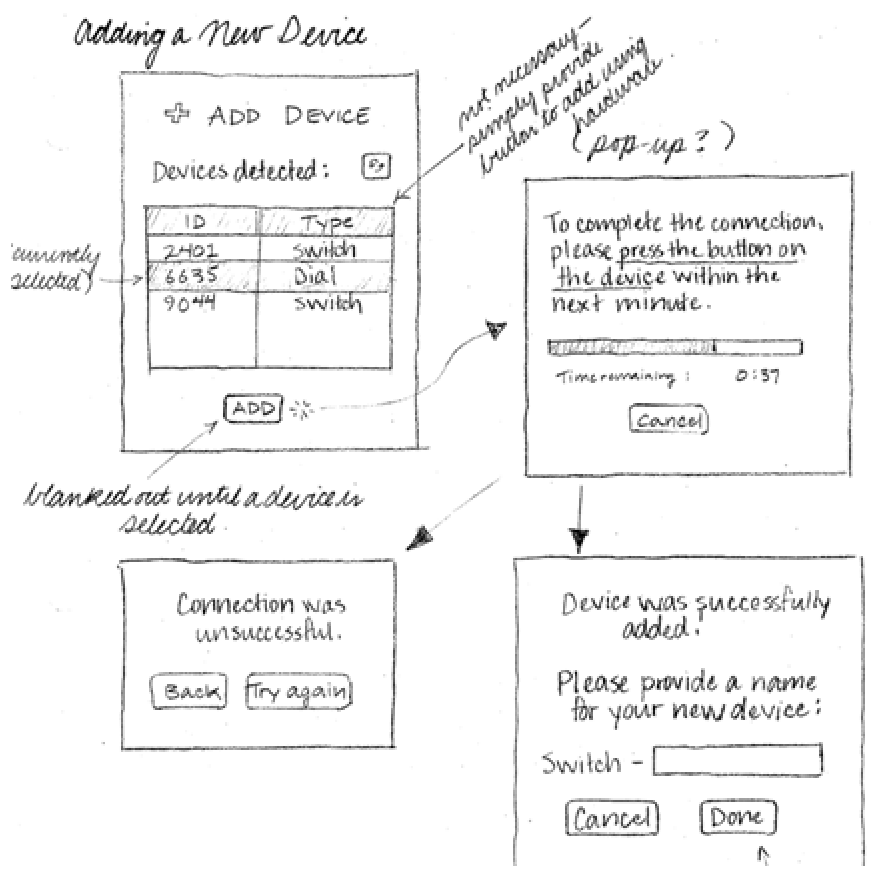

Adding a New Device
ADD DEVICE
Devices detected:
ID
Type
2401
Switch
6635
Dial
9044
Switch
ADD
(currently selected)
blanked out until a device is selected
not necessary — simply provide button to add using hardware
(pop-up?)
To complete the connection, please press the button on the device within the next minute.
Time remaining: 0:37
Cancel
Connection was unsuccessful.
Back
Try again
Device was successfully added!
Please provide a name for your new device:
Switch -
Cancel
Done

Figure 8.1: Prototype of the process for adding a new peripheral to the system

Cascading Style Sheets

All of the front-end programming is contained within several HTML files and a single CSS script. Each HTML file implements a single page while the CSS script defines the common styling for all of the pages. In this way, the CSS script is the core of the front-end programming because it is primarily responsible for the formatting and appearance of features within the user interface. The alternative to maintaining the styling for multiple pages in one CSS script is to define the styling of each page within its corresponding HTML file. Although all of the pages require some unique styling elements, it was decided to make

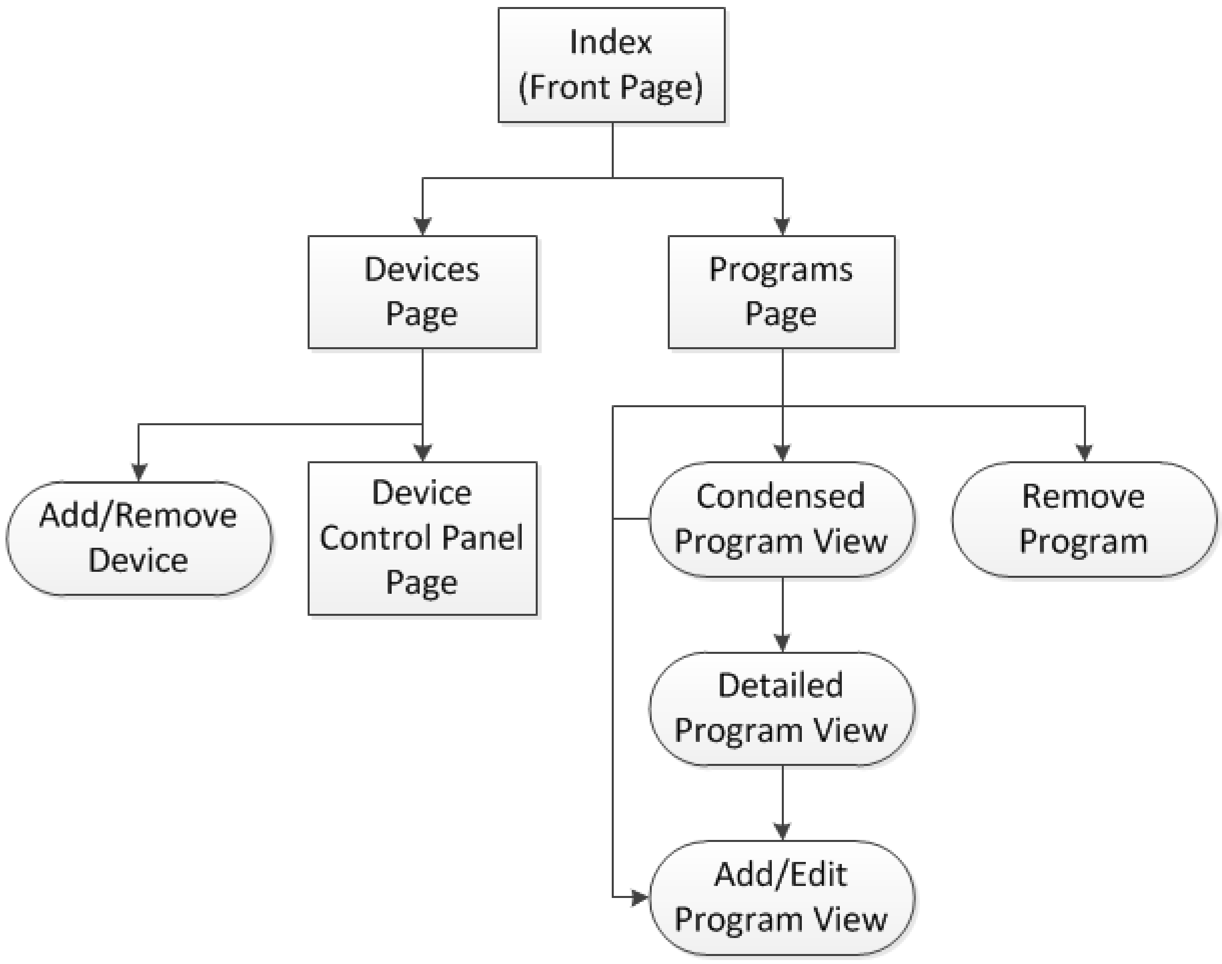


Figure 8.2: Web page hierarchy following the design phase

use of a single CSS script so that shared elements between the pages would not need to be repeated multiple times. This helped to reduce code clutter, improving maintainability and making it easier to work on the user interface. For example, when a new page was added to the user interface, instead of encoding all of the new page's styling within its HTML file, a single line can be added which links the new HTML file to the existing CSS script, automatically causing the appearance of the new page to match the rest of the user interface.

Non-Intrusive Popups

Within the context of front-end programming, a significant implementation problem is how to present a small amount of additional information on a page which already contains a large amount of content. An example of a user wishing to access a small amount of additional information would be, whilst viewing the actions and triggers associated with a particular service, attempting to find out more about any one of the actions or triggers. Such a small amount of information makes retrieving an entirely new page a tedious operation for the user, yet initially all of the relevant information on the current page would results in content which is too cluttered. We solved this problem using non-intrusive popups, which are panels that appear on top of the currently displayed content without opening a new window. This was achieved by adjusting the attributes of various web page components using CSS following specific events, such as a user pressing a button. Specifically, the core concepts involved dimming the currently displayed content and causing a previously invisible panel to appear in the centre of the screen, containing the information requested by the user.

Final Version

Although it very closely matches the flow outlined in Figure 8.2, the final implementation of the front-end was slightly altered from the original design. The web page hierarchy for the final implementation of the front-end can be seen in Figure 8.3. The main features which differ between the original design and the current version are the features accessible from the programs page. Although it is still possible to view a condensed version of a program, the detailed view is now part of the overarching add/edit program page. Additionally, the current version includes the following pages which were not previously considered in detail: the create account page, which is only retrieved if an account has not yet been created; the login page, which is only retrieved if the user did not recently log in; the upload descriptor page; and the edit account settings page.

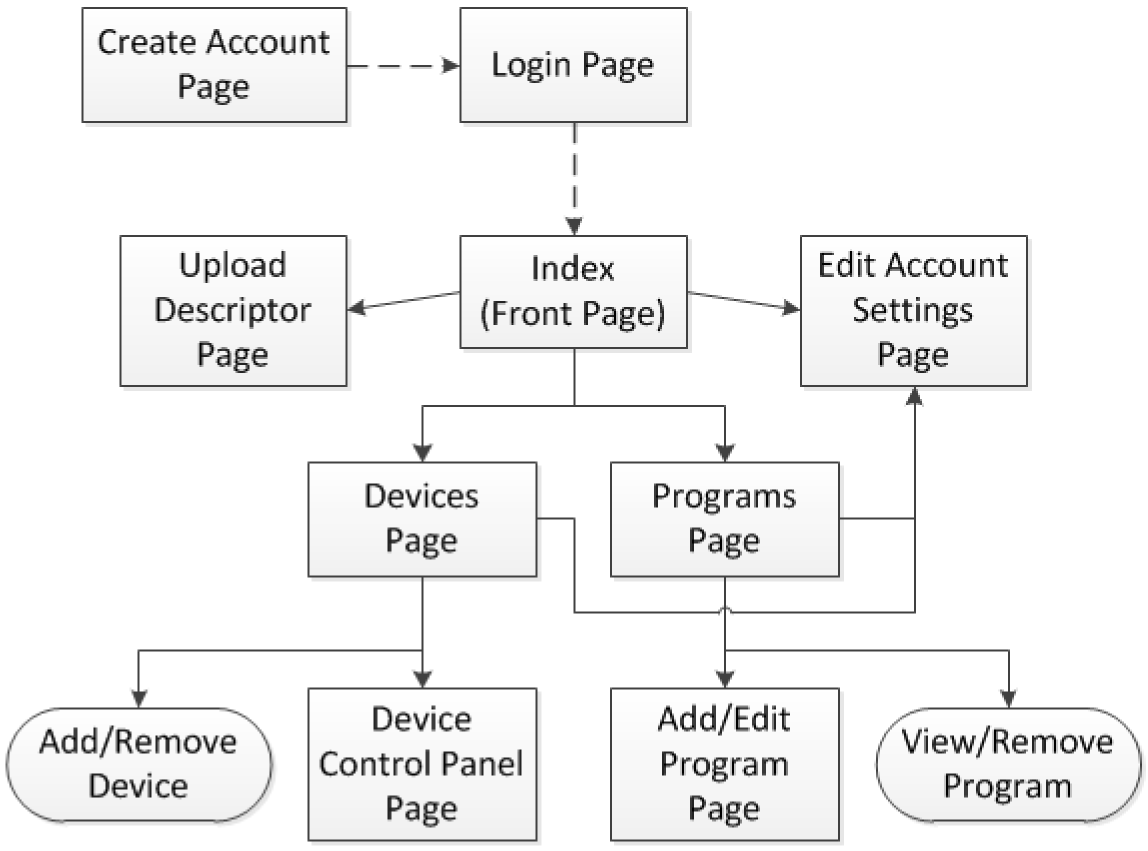


Figure 8.3: Current web page hierarchy

8.4 Back-End Programming

The back-end of the user interface includes all of the code which supports the functionality of the web page. All of the back-end code was written in JavaScript, and is contained within one file which is linked to all of the HTML files. As the largest component of the user interface, the back-end programming involved generating solutions to many major issues.

Communication using HTTP

The most important function of the back-end of the user interface is retrieving data from the base station. Because the user interface is a web page, it is required to use HTTP requests such as HTTP GET and HTTP POST. We did however need to determine how best to use each type of request, and how requests from and responses to the web page should be formatted. Based on how the web server was implemented, which is discussed in Chapter 9, it was determined that GET messages should be used to retrieve pages and POST messages should be used to retrieve any other sort of data. Therefore the URL included in a GET request is simply the URL of the page to be retrieved, whereas the content of a POST indicates what sort of data should be returned. The format of the content for POST requests and responses is also discussed in Chapter 9. To accommodate this method of communication, two JavaScript functions, `get_page` and `http_post_request`, were created for use by any page within the user interface.

Part of the decision to use HTTP POST requests to retrieve any sort of data which was not a new page was the choice to use Asynchronous JavaScript and XML (AJAX) to display dynamic content. AJAX is a JavaScript tool which allows a page's content to change when an HTTP response is received, without needing to refresh the entire page. In addition to this capability, AJAX is well-documented and easy to use, therefore we decided that AJAX should be used in combination with HTTP POST requests to retrieve data and update the user interface with dynamic content, as necessary.

Accepting User Input

In contrast to retrieving data from the base station, another large design problem encountered in programming the back-end of the user interface was retrieving data from the user. In order to complete certain commands, the user is at times required to provide a small amount of data, e.g. the number of seconds a peripheral should wait before executing some action. Naturally, different commands require different types of data, therefore it was necessary to determine how different types of data should be accepted. The two main options considered were to provide the user with a unique input method for each data type, or to provide a generic text input field and to check the submitted data for any errors. We determined that the latter was the better option since it requires less work to implement and it is easier for the user to interact with. An example of a scenario which demonstrates this would be if the user were required to enter an integer. A unique input method for this data type would likely take the form of a default number to be incremented and decremented by two corresponding buttons. Making use of such a pair of buttons would quickly become tedious if the default number was far from the user's desired number. A generic text input field, however, allows a user to enter any data they wish, and if by accident something other than an integer is entered, the user is simply asked to correct the error. As a result of it being easier to implement and to use a generic text input field, an error-checking algorithm was created.

Managing Services

By far the largest section of the back-end code was that which supported the add/edit program page, and the largest implementation consideration in this section was that of interpreting and creating service representations. Since services can contain any number of triggers and actions in an infinite number of combinations, creating a generic set of functions capable of deciphering or constructing the string representation of any service is not trivial. The solution comes in two parts: traversing string representations with recursion, and representing the relationships between triggers and actions as a network of interconnected objects in memory. Recursion is a natural solution for traversing string representations since each service can be viewed as a series of tree structures where the root to each tree is an action, and all of the child nodes are triggers or nodes which combine triggers through logical AND or OR relationships. Figure 8.4 provides a visual representation of such a tree. Objects are necessary in order to differentiate actions, triggers, and logic nodes from one another, and to connect them.

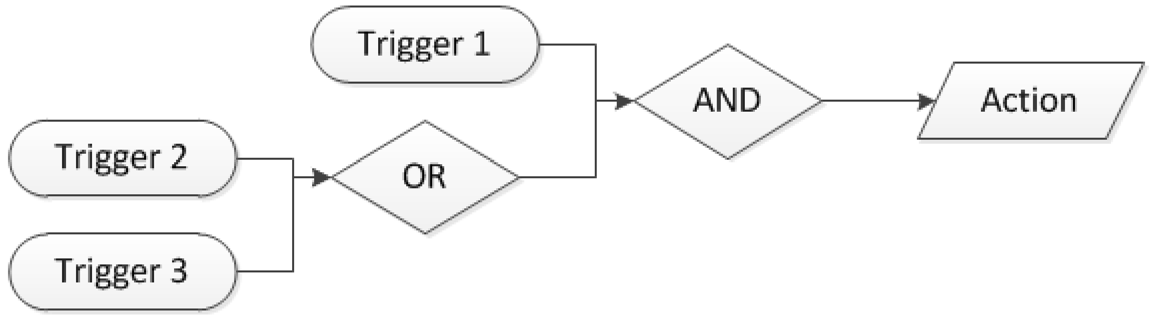


Figure 8.4: Relationship between triggers and actions

8.5 Testing

In order to speed up development, testing the user interface was carried out in two ways: with and without a web server. Each time the back-end support for a feature was written, the feature was first tested in the absence of a server, and afterwards using the web server developed for the base station. In order to accommodate this two-stage method of testing, the JavaScript file was designed to include a built-in test mode. By changing the state of the boolean `server_mode` at the top of the code, the back-end could be set to operate in a mode with or without a server. The code which was primarily altered by the state of this boolean was the function `http_post_request`. If the code was told to operate without a server, `http_post_request` would essentially bypass all HTTP communications by supplying simulated responses to the functions ordinarily meant to handle real responses. In this way, newly added back-end code could be immediately tested without the need to set-up and configure a web server. This streamlined the debugging process because glitches in functionality could be more quickly isolated and dealt with.

Chapter 9

Web Server

A web server was needed to connect the user interface to the peripherals and services on the base station. The following steps were taken in development of the web server:

1. The web server was selected.
2. The ability to communicate to the user interface was established.
3. The ability to communicate with services and peripherals was established.
4. Communications between the web server and web browser were encrypted.
5. User authentication was implemented to deter unwanted access to the system.

9.1 Web Server Selection

Two options were considered for the web server. An already available solution, such as Apache's Tomcat or the Eclipse Foundation's Jetty, and a custom solution were considered[25][26]. One advantage of using an already available solution is that the main code that handles requests and responses is already included. Another advantage of the first option is that the available solutions are well documented although reading through the documentation in order to learn how to install and use an available solution can be time consuming. Another disadvantage of the available solution is that they take up memory with many extra features that are not used by our system. On the other hand, a custom solution offers a memory efficient design because only the required features are implemented. The disadvantage of the custom solution is that time has to be taken to develop, implement, and test the server. The custom solution was chosen in the interest of memory efficiency and because we estimated that implementing a basic web server would take about as much time as to learning how to install and use an already available solution.

There were three options considered in the implementation of the custom web server. The first option was to use servlets; unfortunately since Java SE does not include support for servlets this option had to be discarded. The second option was to program the server using sockets, which requires significant low-level processing. The third option we considered was the `com.sun.net.httpserver` package which provides an HTTP server that already handles the low-level socket programming[27]. The third option was selected because of its easier implementation. To implement the HTTP server a class had to be created that handled requests from the user interface and a new or existing class had to create and start the HTTP server.

9.2 Communication with User Interface

The `WebRequestHandler` class was created to handle requests from the user interface. This class implements the `HttpHandler` class from the `com.sun.net.httpserver` package and implements the `handle` method that handles web requests[27]. In order for the user interface to provide up-to-date information about the system, use of dynamic content was necessary. Two options for providing dynamic content were considered. The first option was to make use of server-side programming that would fill in the necessary information before sending it to the web browser. The second option was to have a client-side script asynchronously request any necessary information that was missing. The primary advantage of the first option is that it reduces network traffic. The main advantage of the second option is that it reduces the workload of the user interface developer since they only need to design one client-side script instead of having to develop one for both the client and server. We chose the second option since it was simpler to implement.

Since our system uses asynchronous information requests the `WebRequestHandler` handles two different kinds of requests: page requests and information requests. The page requests are HTTP GET requests since that is how web browsers request a web page. The information requests are made using HTTP POST requests, which allow an asynchronous information request to include detailed request information. An information request can be recognized by its URL of "program_data" and the body of the request and response follow the protocol outlined in Appendix F.

In addition to all of the information requests, a feature was added to the web server that allows a user to remotely upload a peripheral descriptor to the server. This can be done by accessing the "uploadDescriptor.html" HTML page. This page contains an HTML form with a file input type that makes use of multipart/form-data encoding to send the peripheral descriptor file information to the server. The POST request associated with this form is sent to the URL "uploadDescriptor". The web request handler then parses the request body, removing the leading and trailing information added by multipart/form-data, and saves the file in the plugin folder. After saving the new descriptor the web request handler refreshes the descriptor factory and the plugin is seamlessly loaded by the plugin framework.

9.3 Communication with Peripherals and Services

After receiving and parsing an information request from the user interface the web request handler calls methods in the `MainController` class that provide the necessary information. When the `WebRequestHandler` object is initialized it is passed a reference to a `MainController` object. The main controller contains: a list of all of the `Service` objects in the system; a list of all of the `Peripheral` objects in the system; a `NetworkAdapter` that handles communication to peripherals; and a list of all of the `TriggerLogicBlock` objects in the system, which gives access to every logic block in the system. While most of the main controller functions simply delegate the requests to the `Peripheral`, `Service`, and `NetworkAdapter` classes, some methods perform tasks in addition to delegating requests. For example, the method to create a service in the main controller assigns an unused service ID to the service being created because it has access to all of the services in the system. The main controller also handles all access to the MySql database. Further information on the methods implemented in the `MainController` class can be found in the Javadoc documentation found in the software repository (See Appendix G).

9.4 Communication Encryption

HTTPS was used to encrypt the communications between the web server and the user interface. This feature was provided in the `com.sun.net.httpserver.HttpsServer` class[27]. A custom JKS keystore was created using the *keytool* command. To start the HTTPS server, the keystore is loaded by the program. Then, a key manager factory is initialized to use the custom keystore. Next, the SSL context is initialized using the key manager factory. Lastly, the SSL context is used to configure the HTTPS server. After starting the server the communications are all automatically encrypted using 128-bit encryption. Since the project proposal did not specify any certificate requirements, the certificate is self-signed[2]. Future work on the web server could include investigating the process of getting a signed security certificate. The server is set to listen on port 443, which is the default port used for HTTPS.

