ARDUINO BASED WIRELESS POWER METER

A Design Project Report

Presented to the Engineering Division of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering (Electrical)

By

Christopher McNally

Project Advisor: Bruce Land

Degree Date: May 2010

Abstract

Master of Electrical Engineering Program

Cornell University

Design Project Report

Project Title: Arduino based wireless power meter

Author: Christopher McNally

Abstract:

One avenue through which today's energy problems can be addressed is through the reduction of energy usage in households. The existing utility system only provides feedback at the end of the month in the form of a bill and consumed kilowatt hours (kWh). A homeowner has no way to track their power usage on a more immediate basis.

The Arduino based wireless power meter is a non-invasive current meter for household power with a Matlab interface. Current is measured using split core current transformers. This data is then transmitted over a 802.11b connection through the home's wireless router to the base station and Matlab interface. The project aims to provide a clear picture of a home's current usage, and through this data provide an estimate to power consumption. The project also aims to identify which devices turn on and off by analysis of this current data. The goal of provided such data to a user is that they will optimize and reduce their power usage.

Report Approved by

Project Advisor:_____Date:_____

Executive Summary

The Arduino based wireless power meter centered on the design and development of a current measurement device with an IEEE802.11b link to a base station running a graphical user interface (GUI). The project's goal was to present a user with energy usage information in the hopes that they could use the information to optimize and reduce their energy consumption.

The system was comprised of the current transducers and their rectifier circuitry, the main Arduino board, the WiShield add-on board, and the base station computer. The software portion of the project resided on the Arduino board and on the base station computer. Current was measured from the two current carrying wires into the main power panel, sampled by the Arduino board, and then sent as a UDP packet over the Wi-Fi network to the base station computer. The base station's software then parsed the packet, and converted the raw analog to digital conversion into current data, where it could be used to either display current usage to the user, or estimate apparent and real power for display to the user.

The project succeeded on most fronts. The update rate of data from the embedded side of the project to the base station was 20 Hz, with a measurement resolution of the RMS current of about 47.5 mA. This resolution allowed small changes in the current draw of the house to be visible while examining data. When compared to a consumer grade Wattmeter, the current measurement data showed an error of approximately 4.96% while measuring a purely resistive load consuming 180 W.

The wireless connection between the embedded electronics and base station was reliable for the networking equipment used. However, the 802.11b module used on the embedded side was not compatible with some wireless routers, meaning a user would have to change their router if they happened to be using an incompatible router. Range of the device seemed to be on par with most other 802.11b devices.

The feature of automatically detecting which devices turn on when based on the current data was not successful. Based on the data collected, the change in turn on current only differed significantly between devices by magnitude, not waveform. This meant the originally intended method of cross correlation between the incoming data and reference turn on data would not be effective. A method of identifying devices based on the magnitude of changes also proved to be unreliable. If more than one device powered on at once (such as a light switch turning on a room full of devices), the measurements would only see the net difference in current draw. This would lead to inaccuracies in device detection, and so the feature was dropped from the system.

Table of Contents

1 Introduction

In the existing power utility set up, consumers are presented with usage information only once a month with their bill. The length of time between updates about power usage is far too long for a consumer to observe a changed behavior's effect on power usage. In addition utility bills can be convoluted in how they present usage information, and a consumer may not be able to decipher changes in their power usage from the last bill. An opportunity to educate customers on power usage is lost because of these realities.

If a person can instantaneously see how much power leaving a device on by accident consumes per minute, they may be more careful in the future about letting devices run when not needed. The goal of creating more awareness about energy consumption would be optimization and reduction in energy usage by the user. This would reduce their energy costs, as well as conserve energy.

1.1 Household power

In most US households, power comes into the house through a three wire, split phase connection. Two "hot" wires carry current into the circuit breaker. A neutral wire also provides a connection to ground for the house circuit. Each hot wire has an RMS voltage of 120V +/- 5%. The wires are set up so that the AC waveforms are 180°, so that both lines combined can provide a 240V source for larger household appliances.

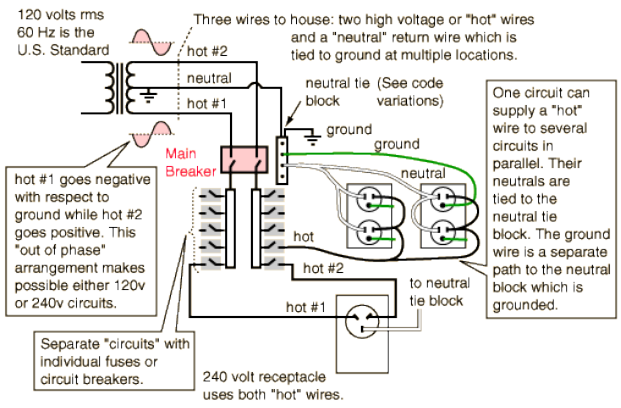

Figure 1. US household three wire power connection [1]

The apparent power consumed by a household can be found by taking the product of the RMS voltage and total RMS current. The real power can be calculated from discrete samples by taking the average of the product of the voltage and current samples over a specified window. Power factor can then be calculated by dividing real power by the apparent power. This project only included current measurements, and thus both power measurements are estimates, as opposed to measurements.

1.2 System requirements

The goals of the project were as follows:

- Accurately collect a house's total current consumption safely, and with a relatively fast update rate.
- Transmit the information back to a base station to be represented visually to a user
- Identify when devices in the household turn on and off based on changes in the current data

To ensure the safety of the user and the ease of installation, a non-invasive method of measuring current was required. This meant that the current measurement circuit could not be in series with the mains power line, and could not require the disconnection of the line for any reason.

The wireless connection had to be able to integrate into the user's home network, and comply with the IEEE802.11b standard. The data rate of the wireless connection also had to be sufficient to support the required update rate. The overall speed of both the wireless connection protocols had to be sufficiently fast that it would not interfere with obtaining the required amount of samples in a given period.

2 System design

2.1 Hardware

2.1.1 Overview

The hardware of the system consists of three parts: sensor capture, the MCU board, and the wireless board. The sensor capture hardware consists of the current transducers and rectifier circuit, connected to the analog to digital converter (ADC) of the MCU. The MCU board is a standalone Arduino Duemilanove development board. The wireless board is a shield designed to pair with the Duemilanove, created by AsyncLabs. A system overview can be seen in the appendix.

2.1.2 Arduino Duemilanove

The Duemilanove is a self contained USB development board centered on an ATMega328P. The operating voltage for most of the board is 5V, provided by an onboard voltage regulator. Power for the board can come from the USB connection, the 9V DC connection, or a battery connected to the V_{IN} terminal. For this project, the 9V DC connection was used for power. A detailed schematic of the board can be found in the appendix.

The current measurement circuit connects to the Arduino board through analog pins 0 and 1. This allows the Arduino to sample the output voltage from the current measurement circuit with the ATMega328's 10-bit ADC. A voltage of 3.3V was applied from the USB to serial converter chip to the voltage reference pin, A_{REF}, in order to provide the ADC with the required reference voltage. Connections to the ground and V_{CC} pins were also provided for the op amp in the precision rectifier.

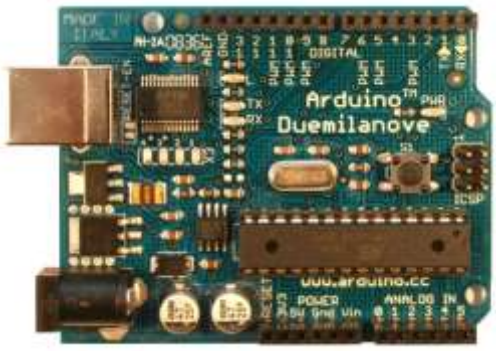

Figure 2. Arduino Duemilanove board [2]

The USB connection on the board was used for programming the chip and getting serial output for troubleshooting. The interface between USB and the ATMega328's UART was provided by the Arduino's built in USB to serial converter chip. The device showed up as a virtual COM port on the host PC, and could be interfaced with any program capable of reading and writing to a serial port.

