NORTHERN ILLINOIS UNIVERSITY

Automated Brewing Process

A Thesis Submitted to the

University Honors Program

In Partial Fulfillment of the

Requirements of the Baccalaureate Degree

With Upper Division Honors

Department Of

Mechanical Engineering

By

Joshua Berger

DeKalb, Illinois

May 11, 2013

University Honors Program

Capstone Approval Page

Capstone Title (print or type)
Automated Brewing Process

Student Name (print or type) Joshua Berger

Faculty Supervisor (print or type) Nicholas Pohlman

Faculty Approval Signature Nicholas A. Pohlman

Department of (print or type) Mechanical Engineering

Date of Approval (print or type) 5/6/13

Honors Thesis Abstract

There are few automated gas control systems that incorporate anything more than proportional control. This thesis will look at the process of making beer at home with a three kettle setup, the possible parts that would need to be incorporated in the system to allow proportional integral derivative control, and how the control theory may work. The research required will look into the basics of control laws. A design hurdle that will need to be met is a control valve that is within the weekend home brewers price range. With a truly robust control system, a home brewer would save money and time. The final conclusion that was approached is that there is room for the product, but the product is not currently producible.

HONORS THESIS ABSTRACT
THESIS SUBMISSION FORM

AUTHOR: Joshua Berger

THESIS TITLE: Automated Brewing Process

ADVISOR: Nicholas Pohlman

ADVISOR'S DEPARTMENT: Mechanical Engineering

DISCIPLINE: Mechanical Engineering YEAR: Senior

PAGE LENGTH:8

BIBLIOGRAPHY: Yes

ILLUSTRATED: Yes

PUBLISHED (YES OR NO): No

LIST PUBLICATION: None

COPIES AVAILABLE (HARD COPY, MICROFILM, DISKETTE): Hard Copy

ABSTRACT (100-200 WORDS):

Automated Brewing Process

Making beer is an art. Fermented beverages have been around for millennia, and it is thought that beer has been around for over six millennia (Hornsey). At times, it was used as a substitute for the poor water supplies, due to the act of heating the water involved to above 175°F. The heating kills most of the bacteria and microorganisms that could live in non-clean water supplies. In modern times, it has become big business, with many beer companies turning out over one million gallons of beer every day (Brewer's Association). These companies have perfected the automated processes that can allow that figure. On the other hand, there are many people around the world that produce their own beer. This can be as little as a gallon at a time or as much as 100 gallons at a time. Many home brewers rely on their own hands to make sure these gallons come together. This is something that our senior design group is working to overcome. We have designed an automated home brew system, along with an associated process. This would allow a person to have the machine repeatability of the larger brewing companies, without having to leave their garage. This paper is in addition to the work done on the engineering senior capstone in a group with Humza Shamsuddin and Nick Skuban. The appendix shows the work completed to finish the requirements in Mechanical Engineering. This system is intended to be a secondary control device, and is only a theoretical review.

The system involves three kettles, a stand, two burners, a heat exchanger, some piping, some wiring, and a processor. The mechanical portion will be very straightforward, and the computerized side will be controlled by an Arduino, a small,

low-cost processor that has a simple coding language. Of the two portions that I had the most interaction with, the heat exchanger in the plumbing and the gas system, I will be focusing this paper on the gas system. I will start by looking at what we have, and how this system works, and then I will go into a discussion of what could be done in the future to make the system better.

It takes time and energy to boil water. We have initially chosen liquid propane (LP) to be our energy source. LP is commercially available, has a good amount of energy stored in it, and has a wide array of products on the market to use for transfer. The LP comes in a pressurized vessel that comes in at around 145 psi. The connection to the tank generally brings it down to an operating range of around 30 psi. This is near what is used on motor homes, gas grills, and the like. From there, it will be further regulated down to a final operating range of less than one-half psi, or less than 11inches water column. This is an industry switch that threw me for a loop at first, but the water column measurement is used when the measurements get to be smaller than one psi. It helps to further delineate small measurements, and meets and industry standard for home appliances fed off of natural gas.

At this point, the gas has been regulated down to one-half psi. It is then piped into a manifold, so as to split the gas between the two burners in an even manner. From here both sides mirror each other so I will only explain it once. There is an automated valve. This valve is generally used in the home heating industry on automatic or semi-automatic heating systems. It takes a signal from a controller to do its job, so there must be a separate control system that is installed with the valve. This valve serves two purposes; it controls the gas to the burner, and it also controls the gas to the pilot for the

burner. This is somewhat important, as it allows for a simpler control system. From this valve, the gas then either flows to the pilot or to the burner. The pilot is also used in the home heating industry. It is the type that has both the igniter and the thermocouple built into it, so it will ignite the gas and then sense whether there is a constant flame. The burner does the job it sounds like it does; it takes the energy trapped in the gas and turns it into a form that can be used for heating the water. The burner is directly fed from the automatic valve. The final piece of the puzzle is the automatic controller for the automatic valve and the pilot. This is designed to work with both, and will perform the operations independent of our controller. The only interaction that will be needed is the signals back and forth over when heat is/is not needed.

The system will be rather simple, and only work in an on/off manner. An explanation is as follows. To start the process, water will be added to kettle one. Once the thermocouple installed senses the water temperature, the Arduino will send a signal to the gas control system. The controller will interpret the signal and send a signal to the automatic valve to release gas to the pilot. After the pilot has gas, it will try to ignite. Following ignition, the pilot will then heat up, showing there is a constant gas flow. When the pilot is warm enough, a signal will go back to the controller. This signal will be interpreted and a different signal will be sent to the automatic valve to open. When the valve opens, the pilot will ignite the gas that is flowing to the burner. Once the water has gained enough energy, and the thermocouple says so, the controller will send a signal to stop the flow of gas. This will extinguish the flame to the pilot and the burner. This process is similar to those in natural gas ovens and furnaces around the country.

This is ultimately where the system is at its weakest. There are essentially two conditions, on and off. There is no in between to maintain a constant gas flow at a lower value. This would allow for less overall temperature fluctuation, and would potentially lead to less gas being used. There will be two parts to this: mechanical and control logic.

The mechanical portion of the system would allow a user to control the amount of gas flowing to the burner. This isn't so important during an initial warm up where the water starts out at room temperature and is going up to 170°F. During most of this heat up period, a mechanical controller, such as a throttle valve, would be wide open allowing full gas flow. Where it would come into play is getting near that 170°F threshold. A conservative person might say that never going over 170°F would be a good thing, so they would cut back the gas flow a bit early, and as temperature climbs slower and slower, they would constantly throttle back. This could be done either by hand, or by a motor operated valve, controlled by a computer with the appropriate logic. Another person might just let the overshoot happen, let the temperature go past 170°F, but on its way down, allow the gas flow to come on in a slow manner. This would allow for an overshoot, and then a controlled descent back to the control level. These are some of the ideas that I will be looking at.

A throttle valve, an orifice, or a restrictor, are ways that flow in a liquid or gas system can be held up. The control method for each is essentially the same; the passage of the flowing fluid is made smaller than the surrounding piping, leading to less flow reaching the intended source. This is used every day in flood gates, turbine systems, automobiles, and any other area where less can be more. For this application,

I would prefer to use a throttle vale, as they can be both motor and hand controlled. There is also the possibility of using a membrane and bladder or spring to control, but they are generally better in high volume, high flow situations. This low flow situation could lead to some damaging situations where gas flow could be messed up and someone could get seriously hurt.

A slightly different way to go would be a valve with an electro-mechanical restrictor. This would have to be completely designed from the ground up. The manner I am speaking of would have an elastic membrane inside the valve, and a motor driven manner for constricting a loop that is in the valve. This would have the effect of essentially tying a rope around a fire hose. As you pull tighter on the rope, the fire hose would emit less and less water. The reason I make this suggestion is that many throttle valves do not have a good linear manner for restricting flow. As a standard gate valve closes, it takes approximate 40% closure to see any appreciable effect on flow, and then is essentially closed at near 20% open. This leads to sudden, jerky changes in the flow through the system. If the constriction of the flow were to happen in a more linear manner, the system would have less oscillation, meaning there could be a more accurate flow, allowing for better use of the energy in the system.

The second portion of this control system is the logic that is used. As Dr. Coller said in his class "Designing control systems is an art (Coller)." After seeing a control system for a coal-fired boiler this past summer, this view was reinforced. The control systems we played with in class had us control an inverted pendulum and move a craft through space. All of these control systems had something in common; there were many ways to make them ultimately work. No two students in the control systems class

had the same numbers, and the boiler was always being tinkered with. Unless there was something that went wrong, as long as a person was on the right path, they could make a system work. Some solutions could make for some interesting moments, but more than likely if you could make your system to Dr. Coller's "robust" standards, then there was very little chance of failure.

The system that I have in mind would start by finding out how much energy would be transferred from the burner to the kettle at any given time. This would consist of many differing heating's from ambient to boiling. By repeating this step more and more, a better data set, and ultimately a better solution could be found for seeing how much energy, time, and gas it takes to heat up an amount of water to a specific value. This would then be intertwined with other ambient factors, such as temperature, wind speed, and humidity, to find a real world value for the amount of energy necessary for the heating of the water. The outside factors would then be programmed into the logic to affect how the system looks at the heating of a kettle.

The other part of the heat up testing that would be performed would be an overshoot test. This would test both how long after removal of heat that the water takes to cool and how far over a set point that the temperature goes in a kettle. The second part of this will again change with differing ambient conditions, so it would need to be researched in many different times of the year. By getting all of this, once the control programming is in place, a person could set the amount of overshoot that they want to see, or even look at the overshoot in comparison to what was predicted by the controlling system. If things seem to be off, a change could be made to bring the system

more in line with what the real world dictates, as there is no real way to cover every situation that is out there (Coller).

Once the data has been taken and analyzed, it would be time to work on process implementation. There is already a system available to piggyback off of to build the signals for the valve. It would be writing the programming to make it all work that would be the hard part. It would start with a look at how the kettles react to the energy being expended to make the water within warmer. After the experimentation, there should be an equation that was fit to the data (Pohlman). This equation would be the characteristic equation, which could end up being long and scary with non-real portions. It would be used to work on the programming. Once the characteristic equation has been found, it will be manipulated in a manner to find out where its roots are. Once these roots are found, one can determine how everything will act when it is applied to the programming by using a method called root locus. Once the behavior has been mapped via root locus, and the programming is done, it will be to test the control system thoroughly to determine if it is robust enough (Coller). This can be as simple as turning on the system and seeing things go awry, or it could be as difficult as fifteen runs to duplicate that small misstep that is causing things to spiral in only precise conditions. This is the portion where being a well-disciplined artist can be helpful. By understanding the system, how it works, and how it is supposed to react, one can see errors when they are getting ready to occur, and change the course for less rocky waters.

The project that this is based off of will more than likely never receive a true control system for the purposes of heating. At this point, it is too expensive and unwieldy. It would take much more experience at building control systems, and much

more ability to control the amount of gas being used. It would also take a rework of the burners, which are already not being used in the proper manner. They are designed to work at near 30 psi. The use of the furnace components at one-half psi means a different orifice is used to connect the piping to the burner. There is still a small matter of understanding what this nozzle does that would have to be overcome, along with the ability to change the system on the fly that will not exist when we are done. The control system that was talked about in the paper was strictly designed for a true test of what has been taught/learned over the last four years of my education. I would love to have the time and ability to sit down and build this control system from the ground up, but I fear that will not happen until I have more time and money available, if at all. The theoretical design that was discussed relied heavily upon the testing of the burner system, and a thorough knowledge of its particulars. The ultimate challenge that arises from trying to make a control system is fully understanding the components that are in the system, and how they interact, not only with each other, but with the entire system.

References

Brewer's Association. *Brewers Association | Facts.* 1 April 2013. website. 2 April 2013.

Coller, Brianno . "Dynamic Systems and Controls 1." DeKalb, 3 May 2012. Class.

Hornsey, Ian S. *A History of Beer and Brewing.* Cambridge: The Royal Society of Chemistry, 2003. Book.

Pohlman, Nicholas. "Experimental Methods in Mechanical Engineering." DeKalb, 1 December 2012. Cass.

MEE482 Project Report:

Automated Brewing Process

Group 13

Josh Berger
Humza Shamsuddin
Nick Skuban

Dr. Nicholas Pohlman, Ph. D.
Dr. Jenn-Terng Gau, Ph.D.

May 4, 2013

Abstract

Good beer is hard to find; it is also becoming more expensive when found. There are companies out there that are willing to either sell their recipes for good beer to anyone that is willing to put in the time. Using a number of standard components, and designing our own where necessary, we designed an automated process to go alongside the components necessary to create a batch of home-brewed beer. This project encompassed most all aspects of mechanical engineering, specifically highlighting: fluids, static structures, control systems, design of experiments, and programming. The design of the heat exchanger and the programming are unique to the design, but could be manipulated for many other areas. The system uses gas components that are shared with the heating industry, and could be adapted to use for cooking. The system is essentially stand alone, and would be a great addition to any home brewers garage.

Contents

List of Figures and Tables

Acknowledgement

We would like start by acknowledging Nick Skuban. This was initially his idea, and he is the primary financial contributor of this project. In addition, Humza Shamsuddin and Josh Berger have contributed additional financial support. We would next like to thank Dr. Nicholas Pohlman and Dr. Jenn-Terng Gau for the advising and direction we received throughout the past two semesters. Our family members have sacrificed countless hours without us throughout this project. A special thanks goes to DeKalb Mechanical, as they made a small donation to the system. We would also like to thank all of our professors through our time in school, whether it is high school, college, or the real world. Without these individuals, we would not have the opportunity for this project.

Chapter 1 Introduction

1.1 Introduction

Beer has been around much longer than consistently clean water supplies. It helped get people through hard times, and has grown into a multi-billion dollar worldwide industry. There are any number of grains and types of hops that go together to make anything from a wheat lager to a rye stout to a soy pale ale. Each brewer, each beer has its own combination of brew times, fermentation schedules, and bottling procedures. These combinations have proven to create a wide array of tastes that are suited to any number of individuals.

As of late, money-thrifty people have chosen to brew their own beers in the comfort of their own house to save some money, and get a little bit of exercise. This process can be done in a kitchen with a big kettle and a bucket, but who wants to deal with possibly getting beer all over their kitchen. This has given way to people getting used kegs, opening them up, and using them as the kettles for bigger productions. This is getting more advanced as there are companies that see this and are taking advantage of this situation.

This advantage needs to be capitalized on by pre-packaging a system for each brewer to use for their own homes. It should take an existing system, and make it even better. We are taking a three keg system with burners under each keg, and automating it. By automating, we are not completely automating it, but that would be something to look at in the future. We will be taking the moving of the liquids, the turning on and off of the burners, and a timer for telling a person when to do things and automating them to the point where the only time a person works is to initially add water, add the grains, add

the hops, add the cooling liquid, and adding the yeast to the fermentation bucket. This should help with making the batches more consistent.

The making of beer consists of four major parts: water, grains, hops, and yeast. The water is the base for the entire process. This is something that will not go away. The grains are normally barley, but can also be rye, wheat, soy, etc. This will affect the flavor and alcoholic content of the beer. The hops are what contribute the smell and the bitterness to the beer. The yeast takes the sugars that come from the grains, and converts them to the alcohol that is the ultimate end game.

1.2 Background for Brewing Process

The process takes around three to three and one-half hours to complete from adding the water to the placing the wort in the fermentation vat, and adding the yeast. This will be discussed below.

To start, five and one-half gallons of water is added to kettle one and the heat is turned on. The burner will now be on until the temperature reaches 170 F. Once the water reaches temperature, four gallons is transferred over to the second kettle, and the grains are added. The first kettle will be maintained at 170 F, and the second kettle will be maintained at 155 F. While maintaining these temperatures, the liquid in the second kettle will be circulated for an hour (or as specified for the grains from the supplier) through the second kettle. After this is complete, the liquid in kettle two will slowly be transferred to the third kettle, while the remaining liquid in kettle one will rinse the grains in the second kettle. All of the liquid, which at this point is wort, is now in the third kettle, and the heaters for the first two kettles have been turned off, while the third heaters has been turned on. With the heat being on, the wort is taken to boiling, and the hops are

added at the rate recommended by the supplier. This will be taking place while the heater is maintaining a boiling condition, most likely taking about an hour. Once this is done, the heat will be removed, and the wort will be cooled to 70 F. This will be done via a cooling coil and some ice water that has been placed in kettle one. This will be recirculated through the coil. Once the wort is at 70 F, the cooled wort will be transferred to the storage vat, and the yeast will be added. Once the yeast is added, clean-up will take place, and the wort with yeast will store for the amount of time that the supplier/producer chooses.

This is right now a heavily man-power intensive project, as the kettles with the liquid in them are rather heavy, and they are also very warm. This is very dangerous, and errors can easily exist. By adding in the pumps, along with some relatively solid plumbed pipes, this lifting of the liquid, along with the necessity of potentially burning ones-self, goes away. The next step is to know when the amount of heat applies to the kettles is enough. By adding thermocouples to the process, we move from having to hold a thermometer over the process or look at a thermometer less than two feet from an open flame, to being able to set-up a remote display that takes the person away from the heat. These thermocouples will be hooked up to a computer, that will allow us to run these temperatures to a solenoid valve on the burners, allowing the burners to be placed in an intermittent state. This will save on the amount of gas used, and also prevent a person from having to possibly light a burner a number of times to keep the temperature constant. The final portion to look at is potential pressure gauges. These, along with the computer, could give a final specific gravity of the liquid. This is

somewhat critical in knowing when the wort is done, and how the beer should taste after the fermentation process is done.

1.3 Initial Thoughts

This project is a serious challenge with the amount of work that is necessary. There are six major components that make up the system: frame, plumbing, ignition, brewing, electrical and the heat exchanger. Each of these systems can be broken into subsystems, which will be discussed in much more detail later. The quickest thing to get done is the frame. We know what we need, which is something to hold the kettles (three), the pumps, the valves, the piping, the heat exchanger, the electrical components, and the gas system. This was made out of steel on site, using the machine shop.

The next step is to get the water side plumbing done. The biggest choice was stainless steel vs. copper vs. silicon tubing. There was a compromise in the end for maintenance ability.

The gas system was next to follow, as it will be allowing for the heating of the kettles throughout the process. By getting something that is pre designed, we do not need to rely on more programming, and we know that is will work as designed.

The process as we designed it needed a heat exchanger that could serve the entire system. This was done to help reduce the gas system cost, and allow for an extra design element. The heat exchanger is a new design, and will take time to ensure it operates as expected.

The valves will need to allow for more than one position, otherwise it will be extremely hard to get the water moved around as we find necessary. We would also like

them to be operated by the programming, as to allow for a better automation, and for better timing of the system.

The electrical components will end up being the biggest challenge. We walked in knowing we would use the Arduino to control the system, but we were completely unsure how everything would go from there. There will need to be connections for all of the valves, the gas system, the temperature and pressure sensors, and a way to power everything. There will also need to be a specialized circuit for this system, as there is nothing on the market that satisfies the requirement we have.

1.4 Professional Responsibilities

As a small group of engineers, we must consider our responsibilities not only to other human beings, but also to the environment. There are also the questions hanging over head if the project will be profitable, will it be manufacture able, will it cause health concerns, and does it fall within societal norms for what everyone expects? These questions should be weighed by all engineers. Without the ability for the engineer in design to see where the project could cause issues, there could be may long nights spent doing unnecessary redesigns that could have been skipped with the appropriate thoughts out of the gate.

The first thing we will look at is the economic component. As with any project, unless there is someone willing to pay more than it is worth, it will need to make a profit. For our project, this may be a small challenge as it will be a customer driven job. We will take orders and customize the system to each customer's specifications. This will cause many headaches during our start-up period, as we cannot expect each customer to want the same thing. There may be components or systems that we decide should be

the same for each customer, but that will take time to see what each customer wants when the system is ordered. There is also the question of how much time will it take to start making money. We will need an amount of capital to ensure we can make each unit in house. As we make more, we can look at what parts we can't replicate on site, and figure out where we will make future purchases.

The environment is something that all people should be able to enjoy for years to come. If we create a product that will harm the environment, then is the harm to the environment ultimately worth it? With our design, our two biggest visible environmental challenges will be a reuse of water, and the burning of a fossil fuel. The water will be a challenge, as there is a need for upwards of 30 gallons per batch, and about half of that will be used to cool the other half. If this water is dumped to the environment, how can we be sure that we are not dumping something that is harmful? If there is nothing harmful, why do we not recycle this water? It would be more economical and better for the water challenged future. There are also a few smaller less slightly issues in the materials used and abused to make the system. The steel has to come from somewhere, so we would like to source a recycled product as much as possible. We are not choosing special steel, so this should be rather easy. It will also help our bottom line, along with our customer's pocketbooks. The other material that will be in high used is silicon. This is starting to become a precious resource with the amount of computer systems it is used in, so if there becomes a different way to make the circuits, we should hop on that to help the environment. There is also a large amount of copper in use. This has become more valuable recently, so if there is a better choice for materials out there, we may want to research it. By using recyclable materials, we are helping with the

sustainability of both our industry, and our product. With the sustainability going up, we can use this as a sales point to help with our futures. There also becomes a chance to find new materials that we can pass on to the industry. By doing this, we can make the entire industry a better servant for the future of the Earth.

The manufacturability and reparability of any product should be weighed by any engineer before a product is made. If there is a brand new design, but it can't be put together due to the intricacy, then what use is it? If something is meant to be put together once and never fixed, what happens when a part on it breaks? This is why engineers should evaluate the project from both perspectives. It is our responsibility to ensure that our customers are happy, so if we design something that is useful and can be fixed simply, they will be more willing to buy a better product from us than shop elsewhere.

To look at ethics, politics, and health and safety, we need to look at them as a whole. There is no single part of any of those that doesn't rely on any other part. The ethics of making beer is tricky, as the alcohol involved can make people act in ways that they shouldn't. If a person that buys our product, and they get drunk off of their beer from our product, does that make us the ultimately responsible party? We would like to think not, but that is not ultimately for us to decide. It involves and amount of politics that are beyond our level/pay grade. By ensuring that we are selling to a mature audience, we can take some of the youthful exuberance out of the equation, but it still remains that beer can make a smart person stupid, and an experienced person into a novice. The politics involved are straight-forward from a government level, as we will need patents, lawyers, and the ability to operate a business. There is then the human level of politics;

do we want to infringe upon a field that can get tricky, or should we market ourselves as something different? If we make our own market, how will that impact both our suppliers, and the competition? Being that we will be producing our systems in house to start, how will the metal cutting, welding, and machine work affect our health? This isn't the only health concern, as alcohol is a poison, as have been made illegal in the past. If we build this system and then sell the beer, this would be a much bigger problem, but we feel that if a person is willing to put the time into brewing a batch of beer, and then waiting for fermentation, they are willing to accept the risk that comes with the alcohol they are about to ingest. As long as we keep the product floor safe, the outside health concerns will take care of themselves.

1.5 Individual Contributions

The group split the work into areas where an individual's knowledge could be the most utilized. This allowed Nick to be our fabrication specialist, Josh to do a majority of the design elements, and Humza to knock the electrical system out of the park.

Josh Berger worked on all parts of the build, as he was willing to help anyone out to get things done. The major parts that he handled were the gas system and the heat exchanger. The gas system required finding a series of components that fit the design requirements of automation, and that could be purchased and put together will minimal challenges. The use of heating system components for most of the gas system was fortunate, as they have been designed and proven useful over time. They also allowed for us to find the parts in many different locations, allowing for simple replacement if something should go wrong. The heat exchanger was the next challenge, as we felt the available ones were no the best for our system. The design was done in SolidWorks,

and then a simple flow analysis was also done there. It showed that the sudden changes of direction would be the biggest problem. The problem is that the heat exchanger was not simulated using the plumbing, so there was no way to see if there would be any restrictions until after the entire system was put together. As for assisting others, Josh helped Nick with the building of the gas system after the components were received, ensuring the heat exchanger was cut in the proper manner, the installation of the heat exchanger, part of the redesign of the plumbing, the installation of the mounting hardware, and the final preparations before painting. Josh helped Humza with the wiring process, the use of SolidWorks to design the electrical, the setup of the wiring component box, and the testing of part of the electrical system. The final portion that Josh took care of was the writing of the papers, ensuring that all worked on it when necessary.