9.5 User Authentication

To deter unauthorized access our system requires user authentication. When a user inputs their user name and password on the log in page and presses the “Log In” button, the user name and password are sent to the server via an HTTP POST request to the URL “login”. The web request handler then queries the main controller to check to see if the user name and password combination is valid. If the combination is valid a session ID is set as a cookie in the web browser. The cookie is called “sid” and contains a Universally Unique ID that represents the session. Whenever the request handler receives an HTTP request, it looks for a valid session ID cookie. If the session ID is not valid or does not exist, the server returns the log in page instead of the requested page. If the HTTP request was an AJAX request, the server returns “not_logged_in” as the response. A log out feature is provided (by requesting the page “logout.html”) which destroys the current session ID and forces the user to log in again. A session ID also becomes invalid after one hour of inactivity. These two features are implemented to combat potential issues that can arise when two people use the same device and only one should have access to the system.

The log in system uses only one user account because session tracking and user privileges are not used. Recognizing that one or both of those features may be needed in future versions of this system, the authentication method was set up to be flexible in this regard. In order to ensure that there is only one user account, there had to be a way of creating the first and only account. We implemented a feature that redirects users to a page that allows them to create a user account if a user navigates to any page and there are no user accounts in the system. The user name and password for the new account are submitted via an HTML form POST to the URL “new_account” with two form fields, “userName” and “password”.

The user name and password combination for the user account as well as the session IDs are stored in two tables in a MySql database in order to keep the information persistent. The first table is named “users” and contains a `VARCHAR(30)` “userName” column and a

`CHAR(64)` “password” column. `CHAR(64)` was chosen for the password column because the password is put through a SHA-256 hash, which output results are always 64 characters long. The passwords are SHA-256 encrypted before being stored in the database so as to prevent malicious attackers from reading the password. When a user attempts to log in the password is hashed and compared to the stored hashed password in the database to verify if the correct password was submitted. The second table is named “sessions” and contains a `CHAR(36)` “sid” column and a `DATETIME` “expiryTime” column. `CHAR(36)` was chosen for the session ID column because every session ID is a Java Universally Unique IDentifier, which is always 36 characters long.

Functions are also provided by the main controller that allow a user to request a change to the user name or password. This ability is accessible to the user interface via the web request handler and is handled as an information request. The protocol for these two functions can be found in Appendix F.

Chapter 10

Conclusion

The primary objective of this project was to create the software framework for a general automation system. In addition to allowing users to directly monitor and control variable conditions in the local environment, the framework provides users with the ability to create their own specialized peripherals and services. The development of such a framework has been presented in this report through in-depth discussion of its major hardware and software components.

10.1 Summary

With respect to hardware, the system was broken down into three main subsystems: the base station hardware, the peripheral cores, and the demonstration system. Once hardware was selected and obtained for these subsystems, each was designed and tested to ensure they met the specifications set out at the beginning of the project. The capacity for peripherals to communicate wirelessly and carry out basic functions was established by the peripheral software. The development of this software primarily involved creating a custom networking protocol and embedding system code into the peripheral cores. The base station software was divided into four components: the peripheral interface, the service engine, the user interface, and the web server. The peripheral interface is responsible for generalizing communication between peripherals and the base station through the use of triggers and actions, and managing peripheral descriptors, which allow developers to build custom peripherals. The service engine allows triggers and actions to be combined in user-defined system behaviours called services, and is able to manipulate peripherals according to the relationships which services define. The user interface is the platform which allows users to interact with the system. It is a web page whose primary functions are to manipulate and configure devices and services. The user interface is hosted by and receives data from the web server, allowing users to control the system using a personal device such as a smart phone or a laptop.

10.2 Future Work

Although the current version of the framework has been built to specification, we recognize that there are a number of areas which could be expanded upon. Within the context of user interaction, the largest issues are security, the minimalistic user interface, and system latency. To date, the minimal amount of security which has been incorporated into the framework is enough to encode all network communications and prevent unauthorized persons from directly accessing the user interface, however it is likely that a dedicated infiltrator could breach these preventative measures. We recognize that this is a significant concern as unauthorized access to an environment such as a home could be very dangerous for the client. Due to the fact that a functional user interface required significant development time, the current version is somewhat lacking in flexibility, especially with regards to controlling devices and editing services. For this reason, the user interface could be upgraded to allow for a better user experience. System latency has been perceived to be relatively high, therefore its reduction would result in a system which is more convenient to use. In regards to networking between peripherals, the system could be expanded upon by using the ZigBee networking layer, allowing for the easy attachment of ZigBee-capable devices. To aid developers in creating custom peripherals, a development environment unique to the framework could be developed. Lastly, there exist two software-related issues which could be improved upon: coupling and lack of persistence. Currently some of the code which makes up the peripheral interface and the service engine includes functions which are highly coupled. In order to reduce complexity and avoid complications in future versions, this code could be improved through better encapsulation. In addition, if a power outage were to occur, all of the peripherals and services would need to be re-added to the system. Enabling persistent information storage would result in a much more reliable system.

References

[1] Rasberry Pi Foundation. Raspberry Pi FAQs. [Online]. Available: http://www.raspberrypi.org/faqs

[2] J. F. Baril, V. Beynon, S. Koop, S. Rohringer, and P. White, "Design and Implementation of a Framework for Remote Automated Environmental Monitoring and Control," January 2013, project Proposal for ECE4600.

[3] Griffith University. Average computer energy usage. [Online]. Available: http://www.griffith.edu.au/sustainability/sustainable-campuses/sustainable-initiatives/energy/average-computer-energy-usage

[4] Arduino Based IMU & AHRS. [Online]. Available: http://code.google.com/p/arduimu/wiki/ArduIMUV4

[5] Atmel Corporation, "ATmega128RFA1," Online, 2012. [Online]. Available: http://www.atmel.ca/Images/doc8266.pdf

[6] Texas Instruments, "CC430F6135," 2014. [Online]. Available: http://www.ti.com/lit/ds/symlink/cc430f6135.pdf

[7] C. A. Balanis, *Antenna Theory Analysis and Design*, 2nd ed. John Wiley & Sons Inc., 1997.

[8] P. Burgess. Power | Digital RGB LED Strip | Adafruit Learning System. [Online]. Available: http://learn.adafruit.com/digital-led-strip/powering

[9] PowerSwitchTail.com LLC, "PowerSwitch Tail II," October 2013. [Online]. Available: http://www.powerswitchtail.com/Documents/PST%20II%20Rev%206A%20Oct%202013.pdf

[10] ——, "PowerSSR Tail," 2011. [Online]. Available: http://www.powerswitchtail.com/Documents/PSSR%20product%20insert.pdf

[11] Oracle. Java SE - downloads. [Online]. Available: http://www.oracle.com/technetwork/java/javase/downloads/index.html

[12] F. Zhao. zigduino-radio: Open source arduino library for zigduino's radio. [Online]. Available: http://msgpack.org/

[13] IconLeak. light-bulb-icon.png. [Online]. Available: http://www.iconarchive.com/show/or-icons-by-iconleak/light-bulb-icon.html

[14] S. Furuhashi. Messagepack: It's like JSON. but fast and small. [Online]. Available: http://msgpack.org/

[15] M. Jasperse. msgpackalt - a simple, fast and lite binary serialisation library. [Online]. Available: http://code.google.com/p/msgpackalt/

[16] S. Alexey. java-simple-serial-connector - jSSC - java serial port communication library. [Online]. Available: http://code.google.com/p/java-simple-serial-connector/

[17] W3C. Extensible markup language (XML). [Online]. Available: http://www.w3.org/XML/

[18] Oracle. Lesson: Annotations. [Online]. Available: http://docs.oracle.com/javase/tutorial/java/annotations/

[19] ——. JDK 8. [Online]. Available: http://openjdk.java.net/projects/jdk8/

[20] M. H. Kay. The saxon XSLT and XQuery processor. [Online]. Available: http://saxon.sourceforge.net/

[21] W3C. XSL transformations (XSLT) version 2.0. [Online]. Available: http://www.w3.org/TR/xslt20/

[22] JPF Team. Java plugin framework (JPF). [Online]. Available: http://jpf.sourceforge.net/

[23] OSGi Alliance. OSGi alliance—main. [Online]. Available: http://www.osgi.org/Main/HomePage

[24] Refsnes Data. w3schools.com. [Online]. Available: http://www.w3schools.com/

[25] The Apache Software Foundation. Apache Tomcat. [Online]. Available: http://tomcat.apache.org

[26] The Eclipse Foundation. Jetty. [Online]. Available: http://www.eclipse.org/jetty

[27] Oracle. Package com.sun.net.httpserver. [Online]. Available: http://docs.oracle.com/javase/7/docs/jre/api/net/httpserver/spec/com/sun/net/httpserver/package-summary.html

[28] J. F. Baril, V. Beynon, S. Koop, S. Rohringer, and P. White, "ECE4600 Design Project Monthly Progress Report," December 2013.

Appendix A

Hardware Specification

During the design of this project our hardware designer proposed a revision to the original hardware design specification to better suit the needs of our client [2, 28]. Table A.I lists the revised specifications used for the final design, as agreed upon by the client and our design team.

TABLE A.I: REVISED HARDWARE SPECIFICATIONS FOR PERIPHERAL CORE

	Property	**Value**	**Unit**	**Notes**
Digital I/O Available to User	GPIO	3	# of pins	shared with SPI channel[1]
	PWM	1	# of pins	shared with SPI channel[1]
	PWM Frequency	31-31k	Hz	min-max.[2]
	PWM Resolution	255	steps	max.
Analog I/O Available to User	Analog Input	1	# of pins	min.
	Resolution	10	bit	min.
Battery Life	Sleeping[3]	7	months	typ.
	Normal Use[3,4]	6	months	typ.
Communication	Frequency Band	915 or 2400	MHz	center frequency of band
	Range	40	m	line-of-sight min.
	SPI	1	# of channels	wired,min.

1. GPIO and PWM are shared, and mutually exclusive with SPI channel.
2. for 8-bit phase correct PWM using 16 [MHz] external oscillator.
3. using a 1200 [mAH] battery, no external circuitry connected.
4. based on continuous transmission of 10 byte packet every second waking to poll internal sensors.

Appendix B

Schematics

Figures B.1 - B.5 are schematics for hardware designed for use in the demonstration system. Figures B.6 - B.10 show proposed schematics for various shields that could be used with the ArduIMU. Since these design idea were archived during the project and never tested, the schematics are included for readers interested in pursuing future work on this project. The associated PCB layout files are made available within the same software code repository mentioned in Appendix G.

Figure B.1 shows the schematic for the light strip peripheral. The associated parts list can be found in Table C.III.

Figure B.2 shows the schematic for the indicator box peripheral. The associated parts list can be found in Table C.IV.

Figure B.3 shows the schematic for the control box peripheral. The associated parts list can be found in Table C.V.

Figure B.4 shows the schematic for the compound peripheral. The associated parts list can be found in Table C.VII. Alternatively, the reader can use the information in this schematic to along with the parts list in Table C.VI to assemble appliance control peripherals. The appliance control peripheral eliminates the buzzer, magnetic contact switch, and RJ-45, instead connecting the PowerSwitch Tails directly to the ArduIMU.

Figure B.5 shows the schematic for a zero crossing detection circuit that could be used with the compound ceripheral.

Figure B.6 shows the schematic for the I2C controlled variant of the GPIO expander shield for the ArduIMU v4.

Figure B.7 shows the schematic for the SPI controlled variant of the GPIO expander shield for the ArduIMU v4.

Figure B.8 shows the schematic for a motor shield for the ArduIMU v4 that could be used to control various stepper, DC and servo motors.

Figure B.9 shows the schematic for an adaptable signal conditioning shield for the ArduIMU v4 configured to simply pass-through the input and output signals.

Figure B.10 shows the same signal conditioning shield instead configured to use an external supply, and scale the input and output signals by 1/4 and 4 times respectively.

