COCKTAIL AUTOMATION MANAGEMENT SYSTEM

A Design Project Report

Presented to the School of Electrical and Computer Engineering of Cornell University

In Partial Fulfillment of the Requirements for the Degree of Master of Engineering, Electrical and Computer Engineering

Submitted by

Cameron Glass (cig23)

MEng Advisor: Bruce Robert Land

Degree Date: August 2014

Abstract

Master of Engineering Program
School of Electrical and Computer Engineering
Cornell University
Design Project Report

Project Title: Cocktail Automation Management System

Author: Cameron Glass

Abstract:
The purpose of the Cocktail Automation Management System (CAMS) is to offer a complete hardware-software system that automatically prepares cocktails for patrons at a bar or restaurant. This systems consists of a central processing board that handles user requests and coordinates communication among modules which each hold one ingredient. The CAMS would complement or replace bartenders for easy-to-mix beverages, which would free up the bartenders for work that could not be automated, such as payment collection and more complex cocktails. The CAMS delivers beverages by acting in two stages – order determination and component delivery. First, a patron orders a cocktail from a list of available drinks using a serial terminal running on a PC. Then, for each ingredient in the cocktail, the CAMS determines what module the ingredient is located in and what volume of that ingredient is needed. A command is issued to each of the relevant modules to dispense the specified ingredient. The modules all dispense ingredients into plastic tubing that runs from the modules to a cocktail glass.

There are several other existing solutions currently on the market. These systems also offer complete hardware-software solutions, but none of these systems are cheap, modular, or scalable. They all require the purchase of a specific sized system, and if at a later point more modules are desired for the system, an entirely new system must be purchased with no way of using the existing system. The CAMS is designed to use as few as 1 ingredient module and as many as 64 ingredient modules. The main controller is separate from the ingredient modules, and the ingredient modules only need connections to power and to the communications bus to add functionality to the system. Additionally, the expected cost of the CAMS is significantly lower than the existing systems per module. The CAMS cost efficiency varies slightly with number of modules as the cost of the main module must be paid regardless of system size. However, the CAMS is more cost effective than all other existing systems for all system sizes.

Executive Summary

The purpose of the Cocktail Automation Management System (CAMS) is to offer a complete hardware-software system that automatically prepares cocktails for patrons at a bar or restaurant. This systems consists of a central processing board that handles user requests and coordinates communication among modules which each hold one ingredient. A module consists of a plastic bottle, an ultrasonic sensor, a valve, a group of tubes and connectors, a structure to mount these components, and a microcontroller board with associated circuitry to run the module components. Some tasks needed to run a bar are difficult to automate, so the system is intended to complement or replace bartenders for easy-to-mix beverages. Preparing more advanced cocktails, collecting payments, swapping out bottles, and many other tasks will still have to be carried out by trained bartenders.

When the CAMS is turned on, it prints a list of all the cocktails available to be made given the current modules, and then prompts the user to select a cocktail to be made. The CAMS delivers beverages by acting in two stages – order determination and component delivery. First, a patron orders a cocktail from a list of available drinks using a serial terminal running on a PC. Then, for each ingredient in the cocktail, the CAMS determines what module the ingredient is located in and what volume of that ingredient is needed. A command is issued over a UART bus to each of the relevant modules to dispense the specified ingredient. The bus does not have any collision management, as the communication method is designed to only have one node communicating on the bus at a time. Delivery of messages is ensured by a system of acknowledges on each command. When a module receive a dispense command, it dispenses its ingredient into plastic tubing that connects directly to the bottle containing the ingredient and ends in an area located above a cocktail glass. Ingredients are dispensed from the bottles using a solenoid valve and the process is controlled by a microcontroller on the module board.

There are several other beverage automation solutions which are either developed by hobbyists or are attempting to attain commercial production through the online crowdfunding platform, Kickstarter. Such systems include the Party Robotics' *Bartendro,* the *Inebriator,* the *BoozeBot*, and *Monsieur.* These systems also offer complete hardware-software solutions, but none of these systems are cheap, modular, or scalable. They all require the purchase of a specific sized system, and if at a later point more modules are desired for the system, an entirely new system must be purchased with no way of using the existing system. For example, the *Monsieur* and the *Inebriator* both only allow for a set number of ingredients. The *BoozeBot* and *Bartendro* both have different models, but provide no way of scaling between models – to increase system size and functionality, a new system must be purchased.