The board came pre-built from the distributor, so no major hardware assembly was required for use. Female headers are attached to each of the I/O and power ports, allowing wires to be inserted rather than soldered to each connection. The connections to the A_{REF} pin, the analog pins, and the ground/V_{CC} pins were all wired in this way. No major modifications of the stock hardware were required.

2.1.3 WiShield

The WiShield contains an 802.11b compliant wireless module that handles the network, data link, and physical layers of the design's wireless connection. The board also includes required loading circuitry and a PCB antenna. A detailed schematic of the board can be found in the appendix.

The WiShield communicated with the Arduino main board through an SPI link and an interrupt line. The bi-directional SPI link allowed the Arduino to issue commands and send data to the 802.11b module, while the module could send received data back to the Arduino. The WiShield's connections to the Arduino are detailed in Table 1.

Description	Arduino pin	ATMega328 pin
Slave select (SS)	10	PortB.2
Clock (SCK)	13	PortB.5
Master in, Slave out (MISO)	12	PortB.4
Master out, slave in (MOSI)	11	PortB.3
MCU interrupt	2	PortD.2
LED connection light	9	PortB.1

Table 1. WiShield connections to Arduino

The board came preassembled from AsyncLabs, thus no major hardware assembly was required for the wireless portion of the circuitry. Female headers with male leads underneath were included on the board, so that the WiShield was inserted into the Arduino main board's female headers with no soldering required.

The system had known compatibility issues with some routers, as detailed on the AsyncLabs wiki page[3]. A Netgear WRN2000 was used as the router for this project because it was confirmed to be compatible by AsyncLabs. The router limitation was directly from limitations of the 802.11b module used, thus no software work around was attempted. For a production system this would represent a problem, but for a proof of concept system this was considered an acceptable limitation.

2.1.4 Current measurement

A split core current transducer was chosen as the current sensor of the project because of its non-invasive nature. The sensor can be clamped onto the mains lines without interrupting power into the circuit breaker. Avoiding interrupting the lines into the circuit breaker makes the installation of the sensors both safer and easier. One sensor is clamped onto each "hot" wire going

Figure 3. CR3110 current transducer [3]

into the circuit breaker. The total current being drawn by the household is the sum of the measured current in both wires.

The output of the current transducer is an AC voltage proportional to the AC current enclosed by the sensor's ring. The amplitude of the voltage waveform is determined by this equation:

$$V = \frac{I*R}{\mathbf{3100}}$$

Where V is RMS AC voltage across the burden resistor, I is the RMS AC current enclosed by the transducer, and R is the resistance of the burden resistor. This equation means that the output range of the sensor is determined by the selection of the burden resistance R. Selecting a higher burden resistance scales the output higher, providing a greater resolution of currents measurement. A higher resistance also means a smaller range of current measurement, and potentially damage to the MCU if the current is higher than intended for the design. When sizing the burden resistor, the range of linear behavior must also be taken into account. **Figure 3** show the linear limit of the sensor with a dotted line. Measurements made beyond this limit will not follow the equation given above.

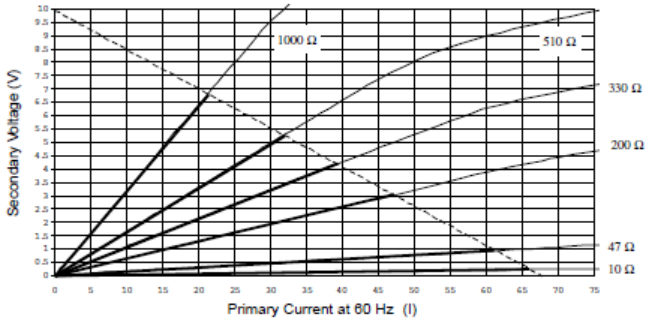

Figure 4. Voltage output from CR3110 [4]

The burden choice for this design rested on two points: maximum resolution given the available reference voltages on the Arduino, and selecting an appropriate range for measuring household current. A reference voltage of 3.3V was selected as the best option for the ADC, which meant that V_{MAX} of the AC voltage signal had to be at or just below 3.3V. A burden resistance of 138.9 Ω would scale the output of the sensor to be 3.3V V_{MAX} at the edge of the linear region, providing the largest range and resolution for the given reference voltage. A resistance of 150 Ω was used as an approximation due to the availability of resistors of that type. In this configuration, the maximum current that could be measured with the system was 48.2 A per line, giving a total range between both sensors of 96.4 A. For an apartment or small to medium household, this range worked well in testing. A larger range may be necessary for larger homes.

The VAC output of the sensor could not directly interface with the Arduino board because of the maximum limits of reverse voltage on the MCU pins. A rectifier was therefore required to ensure that a negative voltage would not be applied to the pins of the MCU. Two rectifier circuits were considered for this role: a full wave rectifier using a diode bridge, and a half wave precision rectifier an op amp. The

diode bridge was also paired with a smoothing capacitor and discharging resistor in hopes of reducing the voltage ripple.

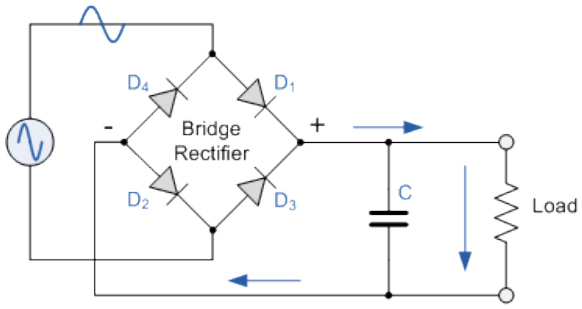

Figure 5. Full wave bridge rectifier with smoothing capacitor [5]

This design proved inadequate due to the forward voltage drop of the bridge rectifier and response to rapid changes in current. The forward voltage drop of the device used was around 1.1V, and depended on the current flowing through the device. A voltage drop this large essentially meant that currents below 1/3 of the maximum voltage would not be detected at all. In addition, accuracy of the voltage measurement was not certain due to the forward voltage drop changing with the amount of current going through the device. The smoothing capacitor also proved problematic because it either greatly reduced the response time of the circuit to rapid changes in the primary current, or let some of the AC waveform pass through to the ADC.

A precision rectifier is built around an op amp, two diodes, and two resistors. Using this circuit instead of the diode bridge solved the forward voltage drop problem, because the op amp corrects the output for any voltage loss over the two diodes. AC ripple was allowed to reach the analog pin and dealt with in software.

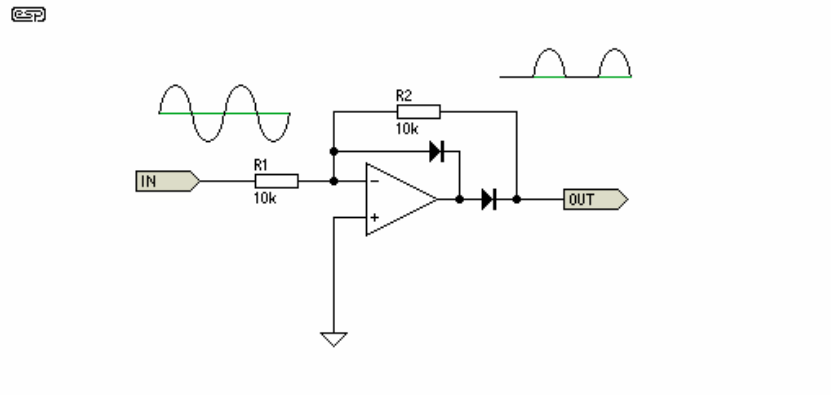

Figure 6. Precision rectifier [6]

2.2 Software

The software side of the system was split into the programs running on the Arduino and the GUI running on the base station. The Arduino was responsible for both capturing the output of both current sensors, and sending the measured data with timestamps over the wireless connection to the base station. The

base station was responsible for reading the packets from the Arduino, converting the raw data into current/power information, and presenting it to the user in a clear fashion.