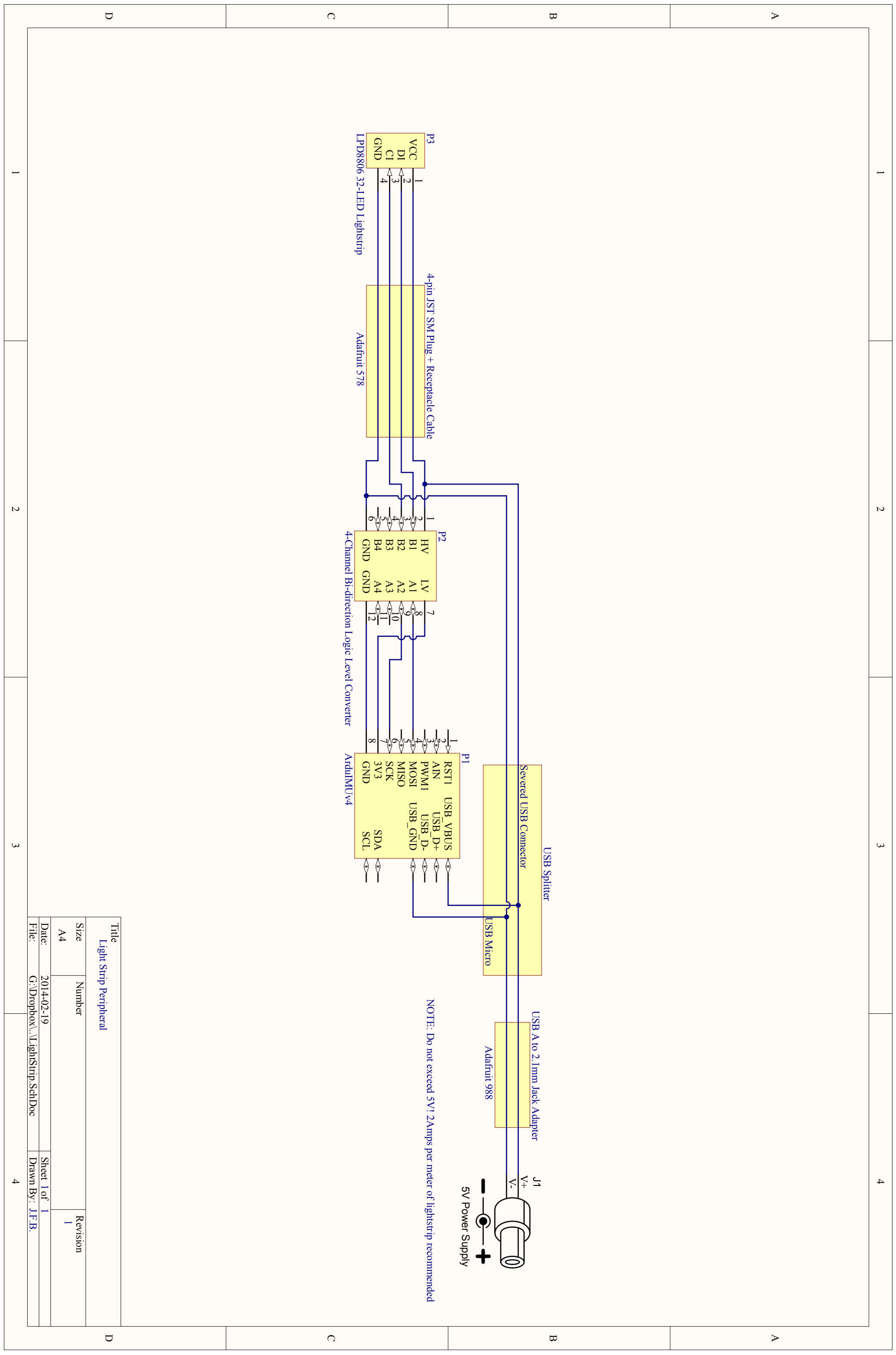


Figure B.1: Light strip peripheral schematic

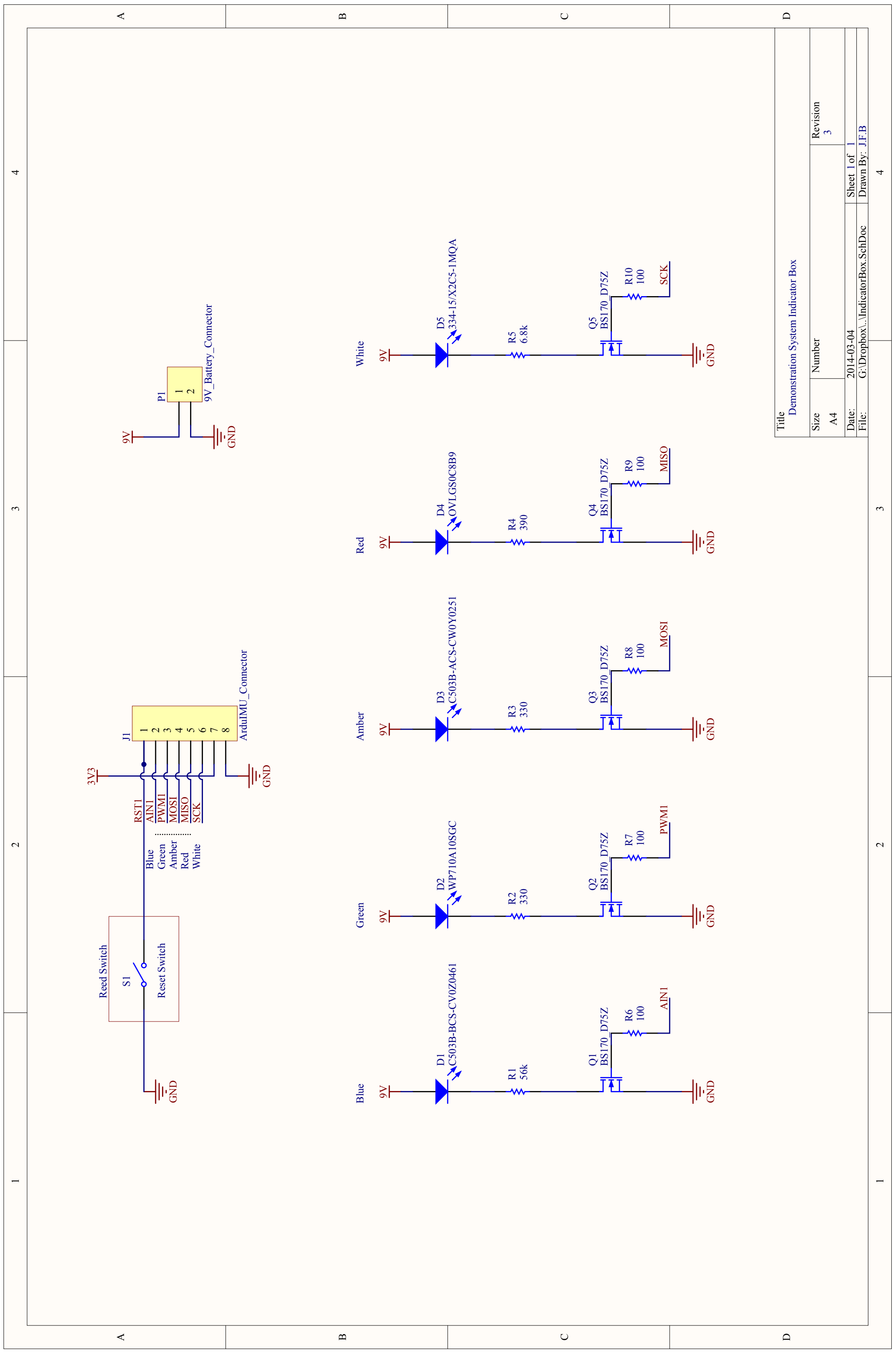


Figure B.2: Indicator box schematic

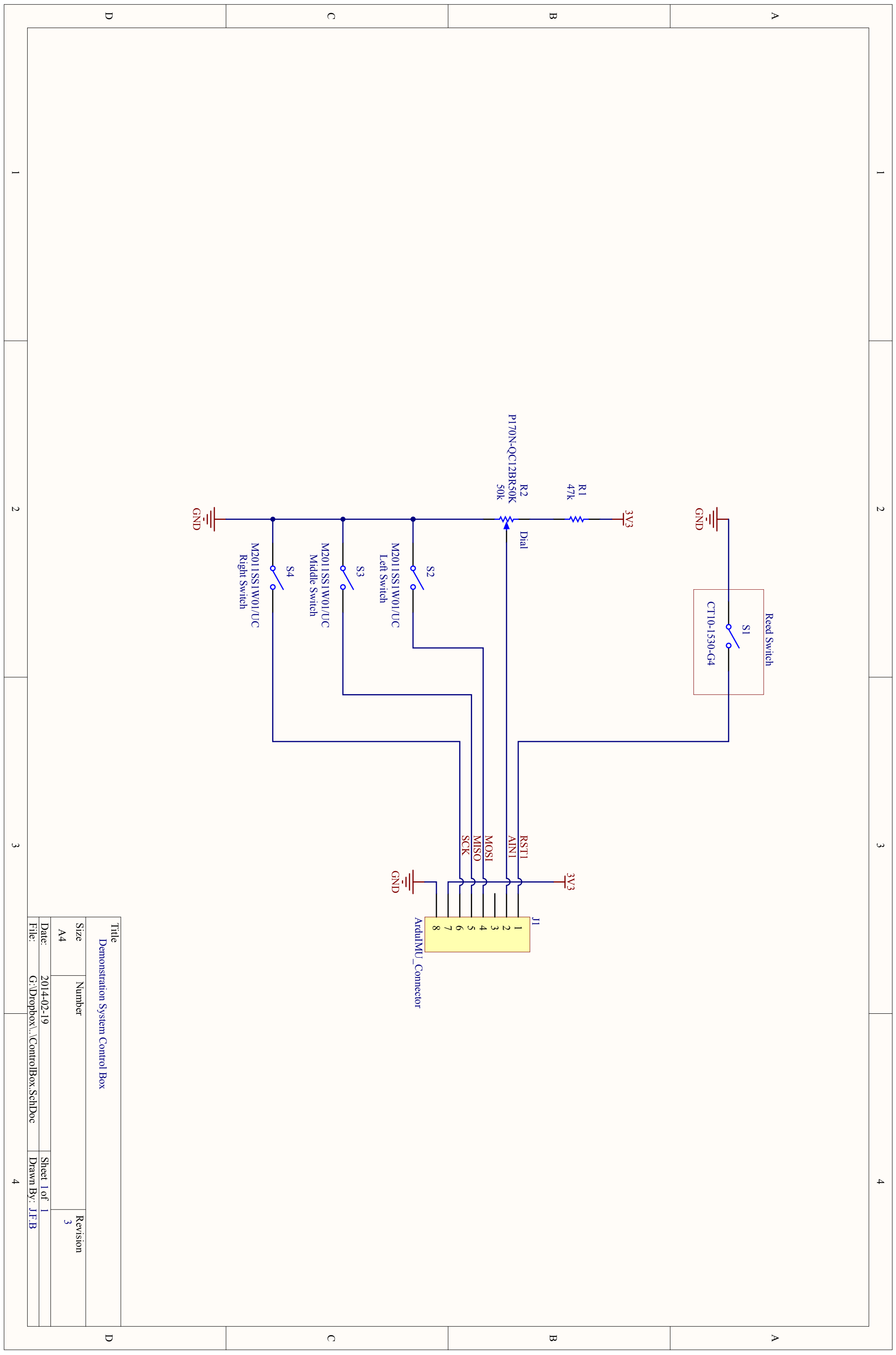


Figure B.3: Control box schematic

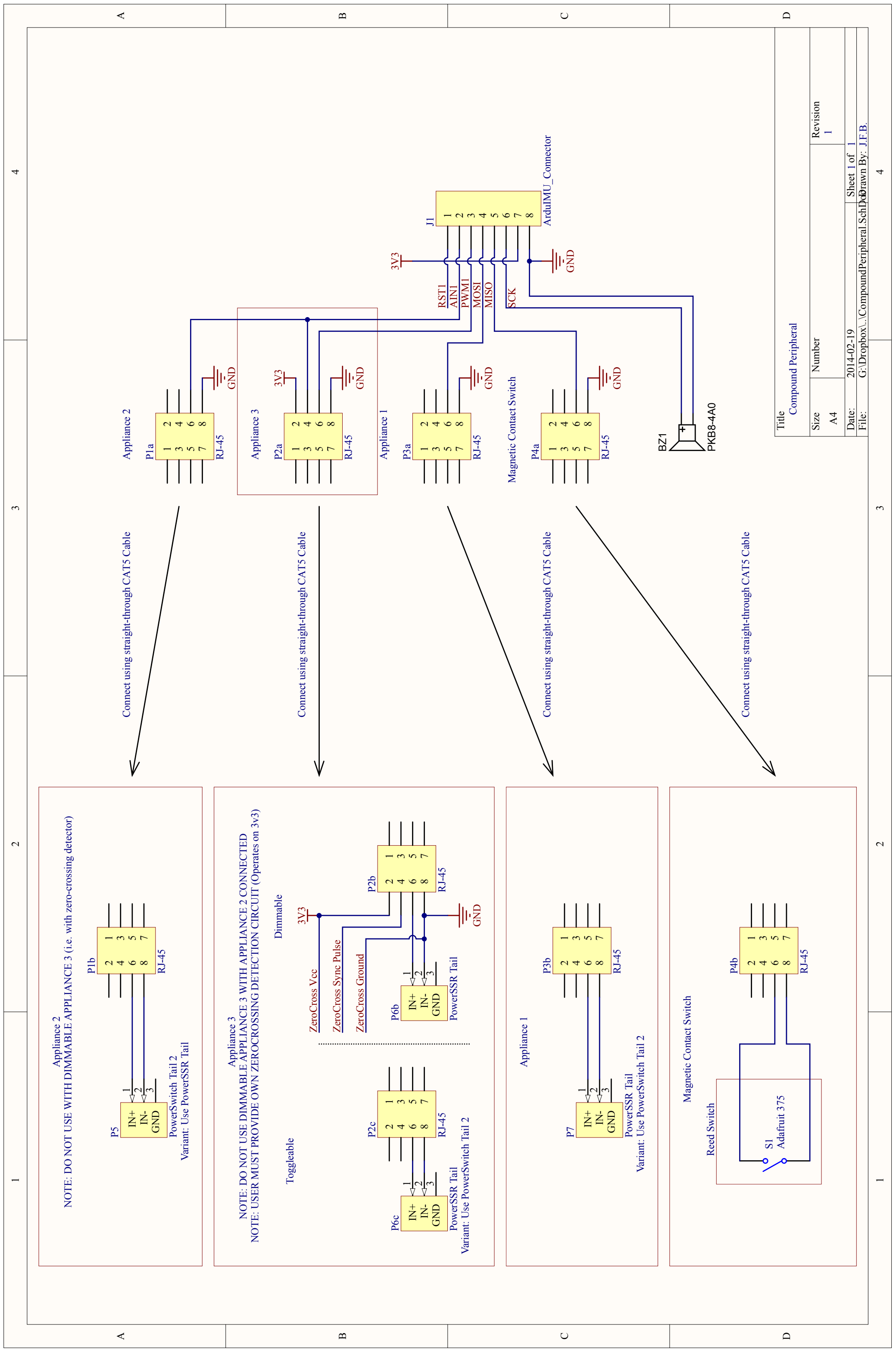


Figure B.4: Compound peripheral schematic

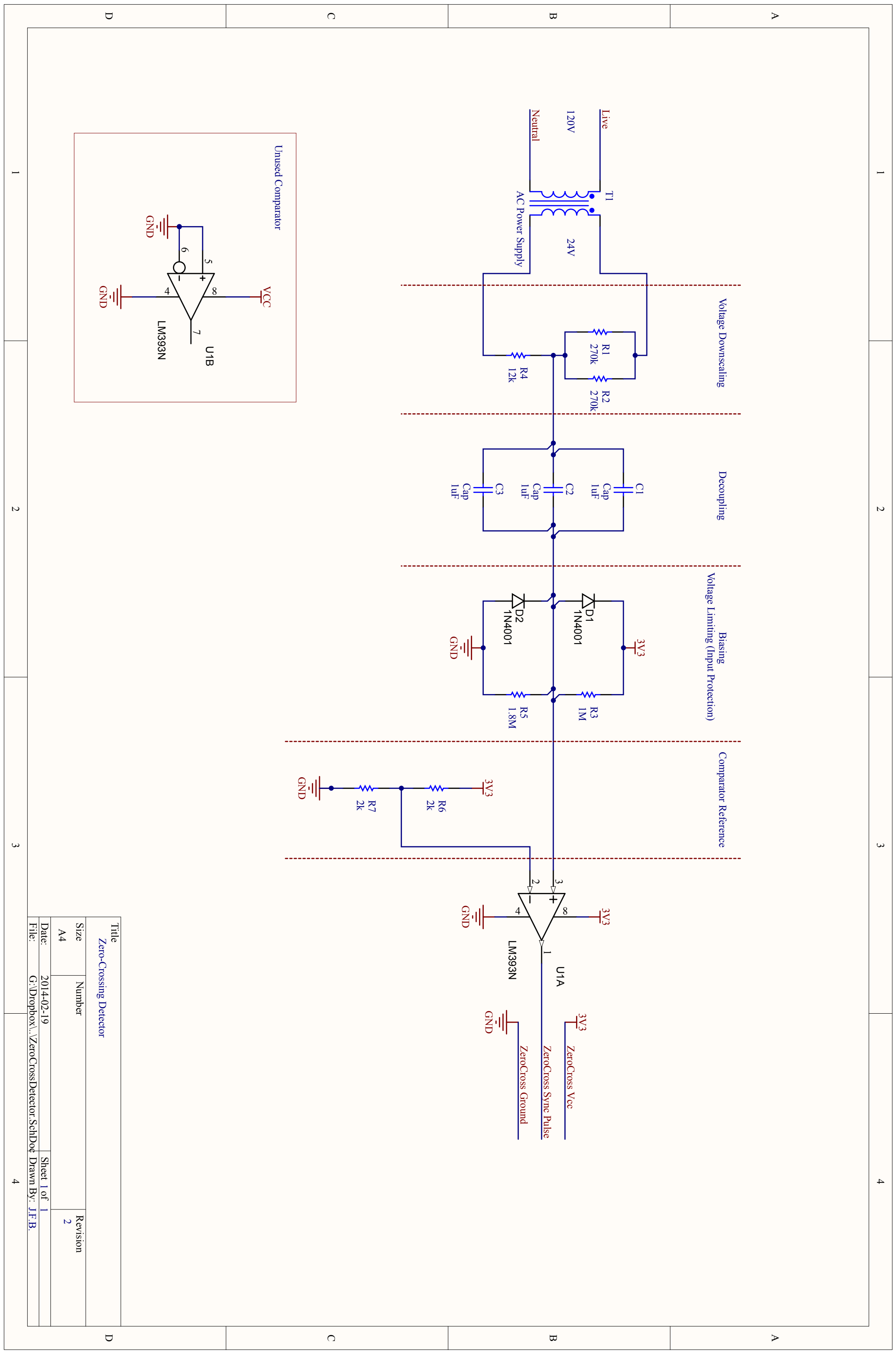


Figure B.5: Zero crossing detection circuit schematic

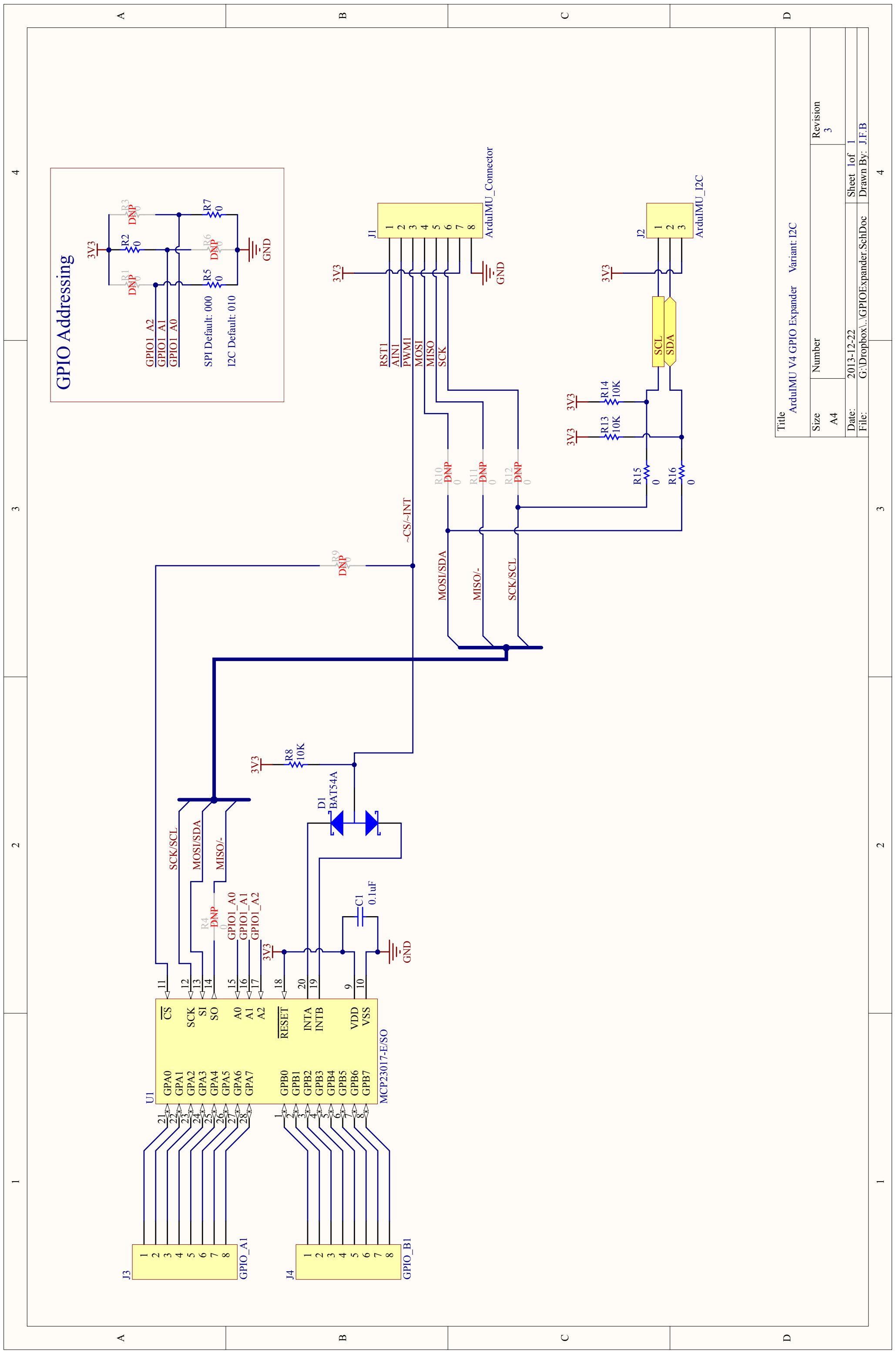


Figure B.6: GPIO expander shield schematic (I2C variant)

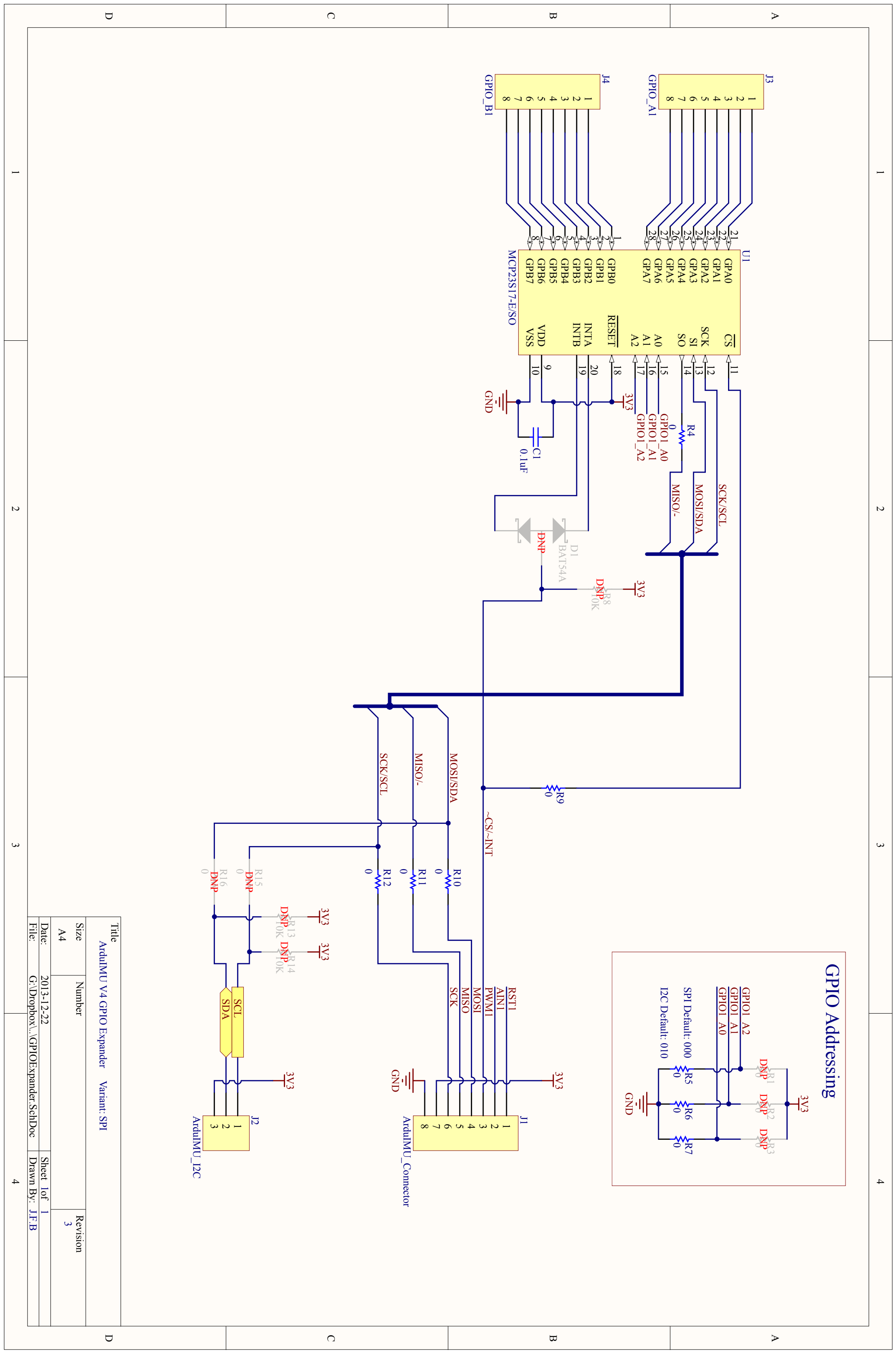


Figure B.7: GPIO expander shield schematic (SPI variant)

