Design for an Internet-Capable DC/AC Power Supply Using Renewable Energy

Junior Design Team D

Charles Boling

Brad Finkbeiner

William Laub

Luis Marquez

Jerry Quiroga

Dr. Erives
Introduction to Design (EE 382)
New Mexico Institute of Mining and Technology
May 11, 2010

Abstract

Renewable energy is an increasingly important source of electrical power for economic and environmental reasons. A device capable of converting the intermittent and variable direct current supplied by wind turbines and photoelectric cells into power suitable for consumer electronics could find applications in industry, home power, and academic research into alternative energy. This spring, teams of students in the New Mexico Tech electrical engineering department's Introduction to Design class were tasked with constructing such a power supply and outfitting it with the capacity to monitor its own power output characteristics in real time. Although Team D made limited progress towards the implementation of the power supply it designed, subsystems capable of measuring and broadcasting power data on the internet have been completed and tested. This paper provides an overview of the motivation and customer specifications for an intelligent, internet-capable DC to DC/AC power converter, followed by an analysis of Team D's design, implementation and testing procedures for each device subsystem.

Index Terms

Renewable energy, Power inverter, Data monitor, Power measurement

Acknowledgments

Thanks to Dr. Hector Erives and Dr. Kevin Wedeward, the advisors and sponsors for this project, for devising an inventive and challenging design specification which drew on many aspects of our electrical engineering education.

Many thanks to Tom McConnell for the donation of an oscilloscope, function generator, and countless other pieces of equipment and components, as well as advice and support.

Thanks to Pat Nystrom for invaluable advice and training in embedded programming.

Thanks to Carrol Teel for ordering components, reserving lecture halls, and being persistently helpful during the course of the semester.

Thanks to Chris Pauli for his consultation and assistance on interacting with the network and webservers he proficiently maintains, as well as making labs and resources available on more than one occasion.

Thanks to Andy Tubesing for his loans of components and materials, his maintenance of labs and equipment, and his advice on practical implementation of an alternative-energy power supply, particularly his comments on the necessity of continuous battery power.

Thanks to Dr. Rene Arechiga, Dr. Robert Bond, Jamey Christy, Dr. Aly El-Osery, Dr. Hector Erives, Dr. Anders Jorgensen, Dr. William Rison, Dr. Scott Teare, Dr. Ronald Thomas, Andy Tubesing, and Dr. Kevin Wedeward for providing lectures and lab exercises which provided us with the instruction necessary to arrive at our present facility with engineering design and implementation.

Thanks to the New Mexico Tech Computing Center for offering free webspace to all students and for graciously setting up a MySQL database for use with this project.

Thanks to the GNU foundation and the mspgcc project for providing free software tools and example code for the MSP430 microcontroller family.

Thanks to the developers of the WireShark packet analyzer for providing a free and robust tool for observing local network traffic.

Contents

List of Tables

List of Figures

I. Introduction

Sources of renewable electrical energy, such as wind turbines and photoelectric cells, are becoming an increasingly popular and important means of producing power. Between 2004 and 2008, the production capacity of photovoltaic panels worldwide increased by a factor of six, and the capacity of wind turbines increased by 250%[5]. This growing popularity is the result of a number of desirable characteristics of green power. Since devices which harness alternative energy draw power from ongoing natural processes, their operation does not incur the costs of fuel and waste disposal which are associated with conventional electric generators. For the same reason, energy sources such as wind turbines cause much less long-term or irreversible damage to the environment than power generation techniques which rely on fossil fuels. Mounting evidence of global climate change continues to motivate the adoption of renewable energy, and the development of reliable green power sources for small-scale home use is a topic of persistent interest to industry and research professionals.

As these generators become more efficient and cost-effective, it is likely that their usage will become even more widespread. However, the power generated by these devices is typically not suitable for consumer electronics. Wind turbines and solar panels typically produce direct-current voltages which may vary widely or become entirely unavailable with environmental conditions. To use these kinds of renewable energy sources on practical loads, it is necessary to minimize downtime by using battery-backup and to regulate the output power of wind and solar devices so that it conforms to standardized characteristics. In particular, 12 volt DC power is used by many consumer products, such as those intended to operate off of car batteries. Additionally, a vast majority of home electronics are designed to operate on alternating-current supplies. A robust alternative-energy power supply should therefore include a high-efficiency DC to AC power inverter capable of providing an output analogous to home wall power (60 hertz, 120 volts RMS in the United States). Furthermore, since devices such as solar panels produce intermittent and variable power, it is important for a supply to measure and log some of the characteristics of the power it produces.

The specification for this spring's Introduction to Design (EE 382) project separates the design of an intelligent power converter into five main modules[6]. The smart battery charger uses power provided by photovoltaic panels and a wind turbine to create and maintain a charge on a 12 volt lead-acid battery without causing damage to the energy sources or the battery. Power from the battery is then used to power other parts of the device. A majority of the power provided to the battery is subsequently consumed by the power supply module, which produces a regulated 12 volt DC output and generates a 120 volt sinusoidal AC output. Some smaller amount of current is used to power an internet-capable microcontroller, as well as signal conditioning circuitry which yields signals the microcontroller can measure.

Since the outputs of the device are expected to supply rather high currents, it is critical that the supply be equipped with a number of safety precautions. Specifications require that the lines which connect to each input device and power output use one-ampere fuses and commercial power connectors. High-power circuitry is contained in a grounded metal enclosure, uses printed circuit boards to ensure consistent operation, and possesses emergency shutoff switches.

II. Background

DC to AC power inverters are well-researched and ubiquitous technology. Modern inverters are capable of very high efficiencies (95% or more), large power outputs (often several thousand watts), and find applications in a wide range of commercial and industrial applications. Inverters allow power companies to generate AC transmission voltages from the DC power they produce and allow consumers to power home electrical appliances in their vehicles. In the digital age, power inverters are often used in uninterruptible power supplies to convert DC battery voltages into temporary backup power for computers and other sensitive devices. Small units use inverters to provide uninterruptible power, while devices for large-scale datacenters often use full-scale generators to guarantee minimal downtime. Often, uninterruptible power supplies are equipped with a software interface which allows these devices to remotely shut down sensitive devices such as computers which rely on them for power [7]. Inverters used for renewable energy harvesting often use computer interfaces for similar purposes, but very few (if any) existing inverter technologies can be directly interfaced with the internet. "Intelligent" power inverter technology typically refers to the computer-controlled efficiency and reliability available in these devices. Internet-capable real-time data logging allows the power characteristics of an inverter to be visible to users anywhere in the world. Additionally, a dynamic webpage which can monitor the output of a power supply can electronically notify users of emergency conditions. These advantages make devices such as those described in this project specification a worthwhile and novel area of research.

III. Design

Although progress has been made on each major design subsystem, those areas which focus on data acquisition and transmission are considerably more complete than those directed towards power adaptation. The three subsystems which deal with measuring, logging, and broadcasting the power characteristics of the device's hypothetical output are presented first, despite their reliance in practical operations on the other modules, because of their more advanced development.

A. Power Measurement Module

The measurement module provides three signals containing voltage and current RMS information, output frequency, and the phase shift between output voltage and current. It is divided into three submodules: voltage, current, and logic, and each module provides one signal. The measurement module is powered entirely by the MSP430 microcontroller, ensuring that its output logic levels match the logic levels the microcontroller expects and minimizing the number of supply rails required to power the circuit.

A.1 Voltage Submodule

The voltage and current modules are nearly identical. Each must provide a voltage signal that does not fall outside the range 0 to 3V. The voltage module generates a signal scaled and centered about 1.5V using the circuit in Figure 1 (p. 20). Superposition analysis shows that the 170V peak input signal is scaled to have a peak voltage of

$$V_{o1} = 170\frac{1.2k||1.2k}{100k + 1.2k||1.2k}V \tag{1}$$

$$V_{o1} = 1.01V \text{ peak}$$

The output for the 3V DC input is

$$V_{o2} = 3\frac{100k||1.2k}{100k||1.2k + 1.2k}V \tag{2}$$

$$V_{o2} = 1.49V$$

The total output is then a maximum 2.02V peak to peak signal centered about 1.49V, which is within the required 0 to 3V range.

The MSP430's built-in analog-to-digital converter has a low input impedance (≈2kΩ), and the scaling and offsetting block has high output impedance (≈1.2kΩ). The offset and scaling factor for the circuit change significantly when the output is attached directly to the ADC, so an emitter-follower buffer is added to give a low output impedance. The buffer shifts the whole signal down by .5 to .6V, giving a new signal with a maximum output swing from 0V to 2V. The output of the follower is biased at about 1V, giving an emitter current of 303μA and r_e of about 85Ω. The total output resistance is less than 100Ω, which is much smaller than the input resistance of the ADC. There is some concern that the bottom of the voltage signal may be too low for the buffer at maximum voltage, resulting in harmonic distortion. In the future, the circuit could be modified to increase the DC offset in the first stage and eliminate the problem.

A.2 Current Submodule

The current stage works in a similar manner, except that it takes a small input signal generated by a current sense resistor. The current sense resistor has a value of .1Ω, and the current at full power has a peak of about 5.5A. This gives a current signal with a .55V amplitude. The signal is conditioned for input to the ADC by the scaling and offsetting block of Figure 2 (p. 20). Analysis by superposition shows that the output for the current signal input is

$$V_{o1} = .55\frac{10k}{10k + 10k}V \tag{3}$$

$$V_{o1} = .275V \text{ peak}$$

and the output for the 3V DC input is

$$V_{o2} = 3\frac{10k}{10k + 10k}V \tag{4}$$

$$V_{o2} = 1.5V$$

The total output is centered at 1.5V and has a maximum swing of 1.225 to 1.775V, which is well within the ADC input range.

The current signal offsetting circuit has a very high output impedance of 10kΩ, and also required a buffer. The buffer in this circuit is identical to the buffer in the voltage submodule and gives an output signal with an offset of .9 to 1V.

The voltage signal varies by a very small amount, and still falls well within the operating range of both the ADC and the buffer.

A.3 Logic Submodule

The logic submodule is shown in Figure 3 (p. 21). It takes input from the voltage and current submodules and outputs a logic signal that can be measured by the MSP430 input capture. The signal has a frequency equal to the frequency of the voltage signal and a duty cycle that varies linearly from 0 to 100% as the phase shift varies from 0 to 2π radians. An LM339 comparator compares the voltage and current signals to two adjustable references set equal to the signal offsets to generate two square waves with 50% duty cycle. The LM339 outputs only 0V and high impedance, so the output is pulled high using a 2.7k resistor and then buffered using a 7414 NOT gate, which drives both high and low outputs.

Since the signals given to the comparator can be very small, noise in the signals and the comparator references can introduce rapid transitions in the comparator output, resulting in a noisy logic signal. Each signal is passed through identical passive low pass filters followed by Schmitt triggered logical NOT gates. This introduces to each signal a roughly identical phase shift, but removes the high frequency noise.

The output from the NOT gate is passed through a passive high-passive filter to generate short pulses at rising and falling edges. That pulse signal goes to another Schmitt triggered NOT gate to generate a single clean, short pulses a few microseconds long on each rising edges. At this point in the circuit, there are two pulse trains. One corresponds to rising zero-crossings in the voltage signal, and the other corresponds to rising zero-crossings in the current signal.

The voltage signal pulse is given to the S input of an SR latch, and the current signal pulse is given to the R input of an SR latch. The latch output signal has a rising edge when the voltage rises above zero volts and a falling edge when the current rising above zero volts. When the current and voltage are perfectly in phase, the SR latch is set and then immediately reset or reset and then immediately set once every period, giving nearly zero phase shift.

There is the possibility that the current and voltage pulses occur at exactly the same time. In this case, the logic signal behaves erratically, but there is sufficient noise in the system that at worst the microcontroller will receive a few erroneous readings, of which no more than one is likely to be logged.

A.4 Design Alternatives

The current measurement module was originally intended to measure current using induction. Inductive current measurement does not modify the output voltage like a current sense resistor does, but it requires more complex circuitry to extract and amplify the signal, and it introduces a phase shift that would have to be characterized and accounted for in the fest of the design. The resistive current sensor was chosen for its simplicity and its well-understood behavior.

Earlier versions of the logic submodule used a single AND gate to combine the current and voltage square waves. The design was simpler, but limited in its ability to distiguish between the current and voltage signals. Though more complex, the current design was chosen because it is capable of measuring any phase shift.

The voltage and current submodules were originally designed to amplify and offset the inputs using op-amp circuits. The op-amps tested with the circuit (741 and LF411) both operated poorly with the given supply rails. The passive resistor circuits now used are simpler, and their high output impedances are easily compensated using emitter followers.

A.5 Components

The current sense resistor for the current module needed to be large enough to provide a measurable voltage signal, but small enough to minimize its impact on the inverter output. A .1Ω resistor gives a peak voltage of .55V at 5.5A, which is only .32% of the 170V peak output. Noise in the measurement system results in a minimum measurable current of about 10mA peak, which corresponds to a maximum load of 17kΩ on the 120V AC output.

The LM339 is a single chip that provides four comparators and operates on a single small supply rail. Only two comparators are needed, but an LM339 was readily available during early design and testing. Its low power requirements are critical in the single-rail design of the measurement circuit. The chip has a very short response time, which makes it susceptible to noise in the input signals[10]. That noise is filtered at a later stage, but since the input signal has a very low frequency, it might have been reasonable to use an op-amp as a comparator and take advantage of its limited slew rate to prevent output noise entirely.

The 74HC14 is an ideal chip for the logic signal filtering stage of the circuit. The circuit requires six NOT gates, all of which are provided by the 7414. In addition, the chip utilizes Schmitt triggers to reduce the effect of noise on the input. The Schmitt triggers are convenient in filtering the high frequency logic noise from the original signal, and they provide switching levels that allow the two pulse generators in the signal to be designed for a specific pulse length. The chip has a combined propagation delay and transition time of about 60ns, so it is more than capable of generating the $1\mu s$ pulses desired from the pulse generators[3].