2.2.1 Arduino software

Data capture

Sampling of the voltage output of the current measurement circuit was done using the ADC on the ATMega328. The Arduino IDE included a family of functions for controlling and retrieving data from the ADC. The initial setup of the ADC was done using the *analogReference(SOURCE)* command, which set which reference voltage the ADC will use in its conversions. The external voltage reference option was set for this project in order to use a reference voltage of 3.3V. The function *analogRead(port)* returned a value between 0 and 1023, reflecting the level of voltage on the specified analog pin. *AnalogRead()* typically took around 100 µs to execute and return the conversion value.

As described in section 2.1.4, the smoothing of the half rectified voltage waveform was handled in software, as opposed to with a smoothing circuit or other filter. In each iteration of the main program, *analogRead()* was called 250 times, and the maximum value returned was saved. The voltage waveform had a period of about (1/60Hz), or 16.7 ms. Sampling 250 times typically took around 25ms, ensuring that at least a full cycle was sampled. Shorter times were experimented with, but sampling periods lower than this sometimes resulted in "dropped" measurements, where the maximum voltage seen was either during the zero portion of the period, or not near the maximum. The maximum value seen was reset when a packet was sent.

Time stamping was handled with the Arduino's *millis()* function, which returned the number of milliseconds since power on or last reset. While not providing an absolute time of each measurement, the times stamps were useful in determining if packets were lost, and in graphing measurements more accurately in the time domain of the user interface. The time was taken to be at the beginning of each sampling period, so that measurements from both lines would be consistent with each other. Without this consistency, combining the waveforms on the base station side would be difficult.

WiFi connectivity and networking

The system uses WiShield's open source implementation of µIP and a modified version of their example UDP communication app. µIP is an open source TCP/IP stack designed for embedded systems, with the ability to run a full TCP/IP stack on an 8 or 16 bit system with very limited RAM and instruction memory. A major reason the WiShield was selected over other 802.11 solutions for embedded systems was because µIP was already ported and released as open source code. This made prototyping of the system far faster and more focus on current and power measurement rather than networking protocols.

The WiShield's core code contained all of the included functionality built in. In order to select a configuration, define statements in the header file "apps-conf" were uncommented. The UDP sending function of the system was based off the example UDP endpoint example, and thus the corresponding

define statement was added to the configuration header. In addition, the configuration header for the µIP stack was altered to turn on UDP support.

The wireless connection set up was defined by a set of global variables at the beginning of the program and by AsyncLab's function *WiFi::init()*. Parameters were recorded in these global variables at compile time and sent to the 802.11b module by *init()*. These parameters included the SSID, passphrase of the network, security type, IP of the WiShield and gateway, and subnet mask. The initialization of the connection was run during *setup()*, and blocked continuation of the program until a connection was established. The system was tested on a WPA2 encrypted network, thus the 802.11b module had to calculate the PSK from the passphrase given, adding substantial time (on the order of 30 seconds) to the set up phase of the program. Using an open or WEP encrypted network would eliminate this, but the added security of WPA2 was considered an acceptable trade off.

After setup, the handling of the wireless connection and data sending is handled by a call to *WiFi::run()* each iteration. This function calls the run process of µIP and checks to see if the send timer has expired. Data transmission is controlled by this send timer. When it expires, a call to *udpapp_appcall()*, and subsequently, *handle_connection()*, is made. In *handle_connection()* , the time stamp and both maximum ADC values are printed into a string buffer, which then gets passed to the µIP function *uip_send()*, which sends a UDP packet containing the passed data. The time out timer was set to be 50 ms, giving the embedded side a maximum update rate of 20 Hz. The packet structure is "timeData reading1 reading2", with a space used as the delimiter between data values.

UDP packets were chosen over TCP/IP for several reasons. µIP's implementation of TCP/IP had issues when working with any system that used delays acknowledgement packets (ACKs), because µIP only allows one packet to be in flight per TCP segment. This limitation can cause delays in transmits up to 500 ms, but typically about 200 ms [7]. A modification of the µIP stack that splits packets into two helps alleviate this issue, but the time required to work this modification into the AsyncLab implementation on the WiShield was not deemed worthwhile. Many of the features of TCP/IP were not critical to the project's goals. The speed and simplicity of UDP was a better fit than the reliability of TCP/IP.

2.2.2 Base station interface

The base station ran as a Matlab figure, with each button having a call back function to execute on press. A screen shot of the interface can be found in the appendix. Each button or field in the interface is created with *uicontrol* and specifying the desired look and behaviors. The graph is created with a call to axes with the desired setup information. The start button begins the data collection routine, and continues until the stop button is pressed. Once the stop button is pressed, the collected data is saved to a MAT file of the time and date of the end of the data collection. The three 'set' buttons each set their respective parameters for the data collection function. The two drop down menus set the value to be graphed (current, estimated apparent power, or estimated real power) and which sensors to graph.

The data collection function operates as a loop, only breaking when the exit flag is set by the stop button, or an error occurs with the UDP feed. Before the loop is started, the UDP socket is opened and a

timeout value is selected. The socket is opened and read from using Java functions imported into Matlab. The data returned by this method is a string of 8 bit unsigned integers reflecting the ASCII values of the characters in the data packet. Data is extracted from this format by first converting to a string, and then using *sscanf* to extract the three integer values.

After extracting the time and raw conversion data, they are each added to the array of data already collection. Time data is not further manipulated by the interface at this point. The ADC data is used to calculate the corresponding value of primary current measured using the equation:

$$Current = \frac{ADC\ value * V_{REF} * 3100}{1023 * \sqrt{2} * R}$$

Where V_{REF} is the reference voltage used by the ADC and R is the burden resistance of both sensors. After calculating this value, it is appended onto the existing current measurement array for each sensor.

Updating the graphs is also done by the data collection function. The program firsts checks to see whether the user has selected current, apparent power, or real power to be graphed. Based on this the current data is converted into the desired value, and then the last n samples are graphed. N is determined by the window size selected. A larger windows size shows a longer history of current data, but the ability to visually discern small changes in current decreases as the window size increases. Apparent power is calculated as

$$P_{apparent} = I_{RMS} V_{RMS}$$

The RMS voltage is an estimate provided by the user, as the system right now does not have the capability of measuring the mains voltage. Real power is calculated as the apparent power times the power factor. Power factor was a user supplied estimate as well.

To create the effect of a sliding graph of the desired data, immediately before graphing the new data the axis was cleared of all items, including the axis labels and scaling. The data was then plotted, and labels assigned based on which data was being graphed. After calling the plot command and appropriate labeling commands, *drawnow* was called. This command flushes the graphics stack and forces Matlab to draw everything in the stack. Without this command Matlab would perform the default behavior of waiting to graph anything until the program exits.

3 Results

The system was successful in some respects, but did not meet all the design goals laid out during the project's conception. The measurement resolution and update rate on the embedded side were both satisfactory for the goal of measuring energy usage, but the ability for the base station to detect different devices from turning on was not achieved, and the speed of execution of the base station software caused problems with receiving the UDP packets in a timely fashion.

The system was tested on a smaller scale than an entire household for accuracy due to a lack of tools available for verifying current flowing into the circuit breaker. A Kill-A-Watt™ was used to measure the

current drawn by a household lamp. At the same time, the live wire of an extension cord was separated from the outer insulation and enclosed in one of the current transformers, allowing the system to measure current drawn by the lamp. Using this method, the system displayed an error of 4.96% from the output of the Kill-A-Watt™. The quantization error of the ADC was 4.75% at 1 A, and the two burden resistors had an error of 1.8% and 1.6%. The error displayed in this test can mostly be attributed to these factors. When measuring higher currents, the quantization error would decrease linearly, meaning measuring larger currents would be more accurate. The error rate observed here can be thought of as a worst case scenario in this regard, with a lower error rate during normal operation.