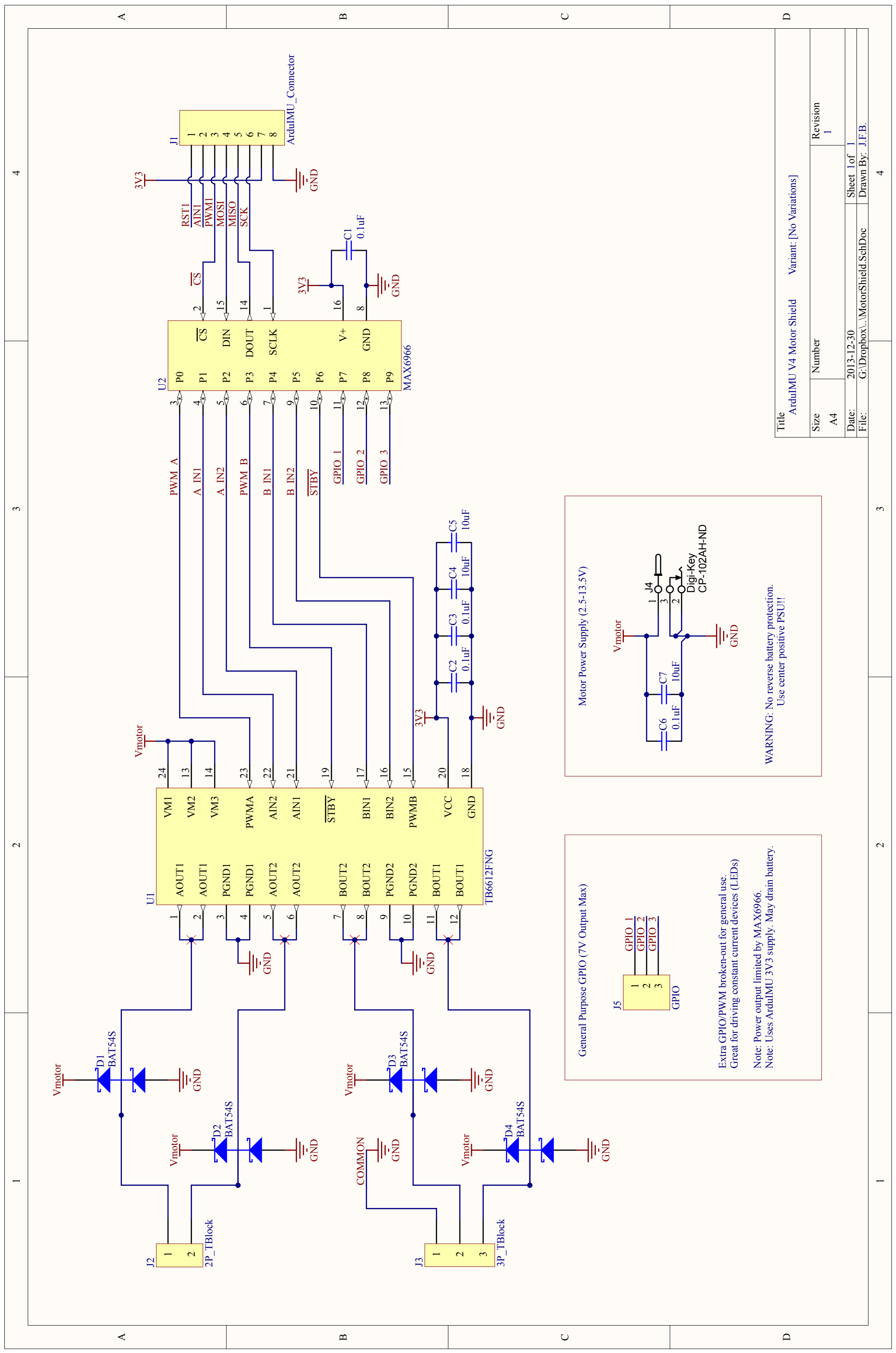


Figure B.8: Motor shield schematic

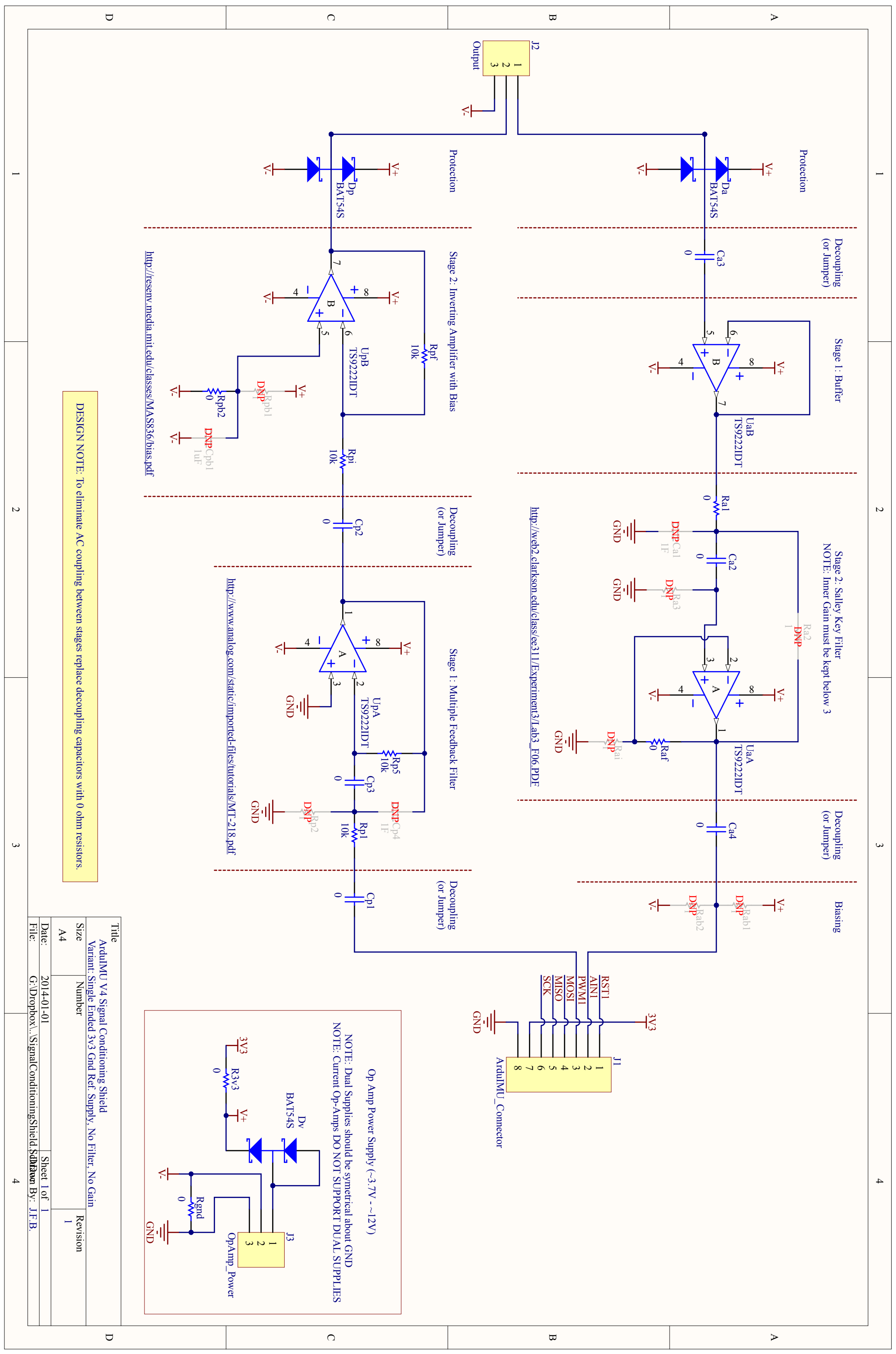
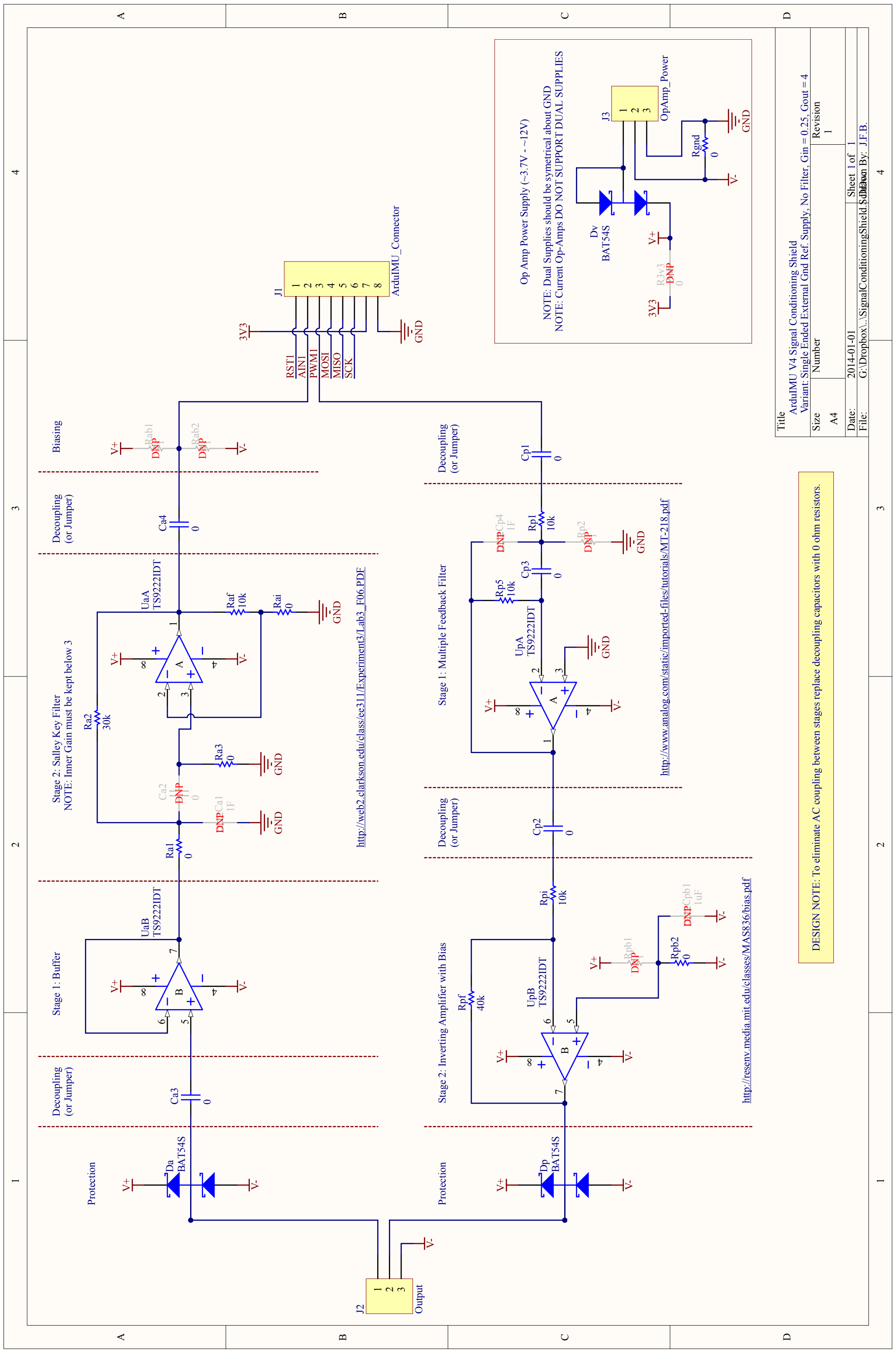

Protection
Decoupling (or Jumper)
Stage 1: Buffer
Stage 2: Salley Key Filter
NOTE: Inner Gain must be kept below 3
Biasing
Stage 2: Inverting Amplifier with Bias
Stage 1: Multiple Feedback Filter
DESIGN NOTE: To eliminate AC coupling between stages replace decoupling capacitors with 0 ohm resistors.
Op Amp Power Supply (~3.7V - ~12V)
NOTE: Dual Supplies should be symetrical about GND
NOTE: Current Op-Amps DO NOT SUPPORT DUAL SUPPLIES
http://web2.clarkson.edu/class/ee311/Experiment3/Lab3_F06.PDF
http://www.analog.com/static/imported-files/tutorials/MT-218.pdf
http://ressrv.media.mit.edu/classes/MAS836/bias.pdf
ArduIMU_Connector
OpAmp_Power
Output
Title
ArduIMU V4 Signal Conditioning Shield
Variant: Single Ended 3v3 Gnd Ref. Supply, No Filter, No Gain
Size A4
Number
Revision 1
Date: 2014-01-01
Sheet 1 of 1
File: G:\Dropbox\..\SignalConditioningShield.SchDoc
Drawn By: J.F.B.

Figure B.9: Signal conditioning shield schematic (single ended 3v3, no filter, no gain)



Protection
Da BAT54S
Decoupling (or Jumper)
Ca3 0
Stage 1: Buffer
UaB TS9222IDT
Ra1 0
Stage 2: Salley Key Filter
NOTE: Inner Gain must be kept below 3
Ra2 30k
Ra3 0
Ca2 DNP
Ca1 DNP 1F
UaA TS9222IDT
Raf 10k
Rai 0
http://web2.clarkson.edu/class/ee311/Experiment3/Lab3_F06.PDF
Decoupling (or Jumper)
Ca4 0
Biasing
Rab1 DNP
Rab2 DNP
J1
RST1 AIN1 PWM1 MOSI MISO SCK
3V3
GND
ArduIMU_Connector
Op Amp Power Supply (~3.7V - ~12V)
NOTE: Dual Supplies should be symetrical about GND
NOTE: Current Op-Amps DO NOT SUPPORT DUAL SUPPLIES
Dv BAT54S
R3v3 DNP 0
J3
OpAmp_Power
Rgnd 0
Protection
Dp BAT54S
Stage 2: Inverting Amplifier with Bias
UpB TS9222IDT
Rpf 40k
Rpb1 DNP
Rpb2 0
Cpb1 DNP 1uF
http://reserv.media.mit.edu/classes/MAS836/bias.pdf
Decoupling (or Jumper)
Rpi 10k
Cp2 0
Stage 1: Multiple Feedback Filter
UpA TS9222IDT
Rp5 10k
Cp3 0
Cp4 DNP 1F
Rp1 10k
Rp2 DNP
http://www.analog.com/static/imported-files/tutorials/MT-218.pdf
Decoupling (or Jumper)
Cp1 0
J2
Output
DESIGN NOTE: To eliminate AC coupling between stages replace decoupling capacitors with 0 ohm resistors.
Title
ArduIMU V4 Signal Conditioning Shield
Variant: Single Ended External Gnd Ref. Supply, No Filter, Gin = 0.25, Gout = 4
Size A4
Number
Revision 1
Date: 2014-01-01
Sheet 1 of 1
File: G:\Dropbox\..\SignalConditioningShield.SchDoc
Drawn By: J.F.B.

Figure B.10: Signal conditioning shield schematic (external supply, no filter, input gain = 1/4, output gain = 4)

Appendix C

Bill of Materials

Tables C.I, C.II, C.III, C.IV, C.V, C.VI, C.VII list the parts and materials that are used to produce the hardware components of this project.

Materials acquired as in-kind donations are listed as having been acquired “In-Kind”. Materials on loan from team members or external parties until the end of the project are listed as acquired “On-Loan” while purchased components are simply listed as “Purchased”.

For the convenience of the reader and where relevant, the part numbers and suggested supplier are listed for many of the parts. Under the columns for supplier and part number, the “-” symbol indicates that the part is generic and can be substituted or replaced by any readily available equivalent. The “N/A” symbol indicates that the supplier or part number was not available at time of writing.

The actual cost of the materials is not included since it may or may not accurately reflect the true cost of replicating or reproducing this design due to large dependence of cost on time, location and quantity ordered.

The “required” column indicates the minimum quantity of an item required to assemble the device associated with the table in question. The “Ordered” column on the other hand indicates how many of that item our group ordered throughout the project. This is included so that between the full budget breakdown in Appendix I and the contents of tables below, the interested reader should have sufficient information to gauge the cost to reproduce this project using their available sources and suppliers.

Table C.I tabulates the items used simply for the development phases of the project. These parts are useful but not required for developing new custom peripheral devices.

TABLE C.I: BILL OF MATERIALS: DEVELOPMENT DEVICES

Item	Required	Ordered	Acquired	Supplier	Part Number
ArduIMU v4 Prototype	0	1	In-Kind	N/A	N/A
ArduIMU Transceiver Proto.	0	1	In-Kind	N/A	N/A
Arduino UNO	0	1	On-Loan	-	-

Table C.II lists the materials required to assemble a base station like the one used in our project.

TABLE C.II: BILL OF MATERIALS: BASE STATION

Item	Required	Ordered	Acquired	Supplier	Part Number
Raspberry Pi Model B	1	2	On-Loan	-	-
USB 2.0 A Male to Micro 5 Pin Male Cable	1	1	In-Kind	www.monoprice.com	5458
ArduIMU v4 Pre-Production Prototype	1	1	In-Kind	[4]	N/A

Table C.III lists the materials required to assemble a light strip peripheral like the one designed and used in the demonstration system.

TABLE C.III: BILL OF MATERIALS: LIGHT STRIP PERIPHERAL

Item	Required	Ordered	Acquired	Supplier	Part Number
ArduIMU v4 Pre-Production Prototype	1	1	Purchased	[4]	N/A
Digital RGB Lightstrip	1	1	Purchased	www.adafruit.com	306
4-Channel Bi-direction Logic Level Converter	1	1	Purchased	www.adafruit.com	757
USB A to 2.1mm Jack Adapter	1	1	Purchased	www.adafruit.com	988
USB Splitter: A Male to 2 A Female, 1 Micro, 1 iPod Connector	1	1	Purchased	Dollarama	-
4-pin JST SM Plug + Receptacle Cable	1	1	Purchased	www.adafruit.com	578
5V Power Supply	1	1	On-Loan	www.adafruit.com	276
Heat Shrink Tubing (1”)	-	-	In-Kind	www.digikey.ca	A100B-4-ND
Heat Shrink Tubing (1/8”)	-	-	In-Kind	www.digikey.ca	A018B-4-ND
Solder	-	-	In-Kind	-	-
Wire (22 AWG)	-	-	In-Kind	-	-

Table C.IV lists the materials required to assemble an indicator box peripheral like the one designed and used in the demonstration system.

Table C.V lists the materials required to assemble a control box peripheral like the one designed and used in the demonstration system.

Table C.VI lists the materials required to assemble an appliance control peripheral. Since we were unable to procure sufficient ArduIMU v4s, the appliance control peripheral was not built for the final demonstration system. Note that this peripheral can support any combination of up to 5 PowerSwitch or PowerSSR Tails as required by the user.

Table C.VII lists the materials required to assemble a compound peripheral like the one

TABLE C.IV: BILL OF MATERIALS: INDICATOR BOX PERIPHERAL

Item	Required	Ordered	Acquired	Supplier	Part Number
ArduIMU v4 Pre-Production Prototype	1	1	In-Kind	[4]	N/A
Red LED	1	5	In-Kind	www.digikey.ca	365-1189-ND
Blue LED	1	3	Purchased	www.digikey.ca	C503B-BCS-CV0Z0461-ND
Green LED	1	5	In-Kind	www.digikey.ca	754-1591-ND
White LED	1	4	Purchased	www.digikey.ca	1080-1011-ND
Yellow LED	1	3	In-Kind	www.digikey.ca	C503B-ACS-CW0Y0251-ND
Transistor	5	20	Purchased	www.digikey.ca	BS170_D75ZCT-ND
100 Ohm Resistor	5	5	In-Kind	-	-
330 Ohm Resistor	2	2	In-Kind	-	-
390 Ohm Resistor	1	1	In-Kind	-	-
6.8k Ohm Resistor	1	1	In-Kind	-	-
56k Ohm Resistor	1	1	In-Kind	-	-
Reed Switch	1	2	Purchased	www.digikey.ca	306-1296-1-ND
LiPo Battery 3.7 - 150mAh	1	2	On-Loan	www.adafruit.com	1317
Enclosure	1	2	Purchased	www.digikey.ca	1553BBK
LED Lens Cap	5	6	Purchased	www.digikey.ca	L30032-ND
Battery Connector	1	3	Purchased	www.digikey.ca	BS12I-ND
Rare-earth Magnet	1	1	On-Loan	Lee Valley Tools	99K36.03
Heat Shrink Tubing (1/4”)	1	1	In-Kind	www.digikey.ca	A014B-4-ND
Heat Shrink Tubing (1/2”)	1	1	In-Kind	www.digikey.ca	A012B-4-ND
Foam Tape	1	1	In-Kind	Canadian Tire	64-2544-2
Double-sided Tape	1	1	In-Kind	-	-
Solder	-	-	In-Kind	-	-
Wire (22 AWG)	-	-	In-Kind	-	-
Stripboard	-	-	In-Kind	-	-

TABLE C.V: BILL OF MATERIALS: CONTROL BOX PERIPHERAL

Item	Required	Ordered	Acquired	Supplier	Part Number
ArduIMU v4 Pre-Production Prototype	1	1	In-Kind	[4]	N/A
Toggle Switch (On-Off SPST)	5	5	Purchased	www.digikey.ca	M2011SS1W01/UC
Potentiometer	1	3	Purchased	www.digikey.ca	987-1396-ND
47k Ohm Resistor	1	1	In-Kind	-	-
Reed Switch	1	1	Purchased	www.digikey.ca	306-1296-1-ND
LiPo Battery 3.7 - 150mAh	1	1	On-Loan	www.adafruit.com	1317
Enclosure	1	1	Purchased	www.digikey.ca	1553BBK
Potentiometer Cap	1	3	Purchased	www.digikey.ca	226-2128-ND
Rare-earth Magnet	1	1	On-Loan	Lee Valley Tools	99K36.03
Heat Shrink Tubing (1/4")	1	1	In-Kind	www.digikey.ca	A014B-4-ND
Heat Shrink Tubing (1/2")	1	1	In-Kind	www.digikey.ca	A012B-4-ND
Foam Tape	1	1	In-Kind	Canadian Tire	64-2544-2
Double-sided Tape	1	1	In-Kind	-	-
Solder	-	-	In-Kind	-	-
Wire (22 AWG)	-	-	In-Kind	-	-

TABLE C.VI: BILL OF MATERIALS: APPLIANCE CONTROL PERIPHERAL

Item	Required	Ordered	Acquired	Supplier	Part Number
ArduIMU v4 Pre-Production Prototype	1	0	Purchased	[4]	N/A
Portable Solar Charger	1	0	On-Loan	ca.cellphoneshop.net	CHR-CSOLAR
USB 2.0 A Male to Micro 5 Pin Male Cable	1	0	In-Kind	www.monoprice.com	5458
Power Switch Tail 2	0 to 5	0	Purchased	www.powerswitchtail.com	PN80135
PowerSSR Tail	0 to 5	0	Purchased	www.powerswitchtail.com	PN80125
Heat Shrink Tubing (1/4")	1	0	In-Kind	www.digikey.ca	A014B-4-ND
Solder	-	-	In-Kind	-	-
Wire (22 AWG)	-	-	In-Kind	-	-

designed and used in the demonstration system. Note that this peripheral can support any combination of up to 3 PowerSwitch or PowerSSR Tails as required by the user.

TABLE C.VII: BILL OF MATERIALS: COMPOUND PERIPHERAL

Item	Required	Ordered	Acquired	Supplier	Part Number
ArduIMU v4 Pre-Production Prototype	1	1	Purchased	[4]	N/A
Portable Solar Charger	1	1	On-Loan	ca.cellphoneshop.net	CHR-CSOLAR
USB 2.0 A Male to Micro 5 Pin Male Cable	1	1	In-Kind	www.monoprice.com	5458
Power Switch Tail 2	0-3	3	Purchased	www.powerswitchtail.com	PN80135
PowerSSR Tail	0-3	3	Purchased	www.powerswitchtail.com	PN80125
Piezo Buzzer	1	1	Purchased	www.abra-electronics.com	PKB8-4A0
Magnetic Contact Switch	1	1	Purchased	www.adafruit.com	375
RJ-45 Connector	8	8	In-Kind	-	-
Heat Shrink Tubing (1/4")	1	1	In-Kind	www.digikey.ca	A014B-4-ND
Heat Shrink Tubing (1/2")	1	1	In-Kind	www.digikey.ca	A012B-4-ND
Hot-Glue	-	-	In-Kind	-	-
Solder	-	-	In-Kind	-	-
Wire (22 AWG)	-	-	In-Kind	-	-
Stripboard	-	-	In-Kind	-	-

Appendix D

Developer Manual

This chapter contains two tutorials aimed at developers:

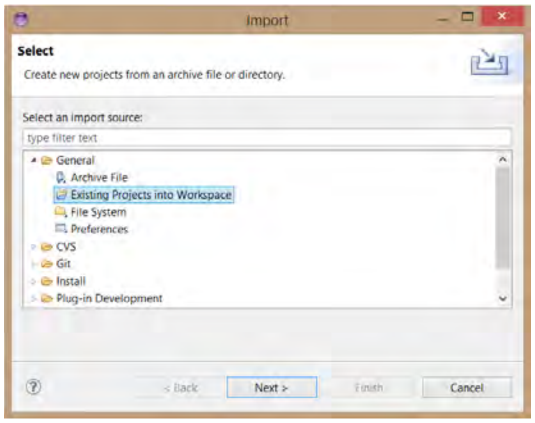
1. The “Peripheral Framework Installation Tutorial” contains instructions to install the tools needed to develop custom peripheral descriptors and custom peripheral code.
2. The “Developer Tutorial” is a brief getting-started guide that walks developers through the creation of a simple custom peripheral.

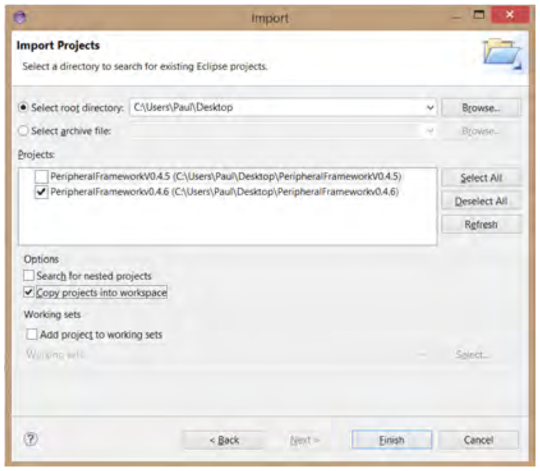
Peripheral Framework Installation Tutorial

We will create a project based off of the peripheral framework, and reference this project in a project for custom peripheral descriptors. Please ensure that you have installed a Java JDK7 or later, and the Eclipse Juno IDE.