Humza designed the electrical system from the ground up and programmed the entire system. With previous experience in working on stereos and lighting for vehicles, this was up his alley. The programming was the first challenge. This was a different sort of challenge due to learning a new programming language. There was some fortune involved, as the language is a simple modification of the C++ programming language, which is taught to all engineers. He also found that the methods taught by Dr. Coller in the Numerical Methods using Computer Programming class could be applied, this allowed for many small codes that came together in the main body of the programming, allowing for each portion to be tested individually, and then the entire system could be hammered out as we all say fit. The biggest challenge that he had was getting the component electrical requirements down, since the last components did not come in

until April. The next step in his design became the connections for all of the components to the Arduino. This became a process that had to be reworked a couple times, until a self-designed circuit board was settled upon. At this point the traces were done, and he soldered everything onto the board. He performed the initial testing, repaired the small mistakes, and then started to put everything together. Once the circuit board was put together, it was time for final assembly. He assembled as much of the electronics as possible. After this was done, Humza and Nick worked together to design a sample process for demonstration after the presentation.

Nick was the man behind the money and the person who wanted the system. His biggest design portion was the frame. This was done early in the process, and was modified late to help with electrical support. He did the design and got help with the FEA from Josh. After the design, he submitted the work order to the College's machine shop, where he completed the frame build on shop time. After this, he brought in the kettles and started designing the plumbing. This was done next, as could mostly be done with what was on hand. After it was finished, he worked on the posters, and was the main contributor for the large poster and worked with Humza to finish the small one. After Josh had the heat exchanger designed, Humza purchased the aluminum, and Nick did all of the machining. It took some time, as the pieces purchased barely fit in any of the machines in the shop. After this, Nick and Josh installed the connections for the heat exchanger, and hooked in the plumbing. He then took to getting the frame ready for paint, by getting all of the support pieces in place, and rolling beads in the pans for stiffness and to channel any spills away from electrical components. After this, he worked on the presentation.

Chapter 2 Design Specifications, Concept Generation, and Evaluation

2.1 Design Specifications

Our design is to automate the home brew process. This will involve all of the system operating from a single controller. There will be built in safety points, allowing the operator to isolate the system for maintenance or cleaning. We are expecting the system to cost less than $2,000 out-of-pocket, while increasing the reliability of the process, decreasing the amount of time to make a batch, ensuring the system follows procedure for food safety and can later be certified, and is mobile for the home brewer.

The automating of brewing is not new. Many international companies are fully automated from the initial water addition to the end of the bottling process. This hasn't translated down the line to a person who wants to make their own beer at home. To this end, we took a standard progression for the home brewer, and we applied the amount of automation we felt was necessary: an Arduino for the programming, electronically controlled liquid valves that allow us to move the water/wort to the next step of the process, a gas system control module that controls the entire gas system with a simple input from the Arduino, and temperature and pressure sensors. All of this adds up to a system that can tell you how much water is where, how warm it is, heat it up, move it around, tell you the specific gravity (important to alcohol creation) and then cool the wort down. The amount of automation is not complete, as there is no direct connection to a water supply, the grains, hops, and ice must be added, the fermentation is done by calendar, and the bottling is still by hand. The extra amount of automation necessary for all of this is beyond the time given for this project. The grain and hops additions are possible for the near future, along with a graphical user interface to allow the brewer to

see where they are in the process, how things are moving along, and to change an input as necessary.

The next design specification is to decrease the brewing time by 30%. This time decrease is in reference to a manual batch done with one or two kettles, one burner, and one person. The time reduction can be broken down into a precision timing issue and a human factor issue. The timing used in the manual process is based on the brewer following a schedule. They must set a number of alarms for when to put the next step into play, generally after setting the alarm after completing the previous step. The error in these times will add up quickly, and make it so a quick three hour batch can go five hours. With a timer integral to the circuit, and know cycle times for valves, we can reduce the error involved. The biggest variable that the circuit cannot account for is the boil time, which is based on the gas system, which will be discussed later. The other part of the brew time is the human element. During a manual process, there can delays when trying to move liquid, light a burner, add the grains, etc. This is essentially eliminated in the automation process, as almost every step is done by the computer. Short of the system breaking down, or a brewer not adding the hops at the right time, the brew will be done at a set time. With future automation additions, these possible misses could also be eliminated, so just the water at the beginning and the chilled water at the end would be on the brewer,.

In keeping the beer free of contaminants, we chose to follow the NSF food safe requirements, with the possibility of gaining food safe certification at a later date. This is huge, as it ensures that what we do will not affect any of the brews in a manner that could harm a customer, and spoil our name. Keeping food safe will also allow for the

potential of selling craft beers in the future. This is not a sure thing, but if we maintain safety and a good head for creating beer, it is definitely a possibility in a long term scenario.

We understand that not all brewers will have a solid, solitary area for their brewing, so we are making the system so it can move around. This was done using casters on the four legs of the frame. This is important for when the water will not reach the product, or if there is need to store it in an area where it cannot be used. Also using the casters allows a brewer to move it into areas that the stand could be used for non-brewing things. We would like to think that the system could be used for making coffee, tea, and sarsaparilla, and it was brought to our attention that it could double as a turkey cooker or a stove with the amount of energy that is tied into the gas system being sent out at any time. This would make it necessary for us to bypass some of the coding, but it shouldn't be too hard to create a simple program that allows for the gas system to be on or off.

The system ultimately needs to be affordable for the home brewer and profitable for our potential investment. This is why we chose to keep the system under $2,000. We achieved this goal by about $50. The biggest costs incurred were in the gas system and the plumbing. The gas system utilizes furnace technology, but the fully integrated system for a single furnace would still be over $200, and with two halves, the costs increase rapidly. The plumbing, which is a mix of copper and silicon tubing, also ran high. This is due to the high cost of copper and the high cost of the machined connectors (barbs) for the silicon tubes. This could be cut for the future with a different set of tubes, but there is a low possibility due to the food safety requirements.

2.2 Concept Generation

Brewing at home is growing in the US, with over 1 million home brewers, so there is a growing market for a product like this. However, many home brewers have either fully manual or partial manual control of their systems. While this does offer a more intimate setting, and a better chance at getting a really good beer, it can eliminate an entire day that some people can ill afford to lose. This brings us to a fully automated system. With the right situation, and a brewer who is experienced, it can be a fully advantageous situation. Imagine waking up on Saturday morning, going outside, starting the brewing process, mowing the lawn, grilling out, and finishing up the brewing process. This sounds like a perfect situation for someone who is short on time. With that in mind, we set out to make the system as simple for the average use as possible.

The easier the system is to operate, the more likely the average American is to use it. We will need to ensure that the interface is simple and elegant in its usage, and that the interface is easy to use. To this, we can allow the brewer to input the times necessary for the proper addition of the materials, and then have an alarm when the brewer needs to take action. The only real challenges at that point are to hook up the gas and add the water to start the process.

The last concept we tried to bring to front is that there hasn't been a good heat exchanger designed for the home brewer. There are a couple different types of chillers on the market: the tube-in-tube, the plate type, and an immersion chiller. They all have their pros and cons, but we felt that each one would have more drawbacks than necessary. This is why we started brainstorming a new chiller. This new chiller would need to maximize the heat transfer surface area while minimizing the amount of space

used. The other thing we wanted to ensure was that it would be food safe. This means that we want a chiller that can be cleaned rather easily, and can come apart if necessary. By being able to take it apart, we can add even more surface area, increasing the amount of heat being transferred.

2.3 Project Evaluation

This project could be considered a breath of fresh air into a stale market. There is room for growth in the home brew market we feel we can capture. By designing a simple system that anyone can run, can be easily maintained, and makes a very delicious final product, we think that this would be a high potential market.

This project also looks to take advantage of many of the courses that are explored in mechanical engineering. Because of the breadth of the course work in mechanical engineering, we could be considered a jack of all trades. This allows us to work in manner different areas, including fluids, thermodynamics, programming, statics structures, electronics, chemistry, and manufacturing. Every one of these topics was covered in detail in this project. This gives us as students a chance to see not only where we excel, but also where we are deficient. This is a good thing, as it gives us a chance to work on things as we progress in our lives and careers.

Chapter 3 Cost/Market Analysis and Patentability

3.1 Cost Analysis

We set out to make an automated home brewery for less than $2,000. This amount was chosen by Nick due to monetary restrictions that he had, as he would be the main support for this project. We also chose this number because it is a low enough number that a customer would be willing to purchase it, but not be willing to go home and try to copy it. This number was an out of pocket number, as we understood that any labor we incurred would be free until we have graduated and no longer have the universities facilities available to our disposal. The cost of production will be discussed later. For now we will get into the cost of the system. The total cost of the system is under our $2,000 goal. The total is shown in Table 1. The shipping cost will not be discussed, as it can change based on sourcing of the materials and time for shipment.

Table 1 Total Costs

System	Cost ($)
Frame	124.87
Plumbing	614.40
Burner	596.75
Brewing	69.85
Shipping	90.03
Electrical	285.99
H. E.	180.00
Cost	$1,961.89

3.1.1 Frame

The frame is the backbone of the structure. It involves square steel tubing, arranged in a manner to support all everything off the ground, except for the gas tank.

The frame material was mainly supplied by DIMCO in DeKalb. The costs will be show in Table 2 below. The casters and drip trays were supplied from the spare stock the university has on hand.

Table 2 Frame Costs

Frame	Qty.	Cost	Supplier
1¼X1¼ 16 Gage Steel	60'	$124.87	DIMCO
Casters	4		CEET
11 Gage Steel	24"X12"		CEET

3.1.2 Plumbing

The plumbing made up of the piping that carries the water/wort, the valves that change the flow patterns and the pumps that move the water. This was all purchased with no spare stock being used from the university. A portion of the parts were locally sourced from a Menards, while the rest comes from specialty stores online. Brewers Hardware and Chugger specialize in helping home brewers with specially designed parts that meet the needs of the average brewer. The costs associated are shown in Table 3 below.

Table 3 Piping Costs

Plumbing	Qty	Cost ($)	Supplier
1/2" X10' Copper Pipe	2	18.18	Menards
1/2" Dielectric Union	6	23.94	Menards
1/2" Close Brass Nipple	5	9.45	Menards
1/2" Red Brass Union FXF	1	10.49	Menards
1/2" X1-1/2 Brass Nipple	1	2.29	Menards
1/2" Copper Tee	2	1.18	Menards
1/2" Threaded Barb	3	23.85	Brewers Hardware
1/2" SS 90° Elbow	3	12.00	Brewers Hardware
1/2" 90° Elbow	20	5.80	Menards
Paste Flux	1	1.69	Menards
Safe Flow Solder	1	8.69	Menards
1/2" Copper Brush	1	0.99	Menards
1/2" FPT Full Coupler	3	11.85	Brewers Hardware
1/2" SS Tube	2	16.00	Brewers Hardware
3-Way SS T Valve	1	51.90	KLD
3-Way SS L Valve	2	99.00	KLD
2-Way SS Valve	1	37.10	KLD
Chugger Pump	2	280.00	Chugger

3.1.3 Electrical

The electrical system was purchased by both Humza and Nick. The biggest reason for this was Humza doing the electrical design; it was easier for him to specify what he needed when he needed it. Many of the parts could be purchased at the local Radio Shack, making it easy to find them in most areas. The sensors that are included came from Brewers Hardware again, due to the specific nature of the product. All purchase costs are shown in

Table 4.

Table 4 Electrical Costs

Electrical	Qty	Cost ($)	Supplier
Temp Sensor	3	51.00	Brewers Hardware
Arduino Mega	1	59.99	Radio Shack
Control box	1	35.00	Menards
Power Supply	1	25.00	MicroCenter
Transformer	1	15.00	Menards
Wire	1	25.00	Open Source Controls Systems
Ground	1	5.00	Menards
Circuit board	1	5.00	Radio Shack
Etchant	1	5.00	Radio Shack
Pressure Sensor	4	60.00	Open Source Controls Systems

3.1.4 Ignition System

The ignition system was almost exclusively purchased online. This was from the research that was all done online. There is a possibility that the right area would have access to all but the specialty parts. This is due to the use of commonly found parts for heating systems in the home and regulation for campers. Costs are shown in Table 5

Table 5 Ignition Costs

Burner	Qty	Cost ($)	Supplier
Valve Orfice	2	14.00	Brewers Hardware
Two Stage Regulator	1	33.99	Amazon
Pilot BCR-18	2	41.90	PexSupply
Pilot Controller	2	174.98	PexSupply
Furnace Valve	2	167.90	PexSupply
Banjo Burner	2	163.98	Northern Brewer

3.1.5 Brewing System

The brewing system is small, as it is just the kettles and the thermowells for the thermocouples in the kettles. All were found online, but with the right contacts, the kettles could become a local find. The costs are in Table 6.

Table 6 Brewing Costs

Brewing	Qty	Cost ($)	Supplier
Thermowell	3	29.85	Brewers Hardware
1/2 Barrel Keg	3	40.00	Craigslist

3.1.6 Heat Exchanger

The heat exchanger, being of a unique design, receives its own section. The 6061 Aluminum was sourced online, as there was a healthy competition in the price of the metal. It can be found locally for a slightly higher price.. The cost is listed below in Table 7.

Table 7 Heat Exchanger Cost

Heat Exchanger	Qty	Cost ($)	Supplier
13"OD X 0.75" 6061 Alum	2	90.00	

3.2 Marketing

The marketing of the system will be multi-faceted project. The key elements that we will look at are product, price, promotion, place, and people. We aim to have a low cost system that works better than anything on the market. It will be seen in trade magazines and in social media, while also being promoted by local friends of the company. We will aim at those who want to brew at home, with a secondary market of those who are looking at a simpler system that they can modify to their liking.

The product we have is high quality. We have chosen materials that will withstand the test of time. The frame is made of steel and will be sealed from the elements by either paint or powder coat. Either way the arms that support the kettles will need a high temp paint to manage the temps from the burners when the flame is dialed in to its operating condition of over 1,000°F. The product we build is smaller in size than our competitors, meaning that less space will be taken up in storage, and there is less

material being used in the construction of the product. This will also allow for more systems being built from a similar amount of material compared to the rivals. As we have more time and money into the system, we can refine the materials used until we have reached an optimum cost vs. quality point. We also plan to have options to change the system to a brewer's choice. This can include a gas tank stand, different color schemes, different piping paths, less automation, and possibly the ability to use materials other than gas to provide the energy to make the water warm.

As for the price point, we feel that we can be 67% of the cost of the competitor, who is at $7,000. This would put us at $4,700 price. This is beyond what we expect the system to cost, and if it is beyond what the market allows, we can be flexible. We will also target the audience that doesn't want full automation. For them, we can offer a frame, a manual gas system, and the piping. There is also the possibility to offer a three burner system with an immersion chiller. This would eliminate the cost of the heat exchanger, but add in burner cost, so essentially a non-change in cost. The cost of $4,700 would allow for rapid recovery of our initial investment into the company, with hopes of being out of debt in two years at most, and six months at the least. Beyond that, and we will need to be a full time business, where we expect to start as a part time venture.

For the promotion of the system, we will use social media to start. Facebook, LinkedIn, and Twitter will allow us to spread out message the fastest. Beyond these, we will also target local brew shops and microbreweries to share our message. If we can get the word out there, where people gather, that would be a huge boon to our future. There is also the possibility of advertising in industry magazines. This would get our

word out to a larger demographic, and could be picked up by any John Q. Public to see a well engineering product that would allow him to stop going to pick up beer at the local store.

When we talk about where we would want to do things, at first it would be out of a garage. This would allow us to build on our own time, and then move the product locally until we were in the black. Once this point hits, we can work into a larger location, and possibly take the business full-time. This would be a huge undertaking, and is still years down the road. It would also require a large amount of capital, along with the support of our families, friends, and local beer aficionados. Once we work up to a regional presence, we would start researching national ties. This could possibly involve selling off to a company that sees this as a great idea, becoming a subsidiary of another company, or turning into a national powerhouse in the home brew industry. If we reach that point, everything would be beyond our wildest dreams.

All of the ideas we have would require the assistance of many people around us. Throughout this process, we all talked about knowing someone that is very interested in this idea. If we can get a foot in the door with our friends, and maybe one or two other small pubs, we can get a small movement going. This could also be supported by local winemakers and other artisans in any field. As we grow, we cannot forget the people that made us a success.

3.3 Patentability

We have a unique product in a unique market. We have already made inquiries with people as to hoe the patent process would go. While parts of the system already

have patents, and other parts are commonly used designs, there are two parts that are not currently available to the public: the heat exchanger and the circuitry.

The heat exchanger will be described in much more detail later. It utilizes square tubing to achieve four wall heat transfer, and Aluminum to allow for a good amount of heat transfer. We chose a material that we are well versed in, in the 6061 Aluminum, but we feel that there are better conductors of heat out there, and the will be part of the focus of our future pans. The heat exchanger also uses both parallel and cross flow elements, allowing for an even greater amount of heat being transferred; this intern allows for a more rapid descent from near boiling wort to the 70°F necessary for the proper addition of the yeast. This rapid descent helps lessen the possibility of bacteria entering the brew, making it bad.

The circuitry is patentable due to the unique design. The circuit is designed to fit onto a single circuit board, allowing multiple power groups to work in unison without disrupting each other. There is also the idea that we have the design. The use of the components in a manner that hasn't been done before makes it different than all other competitors out there. We also have the programming to think about. It isn't that the coding is so much different than what is available, it's that it has never been put together in the manner we have. The code will allow for a number of different inputs, allowing the brewer to brew any number of different recipes. These different recipes will be on the brewer to input, so there will be no claim to that property.

Chapter 4 Frame

The entire system needs to support not only all the components but also the maximum volume of water that can be used. The design of the frame to support these large demands of weight will need to also be in a well-organized fashion to allow ease of use for the brewer. Initially, two main designs were considered: horizontal (Figure 2) and stack (Figure 1)

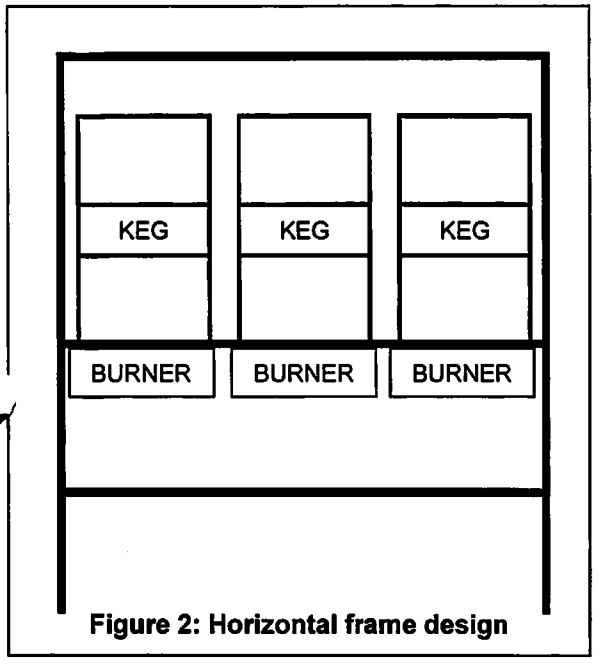


Figure 2: Horizontal frame design

Since liquid must be transferred from kettle to kettle, the stack design would only require one pump. This decrease in plumbing costs is an advantage of the stack design, but the total height of the design proves to be dangerous as the liquid in the top kettle will reach 170°F. If the top kettle were to tip over, severely dangerous burns could potentially occur to the brewer.

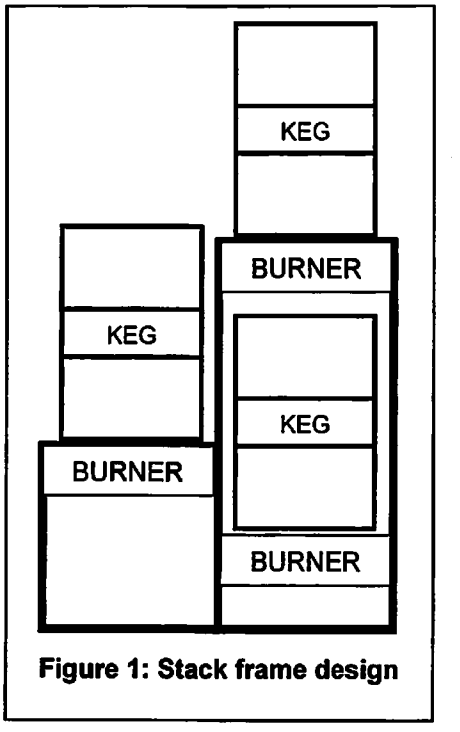


Figure 1: Stack frame design

While the horizontal design has a higher cost and larger footprint then the stack, the increased safety makes this design far superior. After Nick narrowed down to the two designs and presented them to the group, a decision of the horizontal design was agreed upon. Now that the frame design was chosen, the actual design process was to begin. To utilize software offered

to students and is commonly found in industry, Solid Works was chosen. This software is a user friendly program that was preferred by all the members of the group and allows the simplistic use of simulation later used in the design process. Measurements of total kettle height were measured by placing the kettles on boxes of various heights and emulating stirring of the liquids inside was performed until a comfortable height was found for a 6'3" user. Setting a constraint of this total height in Solid Works made the design ergonomically comfortable for the brewer as stirring is a common function performed.

Another consideration in the total size of the system was the location it was going to be stored and commonly used; the garage. The device was to be similar size of storage units as not to be difficult to store in the limited confines of a common garage. The frame's total footprint was set at 26" deep and 64" wide which is a similar size of a workbench. Now that the main size restrictions were defined with constraints the supporting structure and geometry of the frame were to be defined. To be able to support both the kettles and burners, located below the kettles, 4 diagonal horizontal braces were positioned perpendicular to the kettles. These braces were then welded to the heat shield that would support the burners. These bracing members were located evenly across the frame to accommodate the three kettles necessary for the brewing process.

The lower shelf will support the plumbing system in addition to the electrical system that would be mounted below the plumbing. To reduce the risk of electrocution in the event of spillage, a horizontal splash guard was designed on the lower shelf. This shelf would need to be supported properly in addition to mounting the pumps. Two

supports were located such that the pump heads were symmetrically between two kettles. The splash guard was to be attached to these members.

Table 8 Weight per bracing arm

Barley Wine @ 1.100			
Item	Lbs/gal	Gallons	Total (Lbs)
Wort	8.35	10	83.5
Grains			40
Keg			28.6
Burner			13.8
Burner Ring			3.5
		Total (Lbs)	169.4
		FOS (3)	508.2
		Lbs Per Arm	127.05

The frame needed to be made to sufficiently hold the weight of full volume kettles safely with a factor of safety of 3. To calculate the amount of weight each kettle would amount to, the following weights were summed; 10 gallons of wort (1.100SG), keg, grains/barley, burner, and heat shield (Table 8).