A 20 minute test of measuring current through the main breaker yielded the results in figure 7. Figure 7b is the observed current in both sensors. The large initial spike in voltage is possibly an artifact from the current transducer itself reacting to power on. The large changes in current were attributed to the electric oven which was on during the testing. The next smallest change in current is thought to be a refrigerator, but is not known for certain.

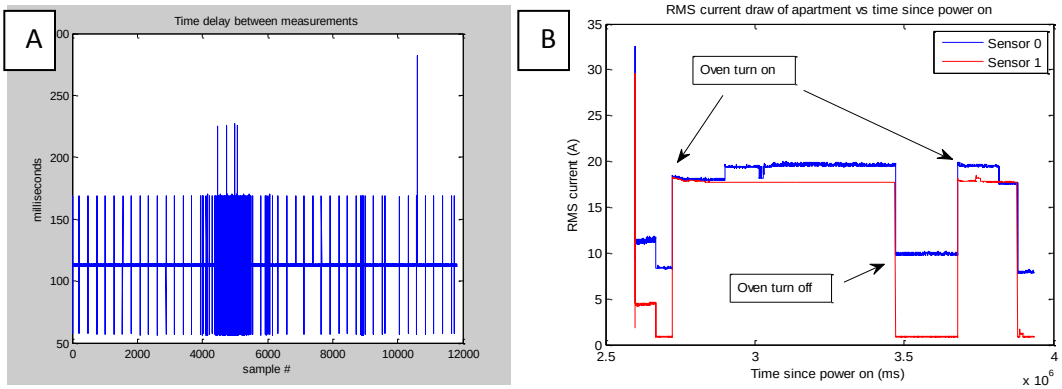

Figure 7. 20 minute system test

Figure 7a is a plot of the time difference between the time stamps of the samples. On average this time difference was 113 ms, with a maximum delay of 282 ms and a minimum of 56 ms. The average time difference was about twice the theoretical maximum based on the transmission timer set in the embedded code. This test suggests that the average update rate of the system was about 8.8Hz. A potential source of slowdown could be extra overhead from the µIP stack that was unforeseen during development. This possibility was ruled out by verifiying the timing of packets by monitoring the network with Wireshark.

Section of code	Min time(ms)	Max Time(ms)	Average (ms)
Overall	47.4	238.2	112.8
Plotting and graphics	36.4	118.9	44.7
UDP receiver	.310	176.5	67.8
All else	.169	5	.269

Table 2. Statistics of time of execution data

The MATLAB commands *tic* and *toc* allow routines to be timed. Three separate instances of this timer were added to the base station software to gain insight into the dropped packet problem. The results of this test can be seen in figure 8a, along with the averages for each portion in table 2. The data observed suggests that the timing problem is a combination of the Java UDP receiver code and plotting code taking too long to execute, and therefore causing packets to be missed. A modified plotting section made use of the *set* command in Matlab by changing the values stored in the data fields for the lineobject created by the initial plot command. The test results with this configuration can be seen in figure 8b. The execution time of the plot section was significantly reduced, with a new average execution time of 16.5 ms. Unfortunately this performance increase does not solve the problem of lost packets, because the receiver function is still sufficiently long in execution time that packets would still be missed. The next version of the system should employ changing the line object each iteration over using the *plot* command, but will also need to find a new implementation of a UDP receiver to achieve better results.

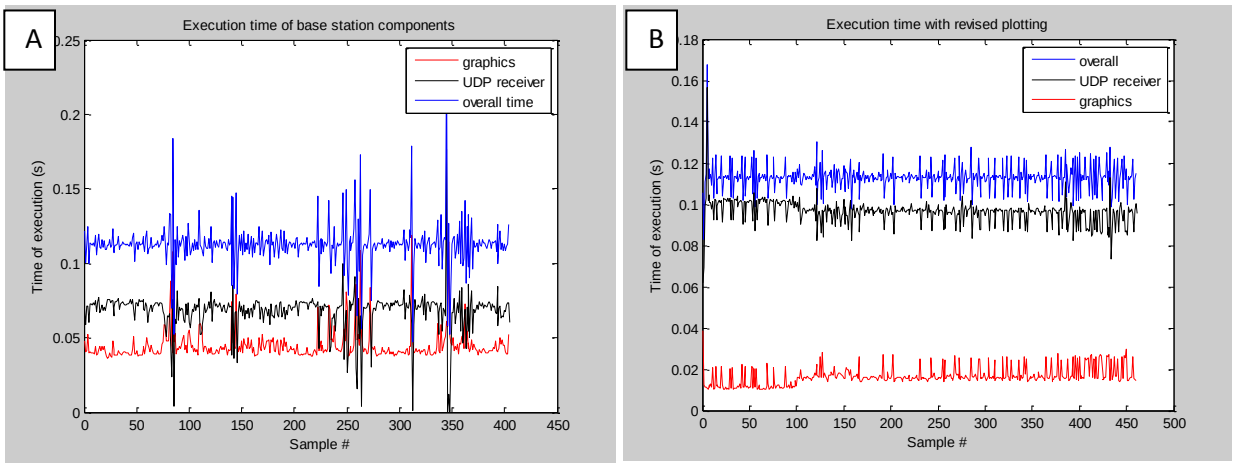

Fig 8. Execution time of base station subroutines

Accuracy of the power estimates depended on the user supplied power factor and RMS voltage. When values were taken from a Kill-A-Watt™, the error was the same as a current only measurement. Unfortunately this method could not be applied when measuring the lines into the circuit breaker, at least for power factor. The addition of a voltage measurement component to the project would add the ability to more accurately estimate both apparent power and real power. The complexity and cost of the system would increase, but the ability to more accurately show user real power data would be valuable. The downside would be that a high voltage interface would be required in order to measure the voltage, increasing the safety risk of installation.

4 Conclusion

The system was successful in measuring current within an acceptable range of error, and sending that information at a higher update rate than previous similar projects have done. Moving time expensive floating point calculations off the Arduino and performing them on the base station helped to speed up

execution time, but difficulties with plotting data and receiving the packets on the base station side lowered performance.

The device detection feature may still be realized if the update rate could be recovered from the lost packet issue, but more likely a more detailed analysis of the shape of the waveform would be needed to do such turn on and turn off detection. Parameters like harmonics and distortion of the waveform for both current and voltage would be useful in identifying which devices are running.

The project was a valuable experience in the design, implementation, and testing of a system that involved several discrete hardware and software components. The use of an open source project for such a central function as the IP stack in the project was initially planned to be a large drain in design time, but ended up greatly accelerating the design of the wireless part of the embedded system. More time was available for the current measurement circuit, which was able to go through several designs before an acceptable one was reached.

Ultimately the system accomplished its primary goal of presenting energy information to a user in a clear way. An individual could watch the interface and visibly see the effect of leaving an appliance or device on, or see the difference in remembering to turn the lights off when they are not needed. More functionality in this regard would have been helpful, specifically long term averages and either numerical or visual comparisons between older data and present data.