INSTALL LIBRARIES AND CONFIGURE WORKSPACE

1. **Get a hold of PeripheralFramework.zip and extract it. Then import the contents into an Eclipse workspace by clicking File > Import:**





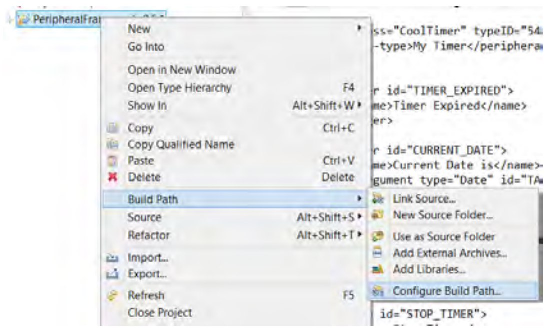
Select root directory: C:\Users\Paul\Desktop [✓]
Select archive file:
Projects:
- ☐ PeripheralFrameworkV0.4.5 (C:\Users\Paul\Desktop\PeripheralFrameworkV0.4.5)
- ✓ PeripheralFrameworkv0.4.6 (C:\Users\Paul\Desktop\PeripheralFrameworkv0.4.6)

Options
- ☐ Search for nested projects
- ✓ Copy projects into workspace

- ☐ Add project to working sets

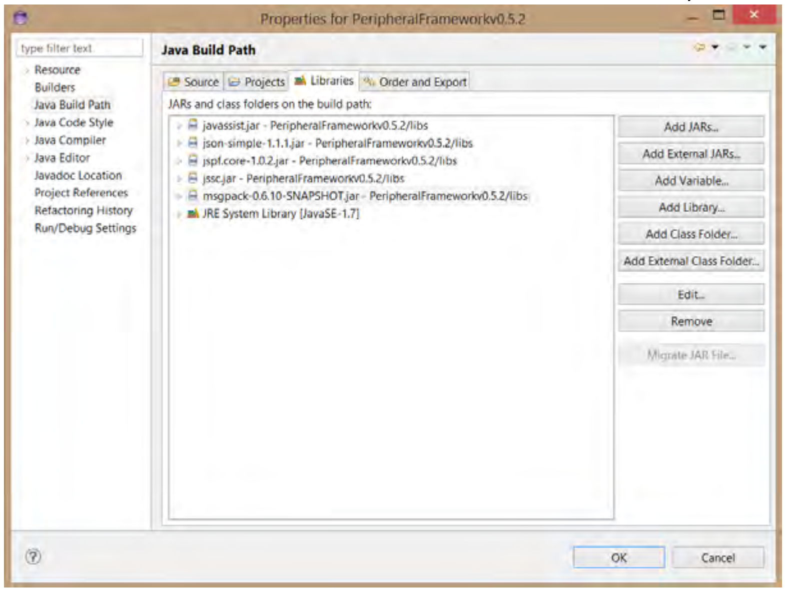
2. **Click 'Finish'**

3. **Now we have to associate some libraries to the java build path. Right-click on the Peripheral Framework project you just imported and select Build Path > Configure Build Path...**



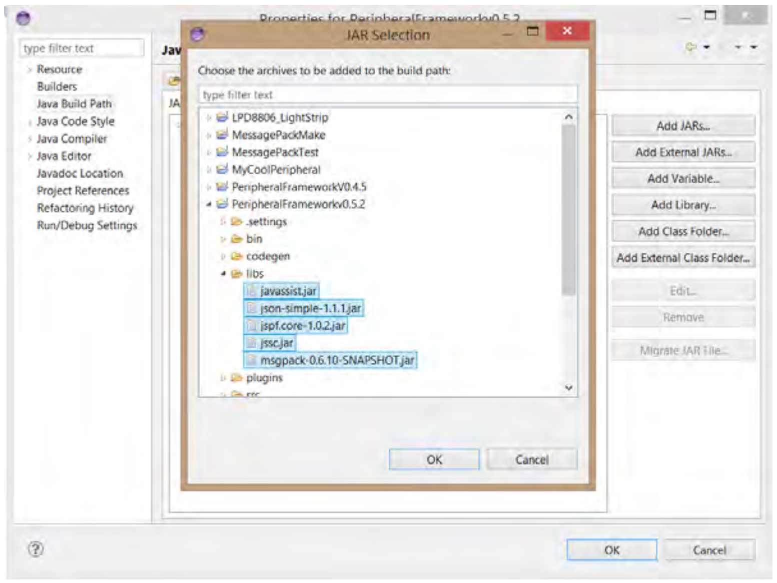
```
ss="CoolTimer" typeID="54
-type>My Timer</periphera

r id="TIMER_EXPIRED">
me>Timer Expired</name>
er>

r id="CURRENT_DATE">
me>Current Date is</name>
gument type="Date" id="TA

id="STOP_TIMER">
```

4. Click the Libraries tab and check to see if the JAR files were added correctly as shown below:



javassist.jar - PeripheralFrameworkv0.5.2/libs
json-simple-1.1.1.jar - PeripheralFrameworkv0.5.2/libs
jspf.core-1.0.2.jar - PeripheralFrameworkv0.5.2/libs
jssc.jar - PeripheralFrameworkv0.5.2/libs
msgpack-0.6.10-SNAPSHOT.jar - PeripheralFrameworkv0.5.2/libs
JRE System Library [JavaSE-1.7]

If not, click 'Add JARs...' and navigate to the 'libs' folder and select the 'javassist', 'json-simple', 'jspf', 'jssc', ' and 'msgpack' jar files.



Choose the archives to be added to the build path:

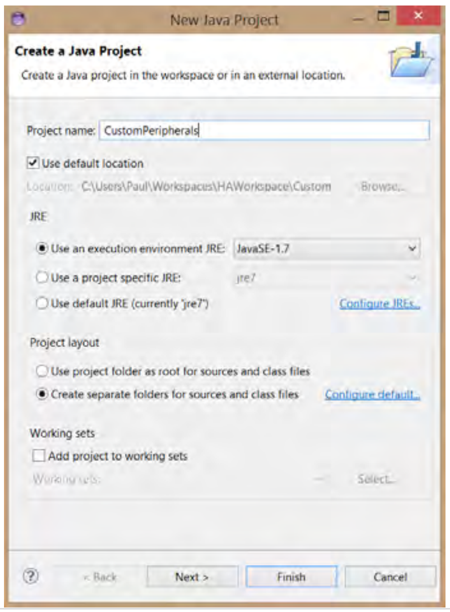
type filter text

- LPD8806_LightStrip
- MessagePackMake
- MessagePackTest
- MyCoolPeripheral
- PeripheralFrameworkV0.4.5
- PeripheralFrameworkv0.5.2
  - .settings
  - bin
  - codegen
  - libs
    - javassist.jar
    - json-simple-1.1.1.jar
    - jspf.core-1.0.2.jar
    - jssc.jar
    - msgpack-0.6.10-SNAPSHOT.jar
  - plugins

Now click 'OK' and click 'OK' again to close the Properties window.

CREATE HOST PROJECT

Now we are ready to make a new project to develop custom peripheral descriptors. You can define the parameters of a peripheral descriptor using an XML file, and the boiler-plate code will be auto-generated for you. Use this project for multiple descriptors, so you don't have to go through these setup steps again.



Project name: CustomPeripherals

[✓] Use default location

Location: C:\Users\Paul\Workspaces\HAWorkspace\Custom

JRE

[●] Use an execution environment JRE: JavaSE-1.7

[ ] Use a project specific JRE: jre7

[ ] Use default JRE (currently 'jre7')

Project layout

[ ] Use project folder as root for sources and class files

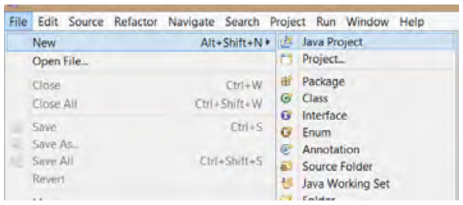
[●] Create separate folders for sources and class files

Working sets

[ ] Add project to working sets

Working sets:

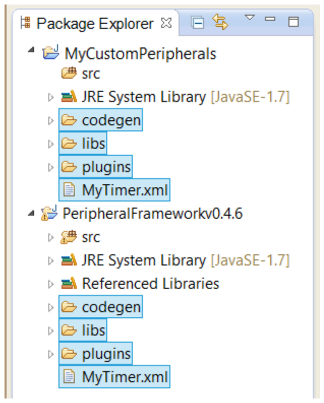
1. **Create a new project:**



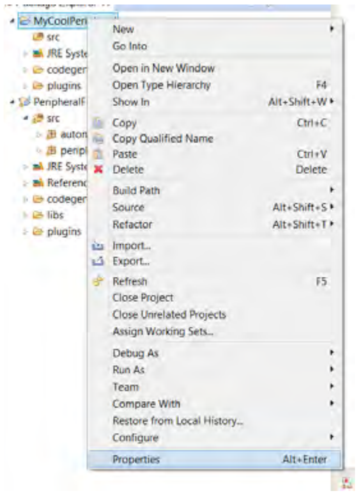
3. **Click 'Finish'.**

4. **Now we will set up auto-generation. We will set up the ANT builder to parse the peripheral descriptor XML and generate**

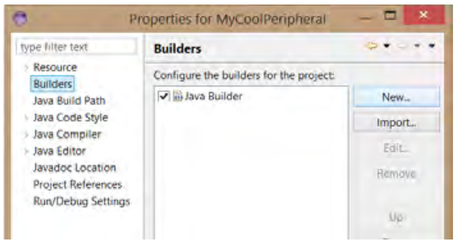
Java source code, then use the Eclipse Java compiler to compile it. First, we need to copy some files into the new project. Look at the Peripheral Framework project that we made earlier and copy over the 'codegen,' 'libs,' and 'plugins' folders, as well as 'MyTimer.xml' to the root of your new project:



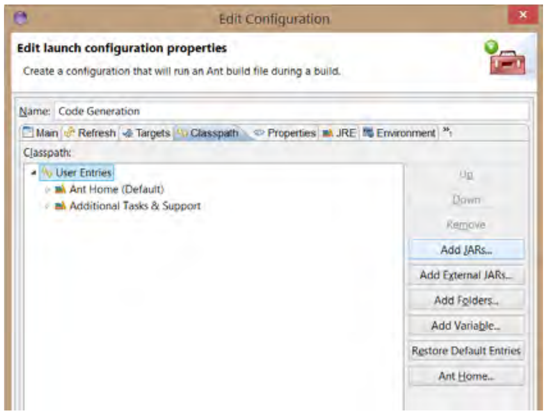
5. Right-click the project and select 'Properties.'



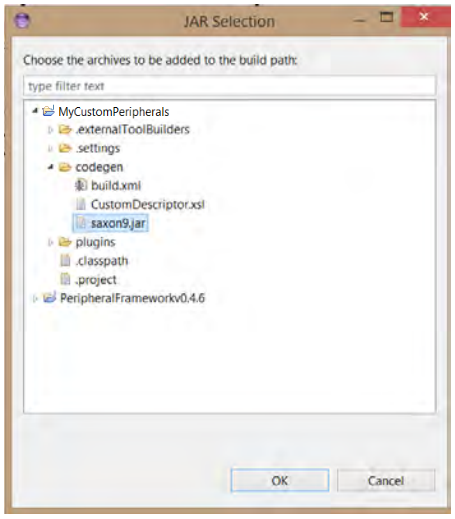
6. Now go to 'Builders' and click the 'New' button.



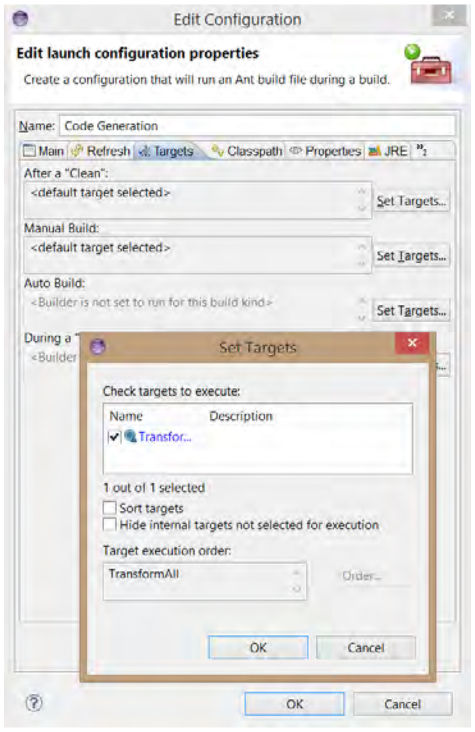
7. Click on 'Ant Builder' and click 'OK.'
8. Give the builder a name (I use 'Code Generation'), then click the 'Classpath' tab. Click on 'User Entries' and then click the 'Add JARs' button.



9. Browse through your project to 'codegen' and select 'saxon9.jar,' and press OK. Be sure not to select the 'saxon9.jar' file located in the Peripheral Framework project.



10. Click the 'Targets' tab and for 'Auto Build' and 'During a "Clean"' click the 'Set Targets…' button and click OK.



Edit Configuration

Edit launch configuration properties

Create a configuration that will run an Ant build file during a build.

Name: Code Generation

After a "Clean": <default target selected> Set Targets...

Manual Build: <default target selected> Set Targets...

Auto Build: <Builder is not set to run for this build kind> Set Targets...

During a "Clean": <Builder

Set Targets

Check targets to execute:

Name	Description
✓ Transfor...	

1 out of 1 selected

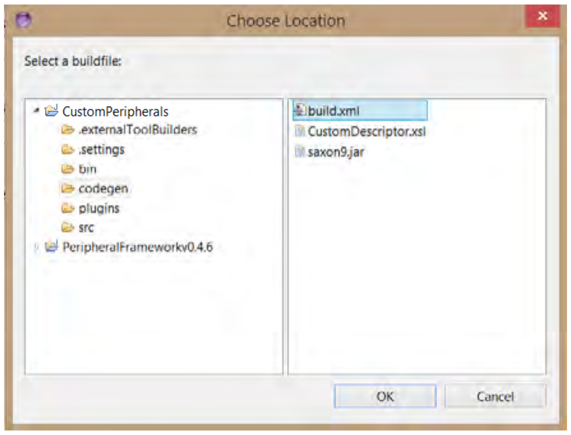
☐ Sort targets

☐ Hide internal targets not selected for execution

Target execution order:

TransformAll

11. Press 'Apply' and go back to the 'Main' tab. We will now fill out the details for Ant code-generation. For 'Build File', click 'Browse Workspace' and choose codegen > build.xml. Then click 'OK.'



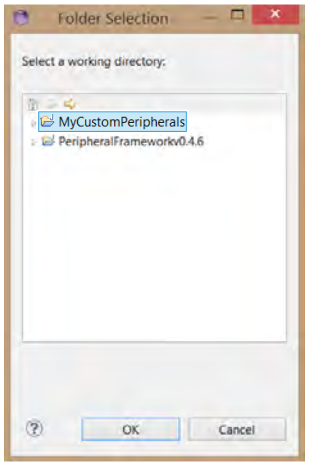
Choose Location

Select a buildfile:

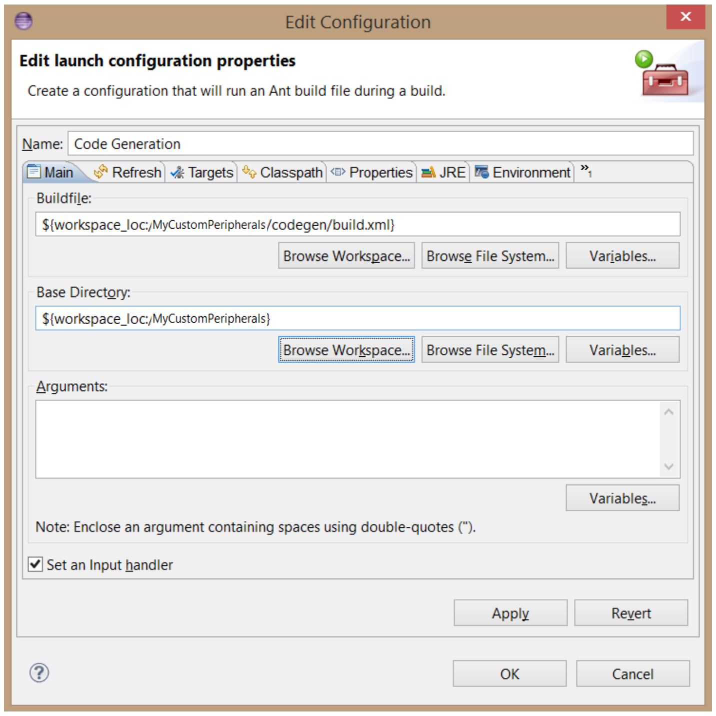
```
CustomPeripherals
    .externalToolBuilders
    .settings
    bin
    codegen
    plugins
    src
PeripheralFrameworkv0.4.6
```

```
build.xml
CustomDescriptor.xsl
saxon9.jar
```

12. For 'Base Directory', click 'Browse Workspace' and select your workspace root. Then press 'OK.'



13. Now press 'Apply', then 'OK.'



Edit launch configuration properties

Create a configuration that will run an Ant build file during a build.

Name: Code Generation

Buildfile: ${workspace_loc:/MyCustomPeripherals/codegen/build.xml}

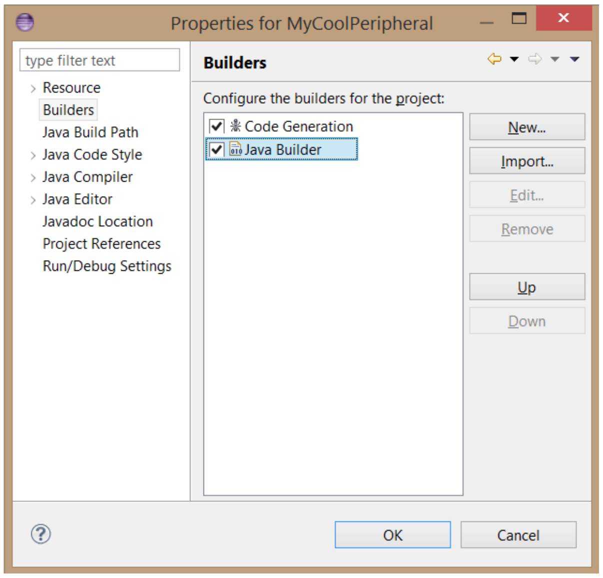
Base Directory: ${workspace_loc:/MyCustomPeripherals}

Arguments:

Note: Enclose an argument containing spaces using double-quotes (").

☑ Set an Input handler

14. Finally, select the Java Builder and click 'Down' to put the Ant Builder at the top of the stack.

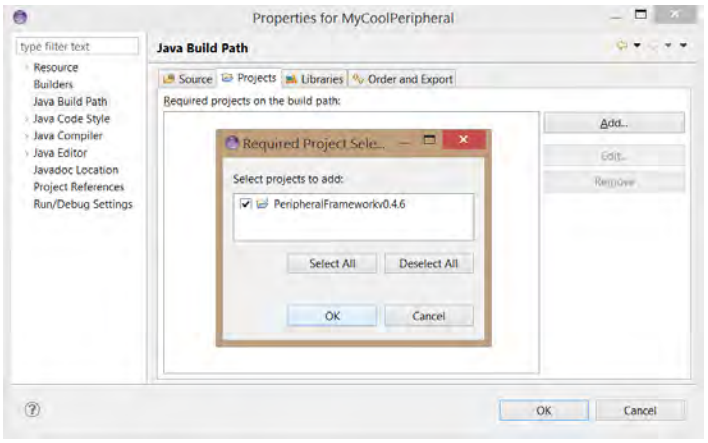

Properties for MyCoolPeripheral

Builders

Configure the builders for the project:

- [✓] Code Generation
- [✓] Java Builder

15. Now we need to add the Peripheral Framework project to the buildpath. Click 'Java Build Path' and click on the 'Projects' tab. Then click 'Add...' and select the Peripheral Framework project. Then click 'OK.'


Properties for MyCoolPeripheral

Java Build Path

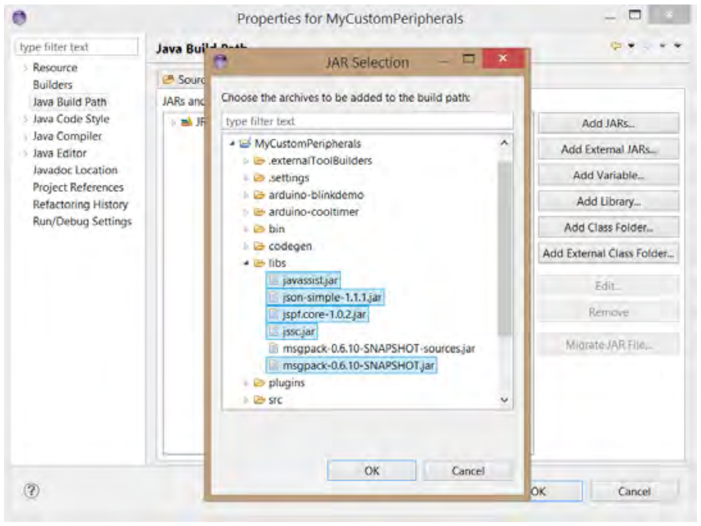
Required projects on the build path:

Required Project Sele...

Select projects to add:

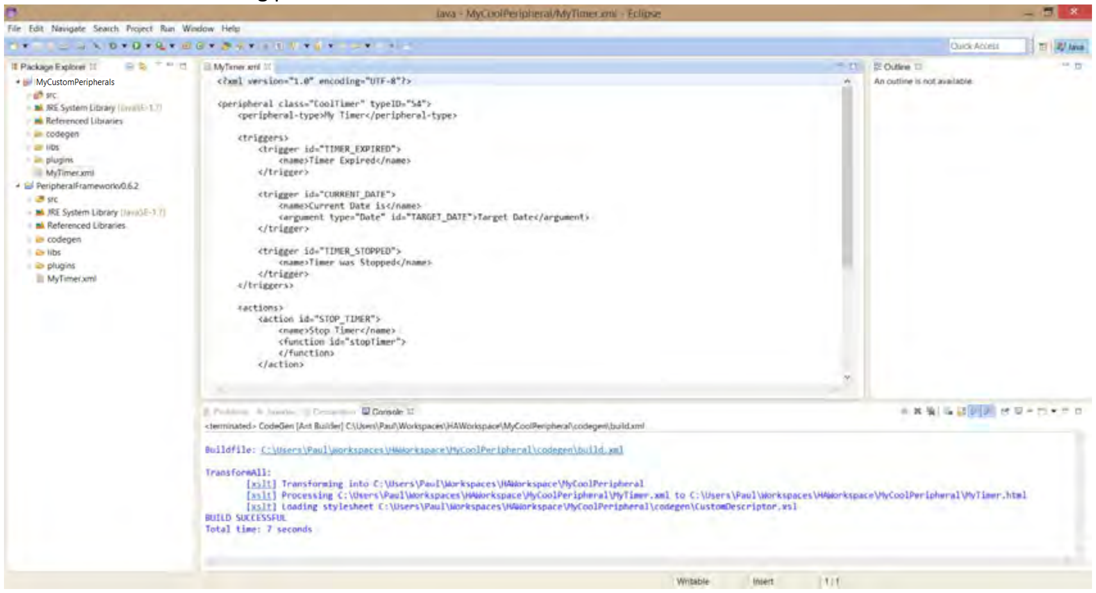
- [✓] PeripheralFrameworkv0.4.6

16. Now we will add the required libraries, like in step 4. Click the 'Libraries' tab and select 'Add Jars...' Then select the required JAR files:



17. Then click 'OK' and you will be done.

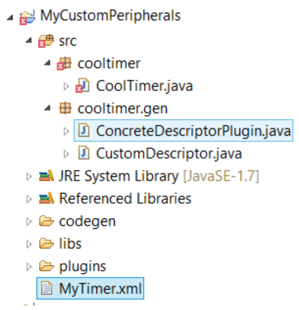
Now let's test it all out. Get (or create) a peripheral descriptor XML file (such as MyTimer.xml) and put it in the root of your project. You should see some blue text in the terminal informing you that an XSLT transform is taking place.