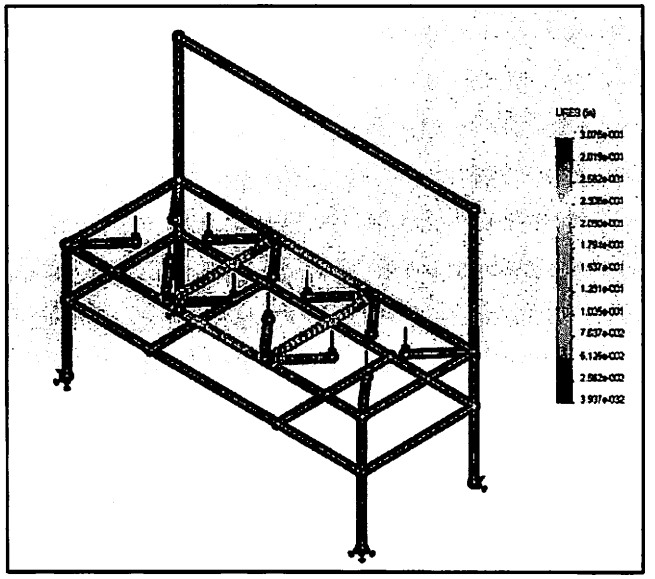


Figure 3: Deflection Plot of 16 gage 1.5" square tube

Using Solid Works force simulation software and 127.05 lbs. per arm were used to analyze the deflection of the frame that was to be made of 1.5"x1.5"x0.60" mild steel square tube (Figure 3). This material was chosen primarily because of the cost to weight ratio. We compared 12, 14, and 16 gage material and 16 gage proved to be the best.

Chapter 5 Plumbing System

During the brewing process a series of liquid movements must be achieved in order to properly make the fermentable wort. This process includes initially filling kettle 1, transferring liquid to kettle 2, recirculating kettle 2, transferring from kettle 2 to kettle 3 while transferring kettle 1 to kettle 2, recirculating kettle 3 while recirculating kettle 1 to chill the liquid, finally, emptying kettle 3 into the fermenting bucket. This process can be very difficult not only to understand but to design a system that allows all these requirements to be achieved.

The use of a multi-purpose heat exchanger allowed the brewer to chill in addition to eliminating the need for a 3rd burner located under the second kettle. This not only reduces the total cost but also prevents the risk of scorching the wort that is due to applying localized heat on the bottom of the second kettle. Using the heat exchanger while recirculating kettle 2 and kettle 1, which has higher water inside, will increase the temperature in kettle 2 to allow proper starch to fermentable sugar conversion without scorching. This heat exchanger will also be used to chill the wort in one of the final processes of brewing by recirculating kettle 3 and kettle 1, which will then have ice water inside.

Another consideration in reducing the total cost of the system is the reduction of valves and pumps need in the plumbing system. The Chugger Pumps will move the liquid at a maximum flow rate of 7.0 gpm and are operated on common 110VAC current. The stainless steel motorized ball valves were found overseas to further decrease cost. One of the primary objectives for the project was to use food grade safe products in which both the valves and pumps achieve. Revisions of the plumbing system allowed

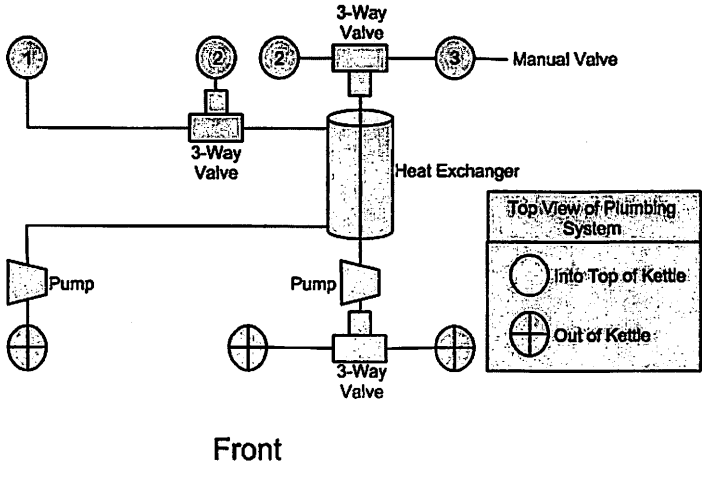


Figure 4: Plumbing diagram

the reduction of the valves and pumps to (3) 3-way and (1) 2-way valves and 2 pumps to be used Figure 4

As the plumbing system was installed, consideration of disassembly and maintenance was thoroughly thought of. The user may need to make improvements, fixes, or a deep cleaning maintenance. Unions, quick disconnects, and removable pump heads were used to insure that ability of full disassembly. Full diagrams of the flow patterns can be found in the appendix.

Chapter 6 Gas System

To properly make a batch of home brew, water must be heated up to a near boil at least twice. This requires a substantive energy addition system. This could be a simple fire from wood, an electric heater, gasoline or another flammable liquid, or it could be a flammable gas. We chose the flammable gas route, as it is widely used and commercially available. This choice then populated all of our other choices within the gas system. We chose a system that is simple, elegant, and easily produced.

6.1 Components

6.1.1 Gas Choice

In a common home in the northern Illinois area, the main heat source is natural gas. In recent years it has become highly available and relatively cheap. There is already a supply network in place that allows residents nearby to hook into the system. This is only a dream for more remote families, who rely on liquid propane (LP) for their home heating. It is delivered in large trucks to stationary tanks in the yard, and piped into the house for heating purposes only due to higher costs. LP is also available in portable, refillable tanks that are available in many locations for either gas grills or mobile homes. There is also a small population of people who use kerosene for heating. This was deemed a moot choice, as there is very little available on the market for kerosene.

The choice fell to LP due to the existing use by Nick in previous home brewing experience. It has a higher energy profile than natural gas, and it fits the mobility value

we have espoused from the beginning. Also by choosing LP, we could find many retail locations that carried parts for our use, if necessary.

6.1.2 Burner

The burner we chose was the Bayou Classic BG 14 Burner. Nick had already purchased one for his own home brew sessions, so after we decided on a two burner setup, he purchased the second. They are cast iron, making them extremely heavy.

The design has a single ¼" NPT inlet with an air adjustment port to help control the O_2 levels for the flame. It has four mount holes that approximately 90 degrees apart. This allowed us to build mounts into the arms of the assembly, along with the shields necessary to prevent wayward fires. The burner surface comes with two hole patterns. The first is two circles that circle the outside of the burner. The second is two lines of holes that go inward along the six arms, stopping just short of the center. The center has no holes, as this is where the gas and air are allowed to mix before being ejected through the holed pattern on the top. Figure 5 [1] shows the setup, and we have modified it so the air inlet is much larger.

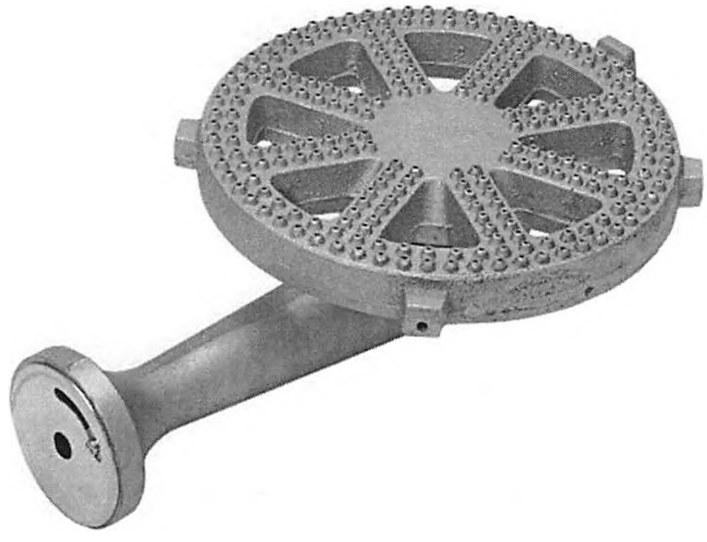


Figure 5 Bayou Classic Banjo Burner BG14

6.1.3 Pilot

The pilot is an essential part of creating an automatic gas system. If the pilot goes out, and gas is stopped, the burn can stop, creating a chance for wasted batch. By finding a good pilot that could be paired with an automatic valve, the system could be made near fool-proof.

The pilot chosen for this system was the Honeywell Q345A1313. This particular model came with a gas inlet and a combination thermocouple/igniter. This choice was made due to cost, control, and size. The pilot is small enough to attach to the burner without creating an extra hole to allow escaping gas, and is low enough in cost that future mass production would not be overly expensive. The control of this pilot will be discussed in more detail shortly; however, we can note that it allows for intermittent operations. This means that if we are trying to maintain a two to four degree temperature band on a kettle, we can shut the gas off and turn it on as needed with the pilot lighting up every time we need gas. The pilot is shown in Figure 6 [5].

Figure 6 Pilot Burner for Low Pressure Gas

6.1.4 Automatic Gas Valve

With the desires to have a fully automated system, we looked at a number of different references for a good, simple solution. We needed something that would be fully automatic, work well with our pilot, fail shut in an emergency, and require almost no interface with the person controlling the system. This is how we stumbled up the Honeywell Standard Dual Intermittent Pilot Gas Valve.

Figure 7 Standard Dual Intermittent Pilot Gas Valve

This valve has the distinction of allowing gas to flow through to the pilot separate from the burner. It is lightweight, compact, and it will also act in a fully automatic state. There are two drawbacks, though. The desire to go fully automatic made the cost go up, well above the expected range. This is a trade-off for the ability to go fully automatic. The other problem is the operating pressure of less than one-half psi of gas. This was an unforeseen problem that caused us more headaches and purchases. There is a bonus; it allows us to switch back and forth between LP and natural gas in a few short minutes. There is a regulating spring installed in the valve. Depending on the gas type that is encountered, a different spring is put in place. When this is done, the brewer has the choice of gas types. The valve is shown in Figure 7 Standard Dual Intermittent Pilot Gas Valve [6].

6.1.5 Ignition System Control

The pilots and the automatic valves must be controlled in order to make the system work. This could be done by the Arduino (detailed later), but this would create an inordinate amount of work, especially since there is already a control module for these components.

Because the pilot and valve are both made by Honeywell, it stands to reason that Honeywell also make a control module. This control module works in unison with the other components to allow the burner to operate. The controller takes an input from the Arduino (acting as a thermostat). It then sends out a signal to the valve to open the pilot

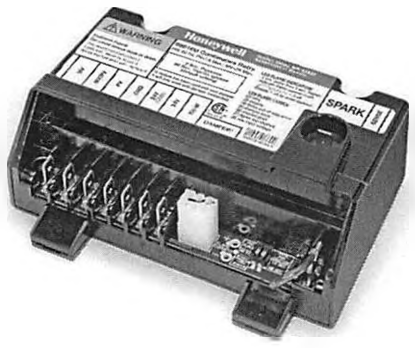


Figure 8 Intermittent Pilot Control

line. This allows gas to the pilot, which then gets an ignition signal from the controller, telling it to ignite. When the thermocouple senses the right temperature, it sends a signal to the controller. This signal is read and then starts another command. This command goes to the automatic valve, allowing it to open. When the control module receives a signal saying that the liquid is warm enough, it will relay the signal to the valve, which shuts both the pilot and main valves. The control module is shown in Figure 8 [4].

6.1.6 Valve and Orifice

Because the burner is not designed to be operated at low pressures, there is a need to ensure that the burner receives enough fuel to ensure proper operation. For this, there is a hand operated control valve with an orifice. The orifice uses the pressure of the gas, and restricts its movement. By creating a small backpressure, the orifice causes the gas exiting to come out at a very high velocity. This helps to ensure that the burner is getting fed well enough for normal operation. The valve portion, along with the air restrictor on the burner, creates a very good control for the size and temperature of the flame at the burner. By having a good, tight flame, the amount of gas used and wasted can be minimized, helping the brewer enjoy a few more beers for the same price. Figure 9 [3] shows the setup, with gas flowing from left to right.

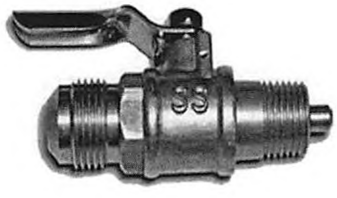


Figure 9 LP Orifice for High Pressure Burner

6.1.7 Regulator

Figure 10 Two Stage Pressure Regulator

It was mentioned in the automatic valve section that the valve had a pressure rating one one-half psi. This is a challenge when the pressure out of an LP tank is ~30 psi. To fix this, a two-stage regulator was found. By using a two-stage regulator, the gas flow out can be more linear with changes in the inlet conditions. It also has an installed vent, so if an overpressure condition occurs, it does not act as a bomb. The regulator was initially designed for motorhome use. Due to our use of the same energy source, we felt it would be a welcome addition to keep us safe. It is shown in Figure 10 [2].

6.1.8 Piping and Wiring

There is a minor component that normally gets overlooked, the connections between components. While the electrical system will be discussed in detail later, and the plumbing was already discussed, the components specific to this system will be described now.

The hard lines for the gas system will be made out of black iron, as it is the standard of the industry. A local hardware store was kind enough to assist with the threading, since it was slow and tedious by hand. After this was done, and all connections were purchased, the system was put together. Gas system tape was used to help prevent leaks. A problem arose when taking out the burners to install the pilots: the burners had a height adjustment the gas system didn't. A trip to the local hardware store for flexible gas connections was made, and the problem was solved. The soft lines

were next. They connected the automatic valve to the pilot. This was rather simple work, as the copper lines were soft and flexible, and the connections were compression fittings that came with the parts.

The electrical was a more challenging part. After receiving the control modules and the pilots, we realized there was no wiring attached. With the spade connections, this wasn't so bad, as the female ends can be found at many hardware stores. The connection for the spark line to the pilot was not found in stores. At this time, we would like to graciously thank DeKalb Mechanical for the donation of the connection wires. They are valued at $25/piece, so this was never included in the total cost.

6.2 Operation

The gas system is a very linear mechanism. It takes multiple inputs and turns them into one output. This final output allows a brewer to boil water, helping to ensure that they have a good, bacteria free batch.

The first thing to happen is that the LP tank gets hooked up to the regulator. This provides the fuel for the system. After the tank valve is opened, and the regulator dialed in, a signal is needed from the Arduino. This signal comes from the kettle that needs hot water. This will only be one kettle at a time, so we do not try to overwhelm the system.

The signal that is received gets interpreted and modified into a call for energy. This call for energy is sent to the control module. It looks at this signal and realizes it needs gas flowing and a flame. The automatic valve receives a signal to flow gas to the pilot, and the pilot gets a spark signal. Once the thermocouple on the pilot starts to get a reading, the ignition is stopped to allow for a clean flame. This clean flame, once hot enough, will start the flame on the burner. It takes the thermocouple getting hot, and

then telling the control module all is good. Upon receipt of this message, a different signal is sent to the automatic valve to open the main line to the burner. This carries the gas to the burner, where it mixes with the air pulled in from the outside, and the flame from the pilot to produce the energy necessary to boil water.

When the kettle's thermocouple realizes the water is hot enough, the Arduino takes this signal and tells the control module to shut off. When this signal is received, the module calls the automatic valve to shut everything down, effectively killing the burner flame along with the pilot flame. Due to the intermittent nature of the system, this can be repeated on a regular basis to keep a kettle at a near constant temperature.

Chapter 7 Electrical

7.1 Introduction

In order to automate the brewing process, numerous electrical and mechanical components had to be combined. One major requirement in completing this objective was to not be tethered to a computer in order to brew a batch. This led to three major component types: processing, automation inputs, and automation outputs. For this reason, the electrical system became rather complex and was one of the last items to actually complete in the project.

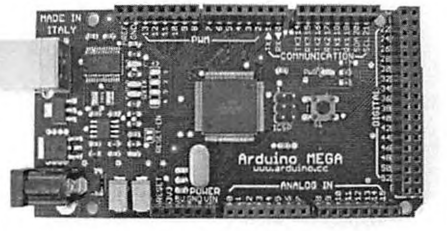


Figure 11: Arduino Mega

7.2 Processing

For the automation system to be successful there had to exist some method to accept sensor inputs and operate electromechanical outputs. An Arduino Mega was chosen to act as the brains for this application (Figure 11). The Arduino operates using open source, C++ programming language. This particular unit was selected due to the number of digital input/output (I/O) pins and analog pins available. This allowed for a large number of possible sensor and electronic control configurations.

This processor operates from 5-volt DC power. This allows the Arduino to be powered by many different power supplies including cell phone chargers or a computer USB port. This created one of the base power supply requirements for the automated process.

7.3 Automation Inputs

The original design called for three temperature sensors in order to monitor the individual kettle temperatures. There were two major kinds of sensor to select from: digital or analog. The advantage of the digital sensor is its ability to share a single digital pin for multiple OneWire-type devices. The disadvantage is the complexity of programming required to obtain data from the sensors. The advantage of the analog sensor was the simplicity of programming required to obtain a temperature read. The disadvantage is that a single pin must be designated for each individual sensor. Overall, the digital sensor was selected in order to reduce the number of I/O pins required (Figure 12).

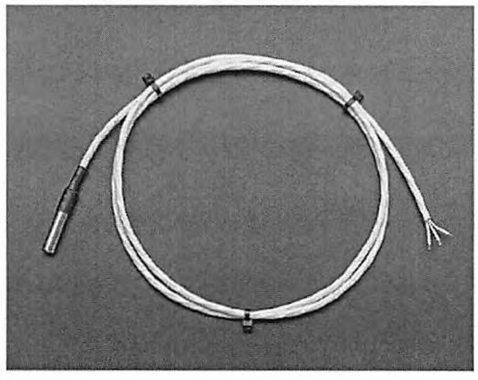


Figure 12: DS18B20 Digital Temperature Sensor

Multiple methods for determining fluid levels within the kettles were considered. These ranged from submersible pressure sensors, flow meters, float switches, and differential pressure sensors. Each one had unique challenges in their usage. Aside from simply determining the level of liquid in each kettle, it was desired to calculate the specific gravity of the wort as it was brewing. This eliminated both the flow meters and float switches as the only sensor used. The largest drawback to stainless steel, submersible pressure sensors were the price tag for each unit. These could be ordered for approximately $110 due to its food grade safe construction method. This became rather cost prohibitive due to needing four sensors. An idea was considered of only

using two of these sensors in order to calculate the specific gravity in kettle two while using another sensor to measure the liquid level in the other containers. A second type of pressure sensor, the MPX5010DP (Figure 13), was found that was not of a submersible type. This was connected using a port welted to the sidewall of the kettles and connected using flexible tubing. This sampled the pressure measured from the port with respect to the ambient air pressure allowing a liquid level to be determined. When two of these ports were mounted at a specific height, the difference between pressures measured would allow for the specific gravity to be determined. This particular sensor transmits data by changing output voltage which is connected and measured by an analog pin on the Arduino.

Figure 13: MPX5010DP Differential Pressure Sensor

A common requirement for both the temperature and pressure sensors was the operating voltage. Both of these sensors operate at 5-volts DC, allowing them to share the same power supply as the Arduino.

7.4 Automation Outputs

In order to facilitate fluid movement and temperature regulation, numerous liquid valves and pumps were required. The actual system called for three three-way valves, one two-way valve, and two liquid transfer pumps. One major requirement for all of these components is the construction material. The material used must be compliant for use in a food grade application.

The first item to be selected was the liquid transfer pump. The pumps are required in order to transfer liquid between the individual kettles. The particular pump was the Chugger Stainless Steel Inlet Pump (Figure 14). This pump operates at 120-volt AC power and is designed for the actual application it will be utilized. This pump features a removable pump head which allows for ease of maintenance and repair.

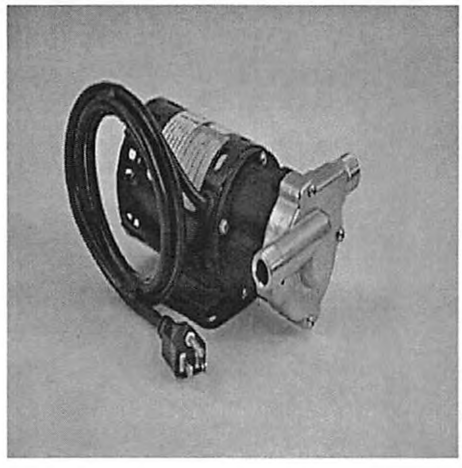


Figure 14: Chugger Stainless Steel Inline Pump

The next item selected was the liquid transfer valves. These are required to change the flow of liquid between the individual kettles. The valves selected were of the KLD20S series in both two- and three-way configuration (Figure 15). These are quarter-turn valves that are controlled by a three-wire circuit to operate a 12-volt DC motorized drive. The construction material was also food safe and readily available. Many valves were considered, but out of the other options considered, this series of valves were both economical and reasonably priced.

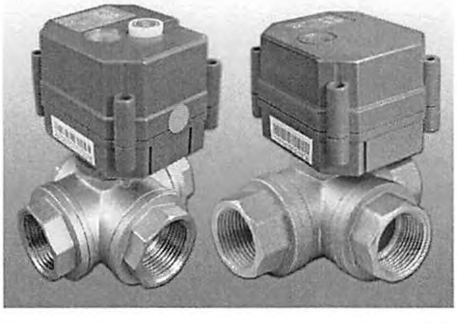


Figure 15: KLD20S Series Liquid Valve

7.5 Circuitry

In order to control the various components mentioned above, a custom control circuit had to be designed. The voltage requirements to operate the automation system were 5-volt DC, 12-volt DC, 24-volt AC, and 120-volt AC. This presented for quite the challenge since the Arduino sends and receives data in a range between 0-5-volts DC. Appendix 3 shows the general wiring schematic designed for this function.

A power distribution system was used to feed 120-volt AC power from a GFI protected outlet to the various components. A 250-watt ATX computer power supply was disassembled and used due to its stable 5-volt and 12-volt DC power output. This power supply has the capacity to run all four liquid control valves, three temperature sensors, four differential pressure sensors, the Arduino, and the actual control circuit. A 24-volt AC doorbell transformer was also fed from the power distribution system in order to provide proper voltage to the ignition and gas control system. The power distribution system was also used to feed the liquid transfer pumps 120-volt AC power.

In order to distribute power and control the numerous components, two circuit boards were proposed. These provided mounting location for the semi-conductor components and a means to wire all of the automation components together. The first variation utilized a pre-drilled project board that all of the semi-conductor components were soldered to (Figure 16). An attempt to solder jumper wires on the backside of the board proved to be difficult and ineffective.

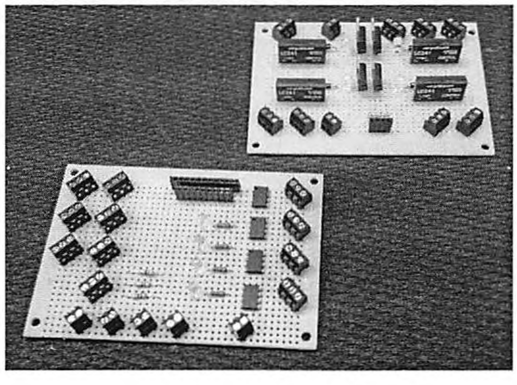
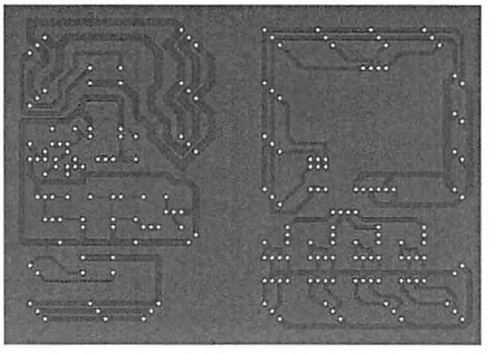
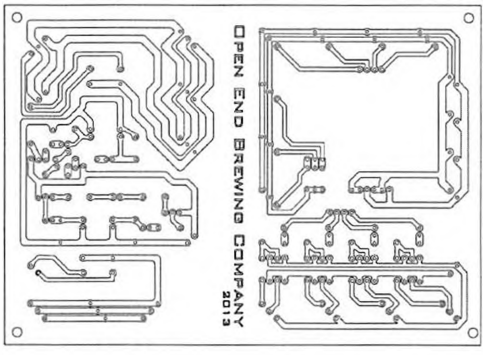


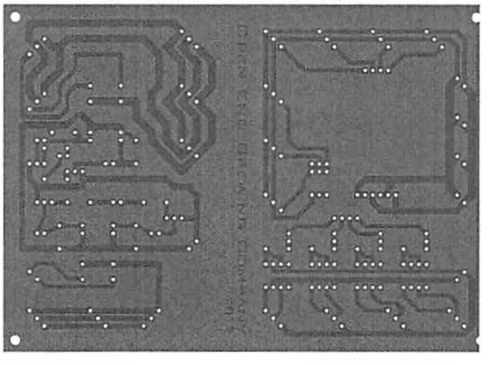
Figure 16: Circuit Board Variation 1



Revision 1



Revision 2



Revision 3

Figure 17: 3-D Circuit Board Traces

The next variation came to be when a 3-dimensional model was created to lay out and plan traces to be placed on a PCB. A total of three revisions to the 3-D model were done to optimize the circuit design (Figure 17). The next part of this circuit design is to manufacture the physical board. There were three main options considered in order to manufacture the board. The first was to outsource the board to an online company. This became both cost- and time-prohibitive. The second option was to use a CNC mill in order to physically remove excess copper and isolate the traces on the copper clad PCB board.