5 Appendix

5.1 Figures

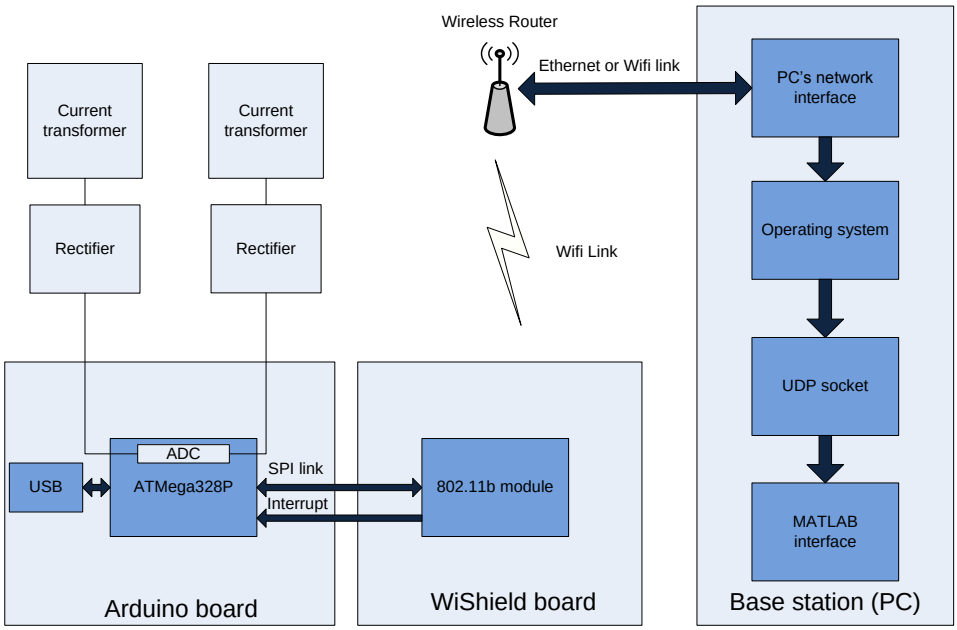

Figure 9. System diagram

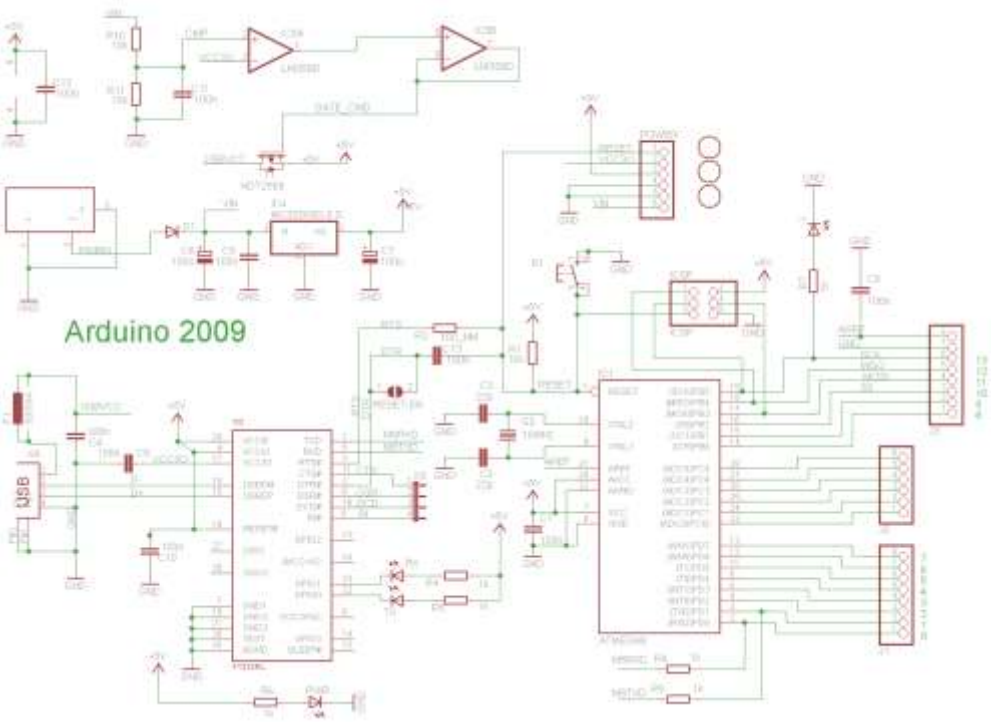

Figure 10. Arduino board schematic [8]

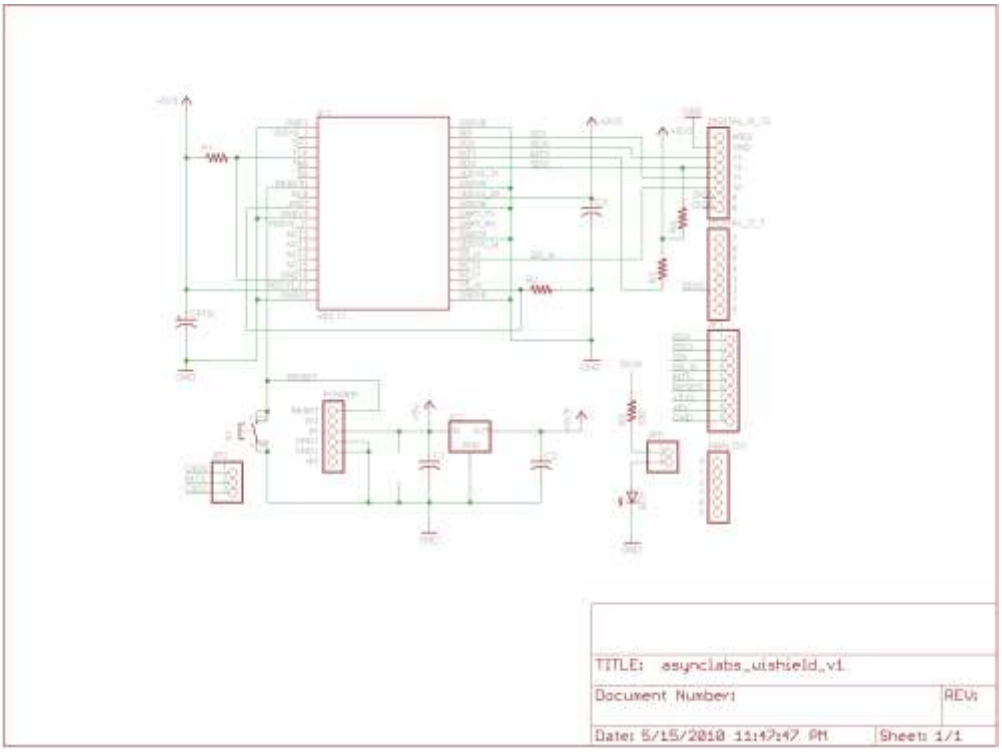

Figure 8 WiShield board schematic [9]