The 7400 series of logic chips provides a few options for SR latches such as the 74118 hex SR latch; however, a 7400

Quad NAND gate chip was readily available during design and testing, and was used to implement the SR latch. The chip still has 2 available NAND gates, which could be used in a more advanced logic design to improve the fidelity of the output logic signal (for example: in the ideal case, the output signal at zero phase sift should be 0. The inverting output of the SR latch could be NANDed with the voltage pulse to ensure that a minimum pulse is always present).

The two emitter-follower buffers are implemented using 2N3904 NPN BJT transistors. The emitter-follower is a robust circuit that requires only a high β in the transistor. The 2N3904 was selected because it meets that criterion and was readily available.

B. Data Logger Module

The data logger module serves to translate electrical signals generated by the measurement module into computerized numerical data. This information can then be used to compute human-readable power data. It is important to ensure that power data is acquired and transmitted efficiently, since accurate measurements rely on very rapid sampling of the source signals. As a result, intensive computational operations, such as multiplication and division, are not performed in this module unless absolutely necessary.

B.1 Computation of Frequency and Phase

The measurement module provides the data logger with a 0 to 3 volt logic signal for the purpose of computing frequency and phase. Since the logic signal has the same frequency as the sinusoidal voltage and current waveforms from the power supply, this signal may be used to measure the period (and therefore the frequency) of the AC power being supplied by the device. Time between rising edges of the logic waveform is measured accurately through use of an input-capture peripheral, resulting in a software representation of the period of the power waveforms. In addition to providing information to users about the frequency of the device's output power, knowledge of the period of these waveforms proves internally useful to the module.

The square wave generated by the logic stage has a duty cycle directly proportional to the phase shift between the inverter's voltage and current output waveforms. Therefore, the input-capture peripheral is configured to capture both rising and falling edges, and the time between subsequent edges is recorded in order to compute duty cycle. Times recorded by the frequency and phase logging software are integer values which represent the number of microcontroller clock cycles that have elapsed between each edge. This simple time measurement scheme requires no mathematical operations more complex than subtraction, allowing input-capture interrupts to execute for as little time as possible.

B.2 Monitoring Peak Waveform Values

An analog-to-digital converter built into the microcontroller allows for software to measure instantaneous points on any analog signal which swings strictly within the range of 0 to 3 volts. This allows for trivial measurement of DC power characteristics: DC voltages are simply scaled to within this range and measured directly, while DC currents are passed through an appropriate current-sense resistor to generate a measurable voltage signal. The task of monitoring AC power, however, is somewhat more sophisticated than logging the battery and DC output voltages.

For AC signals, particularly sinusoids, magnitude is often expressed as a root-mean-square average of the waveform. This average simplifies the computation of power dissipated by a resistive load and provides a convenient method for assessing the average of a signal whose arithmetic mean is zero. For a periodic continuous-time signal $V(t)$ with period T, it is given by

$$V_{RMS} = \sqrt{\frac{1}{T}\int_0^T V(t)^2 dt}. \tag{5}$$

For k discrete-time samples $V_i, i \in \{1, 2, ..., n\}$ of a waveform, however, the formula is

$$V_{RMS} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} V_i^2}. \tag{6}$$

The peak value of a sinusoid is also straightforwardly computed from its RMS average by

$$V_{peak} = \sqrt{2}V_{RMS}. \tag{7}$$

In this case, the desired waveforms are superimposed on constant voltage offsets. It is difficult to directly measure the root-mean-square average of these small sinusoidal waveforms. However, direct measurements using the analog-to-digital converter channels allow for readings of discrete points on the offset signals. That is, although it is not straightforward to measure the small-signal component $v(t)$ of a voltage or current signal provided by the measurement circuitry, it is simple to take measurements of the waveform $V + v(t)$, where V is a constant (but unknown) voltage offset. Input-capture interrupts provide information on the exact times when a new period of each waveform begins, so it is possible for us to take some integer number n measurements of $V + v(t)$. Let $(V + v_i)$ denote the i-th such sample, where $i \in \{1, 2, ..., n\}$. Then the quantities

$$\sum_{i=1}^{n} (V + v_i) \tag{8}$$

and

$$\sum_{i=1}^{n} (V + v_i)^2 \tag{9}$$

are easily computed over a given period. Given only this information, it is possible to compute the root-mean-square average. However, due to the computational complexity of operations such as division and square-root, this arithmetic is performed by server-side scripts following network transmission of the above sums.

B.3 Design Alternatives

Originally, an implementation was considered using only the MSP430's analog-to-digital converter peripheral. The software written for this scheme attempted to determine the slope of the signal, using the zeroes of the slope to find peak and trough points on the input waveform. However, noise in the ADC required that slope values close to zero, rather than strictly zero, be considered indications of waveform extrema. An appropriate tolerance for these measurements proved difficult to find and this method produced inconsistent frequency readings.

Difficulties with ADC-only data logging software precipitated the development of a comparator-based system for measuring time-sensitive power characteristics. Microcontrollers are significantly better equipped to deal with digital signals than analog signals, and the presence of an input-capture peripheral on the MSP430 provides a means for accurate measurement of logic waveform timing characteristics. Since the final scheme uses comparators to detect zero-crossings rather than peaks of an input signal, an alternative method for measuring waveform peaks was devised. Measuring root-mean-square averages of voltage and current signals is a useful and consistent method for determining their magnitudes, since peak values can straightforwardly be computed from RMS values.

When RMS measurements were first being implemented, the software attempted to remove the constant voltage offset from each input signal before appending new points to the mean-square. This offset was measured externally and hard-coded into the microcontroller, causing problems when environmental conditions or variations in the power supply would affect this offset. Computing the time average of each waveform allows this offset to be measured once each period, allowing for more consistent readings.

Despite extremely accurate frequency and power factor measurements and rather consistent readings of root-mean-square, RMS values for voltage and current may still deviate from measurements made with a digital oscilloscope by as much as 10%. It was suspected that this error could be mitigated by using a higher sampling rate. However, since the MSP430's largest integer size is 32-bit and the ADC takes 12-bit readings, it is quite possible to overflow the variable used to store a sum of squares for either waveform. To deal with this problem an additional counter was implemented to keep track of the number of overflows, thereby allowing a server-side script with access to 64-bit integer arithmetic to deal with a higher number of samples.

B.4 Components

All data logging (as well as network transmission) is handled by a simple, commercially manufactured microcontroller board with integrated ethernet hardware. The board's two pin headers allow for simple connection with the measurement circuits. It expects an input between 6 and 9 volts DC, but a "wall-wart" transformer has been used to adapt wall power during programming. This same power supply could be connected to the AC power output of the completed device to avoid the need for voltage regulation down from 12 volts DC.

A JTAG hardware programmer was purchased from the board's manufacturers along with the board. This simple device allows for parallel-port programming and debugging of the microcontroller over a 14-pin JTAG header on the board, while avoiding the expense of a full-featured MSP-family hardware programmer from Texas Instruments. A few non-proprietary software packages exist for programming MSP430 microcontrollers over such an interface, and the GNU-sponsored mspgcc suite was chosen for its full-featured command line interface. Since mspgcc offers a Windows command-line compiler with usage analogous to GCC and a special version of the popular GDB debugger, it provides a simple and familiar development environment.

The EasyWeb board was chosen over other network solutions for simplicity and power economy. The use of a board with built-in ethernet hardware and freely available IP stack code eliminated the need for difficult and error-prone peripheral hardware interfacing. Although instruction sets, peripherals and libraries available on other microcontrollers can greatly simplify the implementation of network code, the MSP430 was determined to be a cost effective and very low-power choice.

C. Internet Interface Module

The internet interface consists of three main stages. First, C code on the microcontroller generates a string containing the integer values which represent measured power characteristics. This string is used to query a server-side PHP (`logger.php`) or Python (`logger.cgi`) script which parses integers out of the string and performs the floating-point arithmetic necessary to translate them into human-readable numbers. The logger script next writes the computed floating-point values into a MySQL database, where they may be read by the user interface page `display.php`. The newest set of power data is written into the database approximately once a second as long as the microcontroller board has a working internet connection.

C.1 Encapsulating and Transmitting Data

The HTTP GET method allows for simple transmission of text data from a client to a dynamic webpage. HTTP GET packets allow clients to request resources, such as webpages or other files, from servers, and it is how webpages are typically accessed by a browser. In addition to transmitting the URI of a resource to be accessed, clients may send named arguments to a server for use with the requested resource. For instance, a web browser might attempt to access a dynamic webpage via the following URI:

```
http://geek.nmt.edu/~aleph0/logger.php?argument1=0&argument2=1
```

This URI specifies first the protocol (`http://`), the name of the server (`geek.nmt.edu`), and the path in the directory tree to the desired resource (`/ aleph0/logger.php`). After this, the `?` symbol indicates that subsequent characters are arguments from the HTTP GET method. The argument name is given, followed by an `=` sign and the value of the argument. Additional argument-value pairs are delimited by `&` symbols.

HTTP GET proves a very simple means of transmitting data to a server-side script because it is possible to complete an entire network transaction using very brief plaintext strings. For backwards-compatibility, most (if not all) servers support GET requests as originally implemented in the oldest HTTP version, HTTP/0.9. In this version, requests were required only to contain the word GET and the request string as shown above. Using this older standard helps minimize the amount of time spent by the microcontroller performing time-consuming string manipulation.

After the microcontroller generates an HTTP GET string containing the argument names the server expects and integers corresponding to the most recent relevant power characteristics, it opens a connection on TCP port 80 (HTTP) and attempts to access a server and script whose names are hard-coded into the MSP430. If a connection is successfully established, the HTTP GET request is transmitted, along with a request to immediately close the connection. In the event of successful transaction termination or a connection timeout, the microcontroller attempts to open another connection with the same server and resource. Repeatedly restoring the connection in this way ensures compatibility and maximizes the reliability of the internet connection, with some sacrifice in speed.

C.2 Computing Power Characteristics

A webpage running on the selected web server passively listens for HTTP connections, parses out the transmitted integers stored as strings, and converts them back to integer data types. The received arguments are: microcontroller clock rate, period length, logic signal high time, scaled ADC readings for DC voltage and current measurements, number of ADC readings taken in a period, the sum of all ADC readings for voltage and for current in that period, and the sum of squares of all ADC readings for voltage and current in the period. From these values it is possible for the server script to compute frequency, power factor, DC voltage, DC current, and AC RMS voltage and current.

Phase is the simplest to compute, since arguments to the script include both the period of the logic signal and the amount of time the waveform stays in a logic high state. Dividing the high time by the total period gives the duty cycle of this logic waveform, which by design of the measurement stage has a direct relationship with phase shift — zero duty cycle corresponds to zero phase shift, while 100% duty cycle corresponds to a phase shift of 2π radians. This phase angle θ is then used to compute power factor S by

$$S = |\cos\theta| \tag{10}$$

and contains information about whether current lags voltage or vice-versa. Since the clock rate of the microcontroller is provided to the server, it is straightforward to compute frequency as well. The period is provided as a number of clock cycles between rising edges, so the frequency may be computed by

$$frequency = \frac{\frac{clock\ cycles}{second}}{clock\ cycles}. \tag{11}$$

RMS measurement of voltage and current requires some slightly more complicated mathematics and relies on the fact that the desired waveform is a sinusoid superimposed on a constant offset. The server-side script receives the sums

$$\sum_{i=1}^{n}(V+v_i) \tag{12}$$

and

$$\sum_{i=1}^{n}(V+v_i)^2 \tag{13}$$

from the microcontroller as integers. The root-mean-square average of both current and voltage waveforms is then computed from these values as follows.

First note that

$$\sum_{i=1}^{n}(V+v_i)^2 = \sum_{i=1}^{n}(V^2+2Vv+v_i^2) = \sum_{i=1}^{n}V^2 + \sum_{i=1}^{n}2Vv_i + \sum_{i=1}^{n}v_i^2. \tag{14}$$

V is a constant offset, so

$$\sum_{i=1}^{n}V^2 = nV^2 \tag{15}$$

and

$$\sum_{i=1}^{n}2Vv_i = 2V\sum_{i=1}^{n}v_i. \tag{16}$$

But $v_i, i \in \{1, 2, .., n\}$ are samples of a sinusoidal voltage signal over a whole period, so their average

$$\frac{1}{n}\sum_{i=1}^{n}v_i = 0. \tag{17}$$

Therefore

$$\frac{1}{n}\sum_{i=1}^{n}(V+v_i)^2 = \frac{1}{n}(nV^2) + \frac{1}{n}(2V\sum_{i=1}^{n}v_i) + \frac{1}{n}\sum_{i=1}^{n}v_i^2 = V^2 + \frac{1}{n}\sum_{i=1}^{n}v_i^2, \tag{18}$$

or

$$\frac{1}{n}\sum_{i=1}^{n}v_i^2 = \frac{1}{n}\sum_{i=1}^{n}(V+v_i)^2 - V^2. \tag{19}$$

However, since the average of a sinusoid is zero,

$$\frac{1}{n}\sum_{i=1}^{n}(V+v_i) = \frac{i}{n}\sum_{i=1}^{n}V + \frac{i}{n}\sum_{i=1}^{n}v_i = \frac{1}{n}(nV) = V. \tag{20}$$

The root-mean-square average of the desired sinusoid may then be computed by

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}v_i^2} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(V+v_i)^2 - \left(\frac{1}{n}\sum_{i=1}^{n}(V+v_i)\right)^2}. \tag{21}$$

However, the units of this RMS reading are dependent on the resolution of the microcontroller's analog-to-digital converter. An RMS voltage can then be computed from the computed value $V_{reading}$ by

$$V_{RMS} = 3\frac{V_{reading}}{2^{12}-1} \tag{22}$$

for a twelve-bit analog-to-digital converter. Finally, it is necessary to multiply this voltage by the inverse of the downwards scaling factor inherent in the measurement circuitry.