This proved to be unrealistic due to the hardware requirements of the CNC mill itself. Due to the minute size of the bit used to remove the copper, the minimum revolutions per minute were higher than the capability of NIU's equipment. The last option was to chemically etch the copper clad boards to remove excess copper. This was a long, grueling process to adequately protect the copper traces from the acid etchant. After three attempts of transferring toner from printer paper to the copper and etching the board, we still did not have an adequate circuit board. After changing to an actual PCB

etchant solution, and using a different toner transfer medium, we successfully produced a PCB board (Figure 18). Once this was complete, the mounting holes were drilled and traces cleaned up. The semiconductors were desoldered from the original project board and moved to the new board and soldered in place. Numerous circuit tests were conducted to verify both conductivity and no existence of short circuits (Figure 19).

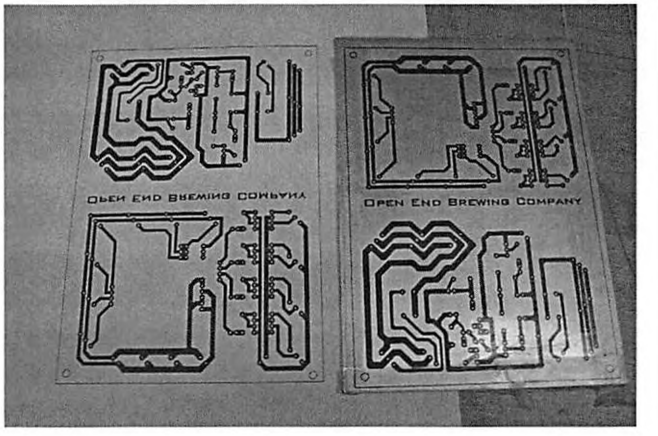


Figure 18: PCB Board Following Etching

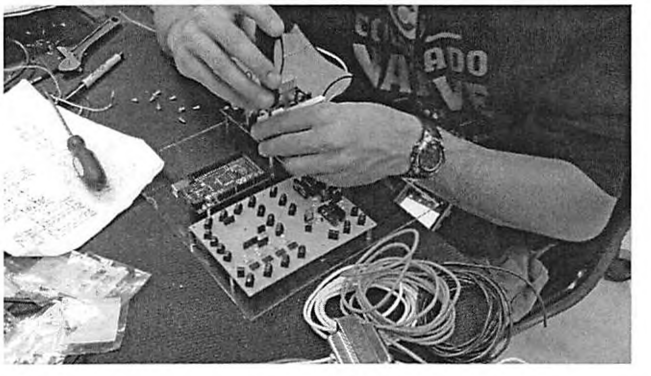


Figure 19: Wiring Electrical Circuits

The final step in creating the control circuitry was to enclose the electronics. A weather tight box was found and customized in order to mount all of the electronic controls. As shown in Figure 19, the components were mounted to plexiglass and then secured inside of the customized enclosure. Figure 20 shows the customizing of the enclosure in order to support and secure the circuit boards. This provided an effective way to prevent water damage and shed any accumulated water.

Figure 20: Customizing the Enclosure

The actual components selected for use have been uploaded to http://goo.gl/b0ns4 to conserve paper and allow for multimedia collaboration.

Chapter 8 Programming and Controls

8.1 Introduction

In order to control all of the equipment using an Arduino, a large amount of programming was required. The principles learned in CSCI240 and MEE321 allowed for a significantly more complex programming style. By layering sections of coding, the overall process logic was greatly simplified. This also allowed for a more robust system control.

8.2 Functional Units

The automated brewing system required essentially four core functions. These functions controlled the individual components selected earlier on. In order to obtain readings from the digital temperature sensors, the OneWire library was loaded and sample coding was modified in order to fit into the overall programming hierarchy.

The Arduino was programmed to measure voltage using the analog input pins where the pressure sensors were connected. A calibrating equation had to be created in order to convert the voltage measured to the height of water. Equation 1 relates the measure pressure to the depth of water from the measurement point. Both ρ and g are density and gravitational constants, respectively.

$$P = \rho g h$$

Equation 1: Pressure

This allowed the volume of liquid to be measured by inserting the height of liquid into the volume equation shown in Equation 2.

$$V = \pi r^2 h$$

Equation 2: Volume

In this equation, height of the liquid is measured using the pressure sensors to equate the current volume as desired in the initial programming.

The specific gravity was calculated using the same pressure sensors and Equation 3.

$$SG = \frac{\left(\frac{Pressure_2 - Pressure_1}{\Delta Height * Gravity}\right)}{1.89}$$

Equation 3: Specific Gravity

This allowed for a measure of pressure difference in order to determine the specific gravity as it relates to tap water.

The next functional group was the control of digital I/O pins in order to send either 0- or 5-volts DC to operate relays or transistors on the control board. This allowed for control over the pumps, valves, and ignition system.

The last main functional group was to obtain a time stamp. The Arduino's time function resets itself after approximately seventy-minutes, making it unusable without significant coding. By using an external clock to maintain track of time allowed for processes to run longer than seventy-minutes without losing track of time.

8.3 Logical Units

In order to simplify individual process coding, logical units were created to perform simple tasks. These tasks included temperature comparisons, liquid level

comparisons, and time duration comparisons between goal and actual values. By doing so, repeat functions could be called using a broad language function.

The next logical units were used to combine valve and pump states in order to achieve goals. An example of these is transfer commands between two kettles or recirculation commands within one kettle.

8.4 Process Sequence

The original intent for this system was to allow for multiple processes to easily occur. These processes included brewing cycles, sanitation cycles, and demonstration cycles. A usable function was created to allow for easy use of one of these three processes.

Using the logical units, it greatly simplified the calling of functions so as to reduce the probable programming errors. Appendix 4 includes the general logic for the brewing process. The actual Arduino code is included in Appendix 5 for reference.

Chapter 9 Heat Exchanger

9.1 Design

The home brew market has a number of chillers available. They are mainly just that: chillers. They are not very useful for trying to keep the energy in the system, such as a recirculation system that only uses two burners. We chose to address as a complex problem.

Each of the chillers had a good idea. The tube in tube style has a large surface area for a compact operational area. The plate type is also compact. The immersion style (drop in) chiller takes direct advantage of the chilled water. We felt that a large surface area with a compact body and good use of the cooling medium would all be necessary. We were also concerned with the amount material being used, as this would mean more money spent.

To this end, we would like to introduce the helical, single pass, parallel and counter flow heat exchanger. This design uses 0.5"x0.5" inch channels that spiral in with each other. As they reach the center, they turn vertically ninety degrees, and then turn parallel to the initial path, only forty-five degrees off of the initial line. This change in direction gets the lines to cross. The flow back to the edge is again parallel flow. The counter flow portion is the transfer between layers. With the liquid moving in on the bottom, and out on the top, it creates a different amount of heat transfer than just following the same lines. Figure 21 shows the flow paths.

The material was a topic of debate also. There are a few good materials for heat transfer: copper, aluminum, and silicon were all debated. In the end we, chose

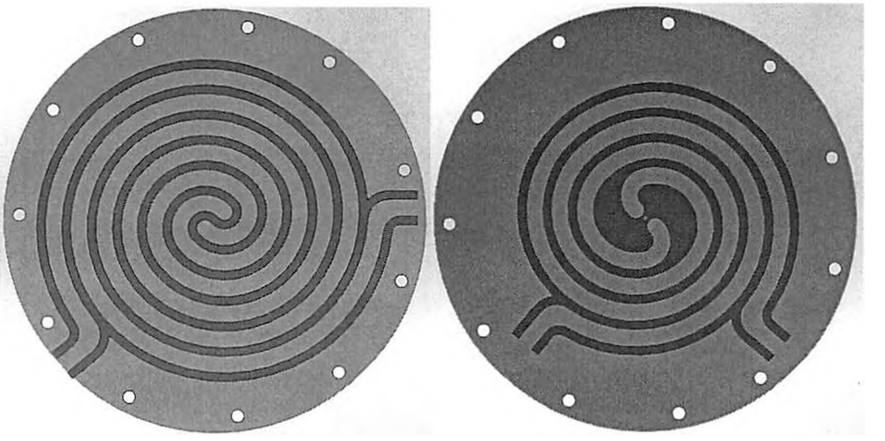


Figure 21 Bottom (left) and Top of Heat Exchanger

aluminum due to its machinability. With copper, we would have needed to figure out how to bend the tubes the exact same, while allowing for a few bends in the middle. This may be looked at in the future, but was beyond the scope for the moment. The aluminum used was 6061, chosen mainly due to familiarity. After discussions with Ken Sparkes, we found there are better choices out there, and they will be further researched in the future.

The initial calculations for the heat exchanger are shown in Appendix . They will show that with a quarter inch wall thickness and approximately forty square inches of surface area, there should be a drop of approximately eighty degrees after a short period of time. This estimate would have the wort cooled down to seventy degrees in only a couple minutes. We feel this is a high estimate, as we feel would need much more surface are to achieve the results expected.

We also looked at a second method for analysis of the temperature gradients. This method used nodal analysis in Excel to figure out what the temperatures would be. We would now like to extend our gratitude to Ken Sparkes for his assistance in teaching

this method to Josh. However, it never came to fruition. The biggest problem was time. It takes a serious amount of time to populate a number of Excel worksheets with a number of equations. This was further complicated by the path of the fluid and the extra set of equations that needed to be derived for the fluids. When time ran out, there was a rather good proof of concept that is now at the link shared in Chapter 7 Electrical, in the 482hx.xls file.

9.2 Testing

After Nick was able to machine the heat exchanger out of aluminum, and we were able to install it into the system properly, Humza used his MEE 390 final experiment to determine the amount of heat being moved.

For this experiment, k-type thermocouples were installed near the inlet and outlet ports of the heat exchanger. Once the system was fully assembled, the functional loops were setup for cooling off kettle three by recirculation of kettle one. Kettle three held near boiling water, and kettle one held near frozen water. The testing was done at a few different flow speeds to see what kind of affect the flow through the system had on the temperature change.

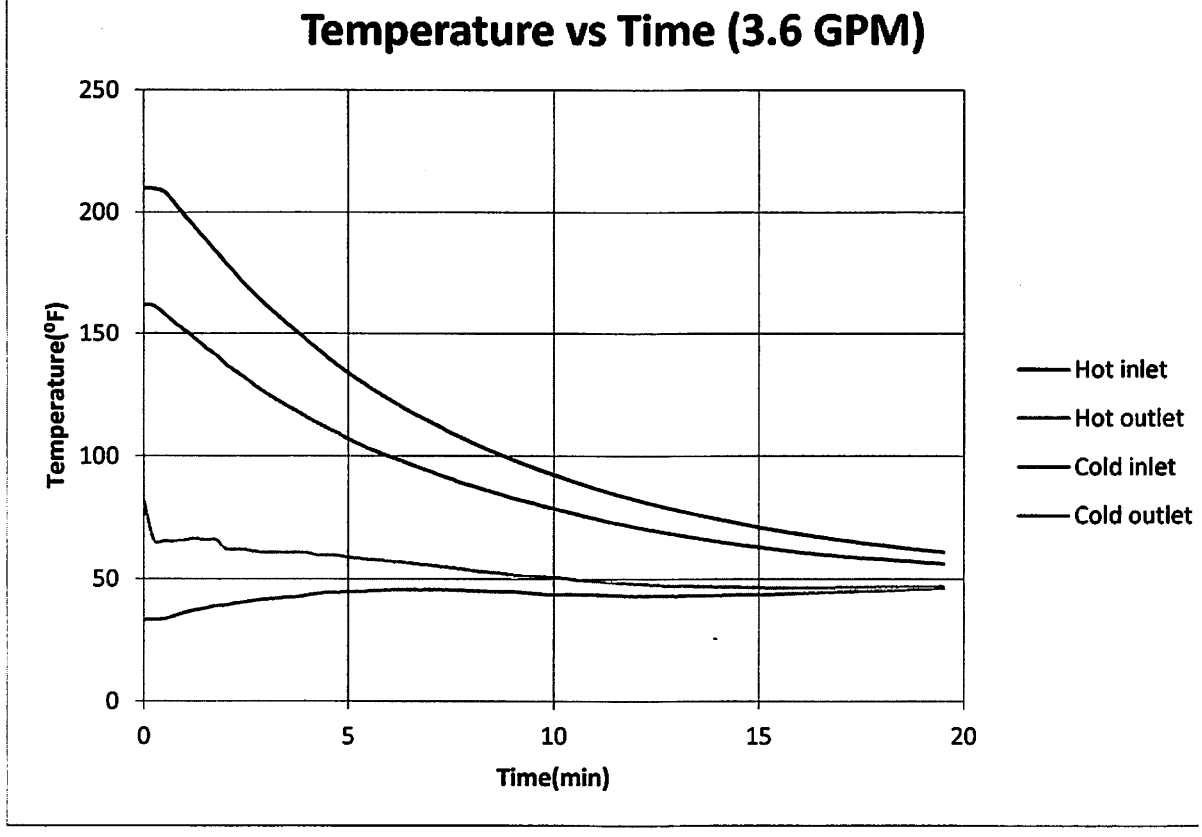


Figure 22 Test Data for the Chiller at 3.6 GPM

The data showed that at a higher speed, more energy as transferred from the hot to the cold side. At a flow rate of 3.6 gallons per minute, the initial temperature drop across the heat exchanger was over 45°F. It took the system 12.5 minutes to reach the final goal of 70°F. This is approximately 40% faster than the immersion chiller that Nick already has. This is a substantial improvement, and with more work could be improved greatly. The data is also in the shared folder mentioned in Chapter 7 Electrical.

Chapter 10 Discussion and Conclusions

This project came together as any engineering endeavor does; with a solution to a problem. Our project had many solutions to each problem that was encountered, yet as a team we came together to decide on the best solution. Initially, it all began with the issue of not inconsistency and the laborious task in brewing beer. Many solutions were discussed to each other, while some ideas were better than others; the main lesson leaded is the ability to respectfully find the best solution.

Trust was another great lesson learned in this project, as each person had their responsibilities, trust in each other to successfully complete one's task was key in the project's success. Each person has their own process of completing a task and trust that they will finish is a very important lesson learned.

Aside from what we learn from this project, some future plans involve using the system to create the best beer known to man!

References

[1] amazon.com. *Amazon.com Bayou Classis BG14 Banjo Burner 10 in. x 10 in.* . 6 May 2013. 6 May 2013. <http://www.amazon.com/Bayou-Classic-BG14-Banjo-Burner/dp/B0009JXYTG>.

—. *RV Motorhome LP Gas Tank Propane Portable Side-Vent two stage Bulk Kit: Amazon*. 6 May 2013. 6 May 2013. <http://www.amazon.com/Motorhome-Propane-Portable-Side-Vent-two-stage/dp/B003VBCZVC/ref=pd_sim_sbs_auto_6>.

Brewers Hardware, LLC. *LPG Valve - Orifice from Brewers Hardware*. 6 may 2013. 6 May 2013. <http://www.brewershardware.com/Valve-and-LPG-Orifice-for-BURN10.html>.

PexSupply.com. *Honeywell S8610U3009 Intermittent Pilot Control*. 6 May 2013. 6 May 2013. <http://www.pexsupply.com/Honeywell-S8610U3009-Intermittent-Pilot-Control-4584000-p>.

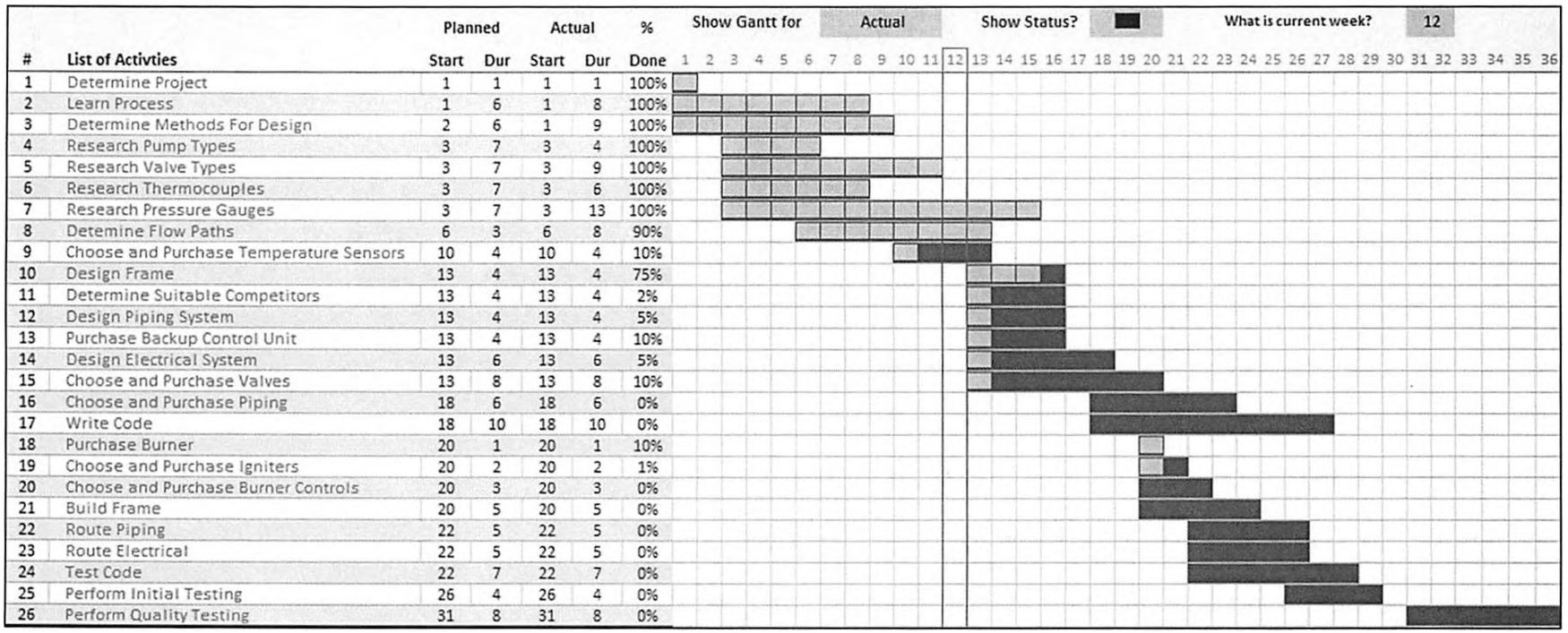
—. *Q345A1313 - Honeywell Q345A1313 - Pilot Burner for natural gas with a BCR-18 orifice*. 6 May 2013. 6 may 2013. <http://www.pexsupply.com/Honeywell-Q345A1313-Pilot-Burner-for-natural-gas-with-a-BCR-18-orifice>.

—. *VR8204A2076 - Honeywell VR8204A2076 - Standard Dual Intermittent Pilot Gas Valve*. 6 May 2013. 6 May 2013. <http://www.pexsupply.com/Honeywell-VR8204A2076-Standard-Dual-Intermittent-Pilot-Gas-Valve-13663000-p>.