The CAMS is designed to use as few as one ingredient module and as many as 64 ingredient modules. The main controller is separate from the ingredient modules, and the ingredient modules only need connections to power and to the communications bus to add functionality to the system. Additionally, the expected cost of the CAMS is significantly lower than the existing systems per module. Across all systems (including CAMS), the greater number of modules purchased, the cheaper the effective module price, as cost of the main controller is distributed over many modules. The most expensive and least cost effective product is the *Monsieur*, which has a rate of $500 per module, and the most cost effective per module is the *BoozeBot*, which costs 62.5 British Pounds, or about $108. The CAMS prototype has a per

module cost of $103.35. The cost of the largest *BoozeBot* system is more than the cost of an equally sized CAMS. Lastly, the prototype cost of the CAMS was driven up significantly by the debug hardware used and buying each part individually. There are several components used for debugging on the module boards that can be removed for production to reduce cost, and purchasing components and producing boards in bulk would significantly drive down the cost per module.

Table of Contents

Cocktail Automation Management System i
Abstract ii
Executive Summary iii
Table of Contents 1
Table of Figures 3
Introduction 4
Alternative Designs Considered 6
Communications Structure 6
Custom Static Network 6
CAN Bus Communications Structure 7
Mechanical Designs Considered 8
Compressed Air 9
Design 11
Electrical Design 11
Prototyping Circuit Board 11
UART Bus 11
Solenoid Valve Driving Circuit 12
Mechanical Design 13
Software Design 15
Data Structures 15
Main Board Software 16
Initialization 16
User Interface 17
Communication Stack 17
Module Board Software 19
Initialization and UART ISR 19
Communication 19
Ultrasonic Sensor and Valve Control 20
Results 22
Sensor Experiments 22

UART Bus Reliability 22
Valve Calibration Tests 23
Future Work 26
Reducing Electronics Cost 26
Mechanical Design Updates 27
User Interface Updates 28
Conclusions 30
References 31
Appendix A: Software and User Interface 32
Appendix B: Mechanical Design 35
Appendix C: Circuit Board Schematics and Design 37
Appendix D: Component Costs 41
Appendix E: Experimental Results 47
Appendix G: Ultrasonic Sensor Datasheet 53

Table of Figures and Tables

Table A1 – List of Ingredient Numbers and Names 32
Table A2 – List of Cocktails 32
Figure A1 – Example Successful Serial Console Output 33
Figure A2 – Example Successful Ordering Sequence Using a Grounded Bus........ 34
Figure B1 – Front Mechanical View of a CAMS module 35
Figure B2 – SolidWorks Model of Mechanical Structure 36
Figure C1 – Main Board Schematic........ 38
Figure C2 – Constructed Main Board 38
Figure C3 – Module Board Schematic........ 39
Figure C4 – Constructed Module Board (Solenoid Valve and Valve Power Rail Connections Disconnected) 39
Figure C5 – AT90CAN128 Breakout Board 40
Table D1 - Electrical Components........ 41
Table D2 – Main Board Components 43
Table D3 – Module Board Components 44
Table D4– Mechanical Components........ 45
Table D5 – Single Valve Module Mechanical Components 45
Table D6 – Compressed Air Module Mechanical Components 45
Table D7 – Compressed Air System Module Costs 46
Table D8 – Single Valve Module Costs 46
Table D9 – Price Comparisons 46
Table E1 – Results of Ultrasonic Sensor Test 47
Figure E1 – Graph of Ultrasonic Sensor Test........ 48
Table E2 – Error of Ultrasonic Sensor Models........ 49
Table E3 – Results of Constant Duration Valve Test........ 50
Figure E2 – Volume Dispensed in 100ms vs Distance from Ultrasonic Sensor to Top of Liquid. 51
Table E4 – Calibration Data for Valve Durations 52