C.3 Storing and Displaying Power Information

Once all the desired power characteristics have been computed as floating-point values, the logger script makes a connection to a remote MySQL database. Power information is inserted into a table which has been configured to store a small number of fixed-size decimal numbers. Insertion therefore forces truncation of the floating point numbers computed by the script. MySQL allows database administrators to separately choose the number of decimal places in the whole-number and fractional parts, so the entry for each table element can be sized to accomodate the degree of precision necessary for the field. Whenever a simultaneous listing of power characteristics is written into a row of the database, the current time is recorded in the row for future reference.

Since the MySQL database is hosted remotely by the New Mexico Tech Computing Center, data written into the table can be read from any internet capable machine. This allows the display script to show users the most recent power information regardless of what server originally computed and logged the power characteristics, and this feature proved useful during testing. The display script queries the MySQL database containing power data, retrieves the most recently entered row, and prints a simple text page with the data and the time it was logged.

C.4 Design Alternatives

A fair amount of time and effort was originally invested into research and attempted implementation of the secure shell (SSH) protocol, for handling data transmission. This protocol has the advantage of being very secure, and many servers — including those maintained by the electrical engineering department — allow clients which connect over this protocol to freely execute shell commands with user privileges. The encryption algorithms required by the secure shell standard, however, are significantly computationally complex, and the limited processing and memory resources of the MSP430 microcontroller make implementation of an SSH client infeasible.

Other HTTP request methods are available (for instance POST) which might allow more secure or reliable data transmission, but these methods were introduced in more recent specifications of the protocol standard. Using GET ensures maximum compatibility and allows for the simplest possible request string. Simple request strings and methods require the least amount of string manipulation, which is typically a memory and time-intensive operation.

Although Python scripts allow for flexible and simple data processing, only a limited number of web servers allow Python scripts to be executed remotely over port 80. PHP, on the other hand, is almost universal on modern servers, and offers significant benefits in speed and ease of database manipulation. Code for the internet interface was originally written in Python, but quickly ported to PHP. However, New Mexico Tech's general-purpose student web server lacks PHP support. Since this server has remained useful for testing, the Python logging script has been kept current with changes in the more practical PHP code.

C.5 Components

Direct interaction with the measurement circuitry is handled entirely by an MSP430 microcontroller on a commercial revision of the EasyWeb networkable microcontroller board originally designed by Texas Instruments. Programmers with the GNU mspgcc compiler project provide free example C code which implements a very lightweight TCP/IP stack for exactly this kind of board. The board offers no secondary memory peripheral, so operation is constrained to within the board's fairly limited (2048-byte) RAM space. Since a large amount of this space is used for variables critical to the operation of the data logging and network transmission subsystems, transmit and receive queues for the TCP/IP stack must be relatively small. Small queues constrain the speed of operation and can cause timing problems when the microcontroller interacts with much faster machines.

Three separate web servers can be used with the board's internet interface, and the server can be selected at compile-time. PHP scripts were tested both on a home desktop computer connected directly to the microcontroller board through a router and on `http://geek.nmt.edu`, a server maintained by the electrical engineering department. Python CGI scripts were used when PHP was unavailable or unreliable, namely on `http://infohost.nmt.edu`, which offers web space to all New Mexico Tech faculty and enrolled students.

D. Regulation Module

The regulation module provides a constant regulated voltage from the sources to be used by other modules such as the battery charger.

D.1 Design

The regulation module is designed around an LTC3780 buck-boost converter chip which operates in buck mode should the input voltage be higher than our desired output voltage or conversely in boost mode should our input be lower than our desired output. This design allows our output voltage to be held constant regardless if the input voltage is lower, higher or equal to our output voltage. We achieve this using only one chip. The overall regulation module uses two identical regulation circuits as shown in Figure 6(p. 22) - one for each source. Their respective outputs are tied together using two 1N5402 diodes to sum the currents and provide the charge current to battery charger circuit.

The circuit has been simulated in LTspice IV using two different DC inputs. Figure 16 (p. 26) shows the output of the regulator during the first few seconds after power is applied with a 25V input. The output starts at zero, then quickly rises to the desired output of roughly 17.8V. Figure 17 (p. 26) shows the output of the same circuit with a 5V input. Again, the output starts at zero, then quickly rises and settles at about 17.5V, which is still close to the desired output.

D.2 Components

The main component of this module is the LTC3780 buck-boost converter chip. We also used several other external components to complete the circuit. We used 4 external IRF7401 mosfets which are needed to provide the buck-boost capabilities of the circuit.

D.3 Design Alternatives

One of the design alternatives we considered was a multi-staged regulation scheme which essentially acted as a buck-boost regulator except it used more chips and was largely inefficient.

E. Battery Charger

The battery charger module takes input from the regulation module and uses it to supply current to a 12V lead-acid battery. It is capable of monitoring the battery voltage and limiting the current provided accordingly.

E.1 Design

The circuit (shown in Figure 7 (p. 22)) is based-on a circuit designed by D. Mohankumar [4]. The original circuit has been modified to use a DC input instead of an AC input. A linear regulator provides a DC voltage to the battery through a diode prevents current from flowing away from the battery toward the regulator as the regulator voltage drops below the battery voltage. As the battery voltage rises, a transistor draws current from the regulator circuit, causing the regulator output to drop. The battery ceases charging.

E.2 Components

The regulator is required to supply a high current to the battery and dissipates a large amount of power. The LM338K uses a large metal can package, and is designed to be used in conjunction with a heat sink to dissipate high power [9]. Likewise, the darlington pair used to control the regulator output also draws a large current and must dissipate a high power. The suitable component for that purpose is the TIP122, which is capable of dissipating 65W [8].

E.3 Design Alternatives

Linear Technologies provides a smart battery charging chip called the LTC1759 [11]. After the chip was order, investigation of the data sheet revealed that it requires a type of battery manufactures with circuitry that allows it to provide information to a smart battery charger via the I^{2}C bus.

F. DC and AC Power Supply Modules

The DC and AC power supply modules provide a 12 volt DC power output and a 120 volt AC power output at 60Hz. It is divided into three submodules: digital, high power AC and high power DC. The high power AC and DC modules are printed on a circuit board to handle the high output current while the digital stage is on. a breadboard.

F.1 Digital Submodule

The digital submodule provides the high power AC submodule with a pulse width modulated sampling of a 60Hz reference sine wave generated with a Wien Bridge oscillator. This submodule consists of a digital comparator which takes in the 1MHz digital oscillator and 60Hz sinusoidal oscillator and outputs two pulse width modulated signals, Q and $\overline{Q}$. This is a basic Class D amplifier circuit, which has high efficiency. This output ranges from 0 to 5V and is fed into the high power AC submodule.

F.2 High Power AC Submodule

The high power AC submodule consists of a LT1160 MOSFET driver which drives two sets of IRF7401 npn MOSFET's in order to amplify the power output of the PWM signal. This higher power signal is then filtered of higher order frequency components with a low pass LC filter consisting of a 420μH inductor in series and two 270μF capacitors in parallel with the output to a transformer which steps up the voltage from 17V peak or 12V AC to 120V AC.

F.3 High Power DC Submodule

The high power DC submodule consists of a voltage regulator to regulate the voltage from the battery to 12V DC. This is output through a 1 amp fuse as specified for safety but the fuse is in a holder so a larger fuse can be placed in if necessary.

F.4 Design Alternatives

For the AC submodule, a design alternative consisted of not using the Class D amplifier deisgn, in favor of a simpler design such as a Class A design or a simple square wave output. These are far more simple and easy to build. However, their output is square and has high THD which is unacceptable for some power applications. The DC submodule could be removed and instead run off of the battery directly as many 12V DC devices are designed to work with common 12V battery voltages ranging from 11V to 14V. This does not provide as consistent an output.

F.5 Components

The oscillator in the digital stage is at a high speed of 1MHz in order to allow the output filter to convert most of the energy back into the AC output signal. The use of this oscillator required the use of the LT1016, a 10ns comparator which provides the 0 to 5V logic signals, Q and $\overline{Q}$. This chip is very fast and has a quick enough output rise time to handle the PWM signal.

The LT1160 MOSFET driver was chose in order to drive the IRF7401 MOSFETs which handle 160W each. These MOSFETs were chosen because of the high power output and price compared to similar products.

The 420μH inductor was chosen because it can handle the large currents which are seen when high power is drawn through the transformer.

IV. Implementation, Testing, and Analysis

A. *Regulation and Battery Charging Modules*

B. *DC and AC Power Supply Modules*

C. *Power Measurement Module*

The measurement module was constructed on a breadboard during design and testing. The final circuit is shown in Figure 4 (p. 21). The circuit is laid out symmetrically about the logic submodule with the voltage submodule on the left and the current submodule on the right. Blocks labeled on the figure correspond to blocks in the schematics presented in chapter 3. The breadboard is suitable for low frequency, low power signals (< 5 Watts). Due to the limitations of the breadboard, the circuit was tested only using small signals.

C.1 Testing

Testing was carried out using voltage signals both from a signal generator and from 120V wall power scaled down by a transformer to 20V or 40V peak. Initial quantitative testing was performed using input from a signal generator. A resistive load was selected according to the typical resistor power rating of .25W to have a peak power less than .125W.

$$\frac{1}{8} = \frac{10V^2}{R_{Load}}$$

$$R_{Load} = 100 \times 8 = 800\Omega$$

A resistor of 1kΩ was chosen for its convenience to give a current of 10mA peak. Three of the resistors were connected in parallel to give a current of 30mA peak. The voltage and current signals were both measured at the non-inverting input of the comparators using an oscilloscope, and are shown in Figures 12 and 13 (p. 25) respectively. The DC offset was measured without any AC input, and then the two amplitudes of the two signals were measured and compared with the expected values. The data recorded are

Submodule	Input Signal	DC Offset	Output Signal	Gain	Expected Gain
Voltage	9.4V	1.14V	52.5mV	5.59mV/V	5.96mV/V
Current	27.0mA	.880V	36.1mV	1.32Ω	.1Ω

TABLE I
Characterization of the power measurement circuitry.

The spurious transresistance reading for the current submodule may be attributed to some unusual noise on the supply rail. The correct signal should have had an amplitude of 3mV peak, which could have been lost in the noise. The noise was later traced to a poor ground connection and eliminated, but new characterization measurements have not yet been conducted.

Since there is no link between the offsetting circuit and the voltage references used to convert the voltage and current signals into square waves, the references must be adjusted to match the signals. The output of the characters was measured on an oscilloscope, and the voltage at the inverting terminals was adjusted until both the voltage and current signals had a duty cycle of 50%. After the input stage of the module was calibrated, the output of the SR Latch was measured to confirm the presence of a pulse train, which is the expected output for a resistive load. One such pulse train is shown in Figure 14 (p. 25).

No other types of loads have yet been tested with the system. When the measured current decreases to about 10 mA, the corresponding voltage signal has a low signal to noise ratio, and, along with the noise present in the comparator voltage reference, produces a logic signal completely lacking distinct edges. It becomes impossible to ensure a 50% duty cycle in the signal, and the logic submodules fails to function correctly. A capacitative load or a load with any significant phase shift requires a bipolar capacitor with a sufficiently low impedance at 60Hz that the current through it exceeds 10mA. During small signal testing, no such capacitor was available.

The measurement module has been tested in conjunction with the data logging module using a 20V peak input derived from 120V AC wall power and a resistive load selected to give a peak current of 73.2mA. The buffer stages on the voltage and current submodules reduce the change in the signals caused by the microcontroller ADC's low input resistance, but the signals are very small, and a change of a few tens of millivolts in the offset can impact significantly the operation of the logic submodule. As a result, the logic submodule must be calibrated with the ADC inputs connected to the circuit. The module operates correctly after the logic submodule has been calibrated.

D. Data Logger Module

Microcontrollers allow for automated interaction with electrical circuits. A number of ready-made microcontroller boards are produced commercially with built-in ethernet hardware, allowing for straightforward acquisition and network transmission of power data. A particularly simple and low-power board of this kind is the EasyWeb, which was designed by Texas Instruments around their MSP430F149 microcontroller to demonstrate the chip's internet capabilities[2]. The designs were made freely available and have since been revised and reproduced by a few manufacturers, including Olimex. Olimex also produces a hardware programmer for the MSP family of microcontrollers, and in conjunction with a free software suite called mspgcc this tool makes it possible to program and use the EasyWeb board without any proprietary software.

D.1 Implementation

The Texas Instruments MSP430 microcontroller on the EasyWeb board (shown in Figure 5, p. 22) executes a fairly simple networking software loop in the C programming language, and all measurements of power characteristics are triggered by interrupts. Code for this subsystem is provided in Appendix D.1 (p. 37). Frequency and phase measurements are performed in an input-capture interrupt service routine which, when an interrupt occurs, reads the triggered input pin to determine whether the edge was rising or falling. When a rising edge is seen, the counter value in the microcontroller's timing peripheral is recorded and the difference is taken between the current rising edge and the previous rising edge. This value is stored for transmission to the server script and represents the period of the input waveform as measured in microcontroller clock cycles. On a falling edge, the time difference between the edge and the previous rising edge is computed and stored, allowing the server data logging script to determine the duty cycle of the logic waveform and therefore the power factor of the load.

The microcontroller also allows its analog-to-digital converter peripheral to trigger an interrupt when a conversion finishes. Each time a measurement is made of an analog input waveform (either voltage or current), two 32-bit integers associated with that waveform are appended with new data. One variable stores a simple sum of each reading received for the waveform while the other stores a sum of squares. These values are ultimately used by a PHP or Python script to compute the arithmetic mean and the mean squared of the total waveform (including offset), which in turn are used to compute root-mean-square averages. Additionally, a counter is incremented each time a conversion completes, allowing the code to keep track of how many points have been taken. Finally, if an overflow is detected in the integers storing sums-of-squares, an overflow counter is incremented for the corresponding waveform. If this counter were omitted, taking as few as 256 points in a period could cause peak voltage and current data to become unusable. When a rising edge is detected by the input-capture peripheral, the interrupt service routine copies each of these variables to new locations, since these variables are likely to change before the network code can transmit them. Each sum and counter is then reset so they may become usable again for the next period.