Appendix

Appendix 1

Table 9 Gantt Chart

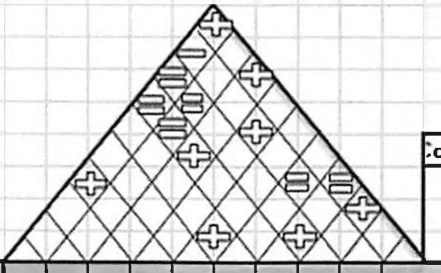


#	List of Activties	Planned Start	Planned Dur	Actual Start	Actual Dur	% Done
1	Determine Project	1	1	1	1	100%
2	Learn Process	1	6	1	8	100%
3	Determine Methods For Design	2	6	1	9	100%
4	Research Pump Types	3	7	3	4	100%
5	Research Valve Types	3	7	3	9	100%
6	Research Thermocouples	3	7	3	6	100%
7	Research Pressure Gauges	3	7	3	13	100%
8	Detemine Flow Paths	6	3	6	8	90%
9	Choose and Purchase Temperature Sensors	10	4	10	4	10%
10	Design Frame	13	4	13	4	75%
11	Determine Suitable Competitors	13	4	13	4	2%
12	Design Piping System	13	4	13	4	5%
13	Purchase Backup Control Unit	13	4	13	4	10%
14	Design Electrical System	13	6	13	6	5%
15	Choose and Purchase Valves	13	8	13	8	10%
16	Choose and Purchase Piping	18	6	18	6	0%
17	Write Code	18	10	18	10	0%
18	Purchase Burner	20	1	20	1	10%
19	Choose and Purchase Igniters	20	2	20	2	1%
20	Choose and Purchase Burner Controls	20	3	20	3	0%
21	Build Frame	20	5	20	5	0%
22	Route Piping	22	5	22	5	0%
23	Route Electrical	22	5	22	5	0%
24	Test Code	22	7	22	7	0%
25	Perform Initial Testing	26	4	26	4	0%
26	Perform Quality Testing	31	8	31	8	0%

Appendix 2

Table 10 House of Quality

PROJECT: Automated Brewing System

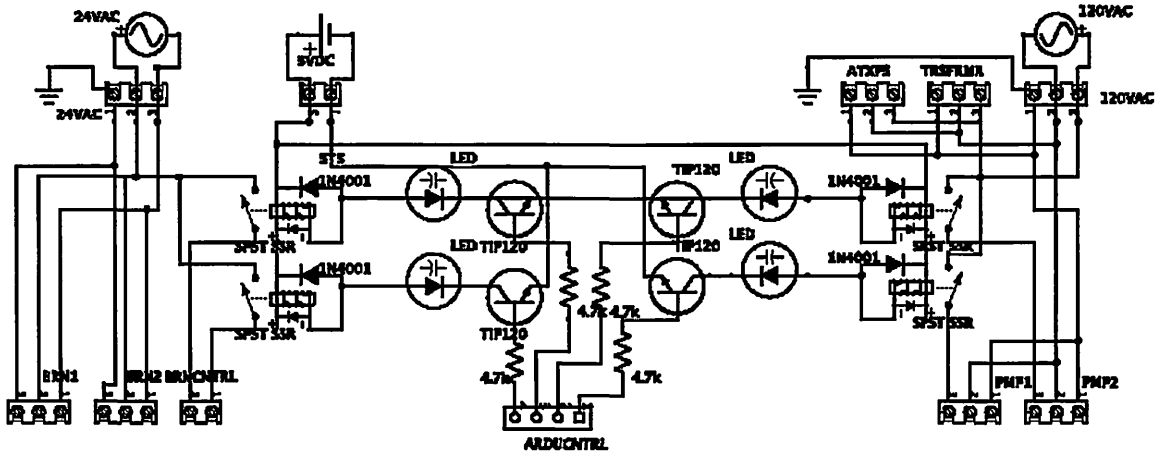


Correlation/Relationshi	
Positive	+
Average	0
Negative	-

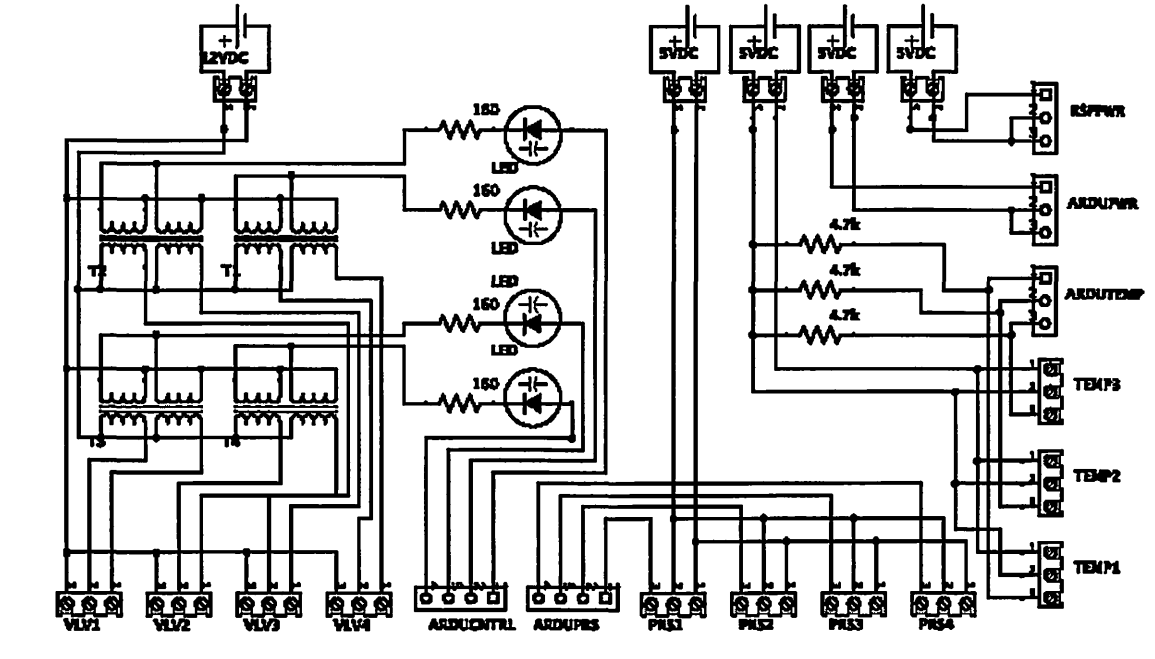
Direction of Improvement												Customer's Feedback					
Product-/Service-Requirements "HOW" / Customer Requirements "WHAT"		Importance (1-5)	pumps	valves	thermocouples	steel frame	food grade tubing	minimize tubing	simple automation logic	easy to use interface	spill-proof	display necessary items(temp, time)	Own Organization	Sabco	Competitor 2	Competitor 3	Competitor 4
	simplicity	3	-	-	-	+	0	+	+	+	-	+	5	4			
	repeatablitiy	3	+	+	+		+	0	+	+		+	4	4			
	low cost	3	-	0	0	0	-	+	+	0			5	2			
	portability	2				0							4	3			
	ease of use	5	+	0	0	0			+	+		+	4	3			
	safety	5	0	0			+	+			+		5	3			
	automated	4	+	+	+				+	+		+	5	3			
	quality	3			+	+	+	+	0		+		4	5			
	volume	1	0			+							5	3			
	asthetics	2	-	-	-	0		+					2	5			
	serviceability	5	-	-	-			+					4	2			
	food grade safe	5	+	+			+						4	4			
ENCHMARK	Own Organization		4	5	4	4	5	4	3	3	4	2	Evaluation				
	Sabco		2	2	3	4	3	4	-	5	4	5	Competition/Importan				
	Competitor 2												1=very weak / not important				
	Competitor 3												2=weak / little important				
	Competitor 4												3=average				
Goal			40	37	28	26	35	48	39	33	16	18	4=strong / important				
Importance (1-5)			3	3	3	5	5	4	5	5	4	4	5=very strong / very important				

Appendix 3

Electrical Wiring Diagram



AC Circuit Design



DC Circuit Design

Appendix 4

Brewing Logic

- Pumps, valves, burners OFF
- //Strike Water Tank1(Step 1)
    - If Temp1<=StrikeTemp //Heat Until Strike Temp
        - Burner1 ON
    - Else //Transfer to Tank2
        - Burner1 OFF
        - Valve1 open to Tank2
        - Pump1 ON
        - Until VolumeTank2 = StrikeVolume //Transfer until Strike Volume
            - Valve1 close
            - Pump1 Off
- //Mash & Recirculation(Step 2)
    - Loop until MashTime/Specific Gravity
        - If Temp2 < MashTemp //Recirculate while Burner2 On
            - Burner2 on
            - Valve2 open to Tank2 //Recirculate
            - Pump2 On
        - Else
            - Burner2 off
            - Pump2 off
- //Transfer & Rinse (AKA Sparge)(Step 3)
    - Until Tank3<= BoilVol
        - //Transfer Tank2 to Tank3
            - Valve2 open to Tank3
            - Pump2 on
        - //Transfer Tank1 to Tank2 to rinse
            - Valve1 open to Tank2
            - Pump1 on
- //Boil (Step 4)
    - Burner3 Intermitted to maintain BoilTemp
        - Wait until Hop1(min)
            - Alarm Hop1
        - Wait until Hop2(min)
            - Alarm Hop2
        - Wait until Hop3(min)
            - Alarm Hop3
        - Wait until Hop3(min) ends
- //Chill (Step5)
    - Until VolTank3 = 0
        - Burner3 Off
        - //Chiller Recirculation
            - Valve 1 open to chiller
            - Pump1 on
        - Valve 3 open to fermenter