Introduction

The purpose of the Cocktail Automation Management System (CAMS) is to offer a complete hardware-software system that automatically prepares cocktails for patrons at a bar or restaurant. This systems consists of a central processing board that handles user requests and coordinates communication among modules which each hold one ingredient. A module consists of a plastic bottle, an ultrasonic sensor, a valve, a group of tubes and connectors, a structure to mount these components, and a microcontroller board with associated circuitry to run the module components. Some tasks needed to run a bar cannot automated, so this system is intended to complement or replace bartenders for easy-to-mix beverages. Preparing more advanced cocktails, collecting payments, swapping out bottles, and many other tasks must still be carried out by trained bartenders.

Turning on the CAMS begins the setup process. Using a serial terminal, the CAMS prompts for which ingredients are located in each ingredient module. These ingredients are encoded by number, so this process requires the user to know the mapping from ingredient names to ingredient numbers. Once the modules and contents have been enumerated, it prints a list of all the cocktails available to be made and then prompts the user to select a cocktail to be made. The CAMS delivers beverages by acting in two stages – order determination and component delivery. First a user orders a cocktail from the presented list by menu item number, which was just displayed. Then, for each ingredient in the cocktail, the CAMS determines what module the ingredient is located in and what volume of that ingredient is needed. A command is issued over a UART bus to each of the relevant modules to dispense the specified ingredient. Each dispensing command executes a small handshake between the main board and the module which allows the main board to specify which module it is attempting to send a command to, followed by the number of parts of liquid to dispense. Specifying a number of parts to dispense allows more complex cocktails of varying volumes to be made. The bus does not have any collision management, as the communication method is designed to only have one node communicating on the bus at a time. Delivery of messages is ensured by a system of acknowledges on each command.

Modules dispense ingredients to the cocktail being made by opening and closing a solenoid valve connected to beverage-grade plastic tubing, which transports the liquid into a cocktail glass. Each time a module gets a dispensing command, it must determine how long to hold the valve open as a function of the current volume of liquid in the bottle and the number of parts of liquid specified by the main board. Ingredient bottles have a hole cut out at the top of them which houses an ultrasonic sensor. When a reading is initiated by the module board, the ultrasonic sensor determines the height of the liquid remaining in the bottle. The microcontroller then reads that value and uses it to index a lookup table to determine how long to hold the valve open. The ultrasonic sensor reading is necessary because the volume of liquid in a bottle determines the flow rate out of the bottle, as described through Bernoulli's Principle of fluid flow.

There are several other beverage automation solutions either developed by hobbyists or are attempting to attain commercial production through the online crowdfunding platform, Kickstarter. Such systems include the Party Robotics' *Bartendro,* the *Inebriator,* the *BoozeBot*, and *Monsieur.* These systems also offer complete hardware-software solutions, but none of these systems are cheap, modular, or scalable. They all require the purchase of a specific-sized system, and if at a later point more modules are desired for the system, an entirely new system must be purchased with no way of using the existing system. For example, the *Monsieur* and the

Inebriator both only allow for a certain number of ingredients. The *BoozeBot* and *Bartendro* both have different models, but provide no way of scaling between models – to increase system size and functionality, a new system has to be purchased.

The CAMS is designed to use as few as one ingredient module and as many as 64 ingredient modules. The main controller is separate from the ingredient modules, and the ingredient modules only need connections to power and to the communications bus to add functionality to the system. Additionally, the expected cost of the CAMS is significantly lower than the existing systems per module. Across all systems (including CAMS), the greater number of modules purchased, the cheaper the effective module price, as cost of the main controller is distributed over many modules. The most expensive and least cost effective product is the *Monsieur*, which has a rate of $500 per module, and the most cost effective per module is the *BoozeBot*, which costs 62.5 British Pounds, or about $108. The CAMS prototype has a per module cost of $103.35. The cost of the largest *BoozeBot* system is more than the cost of an equally sized CAMS, even when taking into account the costs of the main module. A four-module CAMS prototype is expected to cost 57.5% cheaper than the same size BoozeBot, which is the smallest model.