D.2 Testing

Before the present measurement module was in place, frequency measurement was confirmed by passing a sine wave from a function generator through a comparator to generate a square wave with edges at each zero crossing. Next, attempts were made to verify the operation of the phase measurement code on a logic waveform ANDed with a phase-shifted version of itself. By applying a phase shift to the sinusoid with a lowpass filter and feeding this phase-shifted signal into a comparator. Due to signal attenuation in the filter, the resulting waveform did not drive enough current to the comparators at this voltage to trigger them. Frequency measurement shows extremely low error — when a digital multimeter measured the frequency of the function generator output waveform, readings fluctuated between 59 and 60 Hz; the implemented digital frequency measurement showed a reading of 59.5 Hz.

During various stages of the measurement module's completion, the data logging code has been successfully tested on wall power. Initially, wall power, passed through a step-down transformer, was scaled down by a voltage divider and used to generate a logic signal. This allowed for accurate measurement of frequency. The frequency of the logic waveform at later stages, with uneven duty cycle, has also been measured with low error from 60 Hz. Since completion of the measurement module, RMS measurement has successfully been performed on an offset voltage signal. The offset can reliably be measured by computing the arithmetic mean of the ADC samples, and root-mean-square averages with about 10% error from expected values have been measured from sums-of-squares recorded by the microcontroller code.

E. Internet Interface Module

The network code constitutes the primary operation of the microcontroller's software. The EasyWeb board includes a CS8900 hardware ethernet controller, and free example code provided by the mspgcc project (available at [1]) is available to interface with this chip. The simple API provided by this code is used to implement an HTTP client state machine which the microcontroller executes until a hardware interrupt occurs to perform measurements.

During development and testing, several different web servers were used with different software and compatibility.

As a result, two scripts with the same functionality were written, `logger.php` and `logger.cgi`. Since the software focuses primarily on interactions with a MySQL database and retrieving arguments from an HTTP GET request, the main differences are in how output HTML for the display page is generated. Ultimately, the Python display page was discarded for its unnecessary complexity since the database can easily and identically be read using either script. In general, PHP is faster and better suited for both database manipulation and HTML generation. Both forms of logging script and the PHP display script are included in Appendix C (p. 29).

E.1 Implementation

Before the main loop is entered, some basic network setup occurs. The code uses preprocessor directives to select which server to attempt to contact. The viable options include a home desktop computer typically available on the local network occupied by the board and the New Mexico Tech servers `geek.nmt.edu` and `rainbow.nmt.edu`. During this preliminary code, the board also declares its own static network IP, DNS gateway, and subnet mask. Finally, the board randomly generates an ephemeral local port, since insecure HTTP connections require that client devices change their source port after each transaction. This simple measure helps avoid packet spoofing, and each time a new connection is opened the microcontroller selects a new ephemeral port.

In the main loop, the HTTP client consists of two primary operations, each implemented within the void C routine `HTTPClient()`. Between calls to `HTTPClient()`, the loop calls `DoNetworkStuff()`, which serves as the core of the free TCP/IP stack's API. The `DoNetworkStuff()` routine performs all hardware-level network transactions and uses flags to communicate the status of the network to other parts of the program. This allows the HTTP client to continually observe the network status and open a connection to the designated server if no connection exists. Once a connection is established, the most recent power measurement data is formatted into an HTTP request string, including a "Connection: close" header that asks the server to close the connection immediately after the current transaction, and the location of this string is provided to the API for transmission to the server. Since the microcontroller is requesting a webpage from a server, it then anticipates the end of an HTML page and closes the connection once the terminating `</html>` tag is received. The state machine is then reset and a new network connection is opened.

E.2 Testing

In early stages of data logging, code had not yet been written for the microcontroller to transmit data to a server, and server code did not exist to listen for incoming data. Instead, data was displayed by a dynamic webpage hosted on the microcontroller itself and accessible only on the local network. Once transmission code was in place, a simple script was written in Python and run on a desktop computer to listen for incoming HTTP connections and display their content. This allowed for confirmation of the HTTP client's consistent operation and for observation of the packets being transmitted by the microcontroller. Code for this script is provided in appendix A (p. 27). Once data transmission from the board to a networked PC was verified in this way, a similar simple HTTP client script was written in Python and used to connect to PHP and Python CGI scripts running on remote webservers. Using the WireShark packet analyzer, the packet-by-packet transactions between this Python client and each server-side script were observed, and the programming of the microcontroller was altered in order to better match these interactions more closely.

After the board was able to make repeated connections to the server scripts, the insertion of new points into a remote MySQL database and subsequent retrieval by `display.php` proceeds very straightforwardly. In either case the script opens a MySQL connection to `dbhost.nmt.edu` and creates a cursor in the `powerdata` database. The cursor then inserts or retrieves each floating-point power characteristic, along with the date (which in the case of `logger.php` or `logger.cgi` is requested from MySQL by the builtin `NOW()` function). This whole process is operational and the end-user display page is live at `http://geek.nmt.edu/~aleph0/display.php`. Additionally, a screenshot can be found in Figure 15 (p. 26).

The network connection from the board is fairly robust. The ethernet connection may be removed or disabled while the board is running, and transmission of packets proceeds as normal once the connection is restored. The entire program is stored in EEPROM so that it executes automatically when the board is powered on. The board, when communicating on a local network, has been left transmitting data for some matter of hours without failure. Remote connections to real webservers, however, have caused problems after long periods of time, perhaps because of compatibility issues between intermediary routers or switches and the bare-bones network code operating on the microcontroller board.

F. Regulation and Battery Charging Modules

The circuits have not yet been implemented. As a result, no testing has yet been performed on the actual circuits beyond the SPICE simulations presented in section D.1 (p. 9). The circuit schematics have been laid out in Protel, and some work has been performed toward laying out a PCB containing both circuits.

Though no testing has been performed, some thought has been giving to the subject. The regulator outputs should first be characterized using different DC voltages from a power supply. The regulators will then be tested using inputs from the solar cells and wind turbine, and the output will be monitored by the data logging and measurement system to a period of several hours to ensure consistent operation. After confirming that the regulators provide a constant voltage, the output of the battery charger circuit should be characterized using the regulator outputs as a supply and simple resistor circuits to simulate a battery in different states of charging. When both the regulators and the battery charger circuit have been shown to work, the circuit will be applied to a battery with inputs from DC power supplies to confirm that the circuit correctly charges the battery. Finally, the battery will be charged again using only inputs from the solar cells and wind turbine. The charging process will be monitored through completion.

G. DC and AC Power Supply Modules

The digital submodule of the AC power supply module was built on a breadboard as shown in Figure 11 (p. 24). The oscillator and Wien Bridge oscillator are on the left column of the breadboard with the comparator in the center.

There are two wires for Q and $\overline{Q}$ which connect to the printed circuit board. The AC power submodule was laid out in Protel 99 and the printed circuit layout was completed but printing is pending waiting for the battery charger module to be laid out as well. This is because it is more efficient to print out everything on one board and because of the usage of the same high power rails.

The DC submodule was to consist of a simple regulator circuit taking the voltage from the battery and converting it to 12V DC out. This circuit has not been laid out.

G.1 Testing

Testing of the digital submodule was carried out first. This was done by powering it off of a desktop power supply and viewing the outputs of the comparator. The signal appeared to be the correct expected signal.

Testing of the AC power supply on a breadboard was done early on with low power signals and a pure sine wave was seen out of the filter. However, at this point, the board is not printed so an AC 120V output seems unrealistic to achieve on a breadboard. Issues that could be faced include ones related to transformer impedences which can be troublesome with high power transformers. Another issue that could come up would be heat on the MOSFETs. This is why a fan and heatsinks were purchased.

The DC power supply is also not near testing for the same reason as above, and may also encounter heat issues depending on the power required out.

V. Conclusions

A. Summary

Although some of the primary functions of the project, namely accepting power input and providing power output, have not been fulfilled, simulation-proven designs exist for every major circuit and construction is underway. Furthermore, the current state of the data measurement, logging, and broadcast modules, while imperfect, is certainly sufficient to provide real-time measurement of power characteristics to users over the internet. Design and preliminary construction of these modules is complete, and plans exist for expansion and revision of every module. Time proved to be the major constraint in this project. Methods for accessing the budget were not made clear until well into the semester, and as a result the majority of development took place in a very tight timeframe. Nonetheless, significant progress was made on three out of four of the top-level project modules, and testing has shown that these modules should be integrable with other parts of the device.

B. Future Work

B.1 Power Measurement Module

The measurement module can be improved in a number of ways. The current sense resistor approach to current measurement is limited in its precision by the fact that the current sense resistor creates a voltage drop between the ground for AC output and the actual circuit ground. An inductive measurement circuit could be built to have a large transresistance without interrupting or modifying the AC output.

The current approach to logic signal generation relies on the precision or an external reference. If the signal or the reference change offset over time, the circuit will stop working and require recalibration by means not available to the average consumer. It would be worthwhile to design a circuit capable of extracting the DC offset from the AC signal to use as a reference. Even a lowpass filter might be suitable if the filters signal has a very small AC component. If both the voltage and current reference were generated using identical lowpass filters, the result would create a square wave with a slight phase shift from the original signal, but with identical phase shifts in both the current and voltage signals. The tolerance of the components used must be considered, but might be acceptable for common components.

A more effective comparator circuit might utilize an op-amp with a slow slew rate to prevent high frequency noise in its output. Such a circuit would eliminate the need for the filter circuit and inverter, reducing the overall complexity of the module. In addition, the filter circuit introduces a phase shift to the original signal that depends on that filter components. The circuit's accuracy would be improved by the removal of the filter circuit.

Additional testing needs to be performed on the module to characterize its performance. Specifically, data should be collected for a variety of different input voltages and currents as well as different types of loads. There are two points in the circuit where high voltages and high currents may exist under large inputs. Those parts of the circuit need to be placed on a PCB or some other external component capable of handling the required voltages and currents prior to testing.

The entire measurement circuit should be laid out on a PCB so that it can be placed physically near the signals it measures, and so that it can be assembled in a manner that minimizes interference and noise. This is currently only a minor concern, but could make the circuit operate more consistently and improve precision and accuracy of measurements.

B.2 Data Logger Module

Frequency measurement is extremely accurate and reliable. Phase measurement has not been tested extensively due to difficulties in drawing enough current from a stepped-down transformer voltage to observe current through a reactive load. The method used for measuring root-mean-square averages of waveforms has been proven to be effective. However, it remains to be seen whether using a higher sampling rate noticeably improves the RMS readings or whether some other source of error is significant.

B.3 Internet Interface Module

The logging scripts work properly and should not require modification unless the methods used for data measurement are altered significantly. The ability of PHP and Python scripts to handle the case of overflow has not yet been confirmed in practical operation, nor have any observations been made about the potential improvement in accuracy such a change may cause. The current user web interface is rather bland and more functionality, such as a system for retrieving older power measurements by date and time, should be implemented. Networking problems have been persistent, especially when attempting to make connections with servers outside a local network, and further testing and research should be done to determine the source of this difficulty.

B.4 Input Regulation And Battery Charging

This module requires significant work. As no implementation or testing has yet been conducted, the exact nature of that work is unknown. In general, a PCB must be laid out, etched, and populated with components. Testing, as outlined in F (p. 14), must be carried out to verify the functioning of the circuit, and the circuit must be integrated with the other modules as a supply and with a 12V voltage regulator to provide DC output.

B.5 DC and AC Power Supply Modules

The power supply module can be improved in various ways. The Class D amplifier design became difficult to build and resulted in delays in building and testing. A simpler approach would have resulted in a working test bed. The use of a transformer is also a downside of this design in that transformers are lossy and a design which used higher supply rails and created an 120V AC output without a transformer. The AC power supply, however, is a very efficient design and and would be able to power many different loads.

The DC power supply design could be improved by using a buck-boost regulator. This would allow the circuit to work below 12V, however, the batteries usually lose a lot of power below this voltage.

Much more testing needs to be done after the board is printed in order to determine real world output charecteristics for the entire module. Problems may arise in this step and troubleshooting may be necessary.

References

[1] Andreas Dannenberg. Msp430 tcp/ip stack. `http://mspgcc.bzr.sourceforge.net/bzr/mspgcc/mspgcc-examples/revision/39/tcpip/`, November 2003.

[2] Andreas Dannenberg. Msp430 internet connectivity. `http://focus.ti.com/lit/an/slaa137a/slaa137a.pdf`, February 2004.

[3] Texas Instruments. Sn74hc14 hex schmitt-trigger inverters. `http://www.datasheetcatalog.org/datasheet2/b/0dosa061ci0slrkis3hf5etqc8fy.pdf`, December 1982.

[4] D. Mohankumar. Fast charger with auto cut off. `http://electroschematics.com/5614/fast-charger-with-auto-cut-off/`.

[5] Renewable Energy Policy Network. Renewables global status report. `http://www.ren21.net/pdf/RE_GSR_2009_update.pdf`, 2009.

[6] New Mexico Institute of Mining and Technology Electrical Engineering department. An intelligent dc-dc/ac converter. `http://www.ee.nmt.edu/~erives/382_10/Intelligent%20Converter.pdf`, January 2010.

[7] Eric Steven Raymond, Thyrsus Enterprises, and Nick Christenson. Ups howto. `http://tldp.org/HOWTO/UPS-HOWTO/x142.html#AEN187`, 2007.

[8] Fairchild Semiconductor. Tip122 npn epitaxial darlington transistor. `http://www.fairchildsemi.com/ds/TI/TIP122.pdf`, October 2008.

[9] National Semiconductor. Lm338 5-amp adjustable regulators. `http://www.national.com/ds/LM/LM138.pdf`, May 1998.