Java - MyCoolPeripheral/MyTimer.xml - Eclipse
File Edit Navigate Search Project Run Window Help
Package Explorer
MyCustomPeripherals
src
JRE System Library [JavaSE-1.7]
Referenced Libraries
codegen
libs
plugins
MyTimer.xml
PeripheralFrameworkv0.6.2
src
JRE System Library [JavaSE-1.7]
Referenced Libraries
codegen
libs
plugins
MyTimer.xml
MyTimer.xml
Outline
An outline is not available.
Console
Writable
Insert
1 : 1

```
<?xml version="1.0" encoding="UTF-8"?>

<peripheral class="CoolTimer" typeID="54">
    <peripheral-type>My Timer</peripheral-type>

    <triggers>
        <trigger id="TIMER_EXPIRED">
            <name>Timer Expired</name>
        </trigger>

        <trigger id="CURRENT_DATE">
            <name>Current Date is</name>
            <argument type="Date" id="TARGET_DATE">Target Date</argument>
        </trigger>

        <trigger id="TIMER_STOPPED">
            <name>Timer was Stopped</name>
        </trigger>
    </triggers>

    <actions>
        <action id="STOP_TIMER">
            <name>Stop Timer</name>
            <function id="stopTimer">
            </function>
        </action>
```

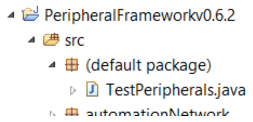
```
<terminated> CodeGen [Ant Builder] C:\Users\Paul\Workspaces\HAWorkspace\MyCoolPeripheral\codegen\build.xml

Buildfile: C:\Users\Paul\Workspaces\HAWorkspace\MyCoolPeripheral\codegen\build.xml

TransformAll:
      [xslt] Transforming into C:\Users\Paul\Workspaces\HAWorkspace\MyCoolPeripheral
      [xslt] Processing C:\Users\Paul\Workspaces\HAWorkspace\MyCoolPeripheral\MyTimer.xml to C:\Users\Paul\Workspaces\HAWorkspace\MyCoolPeripheral\MyTimer.html
      [xslt] Loading stylesheet C:\Users\Paul\Workspaces\HAWorkspace\MyCoolPeripheral\codegen\CustomDescriptor.xsl
BUILD SUCCESSFUL
Total time: 7 seconds
```

Right-click on the root of your project and select 'Refresh' to refresh the package explorer, and you should see your generated code.



In this example, note that 'CoolTimer.java' has some compile errors. These are because it must be made to implement some abstract callback methods. Implement them with your custom 'timer' code.

When you are ready to test, copy the file TestPeripherals.java from the peripheral framework directory. More details will be given about this file in the Developer Tutorial.



Peripheral Framework Developer Tutorial

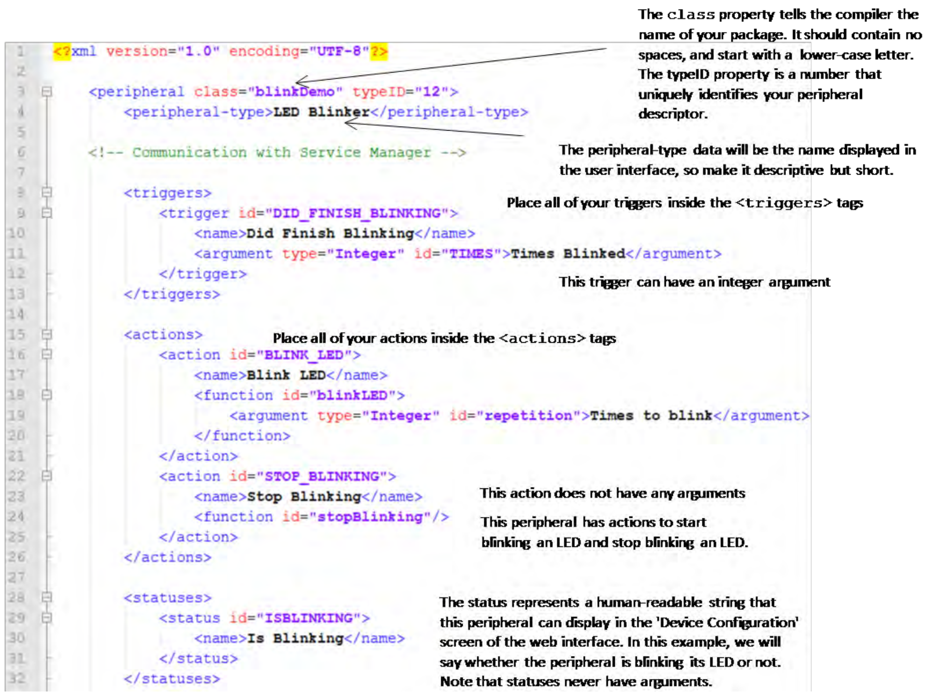
This tutorial will walk you through the development of a simple peripheral. It is capable of blinking the peripheral core's integrated LED. Before you begin, make sure that you have done the following:

- **Install the peripheral framework by following the instructions in the Peripheral Framework Installation Tutorial.**
- **Educate yourself about the basics of callback programming**.
- Get a template for the peripheral core code, newPeripheral.zip

We will begin by generating the boilerplate code for the peripheral descriptor, then we will write the peripheral descriptor methods. Finally, we will write the functions for the peripheral core, and test that everything works correctly.

CODE GENERATION

1. **Create an XML file to describe your new peripheral. This tutorial will use the following XML (blinkDemo.xml):**



```
<?xml version="1.0" encoding="UTF-8"?>

<peripheral class="blinkDemo" typeID="12">
    <peripheral-type>LED Blinker</peripheral-type>

    <!-- Communication with Service Manager -->

    <triggers>
        <trigger id="DID_FINISH_BLINKING">
            <name>Did Finish Blinking</name>
            <argument type="Integer" id="TIMES">Times Blinked</argument>
        </trigger>
    </triggers>

    <actions>
        <action id="BLINK_LED">
            <name>Blink LED</name>
            <function id="blinkLED">
                <argument type="Integer" id="repetition">Times to blink</argument>
            </function>
        </action>
        <action id="STOP_BLINKING">
            <name>Stop Blinking</name>
            <function id="stopBlinking"/>
        </action>
    </actions>

    <statuses>
        <status id="ISBLINKING">
            <name>Is Blinking</name>
        </status>
    </statuses>
```

The `class` property tells the compiler the name of your package. It should contain no spaces, and start with a lower-case letter. The typeID property is a number that uniquely identifies your peripheral descriptor.

The peripheral-type data will be the name displayed in the user interface, so make it descriptive but short.

Place all of your triggers inside the `<triggers>` tags

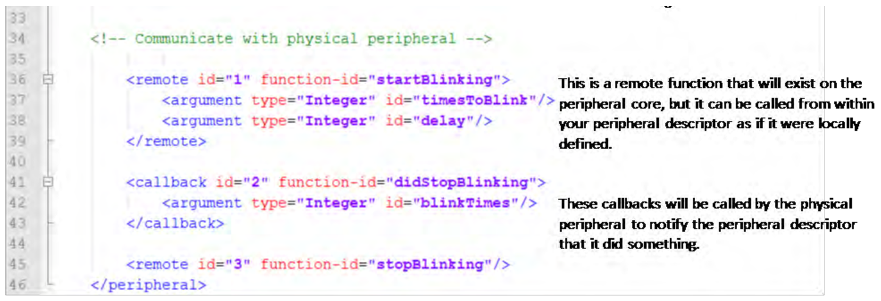
This trigger can have an integer argument

Place all of your actions inside the `<actions>` tags

This action does not have any arguments

This peripheral has actions to start blinking an LED and stop blinking an LED.

The status represents a human-readable string that this peripheral can display in the 'Device Configuration' screen of the web interface. In this example, we will say whether the peripheral is blinking its LED or not. Note that statuses never have arguments.



```
<!-- Communicate with physical peripheral -->

    <remote id="1" function-id="startBlinking">
        <argument type="Integer" id="timesToBlink"/>
        <argument type="Integer" id="delay"/>
    </remote>

    <callback id="2" function-id="didStopBlinking">
        <argument type="Integer" id="blinkTimes"/>
    </callback>

    <remote id="3" function-id="stopBlinking"/>
</peripheral>
```

This is a remote function that will exist on the peripheral core, but it can be called from within your peripheral descriptor as if it were locally defined.

These callbacks will be called by the physical peripheral to notify the peripheral descriptor that it did something.

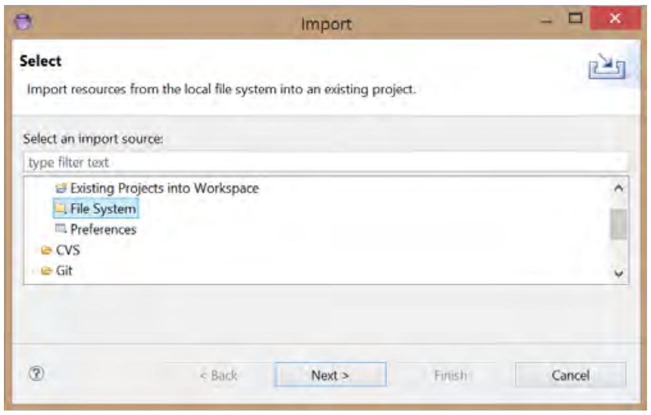
Some peripherals will have sensors on them. Please note that due to limitations in the networking software, it is considered better practice to define a callback that returns all sensor values at once, rather than separate callbacks for each sensor.

Some things to note:

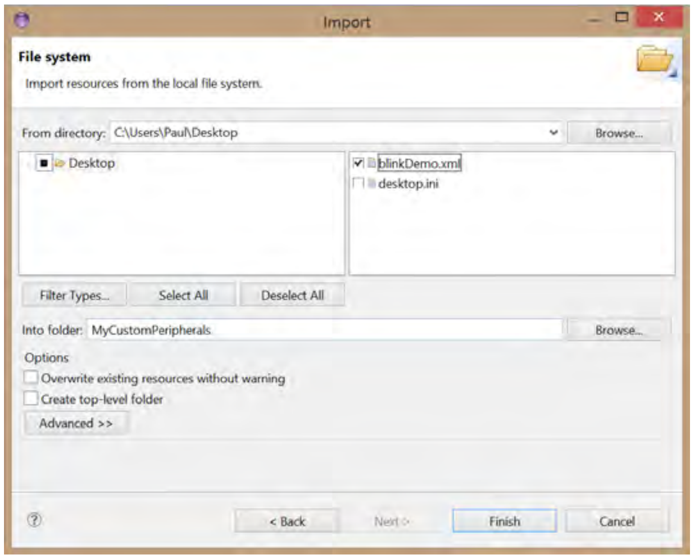
- When coming up with names, please use only letters and numbers.
- The following types are allowed for triggers and actions:
    - Integer
    - Floating-Point
    - String
    - Boolean
- The following arguments are allowed for remote functions and callbacks:
    - Byte
    - Short
    - Integer
    - Long
    - Float
    - String

2. Save the XML file, and import it into the peripheral descriptor project you created in the Installation Tutorial.
    a. Click on your project, then click File > Import… then select File System

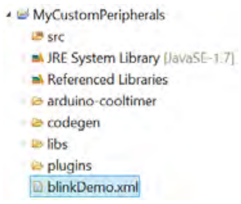
2 | Developer Tutorial



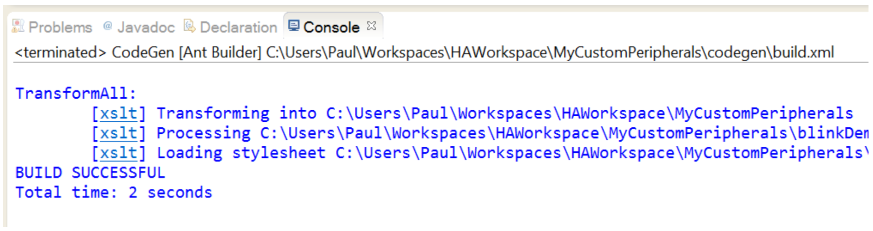
b. Click Next, then click 'Browse' and locate the XML file from step 1.



c. Select your XML file and click 'Finish.'

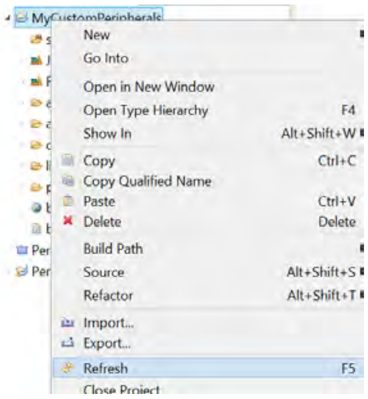


You should see the ANT builder generating code for your peripheral descriptor:

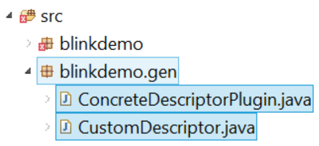


```
<terminated> CodeGen [Ant Builder] C:\Users\Paul\Workspaces\HAWorkspace\MyCustomPeripherals\codegen\build.xml

TransformAll:
        [xslt] Transforming into C:\Users\Paul\Workspaces\HAWorkspace\MyCustomPeripherals
        [xslt] Processing C:\Users\Paul\Workspaces\HAWorkspace\MyCustomPeripherals\blinkDem
        [xslt] Loading stylesheet C:\Users\Paul\Workspaces\HAWorkspace\MyCustomPeripherals\
BUILD SUCCESSFUL
Total time: 2 seconds
```

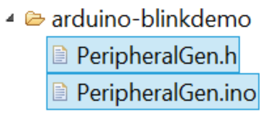
3. Refresh the workspace by pressing F5, or right click the project root folder and select 'Refresh.'



4. Observe the generated code in the 'gen' package. This will become your peripheral descriptor.

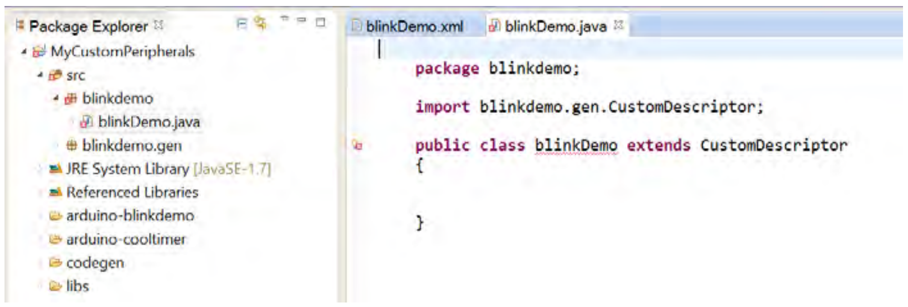


Also observe the generated code in the arduino-lightstripdemo folder. This code will go on your physical peripheral a little later.



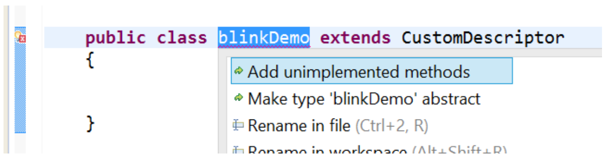
Notice that there are compile errors in the blinkdemo package. This is where you will be writing your custom code for the peripheral descriptor.

5. Open this package and double-click on blinkDemo.java to open it. Notice that there are compile errors.



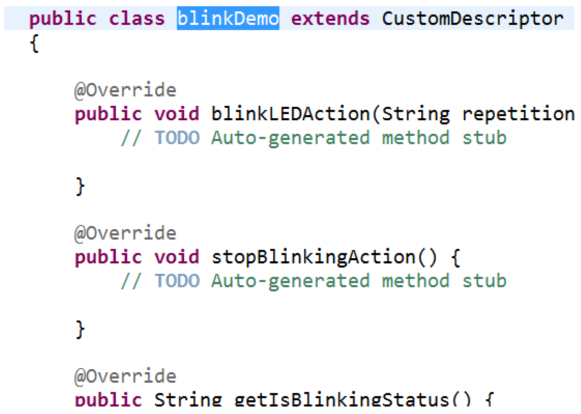
```
package blinkdemo;

import blinkdemo.gen.CustomDescriptor;

public class blinkDemo extends CustomDescriptor
{

}
```

6. Click on the error indicator ( ) and Eclipse will generate the necessary method signatures.



```
public class blinkDemo extends CustomDescriptor
{

}
```

Add unimplemented methods
Make type 'blinkDemo' abstract
Rename in file (Ctrl+2, R)



```
public class blinkDemo extends CustomDescriptor
{

    @Override
    public void blinkLEDAction(String repetition
        // TODO Auto-generated method stub

    }

    @Override
    public void stopBlinkingAction() {
        // TODO Auto-generated method stub

    }

    @Override
    public String getIsBlinkingStatus() {
```

PERIPHERAL DESCRIPTOR CODE

Now that we have generated the boilerplate code, we will fill in the methods. Note that you can use the remote functions that you declared in your XML file to call remote functions on the peripheral core.

1. **We will start with the action method `blinkLEDAction()`. This method will get called when the peripheral descriptor receives the "Blink LED" action from the service engine. Note that the parameters `blinkLEDAction()` accepts are all `Strings`. This is because the service engine treats all triggers, actions, and their respective arguments**

as Strings. We have to parse the String arguments before we can pass them to remote functions on the physical peripheral.

```
public void blinkLEDAction(String repetition)
{
int reps = Integer.parseInt(repetition); // parse the argument
startBlinkingRemote(reps, 2000);
isBlinking = true;
}
```

2. The next two methods we will write are validateTrigger() and validateAction(). These methods get called when the user tries to add a new trigger to the service engine and when the service engine tries to execute an action, respectively.
    a. validateAction() accepts as arguments a number that represents the action ID and an array of strings that represent the parameters that are being passed with the action. validateAction() should return true if the combination of action and parameters is valid and will work when the function corresponding to actionID gets called.

```
protected boolean validateAction(int actionID, String... arguments) {
        boolean isValid = false;

        switch (actionID) {
        case BLINK_LED: // ensure arguments parse correctly
        if (arguments.length == 1)
        {
                try
                {
                int timesToBlink = Integer.parseInt(arguments[0]);
                isValid = (timesToBlink > 0);
                } catch (Exception e)
                {
                isValid = false;
                }
        }
        break;

        case STOP_BLINKING: // this action expects no arguments
        isValid = (arguments.length == 0);
        break;

        default:
        break;
        }

        return isValid;
}
```

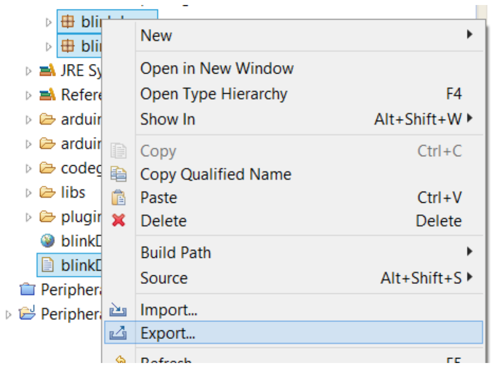
    b. validateTrigger() is very much like validateAction(), in that it should return true if the parameters are valid for the trigger corresponding to triggerID. In addition, it should also register the arguments so that the descriptor can indicate a trigger with them later on.

```
protected boolean validateTrigger(int triggerID, String... arguments)
{
boolean isValid = false;

switch (triggerID) {
case DID_FINISH_BLINKING:
        if (arguments.length == 1) {
                int newBlinkTime = Integer.parseInt(arguments[0]);
                blinkTimeTriggers.add(newBlinkTime);
                     // remember that newBlinkTime is a valid trigger
                isValid = true;
        }
        break;

default:
        break;
}

return isValid;
}
```

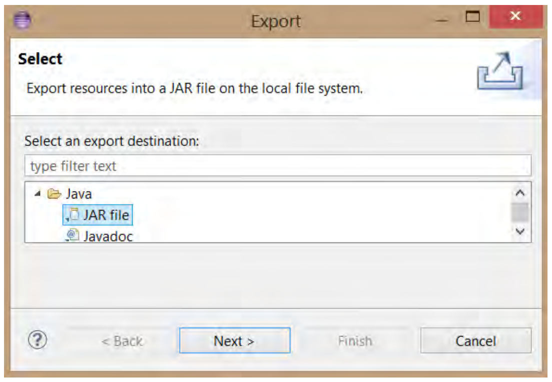
3. Now let's make a callback: look for `didStopBlinkingCallBack()`. This callback function will be called by the remote peripheral once it finishes blinking the LED. When we receive this callback, we will forward a trigger to the service engine. Note that we check to see which blink times have been registered as being of interest.

```
public void didStopBlinkingCallBack(Integer blinkTimes)
{
        if (blinkTimeTriggers.contains(blinkTimes))
        {
                didTrigger(DIDSTOPBLINKING, blinkTimes.toString());
        }
}
```

4. Write the rest of the functions. They can be found at the end of this document.
5. When you are finished, export your peripheral descriptor to a JAR file:
    a. Select your descriptor's package and its XML file. Right-click on one of these files and select 'Export…'



    b. Under Java select 'JAR File' and then click 'Next'


Export

Select

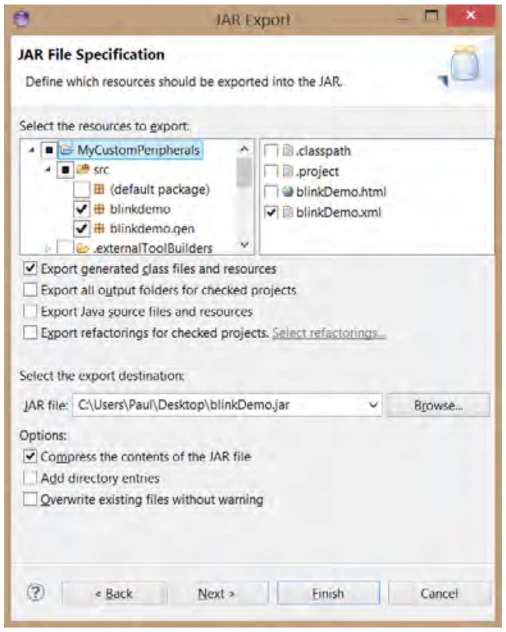
Export resources into a JAR file on the local file system.