Appendix 5

Arduino Code

```
*****Arduino_Sketch*****
#include "Arduino.h"
#include "ControlMe.h"
#include <string>
#include <iostream>
using namespace std;


//Time in minutes
//Temperature in degrees
F


double FillTemp = 40;

double BrewTime = 0;
double BrewTempChange
= 0;
double BrewTemp = 0;
double BrewTemp2 = 0;
double goalSG = 0;

double RinseDelay = 0;

double PasturizeTemp =
210;
double PasturizeTime = 0;

double Hops1 = 0;
double Hops2 = 0;
double Hops3 = 0;
double IceDump = 0;
double CoolTemp = 0;

ControlMe Brew;
ControlMe Setup;

void setup()
{
  Serial.begin(9600);
  Setup.Initialize();
  int x = Demo();
}

void loop()
{
  Brew.Shutdown();
}

int BrewCycle()
{
  ControlMe Brew;
  double TotalStart,
BrewStart, RinseStart,
PasturizeStart;
      TotalStart =
Brew.GetTime(); //Stamps initial
process start time
      Brew.Fill(1);       //Fills
Keg 1


      do
      {
            if (Brew.CheckLevel(1)
== TRUE)
              Brew.RecircKeg(1,1);
//Recirculates Keg 1 once filled

            if
(Brew.CheckTemp(1,FillTemp) ==
FALSE)
              Brew.Burner(1,1);
//Turns Burner 1 on if under desired
temp
            else
              Brew.Burner(1,0);
//Turns Burner 1 off if @ desired
temp
          }
          //Exits loop if Keg 1 is full
and @ desired temp
          while
(Brew.CheckLevel(1) != TRUE &&
Brew.CheckTemp(1,FillTemp)
!=TRUE);

          Brew.Transfer(2);
//Transfers water to Keg 2

          do
          {
            if (Brew.CheckLevel(2)
== TRUE)
              Brew.RecircKeg(1,1);
//Recirculates Keg 1 once Keg 2 is
filled

            if
(Brew.CheckTemp(1,FillTemp) ==
FALSE)
              Brew.Burner(1,1);
//Turns Burner 1 on if under desired
temp
            else
              Brew.Burner(1,0);
//Turns Burner 1 off if @ desired
temp

            if
(Brew.CheckTemp(2,BrewTemp) ==
FALSE)
              Brew.RecircKeg(2,1);
//Recirculates Keg 2 if under desired
temp
            else
              Brew.RecircKeg(2,0);
//Stops recirculating Keg 2 if @
desired temp
          }
          //Exits loop if Keg 2 is full
and @ desired temp
          while
(Brew.CheckLevel(2) != TRUE &&
Brew.CheckTemp(2,BrewTemp)
!=TRUE);

          //***DUMP GRAINS

          BrewStart =
Brew.GetTime(); //Stamps brewing
start time

          do
          {
            if
(Brew.CheckTemp(2,BrewTemp) ==
FALSE)
            {
              Brew.RecircKeg(2,1);
//Recirculates Keg 2 if under desired
temp
              Brew.RecircKeg(1,1);
//Recirculates Keg 1 if Keg 1 is under
desired temp
            }
            else
              Brew.RecircKeg(1,0);
//Stops recirculating Keg 1 if @
desired temp

            if
(Brew.CheckTemp(1,FillTemp) ==
FALSE)
              Brew.Burner(1,1);
//Turns Burner 1 on if under desired
temp
            else
              Brew.Burner(1,0);
//Turns Burner 1 off if under desired
temp

            if
(Brew.CheckTime(BrewStart,BrewTe
mpChange) == TRUE)
              BrewTemp =
BrewTemp2; //Increases brew
temperature after desired time
          }
          //Exits loop if time cycle is
complete or specific gravity is
reached
          while
(Brew.CheckTime(BrewStart,BrewTi
me) != TRUE ||
Brew.CheckSG(goalSG) != TRUE);

          Brew.Burner(1,0);
//Turns Burner 1 off
          Brew.Transfer(3);
//Transfers brewed liquid to Keg 3
          RinseStart =
Brew.GetTime(); //Stamps rinse start
time

          do
          {
            //RINSE DELAY
          }
```

```
            //Exits loop after specified
RinseDelay
            while
(Brew.CheckTime(RinseStart,RinseD
elay) != TRUE);

            do
            {
               Brew.Transfer(2);
//Transfers water from Keg 1 to Keg
2
            }
            //Exits loop if Keg 3 is full
            while
(Brew.CheckLevel(3) != TRUE);

            Brew.RecircKeg(1,0);
//Turns Pump 1 off and closes valve
2

            do
            {
              Brew.Burner(2,1);
//Turns Burner 2 on
            }
            //Exits loop if Keg 3 is @
desired temp
            while
(Brew.CheckTemp(3,PasturizeTemp)
== FALSE);

            Brew.Burner(2,0);
//Turns Burner 2 off

            PasturizeStart =
Brew.GetTime(); //Stamps pasturize
start time

            do
            {
               if
(Brew.CheckTemp(3,PasturizeTemp)
== FALSE)
                  Brew.Burner(2,1);
//Turns Burner 3 on
               else
                  Brew.Burner(2,0);
//Turns Burner 3 off

               if (Brew.CheckLevel(1)
== FALSE)
                  Brew.Fill(1);
//Fills Keg 1 for cooling cycle

               if (Brew.BoilOver() ==
TRUE)
                  Brew.Burner(2,0);
//Turns Burner 2 off if boil over
occurs
                  // *** INSERT
PASTURIZING STEPS
            //    if
(Brew.CheckTime(PasturizeStart,Ice
Dump) == TRUE)
                  // ADD ICE
            //    if
(Brew.CheckTime(PasturizeStart,Ho
ps1) == TRUE)
                  // ADD HOPS
                  // Dump HOPS
            //    if
(Brew.CheckTime(PasturizeStart,Ho
ps2) == TRUE)
                  // ADD HOPS
                  // Dump HOPS
            //    if
(Brew.CheckTime(PasturizeStart,Ho
ps3) == TRUE)
                  // ADD HOPS
                  // Dump HOPS
               }
               //Exits loop after cycle
time is complete
               while
(Brew.CheckTime(PasturizeStart,Pas
turizeTime) != TRUE);

               do
               {
                  Brew.RecircKeg(1,1);
//Recirculate Keg 1 for cooling
                  Brew.RecircKeg(3,1);
//Recirculate Keg 3 for cooling
               }
               //Exits loop after goal
temperature is reached
               while
(Brew.CheckTemp(3,CoolTemp) !=
TRUE);

               Brew.Shutdown();
//Shuts down brewing cycle

               return 0;
            }

            int Demo()
            {
              Brew.Valve(1,1);
              Brew.Valve(2,1);

              delay(7500);

              Brew.Pump(2,1);

              delay(45000);

              Brew.Pump(2,0);
              Brew.Valve(1,0);
              Brew.Valve(2,1);

              delay(7500);

              Brew.Burner(1,1);

              delay(6000);

              Brew.Burner(1,0);

              Brew.Pump(1,1);

              delay(30000);

              Brew.Pump(1,0);
              Brew.Valve(3,1);
              Brew.Valve(4,1);
              Brew.Valve(2,0);

              delay(7500);

              Brew.Pump(1,1);
              Brew.Pump(2,1);

              delay(30000);

              Brew.Pump(1,0);
              Brew.Pump(2,0);
              Brew.Valve(4,0);
              Brew.Valve(2,1);

              delay(7500);

              Brew.Pump(2,1);

              delay(15000);

              Brew.Pump(1,1);

              delay(30000);

              Brew.Pump(1,0);
              Brew.Pump(2,0);
              Brew.Valve(3,0);
              Brew.Valve(2,0);

              delay(7500);

              Brew.Pump(2,1);
              Brew.Burner(2,1);

              delay(6000);

              Brew.Burner(2,0);

              delay(15000);

              Brew.Pump(1,1);

              delay(30000);

              Brew.Pump(2,0);
              Brew.Pump(1,0);

              return 0;
            }
```

```
*****ControlMe.h*****
//============================
============================
=====
// ControlMe.h
//
//
//
//
// Humza Shamsuddin, 2013.
//============================
============================
=====

#include "OneWire.h"
#include "Wire.h"

class ControlMe
{
public:
        ControlMe();
        ~ControlMe();

        void Burner(int Burner, int
IO);
        void Valve(int Valve, int
IO);
        void Pump(int Pump, int
IO);

        void RecircKeg(int Keg, int
IO);
        void Fill(int IO);
        void Transfer(int
TransferTo);

        void Initialize();
        void Shutdown();

    bool CheckLevel(int Keg);
    bool CheckTemp(int Keg,
double GoalTemp);
    bool CheckTime(double Time,
double CycleTime);
    bool CheckSG(double SG);
        bool BoilOver();

        double KegTemp(int Keg);
        double KegPress(int Keg,
int Position);
        double GetTime();
    double AnalogSensor(int Pin);
//      void ElapsedTime(double
Tnow, double Tstart);
//  void ElapsedCycle(double
Tnow, double Tstart);

private:
    //Declares variable for pin
addresses for each component on
the Arduino
        int BURNER1;
        int BURNER2;
        int VALVE1;
        int VALVE2;
        int VALVE3;
        int VALVE4;
        int PUMP1;
        int PUMP2;
    int TEMP1;
    int TEMP2;
    int TEMP3;
    int KEG1PRES1;
    int KEG2PRES1;
    int KEG2PRES2;
    int KEG3PRES1;

    char temp1[8];
    char temp2[8];
    char temp3[8];

    double FillVolume;
    double KegRadius;
};
```

```
*****ControlMe.cpp****
//====================
============================
============
// ControlMe.cpp
//
//
//
//
// Humza Shamsuddin,
2013.
//====================
============================
============

#include "ControlMe.h"

//-------------------------
-----------------------
// constructor
//-------------------------
-----------------------
ControlMe::ControlMe()
{
        //Declares pin
assignments for components
connected to the Arduino
        BURNER1 = 41;
        BURNER2 = 39;

        VALVE1 = 24;
        VALVE2 = 26;
        VALVE3 = 28;
        VALVE4 = 30;

        PUMP1 = 49;
        PUMP2 = 47;

    TEMP1 = 38;
    TEMP2 = 40;
    TEMP3 = 42;

    KEG1PRES1 = A4;
    KEG2PRES1 = A6;
    KEG2PRES2 = A7;
    KEG3PRES1 = A5;

    //Volume in gallons
    FillVolume = 0;
    //Radius in feet
    KegRadius = 1;

    //Sets up pin
assignments within Arduino
        pinMode(BURN
ER1,OUTPUT);
        pinMode(BURN
ER2,OUTPUT);
        pinMode(VALVE
1,OUTPUT);
        pinMode(VALVE
2,OUTPUT);
        pinMode(VALVE
3,OUTPUT);
        pinMode(VALVE
4,OUTPUT);
        pinMode(PUMP
1,OUTPUT);
        pinMode(PUMP
2,OUTPUT);
        pinMode(TEMP1,INPUT);

        pinMode(TEMP2,INPUT);

        pinMode(TEMP3,INPUT);

        pinMode(KEG1PRES1,INPUT);

        pinMode(KEG2PRES1,INPUT);

        pinMode(KEG2PRES2,INPUT);

        pinMode(KEG3PRES1,INPUT);

                //Turns
connected components off at startup
                Burner(1,0);
                Burner(2,0);
                Valve(1,0);
                Valve(2,0);
                Valve(3,0);
                Valve(4,0);
                Pump(1,0);
                Pump(2,0);
        }

        //-------------------------
-----------------------
        // destructor
        //-------------------------
-----------------------
        ControlMe::~ControlMe()
        {

        }

        //-------------------------
-----------------------
        // Initialize
        //    Accepts: Nothing
        //        Returns:
Nothing
        //        Purpose:
Initiates all output pins and ensures
state is off
        //-------------------------
-----------------------
        void ControlMe::Initialize()
        {
          Wire.begin();

          // clear /EOSC bit
          // Sometimes necessary
to ensure that the clock
          // keeps running on just
battery power. Once set,
          // it shouldn't need to be
reset but it's a good
          // idea to make sure.

Wire.beginTransmission(0x68); //
address DS3231
          Wire.write(0x0E); // select
register
          Wire.write(0b00011100);
// write register bitmap, bit 7 is
/EOSC
          Wire.endTransmission();
analogReference(EXTERNAL);

          Valve(1,1);
          Valve(2,1);
          Valve(3,1);
          Valve(4,1);

          delay(7500);

          Valve(1,0);
          Valve(2,0);
          Valve(3,0);
          Valve(4,0);

          delay(7500);
        }

        //-------------------------
-----------------------
        // Burner Control
        //    Accepts: int Burner as
the burner number
        //        int IO as the
burner state (On/Off)
        //        Returns:
Nothing
        //        Purpose:
Evaluates input number and state
and controls appropriate
        //        Arduino pin
state
        //-------------------------
-----------------------
        void ControlMe::Burner(int
Burner, int IO)
        {
                int pin;

                //Converts
burner number to proper Arduino pin
address
                if (Burner == 1)
                        pin =
BURNER1;
                else if (Burner
== 2)
                        pin =
BURNER2;

                //Controls
Arduino pin output (LOW = 0 volts,
HIGH = 5 volts)
                if (IO == 0)

        digitalWrite(pin, LOW);
                else if (IO == 1)

        digitalWrite(pin, HIGH);

                return;
        }

        //-------------------------
-----------------------
        // Valve Control
        //    Accepts: int Valve as
the valve number
        //        int IO as the
valve state (On/Off)
```

```
//          Returns:
Nothing
//          Purpose:
Evaluates input number and state
and controls appropriate
//          Arduino pin
state
//-------------------------------
-------------------------
        void ControlMe::Valve(int
Valve, int IO)
        {
                int pin;

                //Converts valve
number to proper Arduino pin
address
                if (Valve == 1)
                        pin =
VALVE1;
                else if (Valve ==
2)
                        pin =
VALVE2;
                else if (Valve ==
3)
                        pin =
VALVE3;
                else if (Valve ==
4)
                        pin =
VALVE4;

                //Controls
Arduino pin output (LOW = 0 volts,
HIGH = 5 volts)
                if (IO == 0)

        digitalWrite(pin, LOW);
                else if (IO == 1)

        digitalWrite(pin, HIGH);

                return;
        }

        //-------------------------------
-------------------------
        // Pump Control
        //    Accepts: int Pump as
the pump number
        //          int IO as the
pump state (On/Off)
        //          Returns:
Nothing
        //          Purpose:
Evaluates input number and state
and controls appropriate
        //          Arduino pin
state
        //-------------------------------
-------------------------
        void ControlMe::Pump(int
Pump, int IO)
        {
                int pin;

                //Converts pump
number to proper Arduino pin
address
                if (Pump == 1)
                        pin =
PUMP1;
                else if (Pump ==
2)
                        pin =
PUMP2;

                //Controls
Arduino pin output (LOW = 0 volts,
HIGH = 5 volts)
                if (IO == 0)

        digitalWrite(pin, LOW);
                else if (IO == 1)

        digitalWrite(pin, HIGH);

                return;
        }

        //-------------------------------
-------------------------
        // Recirculate
        //    Accepts: int Keg as
the keg number
        //          int IO as the
recirculate state (On/Off)
        //          Returns:
Nothing
        //          Purpose:
Evaluates input number and state
and calls appropriate
        //          functions
        //-------------------------------
-------------------------
        void
ControlMe::RecircKeg(int Keg, int
IO)
        {
                if (Keg == 1 &&
IO == 1)
                {

        Valve(2,0);

        Pump(1,1);
                }
                else if (Keg == 2
&& IO == 1)
                {

        Valve(3,1);

        Valve(4,1);

        Pump(2,1);
                }
                else if (Keg == 3
&& IO == 1)
                {

        Valve(3,0);

        Valve(4,0);

        Pump(2,1);
                }
                if (Keg == 1 &&
IO == 0)
                {
                Pump(1,0);
                Valve(2,0);
        }
                else if (Keg == 2
&& IO == 0)

        Pump(2,0);
                else if (Keg == 3
&& IO == 0)

        Pump(2,0);

                return;
        }

        //-------------------------------
-------------------------
        // Fill Keg 1
        //    Accepts: int IO as the
fill state (On/Off)
        //          Returns:
Nothing
        //          Purpose:
Evaluates input state and calls
appropriate functions
        //-------------------------------
-------------------------
        void ControlMe::Fill(int IO)
        {
                if (IO == 0)

        Valve(1,0);
                else if (IO == 1)

        Valve(1,1);

                return;
        }

        //-------------------------------
-------------------------
        // Transfer Fluid
        //    Accepts: int
TransferTo as the destination keg
        //          Returns:
Nothing
        //          Purpose:
Evaluates input destination and calls
appropriate functions
        //-------------------------------
-------------------------
        void
ControlMe::Transfer(int TransferTo)
        {
                if (TransferTo ==
1)
                        Fill(1);
                else if
(TransferTo == 2)
                {

        Valve(2,1);

        Pump(1,1);
                }
                else if
(TransferTo == 3)
```

```
                {
        Valve(3,1);

        Valve(4,0);

        Pump(2,1);
                }

                return;
        }

        //-------------------------
-------------------------
        // Shutdown Procedure
        //    Accepts: Nothing
        //          Returns:
Nothing
        //          Purpose:
Shutsdown all output pins and
ensures state is off
        //-------------------------
-------------------------
        void
ControlMe::Shutdown()
        {
                Burner(1,0);
                Burner(2,0);
                Valve(1,0);
                Valve(2,0);
                Valve(3,0);
                Valve(4,0);
                Pump(1,0);
                Pump(2,0);
        }

        //-------------------------
-------------------------
        // Get Time Stamp
        //    Accepts: Nothing
        //          Returns: double
Time
        //          Purpose: Gets
time stamp from clock and returns its
value
        //-------------------------
-------------------------
        double
ControlMe::GetTime()
        {
          // send request to receive
data starting at register 0

Wire.beginTransmission(0x68); //
0x68 is DS3231 device address
          Wire.write((byte)0); //
start at register 0
          Wire.endTransmission();
          Wire.requestFrom(0x68,
3); // request three bytes (seconds,
minutes, hours)
          double sec, mins, hrs;
          int seconds, minutes,
hours;

          while(Wire.available())
          {
            seconds = Wire.read();
// get seconds
          minutes = Wire.read(); //
get minutes
          hours = Wire.read();  //
get hours

          sec = (((seconds &
0b11110000)>>4)*10 + (seconds &
0b00001111)); // convert BCD to
decimal
          mins = (((minutes &
0b11110000)>>4)*10 + (minutes &
0b00001111)); // convert BCD to
decimal
          hrs = (((hours &
0b00100000)>>5)*20 + ((hours &
0b00010000)>>4)*10 + (hours &
0b00001111)); // convert BCD to
decimal (assume 24 hour mode)
          }

          double Time = (((hrs * 60)
+ mins) * 60) + sec;
          return Time;
        }

        //-------------------------
-------------------------
        // Get Keg Temperature
        //    Accepts: int Keg as
Keg number
        //          Returns: double
KegTemp
        //          Purpose: Gets
keg temp and returns its value
        //-------------------------
-------------------------
        double
ControlMe::KegTemp(int Keg)
        {
          //Temperature Retrieval
and Conversion Function
          byte type_s;
          byte data[12];
          byte addr[8];

          if (Keg == 1)
          {
            OneWire
Temp1(TEMP1);
            if (!Temp1.search(addr))
            {
              //no more sensors on
chain, reset search
              Temp1.reset_search();
              return -1000;
            }

            if
(OneWire::crc8(addr,7) != addr[7])
            {
              Serial.println("CRC is
not valid!");
              return -1000;
            }

            if (addr[0] != 0x10 &&
addr[0] != 0x28)
            {
              Serial.print("Device is
not recognized");
              return -1000;
            }

            Temp1.reset();
            Temp1.select(addr);
            Temp1.write(0x44,1);
//start conversion, with parasite
power on at the end

            delay(750);

            byte present =
Temp1.reset();
            Temp1.select(addr);
            Temp1.write(0xBE);
//Read Scratchpad

            for (int i = 0; i < 9; i++)
            {
              //We need 9 bytes
              data[i] = Temp1.read();
            }

            Temp1.reset_search();
          }
          if (Keg == 2)
          {
            OneWire
Temp2(TEMP2);
            if (!Temp2.search(addr))
            {
              //no more sensors on
chain, reset search
              Temp2.reset_search();
              return -1000;
            }

            if
(OneWire::crc8(addr,7) != addr[7])
            {
              Serial.println("CRC is
not valid!");
              return -1000;
            }

            if (addr[0] != 0x10 &&
addr[0] != 0x28)
            {
              Serial.print("Device is
not recognized");
              return -1000;
            }

            Temp2.reset();
            Temp2.select(addr);
            Temp2.write(0x44,1);
//start conversion, with parasite
power on at the end

            byte present =
Temp2.reset();
            Temp2.select(addr);
            Temp2.write(0xBE);
//Read Scratchpad

            for (int i = 0; i < 9; i++)
            {
              //We need 9 bytes
              data[i] = Temp2.read();
            }
```

```
            Temp2.reset_search();
          }
          if (Keg == 3)
          {
            OneWire
Temp3(TEMP3);
            if (!Temp3.search(addr))
            {
              //no more sensors on
chain, reset search
              Temp3.reset_search();
              return -1000;
            }

            if
(OneWire::crc8(addr,7) != addr[7])
            {
              Serial.println("CRC is
not valid!");
              return -1000;
            }

            if (addr[0] != 0x10 &&
addr[0] != 0x28)
            {
              Serial.print("Device is
not recognized");
              return -1000;
            }

            Temp3.reset();
            Temp3.select(addr);
            Temp3.write(0x44,1);
//start conversion, with parasite
power on at the end

            byte present =
Temp3.reset();
            Temp3.select(addr);
            Temp3.write(0xBE);
//Read Scratchpad

            for (int i = 0; i < 9; i++)
            {
              //We need 9 bytes
              data[i] = Temp3.read();
            }

            Temp3.reset_search();
          }

          unsigned int raw =
(data[1] << 8) | data[0];
          if (type_s)
          {
            raw = raw << 3; // 9 bit
resolution default
            if (data[7] == 0x10)
            {
              // count remain gives
full 12 bit resolution
              raw = (raw & 0xFFF0)
+ 12 - data[6];
            }
          }
          else
          {
            byte cfg = (data[4] &
0x60);
            if (cfg == 0x00) raw =
raw << 3;  // 9 bit resolution, 93.75
ms
            else if (cfg == 0x20) raw
= raw << 2; // 10 bit res, 187.5 ms
            else if (cfg == 0x40) raw
= raw << 1; // 11 bit res, 375 ms
            // default is 12 bit
resolution, 750 ms conversion time
            }
            byte MSB = data[1];
            byte LSB = data[0];

            float tempRead = ((MSB
<< 8) | LSB);         //using two's
compliment
            float TemperatureSum =
(float)tempRead / 16;
//Temperature in degrees C

            float KegTemp =
(TemperatureSum * 1.8) + 32.0;
//Temperature in degrees F
            return KegTemp;
          }

          //-----------------------------
-------------------------------
          // Check Level
          //    Accepts: int Keg as
which keg
          //            Returns: bool
          //            Purpose:
Checks the level of the keg and
determines if it's full
          //-----------------------------
-------------------------------
          bool
ControlMe::CheckLevel(int Keg)
          {
            //Determines goal height
for fill status in feet
            double GoalHeight =
FillVolume / (3.14159 * KegRadius *
KegRadius);
            double CurrentHeight;
            double
PressureConversionConstant = 0;
            double Voltage;
            double Pressure;

            if (Keg == 1)
            {
              Voltage =
AnalogSensor(KEG1PRES1);
              Pressure = Voltage *
PressureConversionConstant;
              CurrentHeight =
Pressure / (1.936 * 32.2);    //Based
on Spec. Weight @ 70-degrees F
            }
            else if (Keg == 2)
            {
              Voltage =
AnalogSensor(KEG2PRES1);
              Pressure = Voltage *
PressureConversionConstant;
              CurrentHeight =
Pressure / (1.89 * 32.2);    //Based
on Spec. Weight @ 160-degrees F
            }
            else if (Keg == 3)
            {
              Voltage =
AnalogSensor(KEG3PRES1);
              Pressure = Voltage *
PressureConversionConstant;
              CurrentHeight =
Pressure / (1.89 * 32.2);    //Based
on Spec. Weight @ 160-degrees F
            }

            if (CurrentHeight >=
GoalHeight)
              return 1;
            else
              return 0;
          }

          //-----------------------------
-------------------------------
          // Analog Sensor
          //    Accepts: int Pin as
which PIN assignment
          //            Returns: Voltage
          //            Purpose:
Checks the voltage of the sensor
and sends back the value
          //-----------------------------
-------------------------------
          double
ControlMe::AnalogSensor(int Pin)
          {

analogReference(EXTERNAL);
            double Voltage =
0.0048828125 * analogRead(Pin);
            return (Voltage);
          }

          //-----------------------------
-------------------------------
          // Check Temp
          //    Accepts: int Keg as
which keg
          //            double
GoalTemp as the goal temperature
          //            Returns: bool
          //            Purpose:
Checks the temp of the keg and
determines if it's @ temp
          //-----------------------------
-------------------------------
          bool
ControlMe::CheckTemp(int Keg,
double GoalTemp)
          {
            double CurrentTemp =
KegTemp(Keg);

            if (CurrentTemp >=
(GoalTemp - 0.0))
              return 1;
            else
              return 0;
          }

          //-----------------------------
-------------------------------
          // Check Time
```

```
        //    Accepts: double Time
as process start time
        //             double
CycleTime as cycle duration
        //             Returns: bool
        //             Purpose:
Checks the time duration and
determines if cycle is complete
        //-------------------------
-------------------------
        bool
ControlMe::CheckTime(double Time,
double CycleTime)
        {
          double CurrentTime =
GetTime();    //Gets timestamp in
seconds

          if ((Time + CycleTime) >=
CurrentTime)
            return 1;
          else
            return 0;
        }

        //-------------------------
-------------------------
        // Check Specific Gravity
        //    Accepts: double SG
as goal specific gravity
        //             Returns: bool
        //             Purpose:
Checks the specific gravity against
goal value
        //-------------------------
-------------------------
        bool
ControlMe::CheckSG(double SG)
        {
          //Determines goal height
for fill status in feet
          double GoalHeight =
FillVolume / (3.14159 * KegRadius *
KegRadius);
          double CurrentHeight;
          double
PressureConversionConstant = 0;
          double delHeight = 10 /
12;   //Sensor height difference in
inches

          double Voltage1 =
AnalogSensor(KEG2PRES1);
          double Pressure1 =
Voltage1 *
PressureConversionConstant;
          double Voltage2 =
AnalogSensor(KEG2PRES2);
          double Pressure2 =
Voltage2 *
PressureConversionConstant;

          double CurrentSG =
((Pressure2 - Pressure1) / (delHeight
* 32.2)) / 1.89;    //Based on Density
@ 160-degrees F

          if (CurrentSG >= SG)
            return 1;
          else
            return 0;
        }

        //-------------------------
-------------------------
        // Boil Over
        //    Accepts: Nothing
        //             Returns: bool
        //             Purpose:
Checks if boilover condition exists in
Keg 3
        //-------------------------
-------------------------
        bool ControlMe::BoilOver()
        {
          return 0;
        }
```

```
        *****Imported
Libraries*****
        #ifndef OneWire_h
        #define OneWire_h

        #include <inttypes.h>

        #if ARDUINO >= 100
        #include "Arduino.h"      //
for delayMicroseconds,
digitalPinToBitMask, etc
        #else
        #include "WProgram.h"
// for delayMicroseconds
        #include "pins_arduino.h"
// for digitalPinToBitMask, etc
        #endif

        // You can exclude certain
features from OneWire. In theory,
this
        // might save some space.
In practice, the compiler
automatically
        // removes unused code
(technically, the linker, using -fdata-
sections
        // and -ffunction-sections
when compiling, and Wl,--gc-
sections
        // when linking), so most of
these will not result in any code size
        // reduction. Well, unless
you try to use the missing features
        // and redesign your
program to not need them!
ONEWIRE_CRC8_TABLE
        // is the exception,
because it selects a fast but large
algorithm
        // or a small but slow
algorithm.

        // you can exclude
onewire_search by defining that to 0
        #ifndef
ONEWIRE_SEARCH
        #define
ONEWIRE_SEARCH 1
        #endif

        // You can exclude CRC
checks altogether by defining this to
0
        #ifndef ONEWIRE_CRC
        #define ONEWIRE_CRC 1
        #endif

        // Select the table-lookup
method of computing the 8-bit CRC
        // by setting this to 1. The
lookup table enlarges code size by
        // about 250 bytes. It does
NOT consume RAM (but did in very
        // old versions of
OneWire). If you disable this, a
slower
        // but very compact
algorithm is used.
        #ifndef
ONEWIRE_CRC8_TABLE
        #define
ONEWIRE_CRC8_TABLE 1
        #endif

        // You can allow 16-bit
CRC checks by defining this to 1
        // (Note that
ONEWIRE_CRC must also be 1.)
        #ifndef ONEWIRE_CRC16
        #define
ONEWIRE_CRC16 1
        #endif

        #define FALSE 0
        #define TRUE  1

        // Platform specific I/O
definitions

        #if defined(__AVR__)
        #define
PIN_TO_BASEREG(pin)
(portInputRegister(digitalPinToPort(pi
n)))
        #define
PIN_TO_BITMASK(pin)
(digitalPinToBitMask(pin))
        #define IO_REG_TYPE
uint8_t
        #define IO_REG_ASM
asm("r30")
        #define
DIRECT_READ(base, mask)
(((*(base)) & (mask)) ? 1 : 0)
        #define
DIRECT_MODE_INPUT(base,
mask)  ((*(base+1)) &= ~(mask))
        #define
DIRECT_MODE_OUTPUT(base,
mask)  ((*(base+1)) |= (mask))
        #define
DIRECT_WRITE_LOW(base, mask)
((*(base+2)) &= ~(mask))
        #define
DIRECT_WRITE_HIGH(base, mask)
((*(base+2)) |= (mask))

        #elif
defined(__PIC32MX__)
        #include <plib.h>  // is this
necessary?
        #define
PIN_TO_BASEREG(pin)
(portModeRegister(digitalPinToPort(p
in)))
        #define
PIN_TO_BITMASK(pin)
(digitalPinToBitMask(pin))
        #define IO_REG_TYPE
uint32_t
        #define IO_REG_ASM
        #define
DIRECT_READ(base, mask)
(((*(base+4)) & (mask)) ? 1 : 0)
//PORTX + 0x10
        #define
DIRECT_MODE_INPUT(base,
mask)  ((*(base+2)) = (mask))
//TRISXSET + 0x08
        #define
DIRECT_MODE_OUTPUT(base,
mask)  ((*(base+1)) = (mask))
//TRISXCLR + 0x04
        #define
DIRECT_WRITE_LOW(base, mask)
((*(base+8+1)) = (mask))
//LATXCLR  + 0x24
        #define
DIRECT_WRITE_HIGH(base, mask)
((*(base+8+2)) = (mask))
//LATXSET + 0x28

        #else
        #error "Please define I/O
register types here"
        #endif


        class OneWire
        {
          private:
            IO_REG_TYPE
bitmask;
            volatile IO_REG_TYPE
*baseReg;

        #if ONEWIRE_SEARCH
            // global search state
            unsigned char
ROM_NO[8];
            uint8_t
LastDiscrepancy;
            uint8_t
LastFamilyDiscrepancy;
            uint8_t LastDeviceFlag;
        #endif

          public:
            OneWire( uint8_t pin);

            // Perform a 1-Wire
reset cycle. Returns 1 if a device
responds
            // with a presence pulse.
Returns 0 if there is no device or the
            // bus is shorted or
otherwise held low for more than
250uS
            uint8_t reset(void);

            // Issue a 1-Wire rom
select command, you do the reset
first.
            void select( uint8_t
rom[8]);

            // Issue a 1-Wire rom
skip command, to address all on
bus.
            void skip(void);

            // Write a byte. If 'power'
is one then the wire is held high at
            // the end for
parasitically powered devices. You
are responsible
```

```
            // for eventually
depowering it by calling depower() or
doing
            // another read or write.
            void write(uint8_t v,
uint8_t power = 0);

            void write_bytes(const
uint8_t *buf, uint16_t count, bool
power = 0);

            // Read a byte.
            uint8_t read(void);

            void read_bytes(uint8_t
*buf, uint16_t count);

            // Write a bit. The bus is
always left powered at the end, see
            // note in write() about
that.
            void write_bit(uint8_t v);

            // Read a bit.
            uint8_t read_bit(void);

            // Stop forcing power
onto the bus. You only need to do
this if
            // you used the 'power'
flag to write() or used a write_bit()
call
            // and aren't about to do
another read or write. You would
rather
            // not leave this powered
if you don't have to, just in case
            // someone shorts your
bus.
            void depower(void);

        #if ONEWIRE_SEARCH
            // Clear the search state
so that if will start from the beginning
again.
            void reset_search();

            // Look for the next
device. Returns 1 if a new address
has been
            // returned. A zero might
mean that the bus is shorted, there
are
            // no devices, or you
have already retrieved all of them. It
            // might be a good idea
to check the CRC to make sure you
didn't
            // get garbage. The
order is deterministic. You will always
get
            // the same devices in
the same order.
            uint8_t search(uint8_t
*newAddr);
        #endif

        #if ONEWIRE_CRC
            // Compute a Dallas
Semiconductor 8 bit CRC, these are
used in the
            // ROM and scratchpad
registers.
            static uint8_t crc8(
uint8_t *addr, uint8_t len);

        #if ONEWIRE_CRC16
            // Compute the 1-Wire
CRC16 and compare it against the
received CRC.
            // Example usage
(reading a DS2408):
            //  // Put everything in a
buffer so we can compute the CRC
easily.
            //  uint8_t buf[13];
            //  buf[0] = 0xF0;  //
Read PIO Registers
            //  buf[1] = 0x88;  //
LSB address
            //  buf[2] = 0x00;  //
MSB address
            //  WriteBytes(net, buf,
3);  // Write 3 cmd bytes
            //  ReadBytes(net,
buf+3, 10);  // Read 6 data bytes, 2
0xFF, 2 CRC16
            //  if
(!CheckCRC16(buf, 11, &buf[11])) {
            //      // Handle error.
            //  }
            //
            // @param input - Array
of bytes to checksum.
            // @param len - How
many bytes to use.
            // @param inverted_crc
- The two CRC16 bytes in the
received data.
            //              This
should just point into the received
data,
            //              *not* at
a 16-bit integer.
            // @return True, iff the
CRC matches.
            static bool
check_crc16(uint8_t* input, uint16_t
len, uint8_t* inverted_crc);

            // Compute a Dallas
Semiconductor 16 bit CRC.  This is
required to check
            // the integrity of data
received from many 1-Wire devices.
Note that the
            // CRC computed here
is *not* what you'll get from the 1-
Wire network,
            // for two reasons:
            //  1) The CRC is
transmitted bitwise inverted.
            //  2) Depending on the
endian-ness of your processor, the
binary
            //      representation of
the two-byte return value may have a
different
            //      byte order than the
two bytes you get from 1-Wire.
            // @param input - Array
of bytes to checksum.
            // @param len - How
many bytes to use.
            // @return The CRC16,
as defined by Dallas Semiconductor.
            static uint16_t
crc16(uint8_t* input, uint16_t len);
        #endif
        #endif
        };

        #endif

        /*
        Copyright (c) 2007, Jim
Studt  (original old version - many
contributors since)

        The latest version of this
library may be found at:

http://www.pjrc.com/teensy/td_libs_O
neWire.html

        Version 2.1:
          Arduino 1.0 compatibility,
Paul Stoffregen
          Improve temperature
example, Paul Stoffregen
          DS250x_PROM
example, Guillermo Lovato
          PIC32 (chipKit)
compatibility, Jason Dangel,
dangel.jason AT gmail.com
          Improvements from
Glenn Trewitt:
          - crc16() now works
          - check_crc16() does all
of calculation/checking work.
          - Added read_bytes() and
write_bytes(), to reduce tedious
loops.
          - Added ds2408 example.
          Delete very old, out-of-
date readme file (info is here)

        Version 2.0: Modifications
by Paul Stoffregen, January 2010:
          http://www.pjrc.com/teensy
/td_libs_OneWire.html
          Search fix from Robin
James

http://www.arduino.cc/cgi-
bin/yabb2/YaBB.pl?num=123803229
5/27#27
          Use direct optimized I/O
in all cases
          Disable interrupts during
timing critical sections
            (this solves many
random communication errors)
          Disable interrupts during
read-modify-write I/O
          Reduce RAM
consumption by eliminating
unnecessary
```

```
        variables and trimming
many to 8 bits
        Optimize both crc8 - table
version moved to flash

        Modified to work with
larger numbers of devices - avoids
loop.
        Tested in Arduino 11 alpha
with 12 sensors.
        26 Sept 2008 -- Robin
James
        http://www.arduino.cc/cgi-
bin/yabb2/YaBB.pl?num=123803229
5/27#27

        Updated to work with
arduino-0008 and to include skip() as
of
        2007/07/06. --RJL20

        Modified to calculate the 8-
bit CRC directly, avoiding the need
for
        the 256-byte lookup table
to be loaded in RAM.  Tested in
arduino-0010
        -- Tom Pollard, Jan 23,
2008

        Jim Studt's original library
was modified by Josh Larios.

        Tom Pollard,
pollard@alum.mit.edu, contributed
around May 20, 2008

        Permission is hereby
granted, free of charge, to any
person obtaining
        a copy of this software and
associated documentation files (the
        "Software"), to deal in the
Software without restriction,
including
        without limitation the rights
to use, copy, modify, merge, publish,
        distribute, sublicense,
and/or sell copies of the Software,
and to
        permit persons to whom
the Software is furnished to do so,
subject to
        the following conditions:

        The above copyright
notice and this permission notice
shall be
        included in all copies or
substantial portions of the Software.

        THE SOFTWARE IS
PROVIDED "AS IS", WITHOUT
WARRANTY OF ANY KIND,
        EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO
THE WARRANTIES OF
        MERCHANTABILITY,
FITNESS FOR A PARTICULAR
PURPOSE AND
        NONINFRINGEMENT. IN
NO EVENT SHALL THE AUTHORS
OR COPYRIGHT HOLDERS BE
        LIABLE FOR ANY CLAIM,
DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION
        OF CONTRACT, TORT
OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION
        WITH THE SOFTWARE
OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.

        Much of the code was
inspired by Derek Yerger's code,
though I don't
        think much of that
remains.  In any event that was..
            (copyleft) 2006 by
Derek Yerger - Free to distribute
freely.

        The CRC code was
excerpted and inspired by the Dallas
Semiconductor
        sample code bearing this
copyright.
        //---------------------------
-----------------------------------
        // Copyright (C) 2000
Dallas Semiconductor Corporation,
All Rights Reserved.
        //
        // Permission is hereby
granted, free of charge, to any
person obtaining a
        // copy of this software and
associated documentation files (the
"Software"),
        // to deal in the Software
without restriction, including without
limitation
        // the rights to use, copy,
modify, merge, publish, distribute,
sublicense,
        // and/or sell copies of the
Software, and to permit persons to
whom the
        // Software is furnished to
do so, subject to the following
conditions:
        //
        // The above copyright
notice and this permission notice
shall be included
        // in all copies or
substantial portions of the Software.
        //
        // THE SOFTWARE IS
PROVIDED "AS IS", WITHOUT
WARRANTY OF ANY KIND,
EXPRESS
        // OR IMPLIED,
INCLUDING BUT NOT LIMITED TO
THE WARRANTIES OF
        // MERCHANTABILITY,
FITNESS FOR A PARTICULAR
PURPOSE AND
NONINFRINGEMENT.
        // IN NO EVENT SHALL
DALLAS SEMICONDUCTOR BE
LIABLE FOR ANY CLAIM,
DAMAGES
        // OR OTHER LIABILITY,
WHETHER IN AN ACTION OF
CONTRACT, TORT OR
OTHERWISE,
        // ARISING FROM, OUT
OF OR IN CONNECTION WITH
THE SOFTWARE OR THE USE OR
        // OTHER DEALINGS IN
THE SOFTWARE.
        //
        // Except as contained in
this notice, the name of Dallas
Semiconductor
        // shall not be used except
as stated in the Dallas
Semiconductor
        // Branding Policy.
        //---------------------------
-----------------------------------
        */

        #include "OneWire.h"


        OneWire::OneWire(uint8_t
pin)
        {
                pinMode(pin,
INPUT);
                bitmask =
PIN_TO_BITMASK(pin);
                baseReg =
PIN_TO_BASEREG(pin);
        #if ONEWIRE_SEARCH
                reset_search();
        #endif
        }


        // Perform the onewire
reset function.  We will wait up to
250uS for
        // the bus to come high, if it
doesn't then it is broken or shorted
        // and we return a 0;
        //
        // Returns 1 if a device
asserted a presence pulse, 0
otherwise.
        //
        uint8_t
OneWire::reset(void)
        {
                IO_REG_TYPE
mask = bitmask;
                volatile
IO_REG_TYPE *reg IO_REG_ASM
= baseReg;
                uint8_t r;
                uint8_t retries =
125;

                noInterrupts();
                DIRECT_MODE
_INPUT(reg, mask);
                interrupts();
```

```
                    // wait until the
wire is high... just in case
                    do {
                              if (--
retries == 0) return 0;

          delayMicroseconds(2);
                    } while (
!DIRECT_READ(reg, mask));

                    noInterrupts();
                    DIRECT_WRITE
_LOW(reg, mask);
                    DIRECT_MODE
_OUTPUT(reg, mask);         //
drive output low
                    interrupts();
                    delayMicroseco
nds(500);
                    noInterrupts();
                    DIRECT_MODE
_INPUT(reg, mask);// allow it to float
                    delayMicroseco
nds(80);
                    r =
!DIRECT_READ(reg, mask);
                    interrupts();
                    delayMicroseco
nds(420);
                    return r;
          }

          //
          // Write a bit. Port and bit is
used to cut lookup time and provide
          // more certain timing.
          //
          void
OneWire::write_bit(uint8_t v)
          {
                    IO_REG_TYPE
mask=bitmask;
                    volatile
IO_REG_TYPE *reg IO_REG_ASM
= baseReg;

                    if (v & 1) {

          noInterrupts();

          DIRECT_WRITE_LOW(re
g, mask);

          DIRECT_MODE_OUTPUT
(reg, mask);         // drive output
low

          delayMicroseconds(10);

          DIRECT_WRITE_HIGH(re
g, mask); // drive output high

          interrupts();

          delayMicroseconds(55);
                    } else {

          noInterrupts();

          DIRECT_WRITE_LOW(re
g, mask);

          DIRECT_MODE_OUTPUT
(reg, mask);         // drive output
low

          delayMicroseconds(65);

          DIRECT_WRITE_HIGH(re
g, mask); // drive output high

          interrupts();

          delayMicroseconds(5);
                    }
          }

          //
          // Read a bit. Port and bit
is used to cut lookup time and
provide
          // more certain timing.
          //
          uint8_t
OneWire::read_bit(void)
          {
                    IO_REG_TYPE
mask=bitmask;
                    volatile
IO_REG_TYPE *reg IO_REG_ASM
= baseReg;
                    uint8_t r;

                    noInterrupts();
                    DIRECT_MODE
_OUTPUT(reg, mask);
                    DIRECT_WRITE
_LOW(reg, mask);
                    delayMicroseco
nds(3);
                    DIRECT_MODE