[10] National Semiconductor. Lm339 low power low offset voltage quad comparators. `http://www.national.com/ds/LM/LM339.pdf`, March 2004.

[11] Linear Technology. Ltc1759 smart battery charger. `http://cds.linear.com/docs/Datasheet/1759fs.pdf`, 1998.

Appendices

A. Final Budget

The total budget breakdown as of May 11, 2010 is as follows:

Module	Cost
Inverter	\$158.10
Measurement, logging	\$109.28
Battery charger, voltage regulation	\$59.09
Enclosure, wiring	\$28.78
Total	\$355.69

No further expenses are expected except in the manufacturing of PCBs. This breakdown includes parts that were not used in the final project such as the LT1759 smart battery charger. The entire project is underbudget. PCB materials are expected to cost less than \$10, and the cost of etching the PCBs can be mitigated by using available resources. Therefore the total expense to complete the project falls below \$400, and is under budget.

B. Circuit diagrams and photographs

A. *Measurement Subsystem*

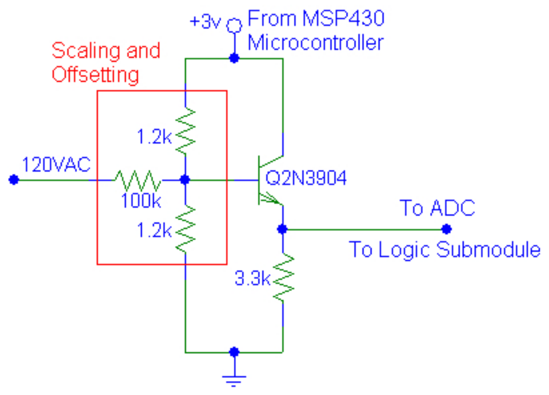


Fig. 1. Voltage measurement submodule.

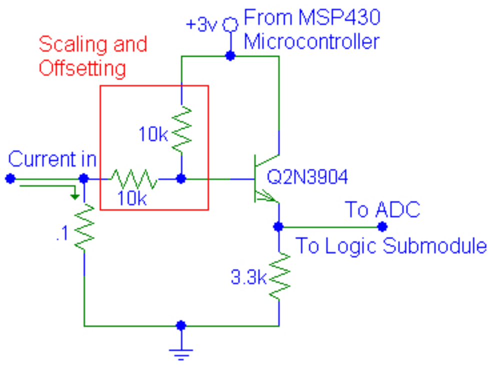


Fig. 2. Current measurement submodule.

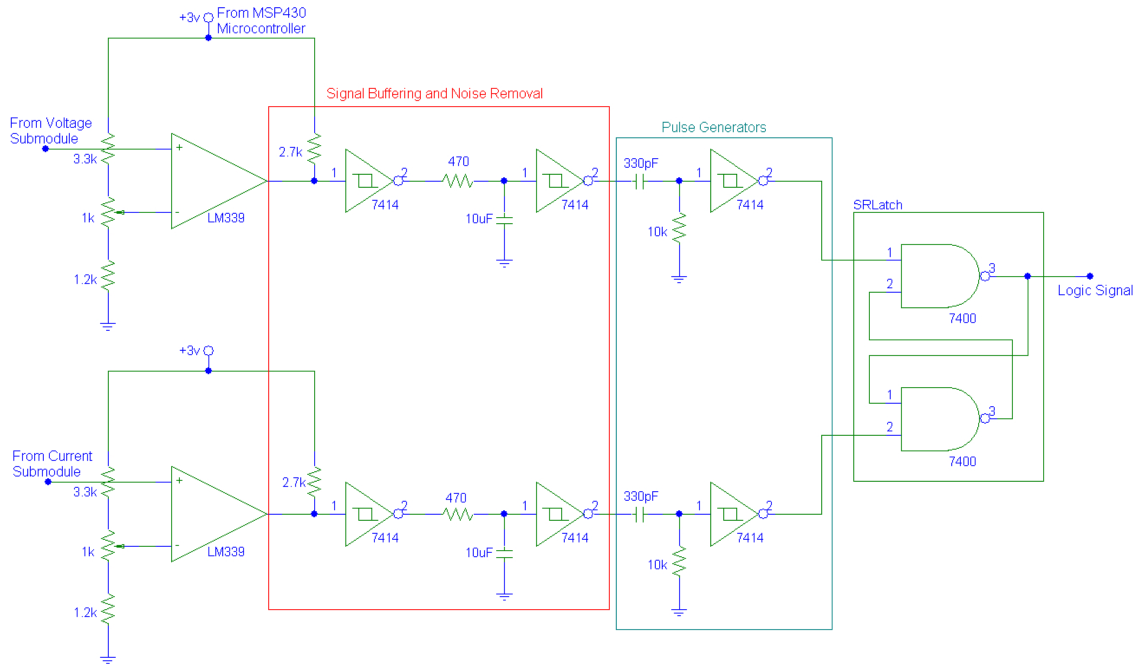


Fig. 3. Logic signal synthesis submodule.

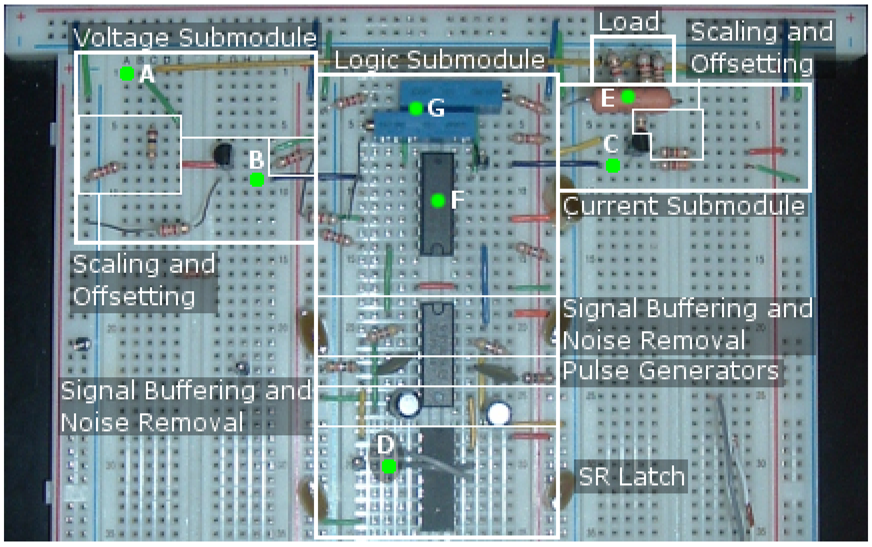


Fig. 4. The measurement module implemented on a breadboard. A: 120V AC Input, B: Voltage signal/Output to ADC, C: Current signal/Output to ADC, D: Logic signal/Output to input capture, E: Current sense resistor, F: LM339 Quad Comparator, G: 1K potentiometers for adjusting voltage references

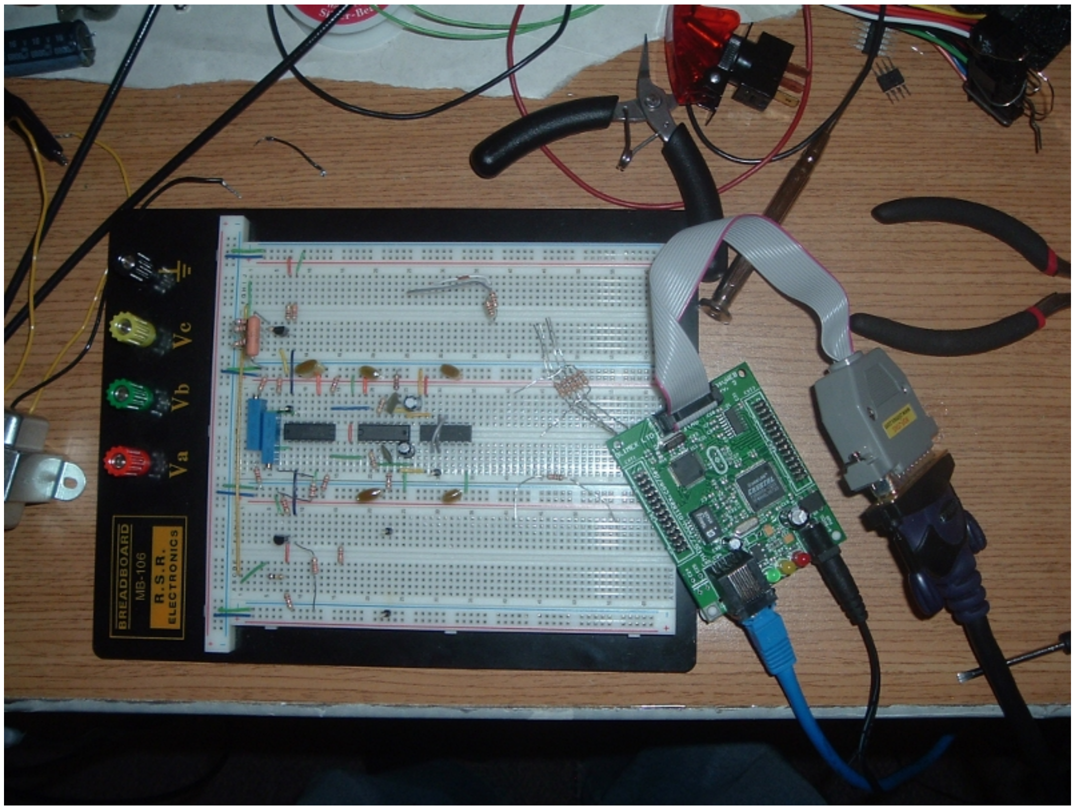
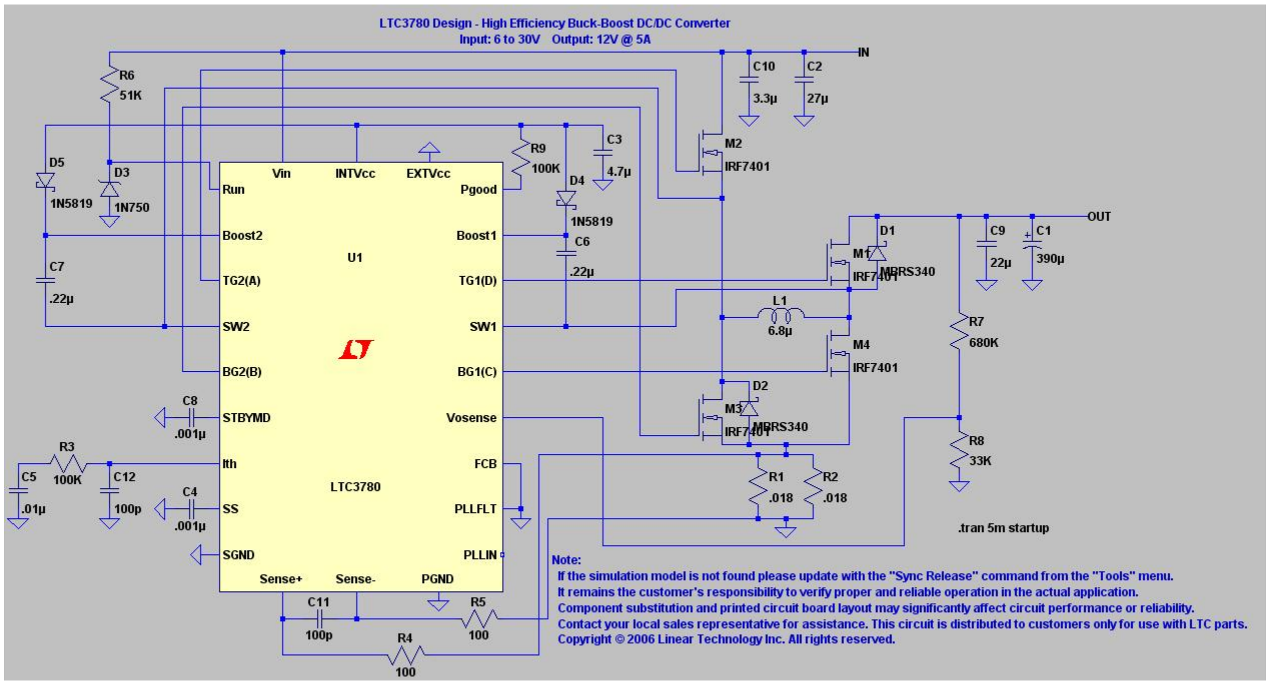


Fig. 5. The EasyWeb 3 ethernet-capable microcontroller board (right), connected to an ethernet cable (blue) and a JTAG hardware programmer (gray), which connects by a parallel cable (black) to a PC (not shown). Shown with breadboard containing measurement circuitry for scale.



Note:
If the simulation model is not found please update with the "Sync Release" command from the "Tools" menu.
It remains the customer's responsibility to verify proper and reliable operation in the actual application.
Component substitution and printed circuit board layout may significantly affect circuit performance or reliability.
Contact your local sales representative for assistance. This circuit is distributed to customers only for use with LTC parts.
Copyright © 2006 Linear Technology Inc. All rights reserved.

Fig. 6. Input regulator circuit built around the LTC3780 Buck-Boot Regulator.

includegraphics[scale=0.5]BattCircuit

Fig. 7. Battery charger circuit. Design provided by [4].