Select an export destination:

type filter text

Java
- JAR file
- Javadoc

c. Choose an export destination, and ensure that all of your files are selected. Then click 'Finish.'


JAR Export

JAR File Specification

Define which resources should be exported into the JAR.

Select the resources to export:

- ■ MyCustomPeripherals
  - ■ src
    - ☐ (default package)
    - ☑ blinkdemo
    - ☑ blinkdemo.gen
  - ☐ .externalToolBuilders

- ☐ .classpath
- ☐ .project
- ☐ blinkDemo.html
- ☑ blinkDemo.xml

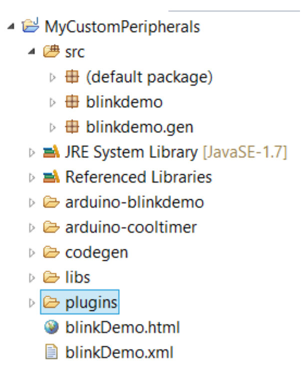
☑ Export generated class files and resources
☐ Export all output folders for checked projects
☐ Export Java source files and resources
☐ Export refactorings for checked projects. Select refactorings...

Select the export destination:

JAR file: C:\Users\Paul\Desktop\blinkDemo.jar

Options:
☑ Compress the contents of the JAR file
☐ Add directory entries
☐ Overwrite existing files without warning

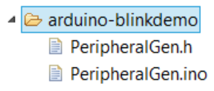
d. Finally, copy the new JAR file into the 'plugins' folder in your project.



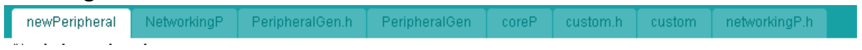
PERIPHERAL CORE CODE

Now we are ready to begin writing code for the peripheral core.

1. **Extract newPeripheral.zip, and copy the generated Arduino files to it. In our example, these files are located in the arduino-blinkdemo directory:**



2. **Open the program by double-clicking newPeripheral.ino. This project should contain the following tabs:**



We will be editing the code under custom.h and custom.ino. We will start with custom.h.

custom.h is used to define special operating modes for your peripheral and to declare pin mappings. Depending on which operating modes you specify, you will have to supply code for some extra functions in custom.ino.

custom.h

`TIME_SLEEP`	This should be defined if the peripheral is going to be used for a monitoring application in which it will be required to perform a task periodically, such as checking the state of a switch, without being told to do so each time by the base station. The `monitor()` function will periodically be called. (See below.)
`AUTO_JOIN`	This option disables the requirement for the user to press the push-button to connect to the network. We do not generally recommended using this option for security reasons, but this mode may be necessary in peripherals where users do not have access to the push button. This mode does not require any extra functions.
`TASK`	This option should be used if you want to delay. This option should be used along with `resumeTask()` (See below).

custom.h is also a good place to declare pin mappings.

Since the light blinker will be delaying in between blinking the light, we need to use the `TASK` option. To expedite the testing process, we also want to avoid the cumbersome push-button authentication so we will also use the `AUTO_JOIN` mode. This means we need to comment out the definition for `TIME_SLEEP`.

```
//#define TIME_SLEEP    // for monitoring peripherals
#define AUTO_JOIN       // used for peripherals that join automatically without user input
                                // not recomended for security reasons
#define TASK            // used for peripherals that require extened use of the cpu
                                // will allow the cpu to be interrupted to check for RF messages
```

We will also define a pin position for the LED. Like the Arduino, there is a built-in LED attached to pin 13, so we can use the following definition:

```
/*

Available I/O ports

label on board : definition in software

Ain     : A1
PWM     : 10
MOSI    : MOSI
MISO    : MISO
SCK     : SCK

Use to define meaningful names.

Example:

#define LAMP 10
#define TEMP A1
#define SWITCH MOSI

*/
#define LED_PIN 13

#endif
```

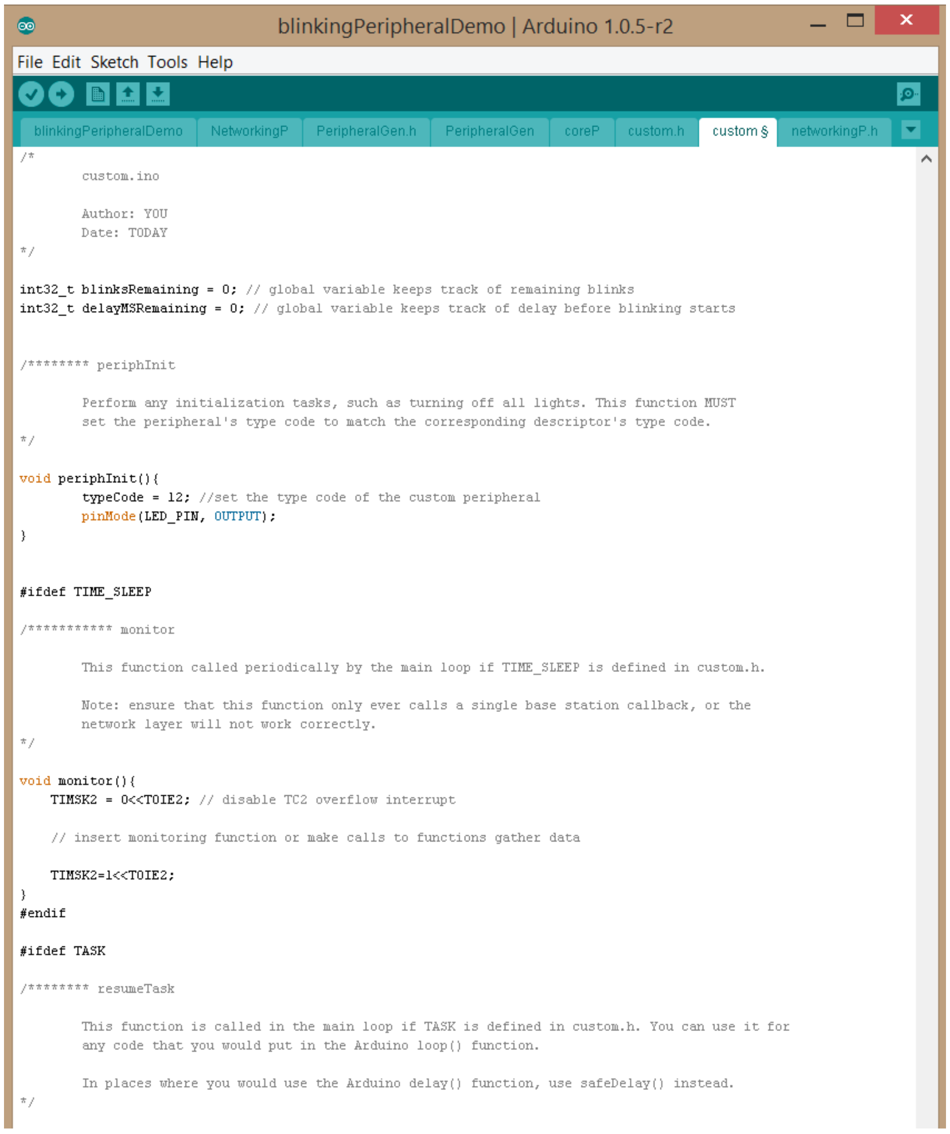
custom.ino

`periphInit()`	This function gets called once when the peripheral starts up, so treat it just like the Arduino `setup()` function.
`resumeTask()`	This function allows long-running operations to be broken up into smaller pieces, and gives the peripheral a chance to abort a long-running operation. `resumeTask()` is called from the main loop, so treat it like the Arduino `loop()` function. Long running tasks are encouraged to save their state and `return` frequently. The more frequently these tasks `return`, the more quickly a peripheral will be able to terminate an unneeded long-running task. In places where you would use the Arduino `delay()` function, use `safeDelay()` instead, or your peripheral will not work correctly.
`monitor()`	This function is periodically called, allowing a peripheral to check its inputs and call a callback on the base station if appropriate. Due to limitations in the network, please ensure that only a single base station callback is called during the execution of this function.

Function stubs are already provided for all three special functions, so it is up to you to know which ones to implement. Since we used the `TASK` mode, we need to provide an implementation for `resumeTask()`. We also need to provide implementations for the remote functions declared in the XML file. To get signatures for these functions, take a look in PeripheralGen.h:

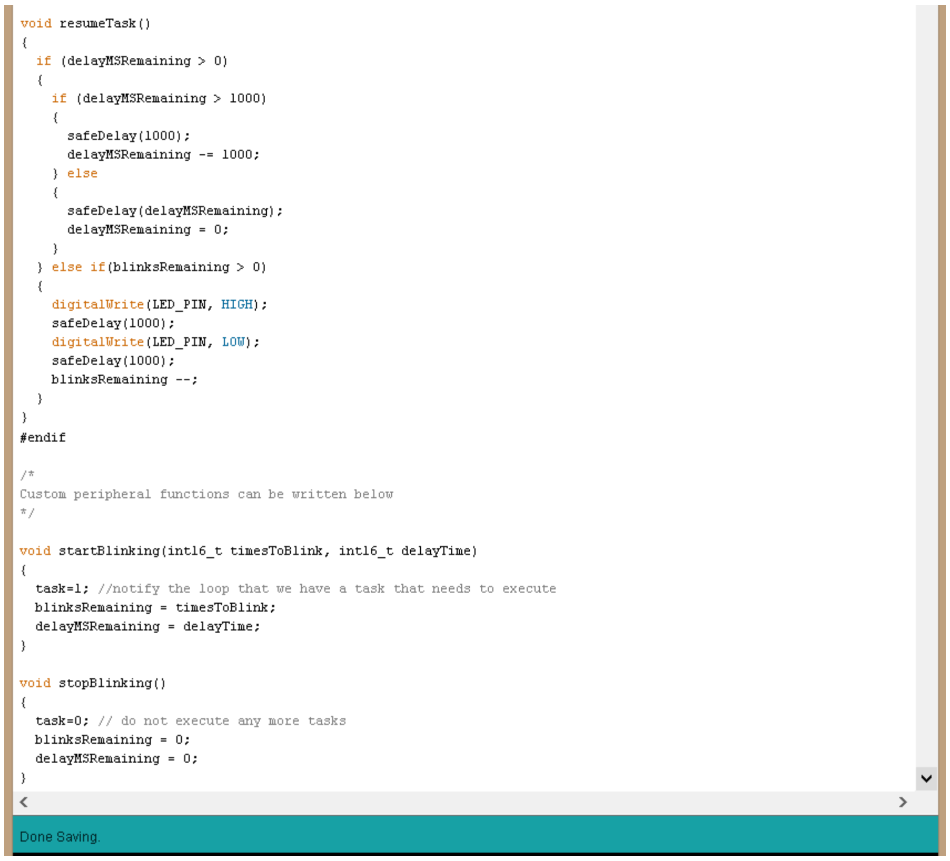
```
/** FUNCTIONS AVAILABLE ON PERIPHERAL **/

extern void startBlinking(int32_t timesToBlink, int32_t delay);

extern void stopBlinking();
```

This means that the functions `stopBlinking()` and `startBlinking()` must be defined in custom.ino. We will use startBlinking to set a global variable that contains the number of blink iterations remaining, and to set another global variable that contains the number of milliseconds to delay. In the loop function `resumeTask()`, we will check the values of these variables and decide whether to blink the light or wait before blinking.



```
/*
        custom.ino

        Author: YOU
        Date: TODAY
*/

int32_t blinksRemaining = 0; // global variable keeps track of remaining blinks
int32_t delayMSRemaining = 0; // global variable keeps track of delay before blinking starts


/******** periphInit

        Perform any initialization tasks, such as turning off all lights. This function MUST
        set the peripheral's type code to match the corresponding descriptor's type code.
*/

void periphInit(){
        typeCode = 12; //set the type code of the custom peripheral
        pinMode(LED_PIN, OUTPUT);
}


#ifdef TIME_SLEEP

/*********** monitor

        This function called periodically by the main loop if TIME_SLEEP is defined in custom.h.

        Note: ensure that this function only ever calls a single base station callback, or the
        network layer will not work correctly.
*/

void monitor(){
    TIMSK2 = 0<<TOIE2; // disable TC2 overflow interrupt

    // insert monitoring function or make calls to functions gather data

    TIMSK2=1<<TOIE2;
}
#endif

#ifdef TASK

/******** resumeTask

        This function is called in the main loop if TASK is defined in custom.h. You can use it for
        any code that you would put in the Arduino loop() function.

        In places where you would use the Arduino delay() function, use safeDelay() instead.
*/
```

12 | Developer Tutorial



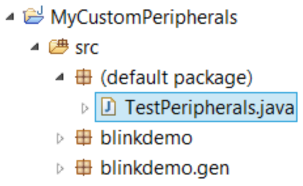
```
void resumeTask()
{
  if (delayMSRemaining > 0)
  {
    if (delayMSRemaining > 1000)
    {
      safeDelay(1000);
      delayMSRemaining -= 1000;
    } else
    {
      safeDelay(delayMSRemaining);
      delayMSRemaining = 0;
    }
  } else if(blinksRemaining > 0)
  {
    digitalWrite(LED_PIN, HIGH);
    safeDelay(1000);
    digitalWrite(LED_PIN, LOW);
    safeDelay(1000);
    blinksRemaining --;
  }
}
#endif

/*
Custom peripheral functions can be written below
*/

void startBlinking(int16_t timesToBlink, int16_t delayTime)
{
  task=1; //notify the loop that we have a task that needs to execute
  blinksRemaining = timesToBlink;
  delayMSRemaining = delayTime;
}

void stopBlinking()
{
  task=0; // do not execute any more tasks
  blinksRemaining = 0;
  delayMSRemaining = 0;
}
```

Done Saving.

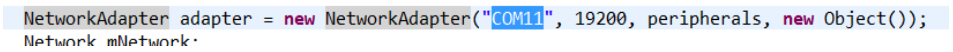
TESTING

Attach a peripheral core with the network master code installed on it to your PC's USB port and check the Serial port ID that it is using.

1. **Ensure that TestPeripherals.java is copied to your workspace from the PeripheralFramework project:**



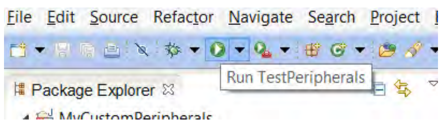
2. **Open TestPeripherals.java and modify the Network Adapter initialization line to initialize to the correct COM port:**



```
NetworkAdapter adapter = new NetworkAdapter("COM11", 19200, peripherals, new Object());
Network mNetwork;
```

3. **Run TestPeripherals.java by clicking the Eclipse 'Run' button:**

13 | Developer Tutorial



4. First, make sure that everything on the base station side is working correctly. In the prompt, check to see if you add your device type

   ```
   add;12
   ```

   where ‘12’ is the typeID that you chose back in your XML file.

   Then type ‘print’ and you should see something like this:

```
Main Network Adapter
add;12
Adding plugins from file:/C:/Users/Paul/Workspaces/HAWorkspace/MyCustomPeripherals/plugins/
got descriptor: 12
print
Peripherals:
LED Blinker0 (0)
```

   This indicates that your descriptor can be detected and loaded correctly.

5. Now type ‘listen’ and wait a moment for your custom peripheral to be detected. Once it is detected, type ‘print’ again to see the list of peripherals that have been added.

```
listen
Network Sending: 0x4e 0xff 0x0
Received 7 bytes
Preamble: 80 1 4 (P)
Data: 0 0 0 12 (    )
Network Sending: 0x53 0x0 0x0
got descriptor: 12
print
Peripherals:
LED Blinker0 (0)
LED Blinker1 (1)
```

6. Test that the blinker can receive an action by typing:

   ```
   action;1;Blink LED;4
   ```

   This tells the tester to perform the action ‘Blink LED’ with argument 4 on peripheral 1 (which should be the one that was added wirelessly). Observe the light on your peripheral blink four times.

Congratulations! You have developed your first peripheral!

Appendix E

Custom Peripheral File Templates

The following section contains the templates that will be provided to developers for the creation of custom peripherals. Explanations of each file can be found in the comments given in the code.

```
/********************************************************************************
custom.h

This file is to be used by peripheral developers for definitions required
by their custom peripheral.

Author: Valerie Beynon
Date: February 2014

********************************************************************************/

#ifndef CUSTOM_H
#define CUSTOM_H

/* choose which option to define below */

#define TIME_SLEEP  // for monitoring peripherals
#define AUTO_JOIN  // used for peripherals that join automatically without
// user input - not recommended for security reasons
#define TASK  // used for peripherals that require extended use of the cpu
// will allow the cpu to be interrupted to check for RF messages
```

```
/*
```

```
Available I/O ports:

label on board : definition in software

Ain  : A1
PWM  : 10
MOSI  : MOSI
MISO : MISO
SCK  : SCK

Use to define meaningful names.

Example:

#define LAMP 10
#define TEMP A1
#define SWITCH MOSI

*/

#endif
```

```
/******************************************************************************
custom.ino

This file is a template that can be used for creating custom peripherals.
The three functions already started in this file are to be filled in by the
peripheral developer to meet there needs. This is also the file in which
the custom remote functions by the base station are to be written as
well as any helper functions.

This is the only file in which source code should be written by a
developer when creating a custom peripheral and goes with the
corresponding header file "custom.h".

Author: Valerie Beynon
Date: February 2014

******************************************************************************/

/******** periphInit

This function is called in the setup function of all peripherals and can be
used to perform any peripheral specific initialization tasks, such as turning
off all lights or reading an initial temperature. The only required task is in
this function is to set the peripheral's type code to match the corresponding
descriptor's type code located on the base station.
*/

void periphInit(){
typeCode = 0; // set the type code of the custom peripheral
}

#ifdef TIME_SLEEP

/*********** monitor

This function called periodically by the main loop if peripherals are designed
for monitoring applications which is indicated by defining TIME_SLEEP. In this
function users may perform any monitoring tasks they wish through reading
various inputs and outputs.

Note: When writing this function ensure that only one packet is transmitted
to the base station at a time, that it the corresponding ACK must have been
```

```
received before the next packet is transmitted. This may require that a call
back function be written to transmit multiple sensors' data back in a single
packet.
*/

void monitor(){
    TIMSK2 = 0<<TOIE2; // disable TC2 overflow interrupt

    // insert monitoring function or make calls to functions gather data

    TIMSK2=1<<TOIE2;
}
#endif

#ifdef TASK

/******** resumeTask

This function is called in the main loop if TASK is defined. This function is used
by peripherals that require extensive use of the cpu, such as peripherals that
must constantly perform tasks until they are told to stop. Custom peripheral
functions must return to the main loop periodically to allow for the peripheral to
check its transceiver buffer. This function should then be written to be able to
call the correct function to resume any tasks that are in process.

For example if peripheral is required to blink lights until told to stop this function
needs to know that it is in the process of blinking lights and call the correct
function to continue. The function it calls is responsible for returning after some
time, for example every 5 blinks.

Use of this method may require variables to be defined outside of the functions
to indicate current tasks and the variables it may need.
*/

void resumeTask(){

// call tasks currently in progress
}

#endif

/* Custom peripheral functions can be entered here */
```

Appendix F

UI Communication Protocol

The following are the functions provided to the user interface by the web server. The request is put in the body of an HTTP POST request posted to URL “program_data”. The response is placed in the body of the HTTP response. The primary response is the response that will be sent if no problems are encountered, while the other responses may be sent based on different conditions that are evident in their naming. If a user is not logged in, the response sent in every case will be “not_logged_in”. If the request is not supported, the response will be “invalid_command”.

(a) Retrieve Message Board Notifications
Request: message_board
Primary Response: [message 1]||[message 2]||[message 3]...
Other Responses: no_notifications

(b) Retrieve List of Devices
Request: device_list
Primary Response: [device name 1]||[device name 2]||[device name 3]...
Other Responses: no_devices

(c) Retrieve Device Information
Request: device_info|[device name]
Primary Response: [XML String representing device]
Other Responses: invalid_name, invalid_num_inputs

(d) Retrieve List of Services
Request: service_list
Primary Response: [service name 1]||[service name 2]||[service name 3]...
Other Responses: no_services

(e) Retrieve Condensed Service Information
Request: service_info|[service name]
Primary Response: [service name]|t‘[device name]‘[trigger name]‘[arg1]‘[arg2]... |a‘[device

name]‘[action name]‘[arg1]‘[arg2]. . . |. . .
Other Responses: invalid_name, invalid_num_inputs

(f) Retrieve Full Service Information
Request: service_representation|[service name]
Primary Response: [String representation of service. See Section 7.2.2]
Other Responses: invalid_name, invalid_num_inputs

(g) Create Service
Request: create_service|[service name]|[String representation of service. See Section 7.2.2]
Primary Response: service_created
Other Responses: invalid_name, creation_failed, invalid_num_inputs

(h) Remove Service
Request: remove_service|[service name]
Primary Response: service_removed
Other Responses: invalid_name, removal_failed, invalid_num_inputs

(i) Edit Service
Request: edit_service|[service name]|[String representation of edited service. See Section 7.2.2]
Primary Response: service_edited
Other Responses: invalid_name, edit_failed, invalid_num_inputs

(j) Edit Service Name
Request: edit_service_name|[old service name]|[new service name]
Primary Response: name_edited
Other Responses: invalid_name, edit_failed, invalid_num_inputs

(k) Search for New Device Signal
Request: wait_for_signal|[time to wait in seconds]
Primary Response: signal_detected|[device type ID]|[device ID]
Other Response: no_signal_detected

(l) Add Device
Request: add_device|[device type ID]|[device ID]|[device name]
Primary Response: device_added
Other Responses: invalid_name, add_failed, invalid_num_inputs

(m) Remove Device
Request: remove_device|[device name]
Primary Response: device_removed
Other Responses: invalid_name, removal_failed, invalid_num_inputs

(n) Do Action
Request: do_action|[device name]|[action name]|[arg1]|[arg2]. . .
Primary Response: action_complete
Other Responses: invalid_peripheral_name, invalid_action, invalid_num_inputs

(o) Edit Device Name
Request: edit_device_name||[old device name]||[new device name]
Primary Response: name_edited
Other Responses: invalid_name, invalid_num_inputs

(p) Get Status Variable From Device
Request: get_status||[device name]||[status name]
Primary Response: [the requested status String]
Other Responses: invalid_name, invalid_status, invalid_num_inputs

(q) Edit User Name
Request: edit_user_name||[old user name]||[new user name]
Primary Response: name_edited
Other Responses: edit_failed, invalid_num_inputs

(r) Edit User Password
Request: edit_password||[old password]||[new password]
Primary Response: password_edited
Other Responses: edit_failed, invalid_num_inputs

Appendix G

Software Repository

All of the software and hardware CAD files for this project along with base station software Javadocs documentation can be found at:
`https://github.com/umkoop27/Remote-Environment-Automation-Framework`

Appendix H

User Manual

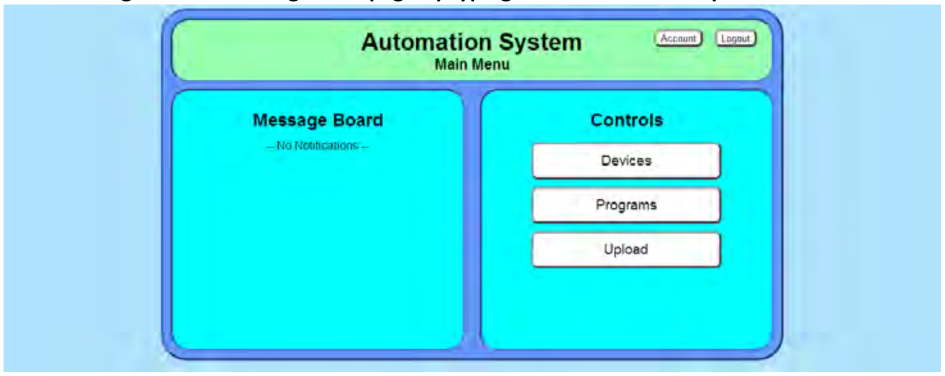
This Appendix contains a brief manual that takes end users through the process of adding a new peripheral and how to create a custom service (or program) using the web interface.