Furthermore, the prototype cost of the CAMS was driven up significantly by the debug hardware used and buying each part individually. Some components, such as the FTDI serial chip, the associated capacitors, and USB connector (totaling about $5.50) are not needed for production model module boards, because the USB connection is used only for debugging. For a production board, the microcontroller would be mounted directly on to the board. Future development work with the CAMS past the first prototype can also likely lead to additional cost efficiencies in the design. It is likely that the code size and timer hardware requirements of the CAMS could be satisfied by a smaller, cheaper microcontroller, which would also remove cost of the system. Designing custom circuit boards for this system would eliminate the need for any breadboards currently used (which each cost $6). Lastly, purchasing all system components and circuit boards in bulk would drastically reduce cost per module, as the prototype cost was determined by individual part costs of every single electrical and mechanical component in the system.

Alternative Designs Considered

Over the course of this design project, several design alternatives were explored to best suit the needs of the CAMS. These design choices demonstrated tradeoffs in cost, system complexity, and required development time. The possible design alternatives were grouped into two categories: communications structure and beverage dispensing methods. For each of these categories, the design possibilities that were not chosen will be discussed while leading up to a description of the final design.

Communications Structure

Custom Static Network

The first network structure that was considered required a custom protocol and network topology. There were three types of network nodes in this topology – the central board, interface boards, and the beverage modules, and these nodes formed a quad-tree structure. The central board forms the root of the tree and thus has four ports, the interface boards each have one port to connect to its parents and have four ports for four branches, and the modules just have a single port. At initialization time of the system, the central board attempts to enumerate all of the modules that are present by querying each of the ports one by one to list the connected modules.

The result of this process is that the main board and the interface boards all know which beverage modules are located at each of their ports, but do not know the exact locations of the modules. For example, the main board might know that there are seven modules that branch off of port zero, but does not know their exact locations. If the main board wants to send a command to one of those seven modules, it sends a request to that port and allows the connected interface boards to handle the request. When an interface board gets a request for module dispensing, it repeats the process. If the desired module is located at one of the ports with just a leaf on it, then the interface board knows the exact location of the module. If the module is located at the port with the next interface board, it just sends a command to dispense an ingredient and allows the next level board to handle the request.

The key aspect of this design is that all of the ports are physically the same pin out and the protocol for requesting commands and responding to commands is the same regardless of whether or not the receiving node is an interface board or a module. This provides two main benefits. First, using the same message standards for sending and receiving from both boards and modules makes the communication software very easy. When a board gets or generates a request, it just needs to look up on which of the four ports the desired module is located on, and sends a request to one of four ports. No board needs to maintain exact location, nor does the protocol require a complicated routing method – the route is deterministic. The second benefit is that the design time of this protocol would be relatively short. All the ports are have same physical and electrical specifications regardless of which board or module they are on, or if they are an input or an output port. Additionally, the simple message types and message structure are relatively easy to program on a microcontroller.

However, there are many negative aspects of this design. Designing a circuit board takes considerable design time, and if there are three different designs needed (main, interface, and module), then the time to just have all the necessary boards produced would triple. The same board design could be used for all three boards, but that would require each board to have five ports and make the module and central boards larger than necessary, which would make the final design physically bulkier. Additionally, three different types of board designs would result in three completely different sets of software, regardless of the physical board design. Lastly and most importantly, developing a tree structure of nodes would require significant harnessing and a large number of boards. Even producing these boards in bulk would likely not reduce the cost enough to validate use of the design. Regardless, requiring an extra board for every multiple of 4 modules used past 5 will unnecessarily drive up the total system cost, which defies one of the principal design points of the CAMS.