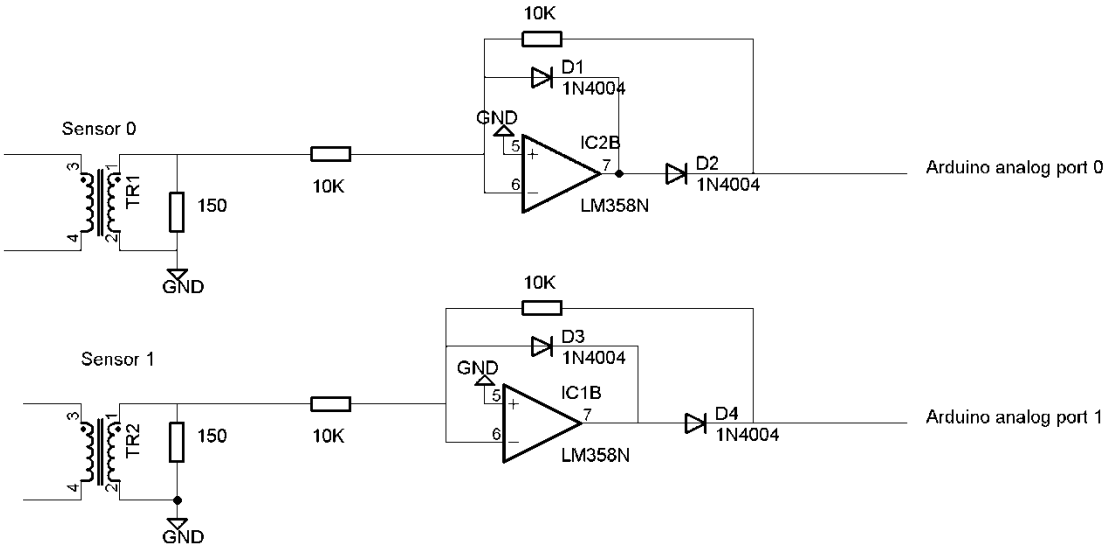

Fig 12. Sensor and rectification circuit

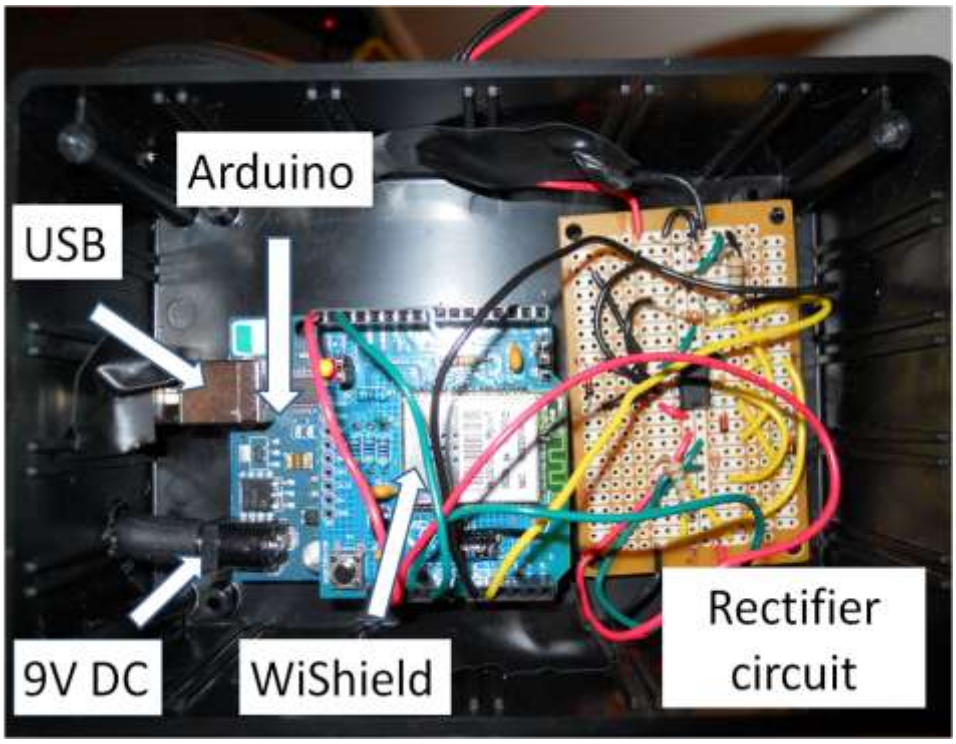

Figure 9. Photograph of completed embedded side

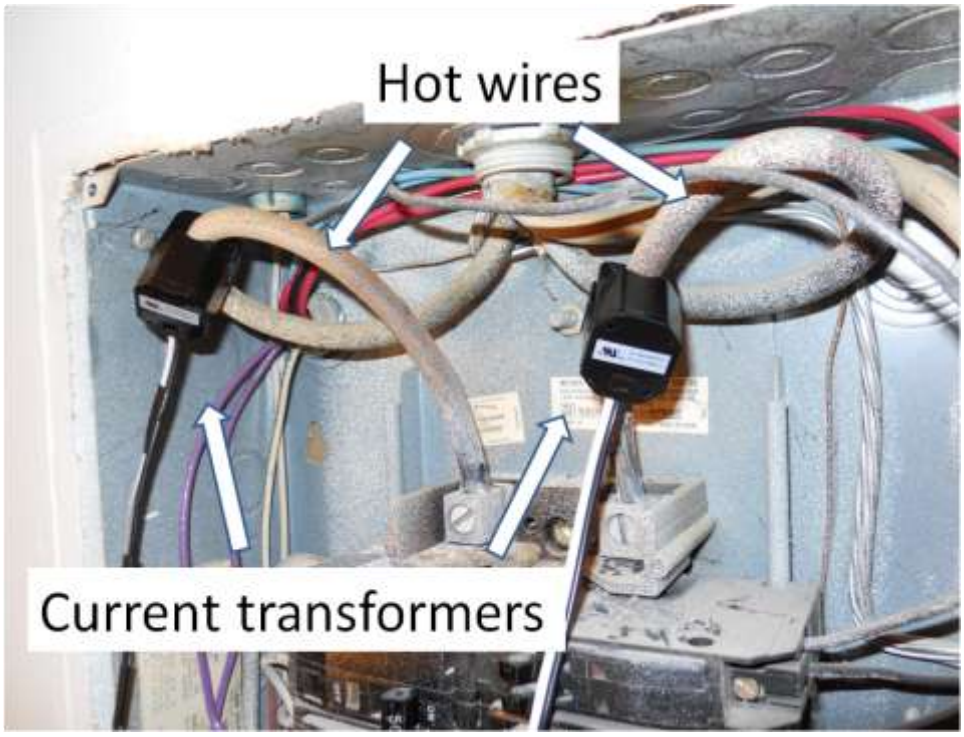

Figure 10. Photograph of installed current transducers

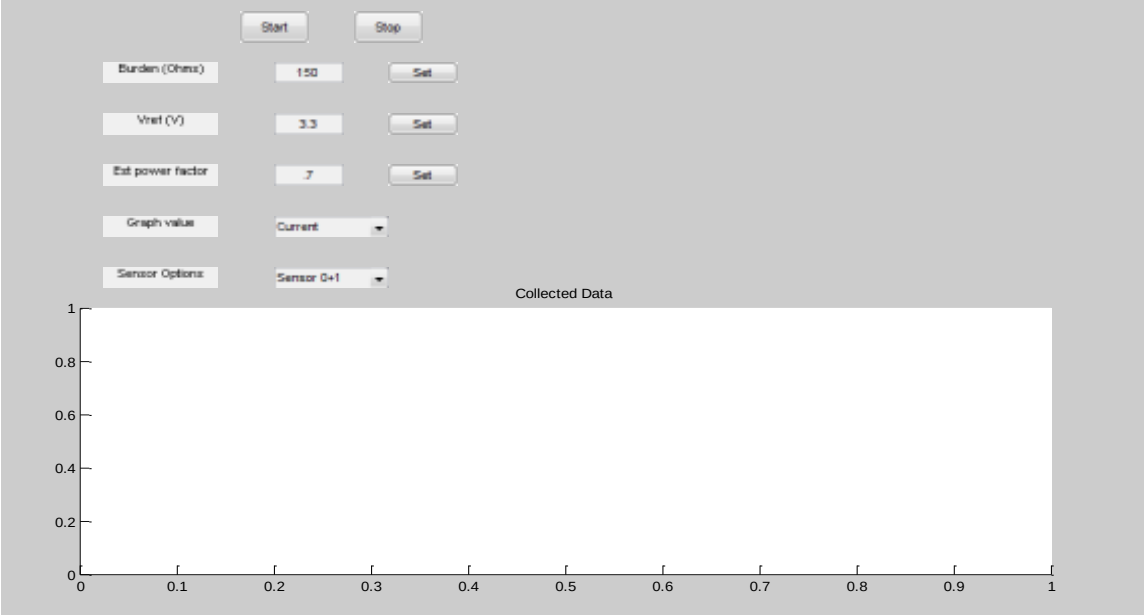

Figure 11. Matlab base station interface

5.2 Code

5.2.1 Arduino main file

```
/*
 * Modified_UDPApp
 * Reads analog ports 0 and 1, and sends a UDP packet with their value
 * Based on UDP endpoint app provided by AsyncLabs
 *
 * Written by Chris McNally - csm44@cornell.edu
 * Cornell University
 *
 *
 *
 * Version History :
 * Version      Author   Date      Comments
 * 1.0          CM       ?         Modified UDP endpoint example code to send packet at regular invervals.
 * 1.1          CM       ?         Added analog read portion,
 * 2.0          CM       4/26/10   Added time stamping, sending time and ADC readings in packet. Also
added max reading
 */

#include <WiShield.h>
```

```
#define WIRELESS_MODE_INFRA	1
#define WIRELESS_MODE_ADHOC	2

// Wireless configuration parameters ----------------------------------------
unsigned char local_ip[] = {192,168,1,100};	// IP address of WiShield
unsigned char gateway_ip[] = {192,168,1, 1};	// router or gateway IP address
unsigned char subnet_mask[] = {255,255,255,0};// subnet mask for the local network
const prog_char ssid[] PROGMEM = {"o_o"};		// max 32 bytes

unsigned char security_type = 3;	// 0 - open; 1 - WEP; 2 - WPA; 3 - WPA2

// WPA/WPA2 passphrase
const prog_char security_passphrase[] PROGMEM = {"dvor@k123"};	// max 64 characters

// setup the wireless mode
// infrastructure - connect to AP
// adhoc - connect to another WiFi device
unsigned char wireless_mode = WIRELESS_MODE_INFRA;

unsigned char ssid_len;
unsigned char security_passphrase_len;
//---------------------------------------------------------------
int maxReading1 = 0;
int maxReading2 = 0;
int tempReading1 = 0;
int tempReading2 = 0;
long readingTime1 = 0;
int packetSent = 0;
//long readingTime2 = 0;
//long tempTime = 0;

void setup()
{
      WiFi.init();
     //Serial.begin(9600);
}

void loop()
{