User Manual

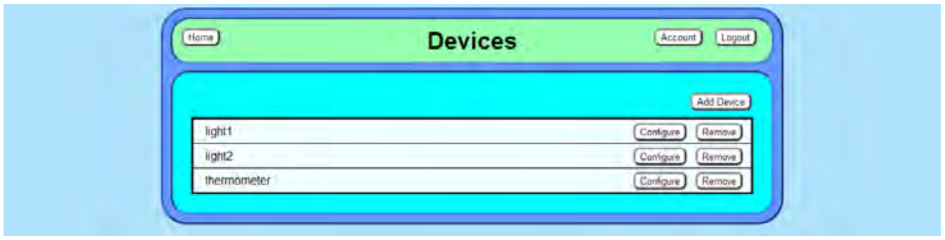
This tutorial will show you how to use the home automation system. For the purposes of demonstrating the functionality of the controls, it is assumed that the system is already up and running, as the following process has not been streamlined for users at the time of writing. Additionally, it should be noted that any devices that are referenced in this tutorial are examples of peripherals, and might not actually exist.

ADD A DEVICE

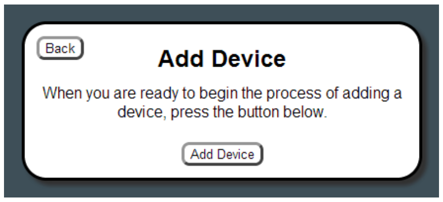
1. Navigate to the management page by typing in the IP address of your base station.



2. Click on Devices, then click Add Device.



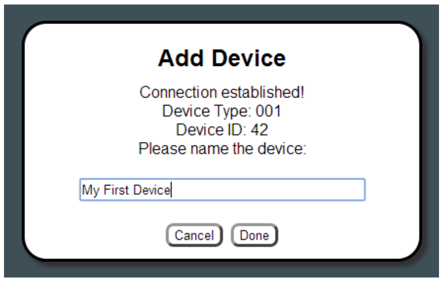
3. Observe this window:



When you are ready to begin the process of adding a device, press the button below.

Turn on your device and click the 'Add Device' button. Locate (but do not press) the sync button on the device.

4. Click the ‘Add Device’ button, and press the ‘sync’ button on your device. When the following window appears, enter a name for your device and press ‘Done’:

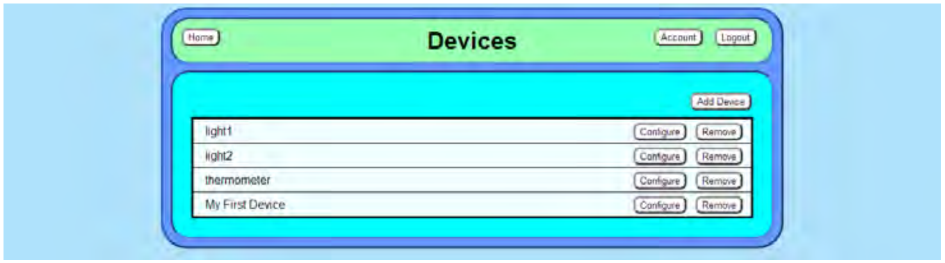


Add Device

Connection established!
Device Type: 001
Device ID: 42
Please name the device:

My First Device

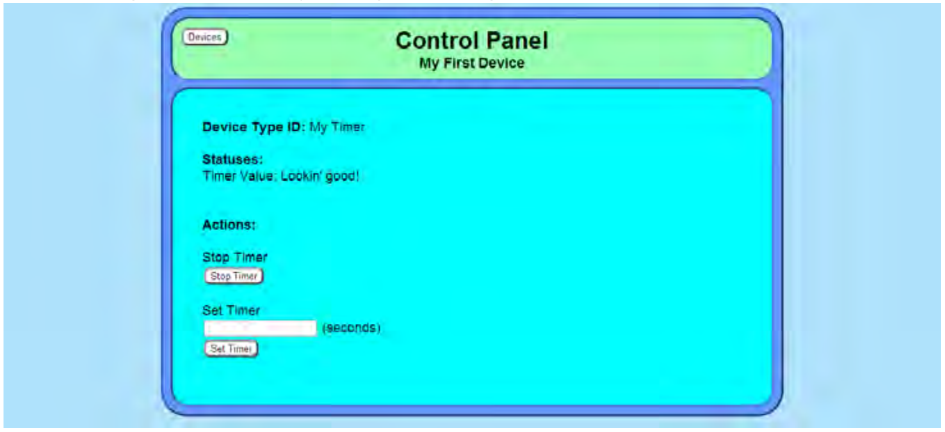
5. Observe that your device has now been added:



Devices

light1	Configure	Remove
light2	Configure	Remove
thermometer	Configure	Remove
My First Device	Configure	Remove

6. Configure the device by clicking the ‘Configure’ button:



Control Panel
My First Device

Device Type ID: My Timer

Statuses:
Timer Value: Lookin' good!

Actions:

Stop Timer

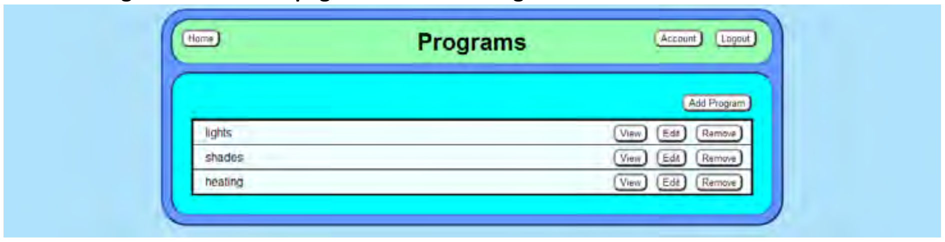
Set Timer
(seconds)

You can test the actions of your device from this screen, and observe its status.

ADD A PROGRAM

Now we will create a custom program.

1. **Navigate to the home page. Click on the 'Programs' button:**

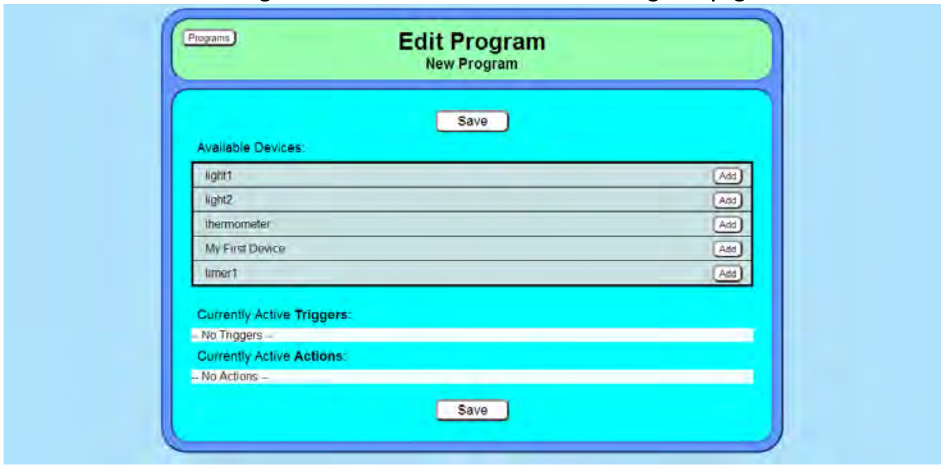


Home Programs Account Logout

Add Program

lights	View	Edit	Remove
shades	View	Edit	Remove
heating	View	Edit	Remove

2. **Now click 'Add Program.' You will be taken to the 'Edit Program' page:**



Programs

Edit Program

New Program

Save

Available Devices:

light1	Add
light2	Add
thermometer	Add
My First Device	Add
timer1	Add

Currently Active **Triggers**:

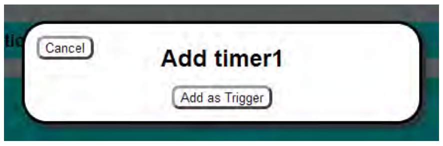
-- No Triggers --

Currently Active **Actions**:

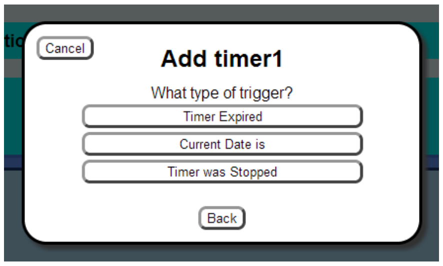
-- No Actions --

Save

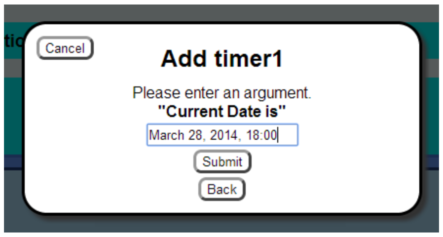
3. **We will create a service to turn on light1 when timer1 detects that the time is 6 pm. Click the 'Add' button next to timer1 and choose 'Add as Trigger'.**



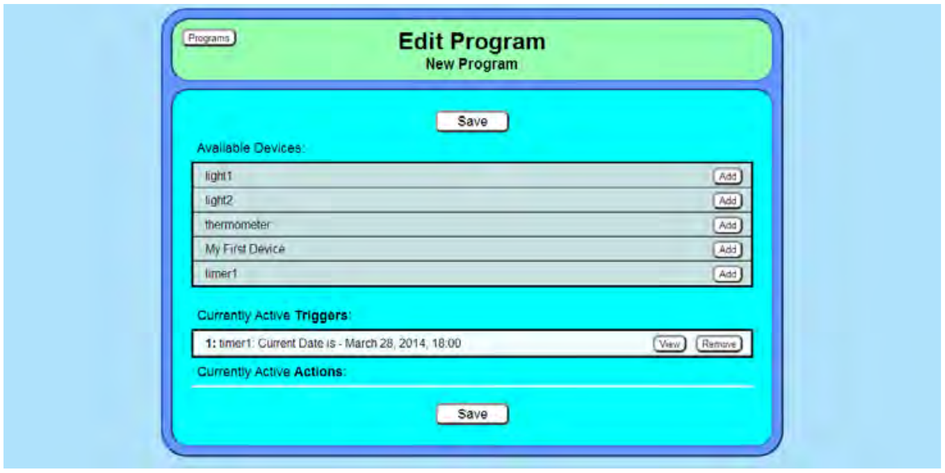
4. **You will be presented with a list of supported triggers. Choose 'Current Date Is'.**



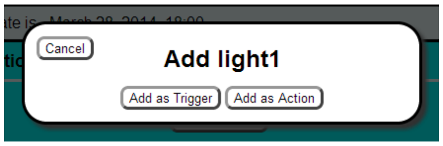
5. Select the date and time you want to use for the trigger:



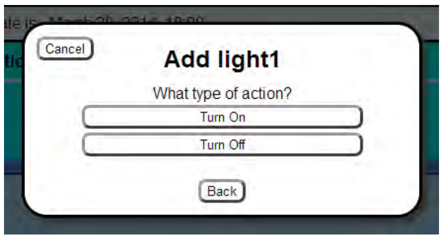
6. Finally, click 'Submit' and see your trigger in the list of current triggers.



7. Now we will add the light action. Click the 'Add' button next to **light1**.
8. Choose 'Add as Action':



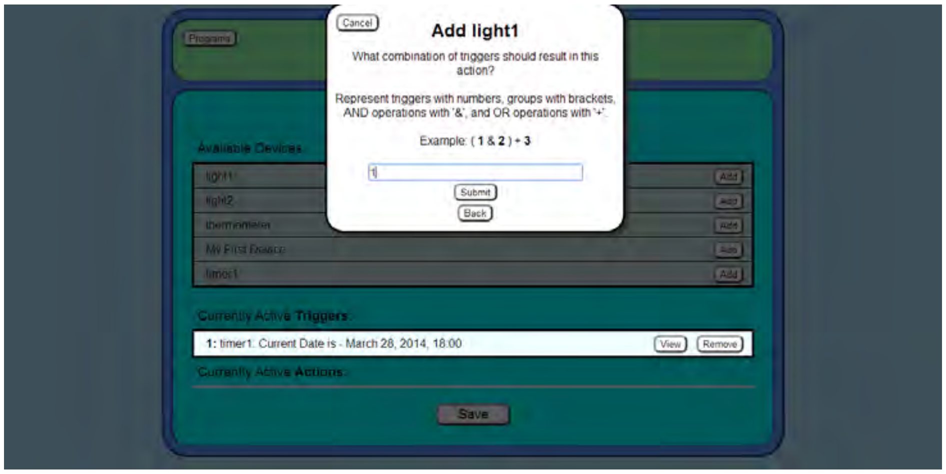
9. Choose 'Turn on':



Add light1

What type of action?

10. You will be prompted to attach the action to a trigger or set of triggers:



Add light1

What combination of triggers should result in this action?

Represent triggers with numbers, groups with brackets, AND operations with '&', and OR operations with '+'.

Example: (**1** & **2**) + **3**

1

Available Devices:

light1	
light2	
thermometer	
My First Device	
timer1	

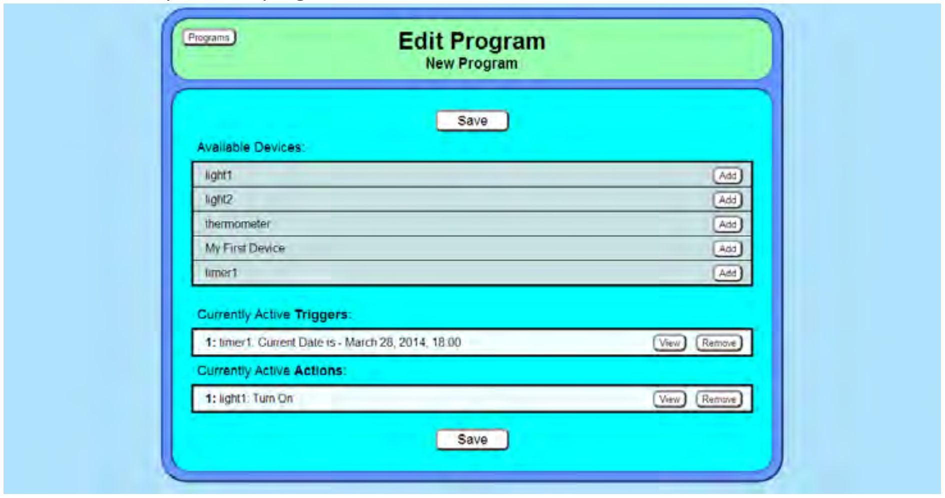
Currently Active **Triggers**:

1: timer1: Current Date is - March 28, 2014, 18:00

Currently Active **Actions**:

Since this action should be triggered by the timer detecting a specific date, simply enter the number '1' and press 'Submit'.

11. Observe your new program:



Edit Program

New Program

Available Devices:

light1	
light2	
thermometer	
My First Device	
timer1	

Currently Active **Triggers**:

1: timer1: Current Date is - March 28, 2014, 18:00

Currently Active **Actions**:

1: light1: Turn On

Appendix I

Budget

The following budget outlines the predicted and final costs as well as the sources of funding for this project. Also discussed are are the predicted and final shop time used.

Since we were able to acquire many of the required components through in-kind donations, or on-loan from external parties or team members, no external funding was required for this project. As such we were able to reduce the R&D costs for this project by \$421.25 or almost 47%. This is reflected in Table I.I.

Shop time remained completely unused. We eliminated the need for assistance from the tech shop time through the use of on-line tutorials. The machine shop time went unused since the modifications required for our enclosures were relatively minor and could be accomplished with the tools we had available at home. This is reflected in Table I.II.

Table I.III summarizes the expenditures associated with the project according to the original proposed budget. Items eliminated from the originally budget are listed as having a cost of "–". Further details about the materials acquired using the allocated funds are found in Appendix C.

TABLE I.I: CAPITAL FUNDING BREAKDOWN

Source	Predicted Cost	Final Cost
ECE Department	\$500.00	\$478.75
External Funding	\$400.00	\$0.00
Total	*\$900.00*	*\$478.75*

TABLE I.II: SHOP TIME

Area	Item	Predicted (hrs)	Final (hrs)
Machine Shop	Enclosure Modifiation	4:00	0:00
Tech Shop	SMD Soldering Lesson	0:30	0:00
	Total	*4:30*	*0:00*

TABLE I.III: EXPENDITURES

Area	Item	Quantity	Cost	
			Expected	Final
Development	Dev. Kit for Peripheral Core	1	\$38.00	–
Electrical	Peripheral Core (Microcontroller)	8	\$260.00	123.63
	Misc. Components	1	\$160.00	–
	Unix-based Computer (Base Station)	1	\$60.00	–
Mechanical	PCB (Peripheral Cores)	8	\$122.00	–
	Peripheral Enclosures	8	\$110.00	\$26.61
Misc.	Peripheral Components	1	\$150.00	\$328.51
		Total	*\$900.00*	*\$478.75*

Appendix J

Curriculum Vitae

Jonathan-F. Baril

PLACE OF BIRTH:	Montréal, Quebec
YEAR OF BIRTH:	1991
SECONDARY EDUCATION:	River East Collegiate (2006-2009)
	Advanced Placement International Diploma (2009)
	Deutsches Sprachdiplom II (2009)
	Advanced Placement National Scholar 2009
	Queen Elizabeth II Entrance Scholarship 2009
	Advanced Placement Entrance Scholarship 2009
	Dean's Honor List 2009-2013
	President Scholar 2009-2014
	Engineering Class of 1946 Scholarship 2010
	NSERC USRA 2011,2014
	Ernest M. and Margaret Scott Memorial Scholar. 2012
	William & Olive Humphrys Scholar. for Elec. Eng. 2012
	Baden-Wuerttemberg Stipendium 2013
	Easton I. Lexier Award for Community Leadership 2013
WORK EXPERIENCE:	Parker Hannifin Electronic Controls Division (2011-2012)

Valerie Beynon

PLACE OF BIRTH:	Winnipeg, Manitoba
YEAR OF BIRTH:	1991
SECONDARY EDUCATION:	Balmoral Hall School (2005-2009)
HONOUR AND AWARDS:	Advanced Placement Scholar 2009
	Queen Elizabeth II Entrance Scholarship 2009
	Dean's Honor List 2009-2013
	President Scholar 2009-2013
	Engineering Academic Excellence Award 2011
	Engineering Class of 1946 50th Anniversary Award 2011
	Faculty of Engineering Second Year Scholarship 2011
	Grettir Eggertson Memorial Scholarship 2011
	President's Scholarship 2012
	University of Manitoba Merit Award 2012
	Engineering Academic Excellence Award 2012
	NSERC Undergraduate Student Research Award 2013
	Dr. Kwan Chi Kao Scholarship 2013
	Faculty of Engineering Centenary Scholarship 2013
	UMSU Scholarship 2013

Shawn Koop

PLACE OF BIRTH:	Winnipeg, Manitoba
YEAR OF BIRTH:	1991
SECONDARY EDUCATION:	River East Collegiate (2006-2009)
HONOUR AND AWARDS:	Advanced Placement Scholar with Honor 2009
	Queen Elizabeth II Entrance Scholarship 2009
	Advanced Placement Entrance Scholarship 2009
	Dean's Honor List 2009-2013
	President Scholar 2009-2013
	Faculty of Eng. Centenary Scholarships 2010,2013
	Dr. Kwan Chi Kao Scholarship 2010,2012
	UMSU Scholarship 2012
	University of Manitoba Merit Award 2012
	Vaughn Betz Engineering Centenary Award 2012
WORK EXPERIENCE:	Undergraduate Student Research Assistant

Sean Rohringer

PLACE OF BIRTH:	Winnipeg, Manitoba
YEAR OF BIRTH:	1991
SECONDARY EDUCATION:	J.H. Bruns Collegiate (2006-2009)
HONOUR AND AWARDS:	Vaughn Betz Engineering Centenary Award 2013
	Engineering Academic Excellence Award 2012
	NSERC USRA 2011-2013
	President's Scholarship 2009-2012

Paul White

PLACE OF BIRTH:	Winnipeg, Manitoba
YEAR OF BIRTH:	1991
SECONDARY EDUCATION:	River East Collegiate (2006-2009)
	Advanced Placement International Diploma (2009)
	Deutsches Sprachdiplom II (2009)
HONOUR AND AWARDS:	Advanced Placement National Scholar 2009
	Queen Elizabeth II Entrance Scholarship 2009
	Advanced Placement Entrance Scholarship 2009
	Dean's Honor List 2009-2013
	President Scholar 2009-2013
	NSERC USRA 2012-2014
WORK EXPERIENCE:	MTS Allstream Intern, 2010-2012
HOBBIES:	Long conversations and sunset walks on the beach
	Pwning n00bs in GTA V