CAN Bus Communications Structure

The main alternative design choice for the board-to-board communications infrastructure considered was the CAN (Controller Area Network) protocol for bus communication. The CAN standard is a bus-based protocol that was developed at Bosch in the 1980's and has been used primarily for automotive systems. CAN is a bus-based multi-master protocol similar to I^2C in that any node can send a message to any other node on the bus. Each node on the bus is assigned a unique identifier (determined at design time) which also denotes priority on the bus. A lower numbered identifier denotes higher priority. A CAN packet consists of many fields including up to 8 bytes of payload and a 15 bit CRC field for error detection. Additionally, CAN has a wide variety of error detection and information to convey to software. The CAN protocol is interesting because it requires two separate controllers to operate the protocol. One controller handles the logic of the protocol and communicates with the second controller, which handles the analog specifics of the CAN bus including performing processing and error detection tasks at the bit level.

There are two versions of the CAN protocol currently in use: CAN version 2A and 2B. The main difference between these two versions of the protocol is that version A only allows for 11 bit identifiers, whereas version B allows for 29 bit identifiers. There are some other changes in packet structure between these two protocols, but they exist only to enable the additional identifier length. Aside from these differences, the protocols operate in the same way. The CAN protocol is broken down into the physical layer, the transfer layer, and the application layer. The physical layer implements differential pair bit signaling, where a 0 is indicated by a high voltage on the bus (called a dominant bit) and a 1 is indicated by a low voltage (recessive bit). The wire harnessing in a CAN bus requires a characteristic impedance of 120 Ohms and requires an additional terminating resistor of 120 Ohms at either end of the bus. Data frames are specified at the transfer layer and consist of a start of frame bit, an identifier, a set of control bits (including payload size), a variable size payload, the CRC and delimiter, an ACK bit and delimiter, and an end-of-frame field. Bus arbitration, error detection, and acknowledgement are all handled at this layer. Lastly, the programmer is able to interface with the CAN-based system at the application layer. Transmission and reception of messages at the application level are handled through a set of message objects that contain all the necessary information about the content and status of a message. For example, a program can format a message object to be a data message designated for another node on the bus, the message can be queued to send as soon as it is ready, and an interrupt flag can be raised when the transmission of the message is successful or if there is an acknowledgement error on the message. The message object system of CAN controllers is complex, but it allows for rich interactions between the software and the lower level protocol.

Originally, it seemed as though the CAN bus was a good design choice for this project. The high noise immunity offered by the bus specifications was attractive for this application because the high power solenoid valves employed are likely to create noise on the bus. The CAN bus is able to detect errors through analog bit errors on the bus, packets formatting errors, CRC errors, and acknowledgement errors. This wide set of error signaling could be useful in debugging to help get the technology running initially as well as in production runtime software to increase system robustness. Additionally, Atmel sells microcontrollers with integrated CAN controllers to reduce the circuit board complexity and allow direct manipulation of message objects and the status of the CAN controller. Atmel also released several resources on how to program for the CAN line of microcontrollers and how to translate register names from other Atmel product lines to the CAN line. Additionally, because CAN packets contain a destination identifier, up to 8 bytes of payload, and built in acknowledgements, the communications software could be simpler because many basic communications tasks could be handled by the built in CAN controller.

A considerable amount of development time was put into enabling use of the CAN bus, but in the end it proved to not be the best technology for this project. Many of the features that initially made CAN attractive made progress slow and halting. First, enabling the use of a CAN microcontroller required designing a printed circuit board, testing it, producing a second revision, and then further testing before any meaningful communications software development could start. The board work provided some exciting work and experience developing a new circuit board but it took development time away from other aspects of the project. Next, the integrated CAN controller on the board actually made it more difficult to debug issues. Because of the way the CAN controller is built in the Atmel architecture, attempting to send any message required a complicated set of register commands which made it difficult to identify the faulty step in the process. The demonstration code offered by Atmel was very difficult to parse, and was also not helpful in trying to develop just a simple board to board communications demonstration. Additionally, I did not have any access to CAN bus analyzers or logic analyzers, which would have made development significantly easier.