      //if we are on a new packet, assign a new time value
      if(packetSent)
```

```
    {
     readingTime1 = millis();
     //Serial.println(maxReading1);
     packetSent = 0;
     maxReading1 = 0;
     maxReading2 = 0;
    }
    int i;
    //Run for 20 samples to capture peak of voltage signal
    for(i=0;i<250;i++)
    {
     //Get ADC reading for both sensors
     tempReading1 = analogRead(0);
     tempReading2 = analogRead(1);
     //if either reading larger than max value seen since
     //last transmission, set as new max value and record
     //time
     if(tempReading1 > maxReading1){
        maxReading1 = tempReading1;
     }
     if(tempReading2 > maxReading2){
        maxReading2 = tempReading2;
     }
    }
    //run wifi routine
        WiFi.run();
}
```

5.2.2 UDP.c

```
#include "uip.h"
#include <string.h>
#include "udpapp.h"
#include "config.h"


#define STATE_INIT                  0
#define STATE_LISTENING     1
#define STATE_HELLO_RECEIVED        2
#define STATE_NAME_RECEIVED         3

//ADC measurements (range 0-1023)
extern int maxReading1;
```

```
extern int maxReading2;
extern long readingTime1;
//extern long readingTime2;
extern int packetSent;

static struct udpapp_state s;

void dummy_app_appcall(void)
{
}

void udpapp_init(void)
{
        uip_ipaddr_t addr;
        struct uip_udp_conn *c;
    //IP address to send packets to
        uip_ipaddr(&addr, 192,168,1,2);
    //create UDP connection to address at port given
        c = uip_udp_new(&addr, HTONS(12344));
        if(c != NULL) {
                uip_udp_bind(c, HTONS(12344));
        }
        s.state = STATE_INIT;
        PT_INIT(&s.pt);
}

static PT_THREAD(handle_connection(void))
{
        //mark beginning of protothreat
        PT_BEGIN(&s.pt);
        //translate data into string
        char data[100];
        sprintf(data,"%lu %u %u",readingTime1,maxReading1,maxReading2);
        //give string over to uIP, send packet
        uip_send(data,strlen(data));
    //reset max readings for next sampling period
    packetSent = 1;
    //mark end of protothread
        PT_END(&s.pt);
}

void udpapp_appcall(void)
```

```
{
    handle_connection();
}
```