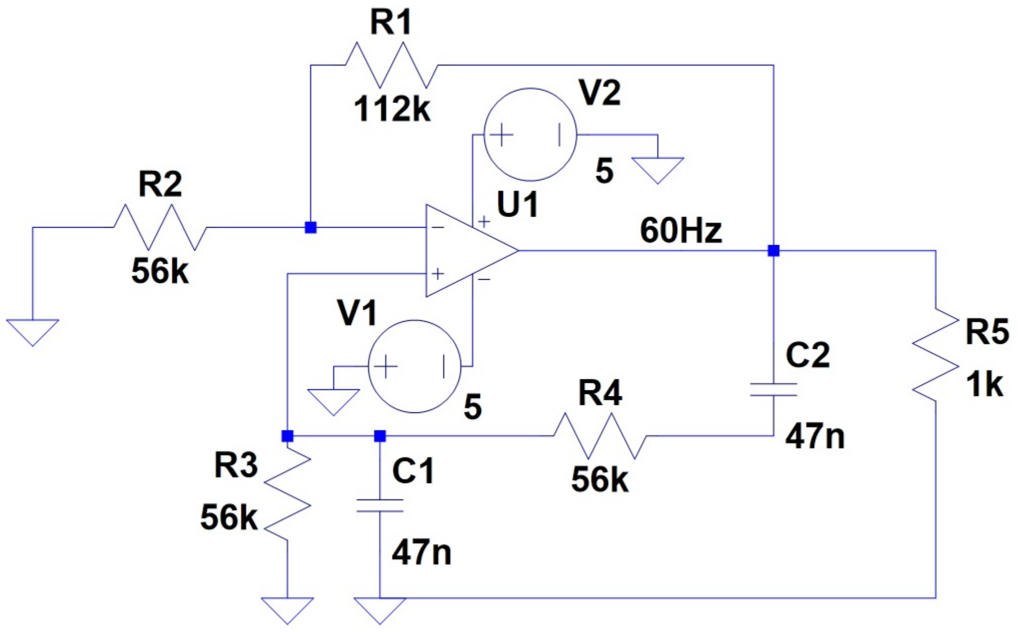


Fig. 8. Circuit diagram for a Wien bridge oscillator designed to produce a 60 Hz sinusoid, which is then sampled and amplified to produce an AC power signal.

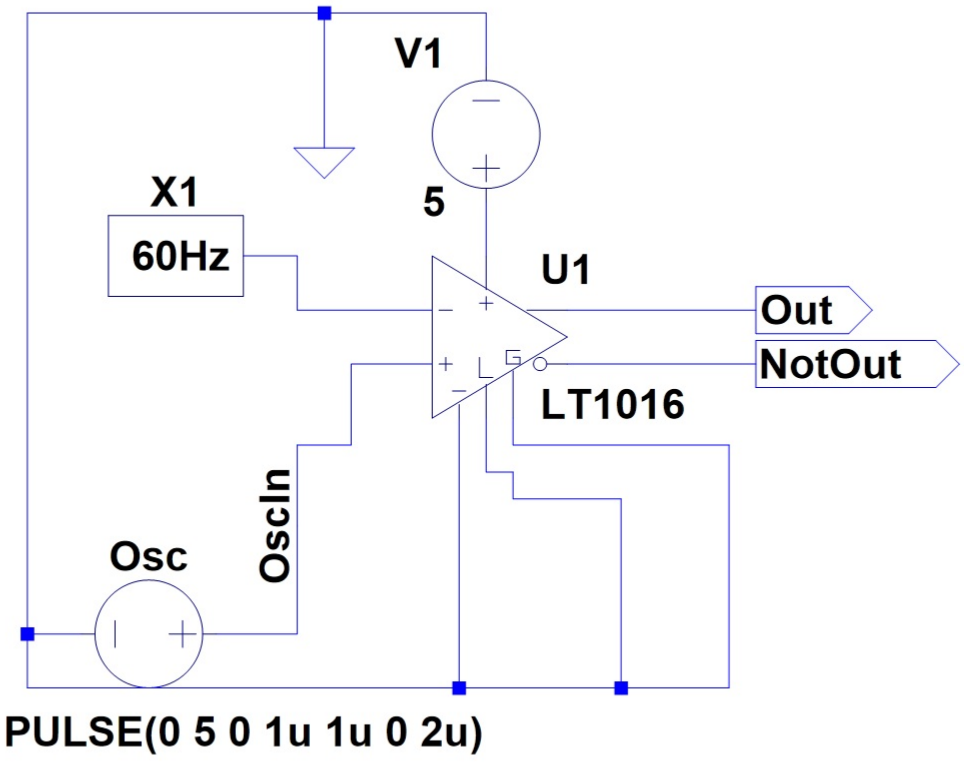


Fig. 9. Generation of a variable duty-cycle (pulse width modulated) digital signal by feeding a comparator with a small sinusoid and a triangle wave.

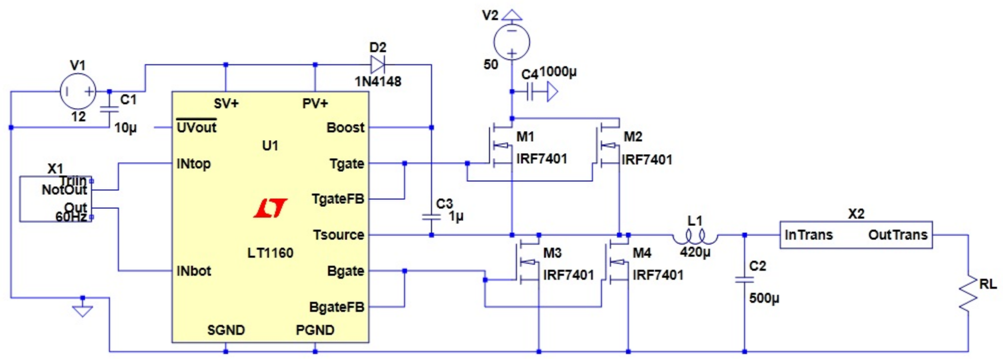


Fig. 10. Amplifying and filtering the PWM signal to produce a power sinusoid.

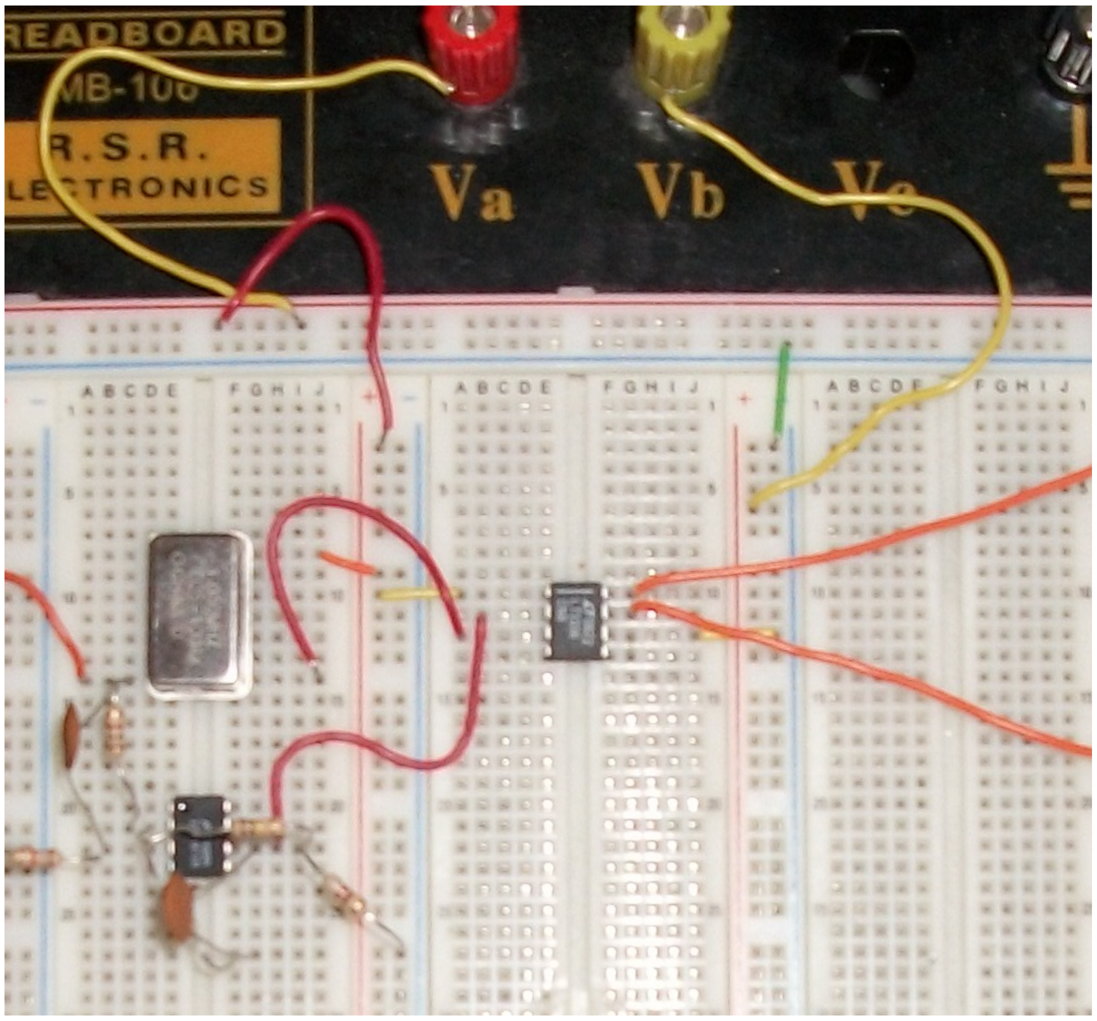


Fig. 11. Prototyped circuit for generating a pulse-width-modulated signal from a Wien bridge and a triangle wave.

C. Waveforms and output

A. Measurement Subsystem

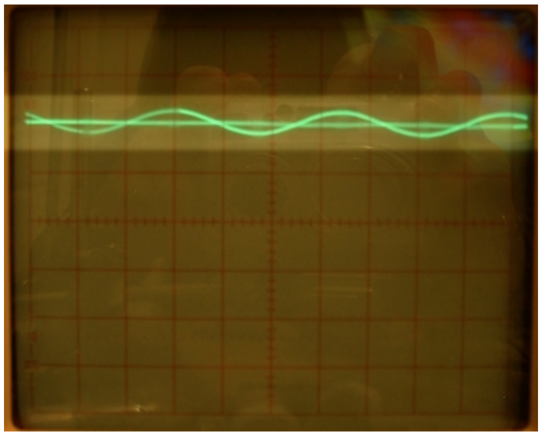


Fig. 12. Voltage signal and comparator reference displayed on a BK Precision 2120 Oscilloscope. .5V/div, 5ms/div

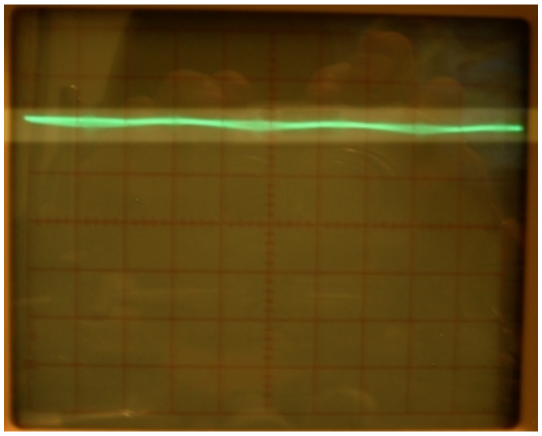


Fig. 13. Current signal and comparator reference displayed on a BK Precision 2120 Oscilloscope. .5V/div, 5ms/div

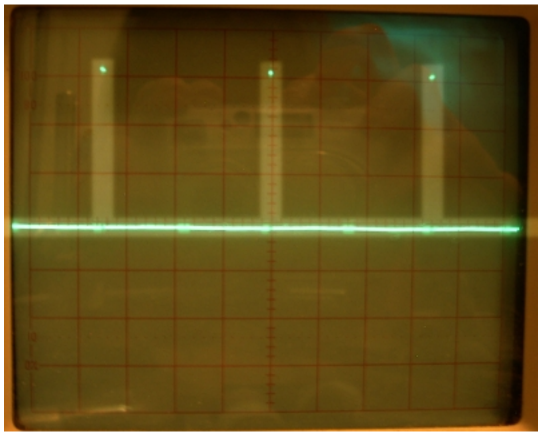
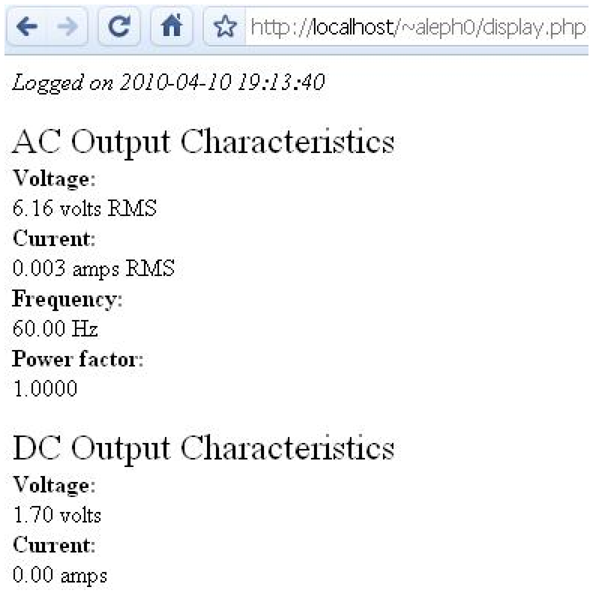


Fig. 14. Logic signal with 0% duty cycle and 60Hz frequency displayed on a BK Precision 2120 Oscilloscope. 1V/div, 5ms/div



http://localhost/~aleph0/display.php

Logged on 2010-04-10 19:13:40

AC Output Characteristics

Voltage:
6.16 volts RMS
Current:
0.003 amps RMS
Frequency:
60.00 Hz
Power factor:
1.0000

DC Output Characteristics

Voltage:
1.70 volts
Current:
0.00 amps

Fig. 15. The `display.php` script, as accessed in a web browser, showing live power data.