_INPUT(reg, mask);// let pin float,
pull up will raise
                    delayMicroseco
nds(10);
                    r =
DIRECT_READ(reg, mask);
                    interrupts();
                    delayMicroseco
nds(53);
                    return r;
          }

          //
          // Write a byte. The writing
code uses the active drivers to raise
the
          // pin high, if you need
power after the write (e.g. DS18S20
in
          // parasite power mode)
then set 'power' to 1, otherwise the
pin will
          // go tri-state at the end of
the write to avoid heating in a short
or
          // other mishap.
          //
          void
OneWire::write(uint8_t v, uint8_t
power /* = 0 */) {
               uint8_t bitMask;

               for (bitMask = 0x01;
bitMask; bitMask <<= 1) {
                    OneWire::write_
bit( (bitMask & v)?1:0);
               }
               if ( !power) {
                    noInterrupts();
                    DIRECT_MODE
_INPUT(baseReg, bitmask);
                    DIRECT_WRITE
_LOW(baseReg, bitmask);
                    interrupts();
               }
          }

          void
OneWire::write_bytes(const uint8_t
*buf, uint16_t count, bool power /* =
0 */) {
               for (uint16_t i = 0 ; i <
count ; i++)
                    write(buf[i]);
               if (!power) {
                    noInterrupts();

DIRECT_MODE_INPUT(baseReg,
bitmask);

DIRECT_WRITE_LOW(baseReg,
bitmask);
                    interrupts();
               }
          }

          //
          // Read a byte
          //
          uint8_t OneWire::read() {
               uint8_t bitMask;
               uint8_t r = 0;

               for (bitMask = 0x01;
bitMask; bitMask <<= 1) {
                    if (
OneWire::read_bit()) r |= bitMask;
               }
               return r;
          }

          void
OneWire::read_bytes(uint8_t *buf,
uint16_t count) {
               for (uint16_t i = 0 ; i <
count ; i++)
                    buf[i] = read();
          }

          //
          // Do a ROM select
          //
          void OneWire::select(
uint8_t rom[8])
          {
               int i;
```

```
            write(0x55);          //
Choose ROM

              for( i = 0; i < 8; i++)
write(rom[i]);
            }

            //
            // Do a ROM skip
            //
            void OneWire::skip()
            {
              write(0xCC);          //
Skip ROM
            }

            void OneWire::depower()
            {
                    noInterrupts();
                    DIRECT_MODE
_INPUT(baseReg, bitmask);
                    interrupts();
            }

            #if ONEWIRE_SEARCH

            //
            // You need to use this
function to start a search again from
the beginning.
            // You do not need to do it
for the first search, though you could.
            //
            void
OneWire::reset_search()
             {
             // reset the search state
             LastDiscrepancy = 0;
             LastDeviceFlag = FALSE;
             LastFamilyDiscrepancy =
0;
             for(int i = 7; ; i--)
              {
              ROM_NO[i] = 0;
              if ( i == 0) break;
              }
             }

            //
            // Perform a search. If this
function returns a '1' then it has
            // enumerated the next
device and you may retrieve the
ROM from the
            // OneWire::address
variable. If there are no devices, no
further
            // devices, or something
horrible happens in the middle of the
            // enumeration then a 0 is
returned.  If a new device is found
then
            // its address is copied to
newAddr.  Use
OneWire::reset_search() to
            // start over.
            //
            // --- Replaced by the one
from the Dallas Semiconductor web
site ---
            //--------------------------
--------------------------------
            // Perform the 1-Wire
Search Algorithm on the 1-Wire bus
using the existing
            // search state.
            // Return TRUE  : device
found, ROM number in ROM_NO
buffer
            //        FALSE : device not
found, end of search
            //
            uint8_t
OneWire::search(uint8_t *newAddr)
            {
              uint8_t id_bit_number;
              uint8_t last_zero,
rom_byte_number, search_result;
              uint8_t id_bit,
cmp_id_bit;

              unsigned char
rom_byte_mask, search_direction;

              // initialize for search
              id_bit_number = 1;
              last_zero = 0;
              rom_byte_number = 0;
              rom_byte_mask = 1;
              search_result = 0;

              // if the last call was not
the last one
              if (!LastDeviceFlag)
              {
                // 1-Wire reset
                if (!reset())
                {
                  // reset the search
                  LastDiscrepancy = 0;
                  LastDeviceFlag =
FALSE;

LastFamilyDiscrepancy = 0;
                  return FALSE;
                }

                // issue the search
command
                write(0xF0);

                // loop to do the search
                do
                {
                  // read a bit and its
complement
                  id_bit = read_bit();
                  cmp_id_bit =
read_bit();

                  // check for no
devices on 1-wire
                  if ((id_bit == 1) &&
(cmp_id_bit == 1))
                    break;
                  else
                  {
                    // all devices
coupled have 0 or 1
                    if (id_bit !=
cmp_id_bit)
                      search_direction
= id_bit;  // bit write value for search
                    else
                    {
                      // if this
discrepancy if before the Last
Discrepancy
                      // on a previous
next then pick the same as last time
                      if (id_bit_number
< LastDiscrepancy)

search_direction =
((ROM_NO[rom_byte_number] &
rom_byte_mask) > 0);
                      else
                        // if equal to
last pick 1, if not then pick 0

search_direction = (id_bit_number
== LastDiscrepancy);

                      // if 0 was picked
then record its position in LastZero
                      if
(search_direction == 0)
                      {
                        last_zero =
id_bit_number;

                        // check for
Last discrepancy in family
                        if (last_zero <
9)

LastFamilyDiscrepancy = last_zero;
                      }
                    }

                    // set or clear the
bit in the ROM byte
rom_byte_number
                    // with mask
rom_byte_mask
                    if (search_direction
== 1)

ROM_NO[rom_byte_number] |=
rom_byte_mask;
                    else

ROM_NO[rom_byte_number] &=
~rom_byte_mask;

                    // serial number
search direction write bit

write_bit(search_direction);

                    // increment the
byte counter id_bit_number
                    // and shift the
mask rom_byte_mask
                    id_bit_number++;
                    rom_byte_mask
<<= 1;
```

```
                    // if the mask is 0
then go to new SerialNum byte
rom_byte_number and reset mask
                    if (rom_byte_mask
== 0)
                    {

rom_byte_number++;
                        rom_byte_mask
= 1;
                    }
                  }
                }

while(rom_byte_number < 8);  // loop
until through all ROM bytes 0-7

                // if the search was
successful then
                if (!(id_bit_number <
65))
                {
                  // search successful
so set
LastDiscrepancy,LastDeviceFlag,sea
rch_result
                  LastDiscrepancy =
last_zero;

                  // check for last
device
                  if (LastDiscrepancy
== 0)
                    LastDeviceFlag =
TRUE;

                  search_result =
TRUE;
                }
              }

              // if no device found then
reset counters so next 'search' will
be like a first
              if (!search_result ||
!ROM_NO[0])
              {
                LastDiscrepancy = 0;
                LastDeviceFlag =
FALSE;

LastFamilyDiscrepancy = 0;
                search_result =
FALSE;
              }
              for (int i = 0; i < 8; i++)
newAddr[i] = ROM_NO[i];
              return search_result;
             }

            #endif

            #if ONEWIRE_CRC
            // The 1-Wire CRC scheme
is described in Maxim Application
Note 27:
            // "Understanding and
Using Cyclic Redundancy Checks
with Maxim iButton Products"
            //

            #if
ONEWIRE_CRC8_TABLE
            // This table comes from
Dallas sample code where it is freely
reusable,
            // though Copyright (C)
2000 Dallas Semiconductor
Corporation
            static const uint8_t
PROGMEM dscrc_table[] = {
                0, 94,188,226, 97,
63,221,131,194,156,126,
32,163,253, 31, 65,
                157,195,
33,127,252,162, 64, 30, 95,
1,227,189, 62, 96,130,220,
                35,125,159,193, 66,
28,254,160,225,191, 93,  3,128,222,
60, 98,
                190,224,  2, 92,223,129,
99, 61,124, 34,192,158, 29,
67,161,255,
                70, 24,250,164,
39,121,155,197,132,218,
56,102,229,187, 89,  7,
                219,133,103,
57,186,228,  6, 88, 25,
71,165,251,120, 38,196,154,
                101, 59,217,135,  4,
90,184,230,167,249, 27,
69,198,152,122, 36,
                248,166, 68,
26,153,199, 37,123, 58,100,134,216,
91,  5,231,185,
                140,210,
48,110,237,179, 81, 15, 78,
16,242,172, 47,113,147,205,
                17, 79,173,243,112,
46,204,146,211,141,111, 49,178,236,
14, 80,
                175,241, 19,
77,206,144,114, 44,109, 51,209,143,
12, 82,176,238,
                50,108,142,208, 83,
13,239,177,240,174, 76, 18,145,207,
45,115,
                202,148,118,
40,171,245, 23, 73,  8,
86,180,234,105, 55,213,139,
                87,  9,235,181,
54,104,138,212,149,203,
41,119,244,170, 72, 22,
                233,183, 85,
11,136,214, 52,106, 43,117,151,201,
74, 20,246,168,
                116, 42,200,150, 21,
75,169,247,182,232, 10,
84,215,137,107, 53};

            //
            // Compute a Dallas
Semiconductor 8 bit CRC. These
show up in the ROM
            // and the registers. (note:
this might better be done without to
            // table, it would probably
be smaller and certainly fast enough
            // compared to all those
delayMicrosecond() calls.  But I got
            // confused, so I use this
table from the examples.)
            //
            uint8_t OneWire::crc8(
uint8_t *addr, uint8_t len)
            {
                uint8_t crc = 0;

                while (len--) {
                    crc =
pgm_read_byte(dscrc_table + (crc ^
*addr++));
                }
                return crc;
            }
            #else
            //
            // Compute a Dallas
Semiconductor 8 bit CRC directly.
            // this is much slower, but
much smaller, than the lookup table.
            //
            uint8_t OneWire::crc8(
uint8_t *addr, uint8_t len)
            {
                uint8_t crc = 0;

                while (len--) {

            uint8_t inbyte = *addr++;
                    for
(uint8_t i = 8; i; i--) {

            uint8_t mix = (crc ^ inbyte)
& 0x01;

            crc >>= 1;

            if (mix) crc ^= 0x8C;

            inbyte >>= 1;
                    }
                }
                return crc;
            }
            #endif

            #if ONEWIRE_CRC16
            bool
OneWire::check_crc16(uint8_t*
input, uint16_t len, uint8_t*
inverted_crc)
            {
                uint16_t crc =
~crc16(input, len);
                return (crc & 0xFF) ==
inverted_crc[0] && (crc >> 8) ==
inverted_crc[1];
            }

            uint16_t
OneWire::crc16(uint8_t* input,
uint16_t len)
            {
                static const uint8_t
oddparity[16] =
                    { 0, 1, 1, 0, 1, 0, 0, 1,
1, 0, 0, 1, 0, 1, 1, 0 };
                uint16_t crc = 0;    //
Starting seed is zero.
```

```
            for (uint16_t i = 0 ; i <
len ; i++) {
            // Even though we're
just copying a byte from the input,
            // we'll be doing 16-bit
computation with it.
            uint16_t cdata =
input[i];
            cdata = (cdata ^ (crc &
0xff)) & 0xff;
            crc >>= 8;

            if (oddparity[cdata &
0x0F] ^ oddparity[cdata >> 4])
              crc ^= 0xC001;

            cdata <<= 6;
            crc ^= cdata;
            cdata <<= 1;
            crc ^= cdata;
          }
          return crc;
        }
        #endif

        #endif

        /*
          TwoWire.h - TWI/I2C
library for Arduino & Wiring
          Copyright (c) 2006
Nicholas Zambetti.  All right
reserved.

          This library is free
software; you can redistribute it
and/or
          modify it under the terms
of the GNU Lesser General Public
          License as published by
the Free Software Foundation; either
          version 2.1 of the
License, or (at your option) any later
version.

          This library is distributed
in the hope that it will be useful,
          but WITHOUT ANY
WARRANTY; without even the
implied warranty of
          MERCHANTABILITY or
FITNESS FOR A PARTICULAR
PURPOSE.  See the GNU
          Lesser General Public
License for more details.

          You should have received
a copy of the GNU Lesser General
Public
          License along with this
library; if not, write to the Free
Software
          Foundation, Inc., 51
Franklin St, Fifth Floor, Boston, MA
02110-1301  USA

          Modified 2012 by Todd
Krein (todd@krein.org) to implement
repeated starts
          */

          #ifndef TwoWire_h
          #define TwoWire_h

          #include <inttypes.h>
          #include "Stream.h"

          #define
BUFFER_LENGTH 32

          class TwoWire : public
Stream
          {
            private:
              static uint8_t rxBuffer[];
              static uint8_t
rxBufferIndex;
              static uint8_t
rxBufferLength;

              static uint8_t txAddress;
              static uint8_t txBuffer[];
              static uint8_t
txBufferIndex;
              static uint8_t
txBufferLength;

              static uint8_t
transmitting;
              static void
(*user_onRequest)(void);
              static void
(*user_onReceive)(int);
              static void
onRequestService(void);
              static void
onReceiveService(uint8_t*, int);
            public:
              TwoWire();
              void begin();
              void begin(uint8_t);
              void begin(int);
              void
beginTransmission(uint8_t);
              void
beginTransmission(int);
              uint8_t
endTransmission(void);
              uint8_t
endTransmission(uint8_t);
              uint8_t
requestFrom(uint8_t, uint8_t);
              uint8_t
requestFrom(uint8_t, uint8_t,
uint8_t);
              uint8_t requestFrom(int,
int);
              uint8_t requestFrom(int,
int, int);
              virtual size_t
write(uint8_t);
              virtual size_t write(const
uint8_t *, size_t);
              virtual int
available(void);
              virtual int read(void);
              virtual int peek(void);
                    virtual void
flush(void);
              void onReceive( void
(*)(int) );
              void onRequest( void
(*)(void) );

              inline size_t
write(unsigned long n) { return
write((uint8_t)n); }
              inline size_t write(long
n) { return write((uint8_t)n); }
              inline size_t
write(unsigned int n) { return
write((uint8_t)n); }
              inline size_t write(int n) {
return write((uint8_t)n); }
              using Print::write;
          };

          extern TwoWire Wire;

          #endif

          /*
            TwoWire.cpp - TWI/I2C
library for Wiring & Arduino
            Copyright (c) 2006
Nicholas Zambetti.  All right
reserved.

            This library is free
software; you can redistribute it
and/or
            modify it under the terms
of the GNU Lesser General Public
            License as published by
the Free Software Foundation; either
            version 2.1 of the
License, or (at your option) any later
version.

            This library is distributed
in the hope that it will be useful,
            but WITHOUT ANY
WARRANTY; without even the
implied warranty of
            MERCHANTABILITY or
FITNESS FOR A PARTICULAR
PURPOSE.  See the GNU
            Lesser General Public
License for more details.

            You should have received
a copy of the GNU Lesser General
Public
            License along with this
library; if not, write to the Free
Software
            Foundation, Inc., 51
Franklin St, Fifth Floor, Boston, MA
02110-1301  USA

            Modified 2012 by Todd
Krein (todd@krein.org) to implement
repeated starts
            */

          extern "C" {
            #include <stdlib.h>
            #include <string.h>
            #include "inttypes.h"
```

```
  #include "twi.h"
}

#include "Wire.h"

// Initialize Class Variables
//////////////////////////////////////////////

uint8_t TwoWire::rxBuffer[BUFFER_LENGTH];
uint8_t TwoWire::rxBufferIndex = 0;
uint8_t TwoWire::rxBufferLength = 0;

uint8_t TwoWire::txAddress = 0;
uint8_t TwoWire::txBuffer[BUFFER_LENGTH];
uint8_t TwoWire::txBufferIndex = 0;
uint8_t TwoWire::txBufferLength = 0;

uint8_t TwoWire::transmitting = 0;
void (*TwoWire::user_onRequest)(void);
void (*TwoWire::user_onReceive)(int);

// Constructors
////////////////////////////////////////////////////////////////
/////

TwoWire::TwoWire()
{
}

// Public Methods
////////////////////////////////////////////////////////////////
///

void TwoWire::begin(void)
{
  rxBufferIndex = 0;
  rxBufferLength = 0;

  txBufferIndex = 0;
  txBufferLength = 0;

  twi_init();
}

void TwoWire::begin(uint8_t address)
{
  twi_setAddress(address);

twi_attachSlaveTxEvent(onRequestService);

twi_attachSlaveRxEvent(onReceiveService);
  begin();
}

void TwoWire::begin(int address)
{
  begin((uint8_t)address);
}

uint8_t TwoWire::requestFrom(uint8_t address, uint8_t quantity, uint8_t sendStop)
{
  // clamp to buffer length
  if(quantity > BUFFER_LENGTH){
    quantity = BUFFER_LENGTH;
  }
  // perform blocking read into buffer
  uint8_t read = twi_readFrom(address, rxBuffer, quantity, sendStop);
  // set rx buffer iterator vars
  rxBufferIndex = 0;
  rxBufferLength = read;

  return read;
}

uint8_t TwoWire::requestFrom(uint8_t address, uint8_t quantity)
{
  return requestFrom((uint8_t)address, (uint8_t)quantity, (uint8_t)true);
}

uint8_t TwoWire::requestFrom(int address, int quantity)
{
  return requestFrom((uint8_t)address, (uint8_t)quantity, (uint8_t)true);
}

uint8_t TwoWire::requestFrom(int address, int quantity, int sendStop)
{
  return requestFrom((uint8_t)address, (uint8_t)quantity, (uint8_t)sendStop);
}

void TwoWire::beginTransmission(uint8_t address)
{
  // indicate that we are transmitting
  transmitting = 1;
  // set address of targeted slave
  txAddress = address;
  // reset tx buffer iterator vars
  txBufferIndex = 0;
  txBufferLength = 0;
}

void TwoWire::beginTransmission(int address)
{

beginTransmission((uint8_t)address);
;
}

//
//  Originally, 'endTransmission' was an f(void) function.
//  It has been modified to take one parameter indicating
//  whether or not a STOP should be performed on the bus.
//  Calling endTransmission(false) allows a sketch to
//  perform a repeated start.
//
//  WARNING: Nothing in the library keeps track of whether
//  the bus tenure has been properly ended with a STOP. It
//  is very possible to leave the bus in a hung state if
//  no call to endTransmission(true) is made. Some I2C
//  devices will behave oddly if they do not see a STOP.
//
uint8_t TwoWire::endTransmission(uint8_t sendStop)
{
  // transmit buffer (blocking)
  int8_t ret = twi_writeTo(txAddress, txBuffer, txBufferLength, 1, sendStop);
  // reset tx buffer iterator vars
  txBufferIndex = 0;
  txBufferLength = 0;
  // indicate that we are done transmitting
  transmitting = 0;
  return ret;
}

//  This provides backwards compatibility with the original
//  definition, and expected behaviour, of endTransmission
//
```

```
uint8_t
TwoWire::endTransmission(void)
{
  return
endTransmission(true);
}

// must be called in:
// slave tx event callback
// or after
beginTransmission(address)
size_t
TwoWire::write(uint8_t data)
{
  if(transmitting){
  // in master transmitter
mode
    // don't bother if buffer is
full
    if(txBufferLength >=
BUFFER_LENGTH){
      setWriteError();
      return 0;
    }
    // put byte in tx buffer
    txBuffer[txBufferIndex] =
data;
    ++txBufferIndex;
    // update amount in
buffer
    txBufferLength =
txBufferIndex;
  }else{
  // in slave send mode
    // reply to master
    twi_transmit(&data, 1);
  }
  return 1;
}

// must be called in:
// slave tx event callback
// or after
beginTransmission(address)
size_t
TwoWire::write(const uint8_t *data,
size_t quantity)
{
  if(transmitting){
  // in master transmitter
mode
    for(size_t i = 0; i <
quantity; ++i){
      write(data[i]);
    }
  }else{
  // in slave send mode
    // reply to master
    twi_transmit(data,
quantity);
  }
  return quantity;
}

// must be called in:
// slave rx event callback
// or after
requestFrom(address, numBytes)
int
TwoWire::available(void)
{
  return rxBufferLength -
rxBufferIndex;
}

// must be called in:
// slave rx event callback
// or after
requestFrom(address, numBytes)
int TwoWire::read(void)
{
  int value = -1;

  // get each successive
byte on each call
  if(rxBufferIndex <
rxBufferLength){
    value =
rxBuffer[rxBufferIndex];
    ++rxBufferIndex;
  }

  return value;
}

// must be called in:
// slave rx event callback
// or after
requestFrom(address, numBytes)
int TwoWire::peek(void)
{
  int value = -1;

  if(rxBufferIndex <
rxBufferLength){
    value =
rxBuffer[rxBufferIndex];
  }

  return value;
}

void TwoWire::flush(void)
{
  // XXX: to be
implemented.
}

// behind the scenes
function that is called when data is
received
void
TwoWire::onReceiveService(uint8_t*
inBytes, int numBytes)
{
  // don't bother if user
hasn't registered a callback
  if(!user_onReceive){
    return;
  }
  // don't bother if rx buffer
is in use by a master requestFrom()
op
  // i know this drops data,
but it allows for slight stupidity
  // meaning, they may not
have read all the master
requestFrom() data yet
  if(rxBufferIndex <
rxBufferLength){
    return;
  }
  // copy twi rx buffer into
local read buffer
  // this enables new reads
to happen in parallel
  for(uint8_t i = 0; i <
numBytes; ++i){
    rxBuffer[i] = inBytes[i];
  }
  // set rx iterator vars
  rxBufferIndex = 0;
  rxBufferLength =
numBytes;
  // alert user program

user_onReceive(numBytes);
}

// behind the scenes
function that is called when data is
requested
void
TwoWire::onRequestService(void)
{
  // don't bother if user
hasn't registered a callback
  if(!user_onRequest){
    return;
  }
  // reset tx buffer iterator
vars
  // !!! this will kill any
pending pre-master sendTo() activity
  txBufferIndex = 0;
  txBufferLength = 0;
  // alert user program
  user_onRequest();
}

// sets function called on
slave write
void TwoWire::onReceive(
void (*function)(int) )
{
  user_onReceive =
function;
}

// sets function called on
slave read
void TwoWire::onRequest(
void (*function)(void) )
{
  user_onRequest =
function;
}

// Preinstantiate Objects
//////////////////////////////////////////////////

TwoWire Wire =
TwoWire();

/*
  twi.h - TWI/I2C library for
Wiring & Arduino
  Copyright (c) 2006
Nicholas Zambetti.  All right
reserved.
```

```
        This library is free
software; you can redistribute it
and/or
        modify it under the terms
of the GNU Lesser General Public
        License as published by
the Free Software Foundation; either
        version 2.1 of the
License, or (at your option) any later
version.

        This library is distributed
in the hope that it will be useful,
        but WITHOUT ANY
WARRANTY; without even the
implied warranty of
        MERCHANTABILITY or
FITNESS FOR A PARTICULAR
PURPOSE.  See the GNU
        Lesser General Public
License for more details.

        You should have received
a copy of the GNU Lesser General
Public
        License along with this
library; if not, write to the Free
Software
        Foundation, Inc., 51
Franklin St, Fifth Floor, Boston, MA
02110-1301  USA
        */

      #ifndef twi_h
      #define twi_h

        #include <inttypes.h>

        //#define ATMEGA8

        #ifndef TWI_FREQ
        #define TWI_FREQ
100000L
        #endif

        #ifndef
TWI_BUFFER_LENGTH
        #define
TWI_BUFFER_LENGTH 32
        #endif

        #define TWI_READY 0
        #define TWI_MRX   1
        #define TWI_MTX   2
        #define TWI_SRX   3
        #define TWI_STX   4

        void twi_init(void);
        void
twi_setAddress(uint8_t);
        uint8_t
twi_readFrom(uint8_t, uint8_t*,
uint8_t, uint8_t);
        uint8_t
twi_writeTo(uint8_t, uint8_t*, uint8_t,
uint8_t, uint8_t);
        uint8_t twi_transmit(const
uint8_t*, uint8_t);
        void
twi_attachSlaveRxEvent( void
(*)(uint8_t*, int) );
        void
twi_attachSlaveTxEvent( void
(*)(void) );
        void twi_reply(uint8_t);
        void twi_stop(void);
        void
twi_releaseBus(void);

      #endif

      /*
        twi.c - TWI/I2C library for
Wiring & Arduino
        Copyright (c) 2006
Nicholas Zambetti.  All right
reserved.

        This library is free
software; you can redistribute it
and/or
        modify it under the terms
of the GNU Lesser General Public
        License as published by
the Free Software Foundation; either
        version 2.1 of the
License, or (at your option) any later
version.

        This library is distributed
in the hope that it will be useful,
        but WITHOUT ANY
WARRANTY; without even the
implied warranty of
        MERCHANTABILITY or
FITNESS FOR A PARTICULAR
PURPOSE.  See the GNU
        Lesser General Public
License for more details.

        You should have received
a copy of the GNU Lesser General
Public
        License along with this
library; if not, write to the Free
Software
        Foundation, Inc., 51
Franklin St, Fifth Floor, Boston, MA
02110-1301  USA

        Modified 2012 by Todd
Krein (todd@krein.org) to implement
repeated starts
        */

      #include <math.h>
      #include <stdlib.h>
      #include <inttypes.h>
      #include <avr/io.h>
      #include <avr/interrupt.h>
      #include <compat/twi.h>
      #include "Arduino.h" // for
digitalWrite

      #ifndef cbi
      #define cbi(sfr, bit)
(_SFR_BYTE(sfr) &= ~_BV(bit))
      #endif
      #ifndef sbi
      #define sbi(sfr, bit)
(_SFR_BYTE(sfr) |= _BV(bit))
      #endif

      #include "pins_arduino.h"
      #include "twi.h"

      static volatile uint8_t
twi_state;
      static volatile uint8_t
twi_slarw;
      static volatile uint8_t
twi_sendStop;
      // should the transaction
end with a stop
      static volatile uint8_t
twi_inRepStart;
      // in the middle of a
repeated start

      static void
(*twi_onSlaveTransmit)(void);
      static void
(*twi_onSlaveReceive)(uint8_t*, int);

      static uint8_t
twi_masterBuffer[TWI_BUFFER_LE
NGTH];
      static volatile uint8_t
twi_masterBufferIndex;
      static volatile uint8_t
twi_masterBufferLength;

      static uint8_t
twi_txBuffer[TWI_BUFFER_LENGTH
];
      static volatile uint8_t
twi_txBufferIndex;
      static volatile uint8_t
twi_txBufferLength;

      static uint8_t
twi_rxBuffer[TWI_BUFFER_LENGT
H];
      static volatile uint8_t
twi_rxBufferIndex;

      static volatile uint8_t
twi_error;

      /*
       * Function twi_init
       * Desc    readys twi pins
and sets twi bitrate
       * Input   none
       * Output  none
       */
      void twi_init(void)
      {
        // initialize state
        twi_state = TWI_READY;
        twi_sendStop = true;
              // default value
        twi_inRepStart = false;

        // activate internal pullups
for twi.
        digitalWrite(SDA, 1);
```

```
        digitalWrite(SCL, 1);

        // initialize twi prescaler
and bit rate
        cbi(TWSR, TWPS0);
        cbi(TWSR, TWPS1);
        TWBR = ((F_CPU /
TWI_FREQ) - 16) / 2;

        /* twi bit rate formula from
atmega128 manual pg 204
        SCL Frequency = CPU
Clock Frequency / (16 + (2 * TWBR))
        note: TWBR should be 10
or higher for master mode
        It is 72 for a 16mhz
Wiring board with 100kHz TWI */

        // enable twi module,
acks, and twi interrupt
        TWCR = _BV(TWEN) |
_BV(TWIE) | _BV(TWEA);
      }

      /*
       * Function twi_slaveInit
       * Desc     sets slave
address and enables interrupt
       * Input    none
       * Output   none
       */
      void
twi_setAddress(uint8_t address)
      {
       // set twi slave address
(skip over TWGCE bit)
       TWAR = address << 1;
      }

      /*
       * Function twi_readFrom
       * Desc     attempts to
become twi bus master and read a
       *        series of bytes
from a device on the bus
       * Input    address: 7bit i2c
device address
       *        data: pointer to
byte array
       *        length: number of
bytes to read into array
       *        sendStop: Boolean
indicating whether to send a stop at
the end
       * Output   number of bytes
read
       */
      uint8_t
twi_readFrom(uint8_t address,
uint8_t* data, uint8_t length, uint8_t
sendStop)
      {
       uint8_t i;

       // ensure data will fit into
buffer

if(TWI_BUFFER_LENGTH < length){
        return 0;
       }

       // wait until twi is ready,
become master receiver
       while(TWI_READY !=
twi_state){
        continue;
       }
       twi_state = TWI_MRX;
       twi_sendStop =
sendStop;
       // reset error state (0xFF..
no error occured)
       twi_error = 0xFF;

       // initialize buffer iteration
vars
       twi_masterBufferIndex =
0;
       twi_masterBufferLength =
length-1;  // This is not intuitive, read
on...
       // On receive, the
previously configured ACK/NACK
setting is transmitted in
       // response to the
received byte before the interrupt is
signalled.
       // Therefor we must
actually set NACK when the _next_
to last byte is
       // received, causing that
NACK to be sent in response to
receiving the last
       // expected byte of data.

       // build sla+w, slave
device address + w bit
       twi_slarw = TW_READ;
       twi_slarw |= address <<
1;

       if (true == twi_inRepStart)
{
        // if we're in the
repeated start state, then we've
already sent the start,
       // (@@@ we hope), and
the TWI statemachine is just waiting
for the address byte.
       // We need to remove
ourselves from the repeated start
state before we enable interrupts,
       // since the ISR is
ASYNC, and we could get confused
if we hit the ISR before cleaning
       // up. Also, don't enable
the START interrupt. There may be
one pending from the
       // repeated start that we
sent outselves, and that would really
confuse things.
        twi_inRepStart = false;
                                //
remember, we're dealing with an
ASYNC ISR
        TWDR = twi_slarw;
        TWCR = _BV(TWINT) |
_BV(TWEA) | _BV(TWEN) |
_BV(TWIE);       // enable INTs,
but not START
       }
       else
        // send start condition
        TWCR = _BV(TWEN) |
_BV(TWIE) | _BV(TWEA) |
_BV(TWINT) | _BV(TWSTA);

       // wait for read operation
to complete
       while(TWI_MRX ==
twi_state){
        continue;
       }

       if (twi_masterBufferIndex
< length)
        length =
twi_masterBufferIndex;

       // copy twi buffer to data
       for(i = 0; i < length; ++i){
        data[i] =
twi_masterBuffer[i];
       }

       return length;
      }

      /*
       * Function twi_writeTo
       * Desc     attempts to
become twi bus master and write a
       *        series of bytes to a
device on the bus
       * Input    address: 7bit i2c
device address
       *        data: pointer to
byte array
       *        length: number of
bytes in array
       *        wait: boolean
indicating to wait for write or not
       *        sendStop: boolean
indicating whether or not to send a
stop at the end
       * Output   0 .. success
       *        1 .. length to long
for buffer
       *        2 .. address send,
NACK received
       *        3 .. data send,
NACK received
       *        4 .. other twi error
(lost bus arbitration, bus error, ..)
       */
      uint8_t twi_writeTo(uint8_t
address, uint8_t* data, uint8_t
length, uint8_t wait, uint8_t
sendStop)
      {
       uint8_t i;

       // ensure data will fit into
buffer

if(TWI_BUFFER_LENGTH < length){
        return 1;
       }
```

```
            // wait until twi is ready,
become master transmitter
            while(TWI_READY !=
twi_state){
              continue;
            }
            twi_state = TWI_MTX;
            twi_sendStop =
sendStop;
            // reset error state (0xFF..
no error occured)
            twi_error = 0xFF;

            // initialize buffer iteration
vars
            twi_masterBufferIndex =
0;
            twi_masterBufferLength =
length;

            // copy data to twi buffer
            for(i = 0; i < length; ++i){
              twi_masterBuffer[i] =
data[i];
            }

            // build sla+w, slave
device address + w bit
            twi_slarw = TW_WRITE;
            twi_slarw |= address <<
1;

            // if we're in a repeated
start, then we've already sent the
START
            // in the ISR. Don't do it
again.
            //
            if (true == twi_inRepStart)
{
              // if we're in the
repeated start state, then we've
already sent the start,
              // (@@@ we hope), and
the TWI statemachine is just waiting
for the address byte.
              // We need to remove
ourselves from the repeated start
state before we enable interrupts,
              // since the ISR is
ASYNC, and we could get confused
if we hit the ISR before cleaning
              // up. Also, don't enable
the START interrupt. There may be
one pending from the
              // repeated start that we
sent outselves, and that would really
confuse things.
              twi_inRepStart = false;
                                //
remember, we're dealing with an
ASYNC ISR
              TWDR = twi_slarw;


              TWCR = _BV(TWINT) |
_BV(TWEA) | _BV(TWEN) |
_BV(TWIE);          // enable INTs,
but not START
            }
            else
              // send start condition
              TWCR = _BV(TWINT) |
_BV(TWEA) | _BV(TWEN) |
_BV(TWIE) | _BV(TWSTA);    //
enable INTs

            // wait for write operation
to complete
            while(wait && (TWI_MTX
== twi_state)){
              continue;
            }

            if (twi_error == 0xFF)
              return 0;           //
success
            else if (twi_error ==
TW_MT_SLA_NACK)
              return 2;           //
error: address send, nack received
            else if (twi_error ==
TW_MT_DATA_NACK)
              return 3;           //
error: data send, nack received
            else
              return 4;           //
other twi error
          }

          /*
           * Function twi_transmit
           * Desc     fills slave tx
buffer with data
           *          must be called in
slave tx event callback
           * Input    data: pointer to
byte array
           *          length: number of
bytes in array
           * Output   1 length too
long for buffer
           *          2 not slave
transmitter
           *          0 ok
           */
          uint8_t twi_transmit(const
uint8_t* data, uint8_t length)
          {
            uint8_t i;

            // ensure data will fit into
buffer

if(TWI_BUFFER_LENGTH < length){
              return 1;
            }

            // ensure we are currently
a slave transmitter
            if(TWI_STX != twi_state){
              return 2;
            }

            // set length and copy
data into tx buffer
            twi_txBufferLength =
length;

            for(i = 0; i < length; ++i){
              twi_txBuffer[i] = data[i];
            }

            return 0;
          }

          /*
           * Function
twi_attachSlaveRxEvent
           * Desc     sets function
called before a slave read operation
           * Input    function: callback
function to use
           * Output   none
           */
          void
twi_attachSlaveRxEvent( void
(*function)(uint8_t*, int) )
          {
            twi_onSlaveReceive =
function;
          }

          /*
           * Function
twi_attachSlaveTxEvent
           * Desc     sets function
called before a slave write operation
           * Input    function: callback
function to use
           * Output   none
           */
          void
twi_attachSlaveTxEvent( void
(*function)(void) )
          {
            twi_onSlaveTransmit =
function;
          }

          /*
           * Function twi_reply
           * Desc     sends byte or
readys receive line
           * Input    ack: byte
indicating to ack or to nack
           * Output   none
           */
          void twi_reply(uint8_t ack)
          {
            // transmit master read
ready signal, with or without ack
            if(ack){
              TWCR = _BV(TWEN) |
_BV(TWIE) | _BV(TWINT) |
_BV(TWEA);
            }else{
                        TWCR =
_BV(TWEN) | _BV(TWIE) |
_BV(TWINT);
            }
          }

          /*
           * Function twi_stop
           * Desc     relinquishes bus
master status
           * Input    none
           * Output   none
           */
          void twi_stop(void)
```

```
        {
          // send stop condition
          TWCR = _BV(TWEN) |
_BV(TWIE) | _BV(TWEA) |
_BV(TWINT) | _BV(TWSTO);

          // wait for stop condition
to be exectued on bus
          // TWINT is not set after a
stop condition!
          while(TWCR &
_BV(TWSTO)){
            continue;
          }

          // update twi state
          twi_state = TWI_READY;
        }

        /*
         * Function twi_releaseBus
         * Desc    releases bus
control
         * Input   none
         * Output  none
         */
        void twi_releaseBus(void)
        {
          // release bus
          TWCR = _BV(TWEN) |
_BV(TWIE) | _BV(TWEA) |
_BV(TWINT);

          // update twi state
          twi_state = TWI_READY;
        }

        SIGNAL(TWI_vect)
        {
          switch(TW_STATUS){
            // All Master
            case TW_START:     //
sent start condition
            case TW_REP_START:
// sent repeated start condition
              // copy device address
and r/w bit to output register and ack
              TWDR = twi_slarw;
              twi_reply(1);
              break;

            // Master Transmitter
            case
TW_MT_SLA_ACK:  // slave receiver
acked address
            case
TW_MT_DATA_ACK: // slave
receiver acked data
              // if there is data to
send, send it, otherwise stop

if(twi_masterBufferIndex <
twi_masterBufferLength){
                // copy data to output
register and ack
                TWDR =
twi_masterBuffer[twi_masterBufferIn
dex++];
                twi_reply(1);
              }else{
                if (twi_sendStop)
                twi_stop();
                else {
                  twi_inRepStart
= true;     // we're gonna send the
START
                  // don't enable
the interrupt. We'll generate the start,
but we
                  // avoid
handling the interrupt until we're in
the next transaction,
                  // at the point
where we would normally issue the
start.
                  TWCR =
_BV(TWINT) | _BV(TWSTA)|
_BV(TWEN) ;
                  twi_state =
TWI_READY;
                }
              }
              break;
            case
TW_MT_SLA_NACK:  // address
sent, nack received
              twi_error =
TW_MT_SLA_NACK;
              twi_stop();
              break;
            case
TW_MT_DATA_NACK: // data sent,
nack received
              twi_error =
TW_MT_DATA_NACK;
              twi_stop();
              break;
            case
TW_MT_ARB_LOST: // lost bus
arbitration
              twi_error =
TW_MT_ARB_LOST;
              twi_releaseBus();
              break;

            // Master Receiver
            case
TW_MR_DATA_ACK: // data
received, ack sent
              // put byte into buffer

twi_masterBuffer[twi_masterBufferIn
dex++] = TWDR;
            case
TW_MR_SLA_ACK:  // address sent,
ack received
              // ack if more bytes are
expected, otherwise nack

if(twi_masterBufferIndex <
twi_masterBufferLength){
                twi_reply(1);
              }else{
                twi_reply(0);
              }
              break;
            case
TW_MR_DATA_NACK: // data
received, nack sent
              // put final byte into
buffer

twi_masterBuffer[twi_masterBufferIn
dex++] = TWDR;
                if (twi_sendStop)
                twi_stop();
                else {
                  twi_inRepStart
= true;     // we're gonna send the
START
                  // don't enable
the interrupt. We'll generate the start,
but we
                  // avoid
handling the interrupt until we're in
the next transaction,
                  // at the point
where we would normally issue the
start.
                  TWCR =
_BV(TWINT) | _BV(TWSTA)|
_BV(TWEN) ;
                  twi_state =
TWI_READY;
                }
                break;
            case
TW_MR_SLA_NACK: // address
sent, nack received
              twi_stop();
              break;
            // TW_MR_ARB_LOST
handled by TW_MT_ARB_LOST
case

            // Slave Receiver
            case
TW_SR_SLA_ACK:   // addressed,
returned ack
            case
TW_SR_GCALL_ACK: // addressed
generally, returned ack
            case
TW_SR_ARB_LOST_SLA_ACK:  //
lost arbitration, returned ack
            case
TW_SR_ARB_LOST_GCALL_ACK:
// lost arbitration, returned ack
              // enter slave receiver
mode
              twi_state = TWI_SRX;
              // indicate that rx buffer
can be overwritten and ack
              twi_rxBufferIndex = 0;
              twi_reply(1);
              break;
            case
TW_SR_DATA_ACK:       // data
received, returned ack
            case
TW_SR_GCALL_DATA_ACK: // data
received generally, returned ack
              // if there is still room in
the rx buffer
              if(twi_rxBufferIndex <
TWI_BUFFER_LENGTH){
                // put byte in buffer
and ack
```

```
twi_rxBuffer[twi_rxBufferIndex++] =
TWDR;
                    twi_reply(1);
                  }else{
                    // otherwise nack
                    twi_reply(0);
                  }
                  break;
                case TW_SR_STOP: //
stop or repeated start condition
received
                  // put a null char after
data if there's room
                  if(twi_rxBufferIndex <
TWI_BUFFER_LENGTH){

twi_rxBuffer[twi_rxBufferIndex] = '\0';
                  }
                  // sends ack and stops
interface for clock stretching
                  twi_stop();
                  // callback to user
defined callback

twi_onSlaveReceive(twi_rxBuffer,
twi_rxBufferIndex);
                  // since we submit rx
buffer to "wire" library, we can reset it
                  twi_rxBufferIndex = 0;
                  // ack future responses
and leave slave receiver state
                  twi_releaseBus();
                  break;
                case
TW_SR_DATA_NACK:      // data
received, returned nack
                case
TW_SR_GCALL_DATA_NACK: //
data received generally, returned
nack
                  // nack back at master
                  twi_reply(0);
                  break;

                // Slave Transmitter
                case
TW_ST_SLA_ACK:        //
addressed, returned ack
                case
TW_ST_ARB_LOST_SLA_ACK: //
arbitration lost, returned ack
                  // enter slave
transmitter mode
                  twi_state = TWI_STX;
                  // ready the tx buffer
index for iteration
                  twi_txBufferIndex = 0;
                  // set tx buffer length to
be zero, to verify if user changes it
                  twi_txBufferLength = 0;
                  // request for txBuffer
to be filled and length to be set
                  // note: user must call
twi_transmit(bytes, length) to do this
                  twi_onSlaveTransmit();
                  // if they didn't change
buffer & length, initialize it
                  if(0 ==
twi_txBufferLength){
                    twi_txBufferLength =
1;
                    twi_txBuffer[0] =
0x00;
                  }
                  // transmit first byte
from buffer, fall
                case
TW_ST_DATA_ACK: // byte sent,
ack returned
                  // copy data to output
register
                  TWDR =
twi_txBuffer[twi_txBufferIndex++];
                  // if there is more to
send, ack, otherwise nack
                  if(twi_txBufferIndex <
twi_txBufferLength){
                    twi_reply(1);
                  }else{
                    twi_reply(0);
                  }
                  break;
                case
TW_ST_DATA_NACK: // received
nack, we are done
                case
TW_ST_LAST_DATA: // received
ack, but we are done already!
                  // ack future responses
                  twi_reply(1);
                  // leave slave receiver
state
                  twi_state =
TWI_READY;
                  break;

                // All
                case TW_NO_INFO:   //
no state information
                  break;
                case
TW_BUS_ERROR: // bus error,
illegal stop/start
                  twi_error =
TW_BUS_ERROR;
                  twi_stop();
                  break;
              }
            }
```