5.2.3 Matlab interface

```
function PowerMeter
%Powermeter - GUI interface for Wireless Power Meter project
%Created by Chris McNally - csm44
%Cornell University
%%%%%%%%%%%%%%%%%%%%%%%%%%%%Revision history%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Date       Version       Notes
% 4-13-10     1.0          Initial version
% 5-11-10     2.0          Update graphical elements and new data packet
format
% 5-13-10     2.1          Changed graph axis to look better, added sensor
selector

%declare main figure window at position 300,300
%800x600 windows
meter = figure('Position',[ 5 5 1000 600]);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%layout parameters%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Graph
%powerGraphPos1 = [70 20 400 200];
powerGraphPos2 = [70 40 850 260];
%powerGraphPos3 = [70 560 400 200];

%%%%%%%%%Parameters%%%%%%%%%%%%
%reference voltage on arduino
vRef = 3.3;
%Burden resistance
bR = 150;
%Estimated power factor
pFactor = .7;
%Estimated mains voltage
Vrms = 121;
%#of samples to display on graph
windowSize = 100;
samplesCollected = 0;
%stop flag
keepTruckin = 1;
%1 = Current
%2 = Apparent power
%3 = Real power
whatToGraph = 1;
offset = 450;
%current data
timeData1 = [];
currentData1 = [];
%timeData2 = [];
currentData2 = [];
avg = '';

graphData = [];
graphTime = [];
```

```
%%%%%%%%Button positions%%%%%%%%%
startButtonPos = [660-offset 560 60 30];
stopButtonPos = [760-offset 560 60 30];
setRPos = [790-offset 500+20 60 20];
setVrefPos = [790-offset 450+20 60 20];
setPfPos = [790-offset 400+20 60 20];

%%%%Parameter UI positions%%%%%
rTextBoxPos = [690-offset 500+20 60 20];
rLabel = [540-offset 500+20 100 20];
vRefTextBoxPos = [690-offset 450+20 60 20];
vRefLabel = [540-offset 450+20 100 20];
pfTextBoxPos = [690-offset 400+20 60 20];
pfLabel = [540-offset 400+20 100 20];
avgTextBoxPos = [];
avgLabel = [];
graphSelectPos = [690-offset 350+20 100 20];
graphSelectLabelPos = [540-offset 350+20 100 20];
sensorSelectPos = [690-offset 300+20 100 20];
sensorSelectLabelPos = [540-offset 300+20 100 20];
%%%%%%%%%%%%%%%%%%%%%%%%%%%Object declarations%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%Graphs%%%%%%%%%
%powerGraph1 =
axes('Parent',meter,'Units','pixels','Position',powerGraphPos1);
%title('Sensor 1');
powerGraph2 =
axes('Parent',meter,'Units','pixels','Position',powerGraphPos2);
title('Collected Data');
%powerGraph3 =
axes('Parent',meter,'Units','pixels','Position',powerGraphPos3);
%title('Total');

%%%%%%%%%%Buttons%%%%%%%%%%
startButton = uicontrol(meter,'Position',startButtonPos,...
                'String','Start',...
                'Callback',@collectData);
stopButton = uicontrol(meter,'Position',stopButtonPos,...
                'String','Stop',...
                'Callback',@stopCollectData);

setRButton = uicontrol(meter,'Position',setRPos,...
                'String','Set',...
                'Callback',@setR);

setVButton = uicontrol(meter,'Position',setVrefPos,...
                'String','Set',...
                'Callback',@setV);

setPFButton = uicontrol(meter,'Position',setPfPos,...
                'String','Set',...
                'Callback',@setPF);
%%%%%%%%Text fields%%%%%%%%
%%%User definable%%%%
burdenLabel = uicontrol(meter,'Style','text',...
                'String','Burden (Ohms)',...
```

```
                'Position',rLabel);

burdenR = uicontrol(meter,'Style','edit',...
                'String','150',...
                'Max',1,'Min',0,...
                'Position',rTextBoxPos);

voltageLabel = uicontrol(meter,'Style','text',...
                'String','Vref (V)',...
                'Position',vRefLabel);

voltageReference = uicontrol(meter,'Style','edit',...
                'String','3.3',...
                'Max',1,'Min',0,...
                'Position',vRefTextBoxPos);

pfLabel = uicontrol(meter,'Style','text',...
                'String','Est power factor',...
                'Position',pfLabel);

powerFactor = uicontrol(meter,'Style','edit',...
                'String','.7',...
                'Max',1,'Min',0,...
                'Position',pfTextBoxPos);

graphSelectLabel = uicontrol(meter,'Style','text',...
                'String','Graph value',...
                'Position',graphSelectLabelPos);


graphSelect = uicontrol(meter,'Style','popupmenu',...
                'String',{'Current','Est App Pow','Est Real Pow'},...
                'Value',1,'Position',graphSelectPos);

sensorSelectLabel = uicontrol(meter,'Style','text',...
                'String','Sensor Options',...
                'Position',sensorSelectLabelPos);

sensorSelect = uicontrol(meter,'Style','popupmenu',...
                'String',{'Sensor 0+1','Sensor 0','Sensor 1'},...
                'Value',1,'Position',sensorSelectPos);
%%%Outputs%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Functions%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %receives UDP packets from Arduino
%     %function data = receiveData()
%
%     end
%
%     %grabs new data and graphs data
%     function updateGraph()
%
%     end
```

```
    %loops, checks whether to collect data or not
    function collectData(hObject,eventdata)
        keepTruckin = 1;
        %grab new packet
        import java.io.*
        import java.net.DatagramSocket
        import java.net.DatagramPacket
        import java.net.InetAddress
        %receieve port, set on Arduino. Default is 12344.
        port = 12344;
        socket = DatagramSocket(port);
        %timeout if no packets are received (milliseconds)
        timeout = 1000;
        %estimated length of incoming packet
        packetLength = 100;
        socket.setSoTimeout(timeout);
        while(keepTruckin)
            try
                %grab the packet
                packet =
DatagramPacket(zeros(1,packetLength,'int8'),packetLength);
                %close socket
                socket.receive(packet);
                %extract data from packet
                mssg = packet.getData;
                mssg = mssg(1:packet.getLength);
                newData = mssg;
            catch receiveError
                % Determine whether error occurred because of a timeout.
                if
~isempty(strfind(receiveError.message,'java.net.SocketTimeoutException'))
                    errorStr = sprintf('Failed to receive UDP packet;
connection timed out.\n');
                else
                    errorStr = sprintf('Failed to receive UDP packet.\nJava
error message follows:\n%s',receiveError.message);
                end % if
                try
                    socket.close;
                end % try
                error(errorStr);
            end % try
            %sort new data
            parsedNewData = cellstr(char(newData'));
            parsedNewData = parsedNewData{1};
            parsedNewData = sscanf(parsedNewData,'%d');
            %add new power data to data set
            timeData1 = [timeData1 ; parsedNewData(1)];
            currentData1 = [currentData1 ; parsedNewData(2) ./
(1023.*sqrt(2)*bR) .* vRef .* 3100];
            %timeData2 = [timeData2 ; parsedNewData(3)];
            currentData2 = [currentData2 ; parsedNewData(3)  ./
(1023.*sqrt(2)*bR) .* vRef .* 3100];
            whatToGraph = get(graphSelect,'Value');
            switch whatToGraph
                %%%%%%%%Graphing current data%%%%%%%%
                case 1
```

```
                    %samplesCollected = samplesCollected + 1;
                    if(windowSize > samplesCollected)
                        graphData =  currentData1 + currentData2;
                        graphTime = timeData1;
                    else
                        graphData = currentData1(samplesCollected+1-
windowSize:end) + ...
                            currentData2(samplesCollected+1-windowSize:end);
                        graphTime = timeData1(samplesCollected+1-
windowSize:end);
                    end
                    yString = 'RMS current (amps)';
                    xMax = 80;
                %%%%%Graphing apparent power%%%%%%%%
                case 2
                %Apparent power = Vrms * Irms
                    if(windowSize > samplesCollected)
                        graphData =  (currentData1 + currentData2) .* Vrms;
                        graphTime = timeData1;
                    else
                        graphData = (currentData1(samplesCollected+1-
windowSize:end,:) + ...
                            currentData2(samplesCollected+1-
windowSize:end,:)) .* Vrms;
                        graphTime = timeData1(samplesCollected+1-
windowSize:end);
                    end
                    yString = 'Apparent power (VA)';
                    xMax = max(graphData);
                %%%%%%graphing real power%%%%%%%
                case 3
                    %Real power = Apparent power * power factor
                    if(windowSize > samplesCollected)
                        graphData =  (currentData1 + currentData2) .* Vrms
.*pFactor;
                        graphTime = timeData1;
                    else
                        graphData = (currentData1(samplesCollected+1-
windowSize:end,:) + ...
                        currentData2(samplesCollected+1-windowSize:end,:)).*
Vrms .*pFactor;
                        graphTime = timeData1(samplesCollected+1-
windowSize:end);
                    end
                    yString = 'Real power (W)';
                    xMax = max(graphData);
            end
            samplesCollected = samplesCollected + 1;
            cla;
            plot(powerGraph2,graphTime,graphData);
            title('Collected Data');
            xlabel('time from power up (ms)');
            ylabel(yString);
            axis(powerGraph2,[min(graphTime) max(graphTime)+1 0 xMax+1]);
            axis manual;
            drawnow;
        end
```

```
        socket.close;
    end

    %sets stop flag
    function stopCollectData(hObject,eventdata)
        %grab current time
        c = datestr(now, 'mmm dd, yyyy HH MM SS');
        %save workspace
        save(c);
        keepTruckin = 0;
        samplesCollected = 0;
        currentData1 = [];
        currentData2 = [];
        timeData1 = [];
        graphData = [];
        graphTime = [];
        cla;
    end

    function setR(hObject,eventdata)
        bR = str2num(get(burdenR,'String'));
    end

    function setV(hObject,eventdata)
        vRef = str2num(get(voltageReference,'String'));
    end

    function setPF(hObject,eventdata)
        pFactor = str2num(get(powerFactor,'String'));
    end

end
```

References

[1] R. Nave. Household Wiring. http://hyperphysics.phy-astr.gsu.edu/HBASE/electric/hsehld.html , 1999

[2] Arduino. http://arduino.cc/en/Main/ArduinoBoardDuemilanove. 2009

[3] CR Magnetics. CR-3110 data sheet. http://www.crmagnetics.com/products/Assets/ProductPDFs/CR3100.pdf. 2010.

[4] CR Magnetics. CR-3110 data sheet. http://www.crmagnetics.com/products/Assets/ProductPDFs/CR3100.pdf. 2010.

[5] Wayne Storr. Full-wave Rectifiers. http://www.electronics-tutorials.ws/diode/diode_6.html. 2010

[6] Rod Elliot. Precision Rectifiers. http://sound.westhost.com/appnotes/an001.htm. 2009.

[7] Adam Dunkels. http://www.sics.se/~adam/uip/uip-1.0-refman/.

[8] Arduino Duemilanove. http://arduino.cc/en/Main/ArduinoBoardDuemilanove. 2009

[9] AsyncLabs. http://asynclabs.com/wiki/index.php?title=WiShield_1.0 2009