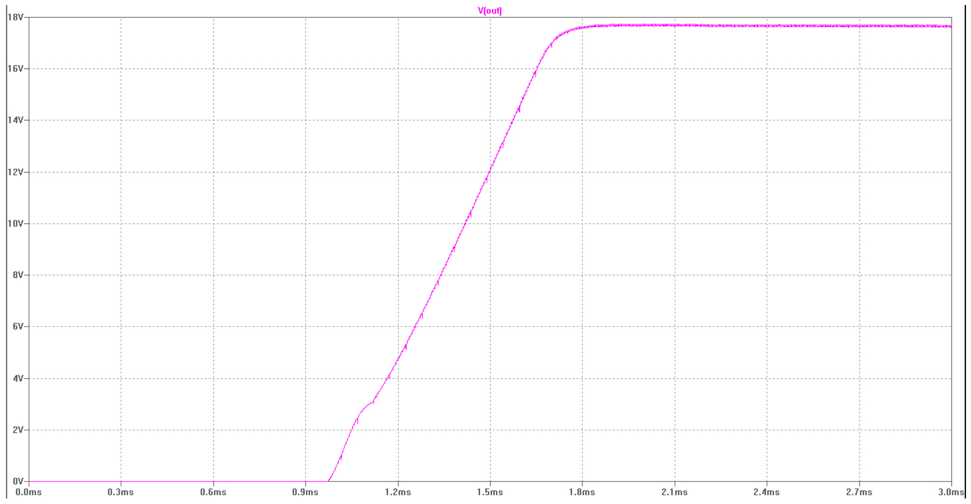


Fig. 16. Simulation showing the input regulator producing a 17.8V output given a 25V input.

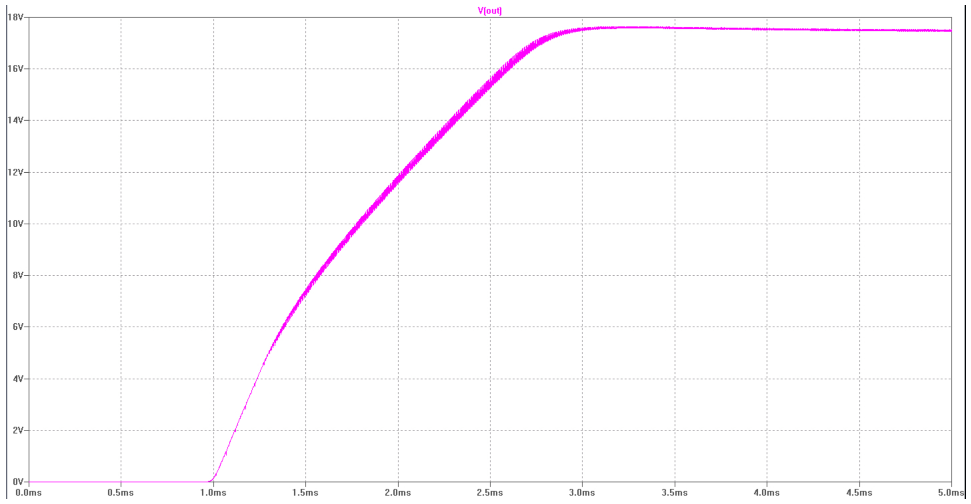


Fig. 17. Simulation showing the input regulator producing a 17.5V output given a 5V input.

D. Code

A. Simple server emulation (Python)

```
import socket

host = ''
port = 80
backlog = 5
size = 1024
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host,port))
s.listen(backlog)

num_conns = 0
response  = 'HTTP/1.1 200 OK\r\n'
response += 'Date: Sat, 10 Apr 2010 19:43:20 GMT\r\
response += 'Server: Apache/2.2.9 (Ubuntu) PHP/5.2.6-2ubuntu4.5 with Suhosin-Patch\r\n'
response += 'X-Powered-By: PHP/5.2.6-2ubuntu4.5\r\n'
response += 'Vary: Accept-Encoding\r\n'
response += 'Content-Length: 166\r\n'
response += 'Connection: close\r\n'
response += 'Content-Type: text/html\r\n\r\n'
html = '<html>\n  <head>\n    <title>Power Data Logging Script</title>\n  </head>\n  <body>\n       Thank yo

print len(html)
response += html

while 1:
    client, address = s.accept()
    if client:
        num_conns += 1

        data1 = client.recv(size)
        if data1:
            client.send(response)
            print '%d: client made connection on port %d' % (num_conns,port)
            print 'client sent %s' % data1
            client.close()
```

B. Simple client emulation (Python)

```
hostname = 'geek.nmt.edu'
packet_size  = 262144
timeout = 5

import socket


path = '/~aleph0/'
script = 'logger.php'

data = {
    'clockrate': 1250000,
    'period':     65500,
    'uptime':     65500,
    'atd0':       0,
    'atd1':       1,
    'atd2':       2,
    'atd3':       3,
    'atd4':       4,
    'atd5':       5,
    'atd6':       6,
    'atd7':       7,
    'vsqsum':      4597701,
    'isqsum':      4095,
    'dcvolts':      4095,
    'dcamps':      4095,
    'points':      119
}

IP_HOST = (hostname,80)
mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect(IP_HOST)

datastring = 'GET %s%s?' % (path,script)
dataElems = []
for key in data.keys():
     dataElems.append(key + '=' + str(data[key]))
datastring += '&'.join(dataElems)
datastring += ' HTTP/0.9\r\n'
datastring += 'Connection: close\r\n'
datastring += '\r\n'

print datastring

mysock.send(datastring)
returndata = mysock.recv(packet_size)
if returndata:
    print returndata
mysock.close()
```

C. Server-side dynamic webpages

C.1 Server-side logging code (Python)

```
#!/usr/local/bin/python

import cgi, cgitb, urllib, sys, math, MySQLdb
import lxml.etree as et

cgitb.enable()

ACVScale = 173.25
ACIScale = 1.225

f = open('output.txt','w')
f.write('hello\n')

def main():
    form       = cgi.FieldStorage()

    clock   = Clock(form)
    freq    = Freq(form,clock)

    phase   = Phase(form)
    pf      = PF(phase)
    lagging = Lag(phase)

    f.write('clock, freq, phase: %d %f %f\n' % (clock,freq,phase))


    for x in form.keys():
f.write('%s\n' % x)

    dcvolts  = DCVolts(form)
    f.write('computed dc volts: %f\n' % dcvolts)
    dcamps   = DCAmps(form)
    f.write('computed dc amps: %f\n' % dcamps)
    rmsvolts = ACVScale*ACVolts(form)
    f.write('computed rms volts: %f\n' % rmsvolts)
    rmsamps  = ACIScale*ACAmps(form)
    f.write('computed rms amps: %f\n' % rmsamps)

    f.write('computation complete\n')

    conn    = MySQLdb.connect(host='dbhost.nmt.edu',
                              user='cboling',
                              passwd='4l3phNu11',
                              db='cboling')

    f.write('frequency: %f\n' % freq)

    cursor  = conn.cursor()

    f.write('made MySQL cursor\n')

    cursor.execute("""
INSERT INTO powerdata (timestamp,dcvoltage,dccurrent,peakvoltage,peakcurrent,frequency,pf,lagging)
VALUES
  (NOW(),%f,%f,%f,%f,%f,%f,%d)
```

```
      """ % (dcvolts,dcamps,rmsvolts,rmsamps,freq,pf,lagging))
    cursor.close()
    conn.close()


    print 'Content-type: text/html'
    print

    html       = et.Element('html')
    page       = et.ElementTree(html)
    head       = et.SubElement(html,'head')
    title      = et.SubElement(head,'title')
    title.text = 'Power Logging'

    body       = et.SubElement(html,'body')

    nameList   = form.keys()
    nameList.sort()

    notice      = et.SubElement(body,'nothing')
    notice.text = 'Power data logged successfully.'

    page.write(sys.stdout)

def Clock(form):
    clock = form.getlist('clockrate')
    return int(clock[0])

def Freq(form, clockrate):
    inticks = form.getlist('period')
    f.write('ticks per period: %d\n' % int(inticks[0]))
    if int(inticks[0]) == 0:
       return 0
    return  clockrate/float(inticks[0])

def Phase(form):
    timediff = form.getlist('uptime')
    period   = form.getlist('period')
    if int(period[0]) == 0:
       return 0
    duty     = float(timediff[0])/float(period[0])
    return   duty*2*math.pi

def PF(phase):
    return abs(math.cos(phase))

def Lag(phase):
    if phase > math.pi:
        return 1
    else:
return 0

def DCVolts(form):
    value = form.getlist('dcvolts')
    return 3*float(value[0])/4095

def DCAmps(form):
    value = form.getlist('dcamps')
    return 3*float(value[0])/4095

def ACVolts(form):
    sqsum = int(form.getlist('vsqsum')[0]) + int(form.getlist('vovflows')[0])<<32
    avg   = int(form.getlist('vacsum')[0])
    per   = int(form.getlist('points')[0])

    f.write('\nAC Volts Activate:\n')
    f.write('Sum of squares: %d\n' % sqsum)
    f.write('Sum: %d\n' % avg)
    f.write('Period length: %d\n' % per)

    if (per == 0):
        return 0.0

    result = sqsum - (avg**2)/float(per)
    f.write('unscaled MS volts: %f\n' % result)
    if (result < 0):
        return 0.0
    rms = math.sqrt(result/per)
    return (3*rms/4095)

def ACAmps(form):
    sqsum = int(form.getlist('isqsum')[0]) + int(form.getlist('iovflows')[0])<<32
    avg   = int(form.getlist('iacsum')[0])
    per   = int(form.getlist('points')[0])

    f.write('\nAC Amps Activate:\n')
    f.write('Sum of squares: %d\n' % sqsum)
    f.write('Sum: %d\n' % avg)
    f.write('Period length: %d\n' % per)

    if (per == 0):
        return 0.0

    result = sqsum - (avg**2)/float(per)
    f.write('unscaled MS amps: %f\n' % result)
    if (result < 0):
return 0.0
    rms = math.sqrt(result/per)
    return (3*rms/4095)

def ATD(form, channel):
    value    = form.getlist('atd%d' % channel)
    return 3*float(value[0])/4095

main()
f.close()
```

C.2 Server-side logging code (PHP)

```
<html>
  <head>
    <title>Power Data Logging Script</title>
  </head>
  <body>
    <?php
     define("VACScale", 1.0);
     define("IACScale", 1.0);
     define("VDCScale", 1.0);
     define("IACScale", 1.0);
 define("intmax",4294967296);

     $hostname = "dbhost.nmt.edu";
     $username = "cboling";
     $password = "4l3phNu11";
     $database = "cboling";

     function Freq($clockrate)
     {
       $inticks = $_GET['period'];
       if ((int)$inticks == 0)
       {
         return 0;
       }
       else
       {
         return ($clockrate/(int)$inticks);
       }
     }

     function Phase()
     {
       $timediff = $_GET['uptime'];
       $period   = $_GET['period'];

       if ((int)$period == 0)
       {
         return 0;
       }

       $duty     = (float)$timediff/(int)$period;
    $duty*2*M_PI;
     }

     function PF($phaseang)
     {
       return abs(cos($phaseang));
     }

     function Lag($phaseang)
     {
       return (($phaseang > M_PI) ? 1 : 0);
     }

     function VRMS($handle)
     {
$sqsum = (int)$_GET['vsqsum'] + (int)$_GET['vovflows']*intmax;
$avg   = (int)$_GET['vacsum'] + (int)$_GET['iovflows']*intmax;
$per   = (int)$_GET['points'];
        if ($per == 0)
{
return 0.0;
}

$result = $sqsum - ($avg*$avg)/((float)$per);
if ($result < 0)
{
return 0.0;
}
$rms = sqrt($result/$per);
return 3*VACScale*$rms/4095;
  }

     function IRMS()
     {
       $sqsum    = (int)$_GET['isqsum'];
       $avg    = (int)$_GET['iacsum'];
   $per  = (int)$_GET['points'];

       if ($per == 0)
       {
         return 0.0;
       }

   $result = $sqsum - ($avg*$avg)/((float)$per);
   if ($result < 0)
   {
    return 0.0;
   }
   $rms = sqrt($result/$per);
   return 3*IACScale*$rms/4095;
     }

     function VDC()
     {
       $value = $_GET['dcvolts'];
       return 3*VDCScale*((float)$value)/4095;
     }

     function IDC()
     {
       $value = $_GET['dcamps'];
       return 3*IDCScale*((float)$value)/4095;
     }

     function ATD($channel)
     {
       $value = $_GET["atd$channel"];
       return 3*((float)$value)/4095;
     }

     $clock   = (int)$_GET['clockrate'];

     $freq    = Freq($clock);
     $phase   = Phase();
     $pf      = PF($phase);
     $lagging = Lag($phase);


 $acVolts = VRMS();
 $acAmps  = IRMS();

     $dcVolts = VDC();
     $dcAmps  = IDC();


     mysql_connect($hostname,$username,$password);
     @mysql_select_db($database)
       or die("Database $database not found at $hostname.");

     $query = "INSERT INTO powerdata (timestamp,dcvoltage,dccurrent,peakvoltage,peakcurrent,frequency,pf,lag
               VALUES
                 (NOW(),$dcVolts,$dcAmps,$acVolts,$acAmps,$freq,$pf,$lagging)";
     mysql_query($query);
     mysql_close();
    ?>
   Thank you for your contribution to our prestigious database.
  </body>
</html>
```

C.3 Server-side display code (PHP)

```
<html>
  <head>
    <title>Smart Battery Charger Power Data</title>
    <meta http-equiv="refresh" content="5" />
  </head>
  <body>

    <?php
     $hostname = "dbhost.nmt.edu";
     $user     = "cboling";
     $password = "4l3phNu11";
     $database = "cboling";

     mysql_connect($hostname,$user,$password);
     @mysql_select_db($database)
       or die("Database $database not found at $hostname.");
     $query = "SELECT * FROM powerdata
               ORDER BY timestamp DESC
               limit 1";
     $row     = mysql_query($query);

     $time    = mysql_result($row,0,"timestamp");

     $acvolts = mysql_result($row,0,"peakvoltage");
     $acamps  = mysql_result($row,0,"peakcurrent");
     $freq    = mysql_result($row,0,"frequency");
     $pf      = mysql_result($row,0,"pf");

     $dcvolts = mysql_result($row,0,"dcvoltage");
     $dcamps  = mysql_result($row,0,"dccurrent");

     mysql_close();
    ?>

    <p>
      <i>Logged on <?php echo $time; ?></i>
    </p>

    <p>
      <font size="5">AC Output Characteristics</font>
        <br /><b>Voltage:</b>
<br /><?php echo $acvolts; ?> volts RMS

        <br /><b>Current:</b>
        <br /><?php echo $acamps; ?> amps RMS

        <br /><b>Frequency:</b>
        <br /><?php echo $freq; ?> Hz

        <br /><b>Power factor:</b>
        <br /><?php echo $pf; ?>
    </p>

    <p>
      <font size="5">DC Output Characteristics</font>
        <br /><b>Voltage:</b>
        <br /><?php echo $dcvolts; ?> volts
        <br /><b>Current:</b>
        <br /><?php echo $dcamps; ?> amps
    </p>

  </body>
</html>
```