Appendix 6

$$T_{hot_{in}} = 212\ F$$

$$T_{cold_{in}} = 32\ F$$

$$\dot{m} = 3\ \frac{gallons}{minute}$$

$$c_{p_{cold}} = 1.01\ \frac{Btu}{lb_m * F}$$

$$c_{p_{hot}} = 1.007\ \frac{Btu}{lb_m * F}$$

$$h_{hot} = 3000\ \frac{Btu}{ft^2 * hr * F}$$

$$h_{cold} = 50\ \frac{Btu}{ft^2 * hr * F}$$

$$k = 1160\ \frac{Btu}{ft * hr * F}$$

$$A = 40\ in^2$$

$$L = .25\ in$$

$$C_{cold} = c_{p_{cold}} * \dot{m} * 60\ \frac{minutes}{hour} * 8.3436\ \frac{lb_m}{gallon} = 1516.87\ \frac{Btu}{F * hour}$$

$$C_{hot} = c_{p_{hot}} * \dot{m_{hot}} * 60\ \frac{minutes}{hour} * 8.3436\ \frac{lb_m}{gallon} = 1449.30\ \frac{Btu}{F * hour}$$

$$C = \frac{C_{min}}{C_{hot}} = .955$$

$$U = \frac{1}{\frac{1}{h_{cold}} + \frac{L}{k} + \frac{1}{h_{hot}}} = 48.66\ \frac{Btu}{ft^2 * hr * F}$$

$$NTU = \frac{U * A}{C_{hot}} = 1.343$$

$$\xi = \frac{1 - e^{-NTU*(1-C)}}{1 - C * e^{-NTU*(1-C)}} = .5806$$

$$T_{hot_{out}} = T_{hot_{in}} - \xi * \left(T_{hot_{in}} - T_{cold_{in}}\right) = 107.50\ F$$

$$T_{cold_{out}} = T_{cold_{in}} + \xi * C_{cold} * \frac{\left(T_{hot_{in}} - T_{cold_{in}}\right)}{C_{hot}} = 141.37\ F$$