D. Microcontroller code (C)

D.1 `easyweb.c` excerpt – Measurement Interrupts and Configuration

```
void InitTimer(void) {
    int i;

    freqCount = upEdge = upTime = 0;

    _DINT();

    TBCTL = TBSSEL_1 | ID_3 | MC_2 | TBCLR; //ACLK, divide by 8, continuous mode, reset counter (500 kHz clo
    TBCCTL0 = CM_3 | CCIS_1 | CAP | CCIE;   //capture on both edges on P4.0

    _EINT();
}

void InitADC(void)
{
  unsigned char   i;
  unsigned char * p;

  for (i=0;i<8;i++)
        result[i] = 0;
  vACTotal = iACTotal = vSqTotal = iSqTotal
  = vACSum = iACSum = vSquareSum = iSquareSum
  = pointsCt = pointsPerT = 0;

  _DINT();

  ADC12CTL0 &= ~ENC;

  ADC12CTL0 = ADC12ON | SHT0_2 | MSC;                                      //ADC on, 2.5V ref, sample conti
  ADC12CTL1 = ADC12SSEL_0 | ADC12DIV_0 | CSTARTADD_0 | SHP | CONSEQ_3;      //repeat sequence of channels m

  p = (unsigned char *)&ADC12MCTL0;
  for (i=0;i<8;i++,p++)
    *p = SREF_0 | i;
  ADC12MCTL7 |= EOS;
  ADC12IE |= BIT7;                                                          //turn on ch0 interrupt

  ADC12CTL0 |= ENC;                                                         //start sampling
  ADC12CTL0 |= ADC12SC;

  _EINT();                                                                  //turn on interrupts in general
}

interrupt (TIMERB0_VECTOR) isr_ch0capture (void)
{
    unsigned int timerval = TBCCR0;
    unsigned int dummy = TBIV;
    if (P4IN & 0x01)     //we just saw a rising edge
    {
        if (upEdge)         //if we've seen one already
        {                   //then we know the period, so compute it
            if (timerval <= upEdge)
            {
                freqCount = timerval + (0xffff - upEdge);
            }
            else
            {
                freqCount = timerval - upEdge;
            }

            upEdge = 0; //and reset the rising edge
        }
        else
        {
            upEdge = timerval;
        }

        pointsPerT = pointsCt;
vACSum    = vACTotal;
iACSum    = iACTotal;
        vSquareSum = vSqTotal;
        iSquareSum = iSqTotal;
vOverSum   = vOverTotal;
iOverSum   = iOverTotal;

        pointsCt = vACTotal = iACTotal = vSqTotal = iSqTotal = 0;

    }
    else                    //we just saw a falling edge
    {
        if (upEdge)         //if we've seen a rising edge
        {                   //then we know the duty cycle, compute the high time
            if (timerval <= upEdge)
            {
                upTime = timerval + (0xffff - upEdge);
            }
            else
            {
                upTime = timerval - upEdge;
            }
        }
    }

    TBCTL &= ~TBIFG;
}

interrupt (ADC_VECTOR) isr_convcomplete (void)
{
unsigned long vSquared,iSquared;
    unsigned int * resultPtr = (unsigned int *)&ADC12MEM0;
    unsigned char i;
    for (i=0;i<8;i++)
        result[i] = *(resultPtr++);

vACTotal += (unsigned long)result[2];
iACTotal += (unsigned long)result[3];

vSquared = (unsigned long)result[2]*result[2];
iSquared = (unsigned long)result[3]*result[3];

    vSqTotal += vSquared;
if (vSqTotal < vSquared)
```

```
{
vOverTotal++;
}

    iSqTotal += iSquared;
if (iSqTotal < iSquared)
{
iOverTotal++;
}

    pointsCt++;
}
```

D.2 `httpquery.h` – HTTP Client (header)

```
/*************************************************************
 *****                                                   *****
 *****  Name: httpquery.h                                *****
 *****  Date: March 2010                                 *****
 *****  Auth: Charles Boling                             *****
 *****  Func: header-file for httpquery.c                *****
 *****                                                   *****
 *************************************************************/

#ifndef __HTTPQUERY_H
#define __HTTPQUERY_H

#include <io.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "tcpip.h"

#define TX_BUFFER_SIZE       170
#define NUM_Q_ELEMS          5
#define NUM_FIELDS           11

#define CLOCK_CYCLES_PER_SEC 250000
#define AC_VOLTS_OFFSET      1925             //the offsets applied to the real waveforms in ADC ticks (1.41
#define AC_AMPS_OFFSET       2184          //

#define HTTP_SEND_STARTED    0x01
#define HTTP_CLOSE_REQUESTED 0x02

void InitHTTPGet(void);
void HTTPClient(void);
unsigned char * CreateGetString(unsigned char * ptr);

typedef struct dataStruct {
    unsigned int values[NUM_FIELDS-1];
} powerdata;

void          qpush (unsigned char toAdd);
unsigned char qpop  (void);

typedef enum {
    CLOSE_EXISTING,
    WAIT_TO_CLOSE,

    MAKE_CONN,
    ESTABLISH_CONN,

    MAKE_STRING,
    SEND_STRING,
    IGNORE_PAGE,

    ERROR
} THTTPStateMachine;


extern unsigned long vSqTotal,iSqTotal,vACTotal,iACTotal,vSquareSum,iSquareSum,vACSum,iACSum;
extern unsigned int  freqCount,upTime,pointsCt,pointsPerT,vOverTotal,iOverTotal,vOverSum,iOverSum;
extern unsigned char done;
extern signed int    result[8];

THTTPStateMachine HTTPStateMachine;

#endif
```

D.3 httpquery.c – HTTP Client (implementation)

```
#define __ME
//#define __GEEK
//#define __RAINBOW
//#define __MYPC
//#define __WILLIAM

#include "httpquery.h"

unsigned long vTotal,iTotal,vSquareSum,iSquareSum;
unsigned int  HTTPBufferLength;
unsigned char done,start,end,fill,buffer[NUM_Q_ELEMS],HTTPSent;
unsigned char *txBuffer, *HTTPPointer;

#ifdef __RAINBOW
const unsigned char scriptpath[] = "/~cboling/ee382/logger.cgi";
#else
const unsigned char scriptpath[] = "/~aleph0/logger.php";
#endif

const unsigned char * fields[] = {
"clockrate",
"vsqsum",
"isqsum",
"vacsum",
"iacsum",
"vovflows",
"iovflows",
"period",
"uptime",
"points",
"dcvolts",
"dcamps",
"adc2",
"adc3",
"adc4",
"adc5",
"adc6",
"adc7"
};

const unsigned char footer[] = {
" HTTP/0.9\r\n"
"Connection: close\r\n"
"\r\n"
};

const unsigned char response[] = {
"</html>\n"
};

void OutOfMemory(void)
{
    return;
}

void InitHTTPGet(void)
{
    srand(result[0]);

    #ifdef __ME
    //my desktop PC
    *(unsigned char *)RemoteIP       = 192;
    *((unsigned char *)RemoteIP + 1) = 168;
    *((unsigned char *)RemoteIP + 2) = 2;
    *((unsigned char *)RemoteIP + 3) = 3;
    #endif

#ifdef __MYPC
*(unsigned char *)RemoteIP       = 129;
    *((unsigned char *)RemoteIP + 1) = 138;
    *((unsigned char *)RemoteIP + 2) = 36;
    *((unsigned char *)RemoteIP + 3) = 192;
#endif

#ifdef __GEEK
    //geek.nmt.edu
    *(unsigned char *)RemoteIP       = 129;
    *((unsigned char *)RemoteIP + 1) = 138;
    *((unsigned char *)RemoteIP + 2) = 40;
    *((unsigned char *)RemoteIP + 3) = 11;
#endif

    #ifdef __RAINBOW
    //infohost.nmt.edu
    *(unsigned char *)RemoteIP       = 129;
    *((unsigned char *)RemoteIP + 1) = 138;
    *((unsigned char *)RemoteIP + 2) = 4;
    *((unsigned char *)RemoteIP + 3) = 51;
#endif

#ifdef __WILLIAM
*(unsigned char *)RemoteIP       = 192;
    *((unsigned char *)RemoteIP + 1) = 168;
    *((unsigned char *)RemoteIP + 2) = 2;
    *((unsigned char *)RemoteIP + 3) = 20;
#endif

    TCPLocalPort  = (rand() % (65535-1024)) + 1024;
    TCPRemotePort = 80;
    HTTPStateMachine = MAKE_STRING;
    done = HTTPBufferLength = HTTPSent = 0;
    return;
}

void HTTPClient(void)
{
    unsigned int HTTPBytesToSend;

if (SocketStatus & SOCK_ERROR_MASK)
{
OutOfMemory();
}

    if (!(SocketStatus & SOCK_CONNECTED))
    {
        if (!(HTTPSent & HTTP_SEND_STARTED))
        {
            TCPLocalPort = (rand() % (65535-1024)) + 1024;
            /*
            if ((TCPLocalPort < 1024) || (TCPLocalPort > 65530))
            {
                TCPLocalPort = 1024;
            }
            else
            {
                TCPLocalPort += rand()%5 + 1;
            }
            */
            HTTPSent         |= HTTP_SEND_STARTED;
            HTTPStateMachine  = MAKE_STRING;
        }
    }
    else
    {

        /*
        if (SocketStatus & SOCK_DATA_AVAILABLE)
        {
            TCPReleaseRxBuffer();
        }
        */

        switch(HTTPStateMachine)
        {
            case MAKE_STRING:
            {
                txBuffer = (unsigned char *) malloc(TX_BUFFER_SIZE);
                if (!txBuffer)
                    OutOfMemory();
                memset(txBuffer,0,TX_BUFFER_SIZE);
                txBuffer = CreateGetString(txBuffer);
                HTTPPointer = txBuffer;
                HTTPBufferLength = strlen(txBuffer);
                HTTPStateMachine = SEND_STRING;
                break;
            }
            case SEND_STRING:
            {
if (!HTTPBufferLength)
{
HTTPStateMachine = CLOSE_EXISTING;
break;
}

                if (SocketStatus & SOCK_TX_BUF_RELEASED)
                {
                    if (HTTPBufferLength > MAX_TCP_TX_DATA_SIZE)
                    {
                        HTTPBytesToSend = MAX_TCP_TX_DATA_SIZE;
                    }
                    else
                    {
                        HTTPBytesToSend = HTTPBufferLength;
                    }

                    memcpy(TCP_TX_BUF, HTTPPointer, HTTPBytesToSend);
                    TCPTxDataCount = HTTPBytesToSend;
                    TCPTransmitTxBuffer();

                    HTTPPointer      += HTTPBytesToSend;
                    HTTPBufferLength -= HTTPBytesToSend;
                }
                break;
            }
            case IGNORE_PAGE:
            {
                if (SocketStatus & SOCK_DATA_AVAILABLE)
                {
                    if(strstr(RxTCPBuffer,response))
                    {
                        HTTPStateMachine = CLOSE_EXISTING;
                    }
                    TCPReleaseRxBuffer();
                }
                break;
            }
            case CLOSE_EXISTING:
            {
                if (SocketStatus & SOCK_DATA_AVAILABLE)
                {
                    TCPReleaseRxBuffer();
                }

                if (HTTPSent & HTTP_SEND_STARTED)
                {
                    free(txBuffer);

                    TCPClose();
                    HTTPSent &= ~HTTP_SEND_STARTED;
                }
                break;
            }
        }
    }
}

unsigned char * CreateGetString (unsigned char * arg)
{

    unsigned char i,offset,*p = arg;
unsigned long lvalues[4] =
{
vSquareSum,
        iSquareSum,
vACSum,
iACSum
};
    unsigned int values[sizeof(fields)/sizeof(char *) - 1] = {
vOverSum,
iOverSum,
        freqCount,
        upTime,
        pointsPerT,
        result[0],
        result[1],
        result[2],
        result[3],
        result[4],
        result[5],
        result[6],
        result[7],
    };


    p += sprintf(p,"GET %s?",scriptpath);

p += sprintf(p,"%s=%lu",fields[0],CLOCK_CYCLES_PER_SEC);

i = offset = 1;
do
{
p += sprintf(p,"&%s=%lu",fields[i],lvalues[i-offset]);
i++;
} while (i<=sizeof(lvalues)/sizeof(unsigned long));

    i = offset = offset + sizeof(lvalues)/sizeof(long);
    do
    {
        p += sprintf(p,"&%s=%u",fields[i],values[i-offset]);
        i++;
    } while (i<(sizeof(fields)/sizeof(char *)));

    p += sprintf(p,"%s",footer);

    return arg;
}

void qpush (unsigned char toAdd)
{
    if (fill < NUM_Q_ELEMS)
    {
        fill++;
        buffer[end] = toAdd;
        end = (end + 1) % NUM_Q_ELEMS;
    }
}

unsigned char qpop (void)
{
    unsigned char ret;
    if (fill)
    {
        fill--;
        ret = buffer[start];
        start = (start + 1) % NUM_Q_ELEMS;
        return ret;
    }
    return -1;
}
```