The largest error I made with trying to use the CAN bus on this project was underestimating the difficulty in enabling basic use of the technology. Specifically, I was trying to simultaneously develop new software while also trying to develop the hardware that the software was running on. Developing both at once made it incredibly difficult to track down errors. I did not have any CAN bus communications working, which was due primarily to my inability to either guarantee that board design was correct and that I had a software issue, or to verify the software was working and that I maybe have had the wrong pin mappings on one of the chips on the board. Although the design did not work, the schematics of the board can be found in Figure C5. Despite these shortcomings, using a CAN bus would have still had its benefits. The final software version for the CAMS has a complicated software communication architecture, which must include acknowledgement, timeouts, and retries in software, which could have been handled by proper configuration of the CAN controller. However, I did not have the proper development tools, expertise, and (most importantly) time to enable the use of CAN technology for the CAMS, so a simple UART bus scheme was implemented. The selection rationale and implementation details for the UART bus can be found in the *Design* section of this report.

Mechanical Designs Considered

The first mechanical design option was a conveyor belt based system. The modules would be mounted above a conveyor belt and the tubes would go from the modules down to near the belt. When a cocktail was ordered, the bartender would place a glass on the belt, and the system would move the glass under the appropriate tube and then dispense the liquid. Once the beverage was ready, the bartender would grab the cocktail from off the belt. Mounting the modules directly above the belt would eliminate the possible need for compressed air and tubing. From a patron's perspective, it might also be fun to watch the cocktail be made as it travels down the belt.

However, this method is not very suitable for a modular system, and it is not expected to have a high throughput of cocktails. The conveyor belt system is not very modular because the belt assembly would have to be resized whenever additional modules were added. It would be possible to use a larger belt than is necessary to accommodate for a variable number of modules, but it would be wasteful to have a conveyor belt that could fit ten modules on it when only four modules are installed. Next, the conveyor belt would also have to be run very slowly to not spill any liquids. One of the goals of the CAMS is to aid bartenders during peak demand, and so the system would not be useful if it had to be run slowly to make sure the cocktails would not spill. The conveyor belt would require a calibration method to make sure the glass was aligned properly under the tubes.

Compressed Air

It was apparent that the conveyor belt system was not the optimal design choice, so additional design choices were explored. Another such choice was to use a system of compressed air and tubes to be able to dispense ingredients from modules that were physically distant. This design requires many mechanical components. First, the system would require a compressed air tank and a method of delivering the compressed air to the beverage modules. This method also has to allow air flow control from the beverage modules, as well as a way to control the dispensing of liquids out of the modules. None of these pose particularly difficult mechanical challenges. The regulator output of a compressed air tank is a National Pipe Thread (NPT) standard, so the tank can be easily interfaced with a variety of beverage safe plastic standard connectors. A tree of tube splitter components, such as Y and T splitters, could be used to deliver compressed air to tubes for each of the modules, and the fluid flow could be controller by a microcontroller by using a solenoid valve and appropriate driving circuit. Another solenoid valve could be used to dispense the liquid from the module itself. A minor amount of coordination software would be required by the module boards, but having one dedicated valve for liquid dispensing and one for compressed air would make the coordination relatively simple.

The valve and tubes solution had many advantages for the CAMS. First, this design was extremely modular. The connections coming from the compressed air tank could be resized at will using additional tube splitters, and an appropriate structure design would allow each module to be built and installed separately. Based on previous designs available through the Cornell ECE 4760 course page the solenoid valve control would use an opto-isolator circuit that was controlled by a microcontroller pin and powered from a power supply large enough to quickly switch on and off the valves (Land, 2013). This seemed like an obvious choice for the mechanical design of the CAMS.

There were two main issues that arose through development that resulted in only partial adoption of this plan. The first is that there was not sufficient development time to fully design and construct the system of valves, tubes, and splitters that would make up the core of the liquid and compressed air delivery system. Additionally, the cost of the compressed air tank, regulator, additional tubing, connectors, valves, and required circuitry would cause the cost of the CAMS to skyrocket. Including the cost of the valve for beverage dispensing, the mechanical components of each additional module would cost $66.98 (see Appendix D: Component Costs). Between this and the total BOM cost for the module boards, each module could cost up to $139.50, which is was well above the desired price point of each module.

Once I resolved not to use compressed air I wanted to determine how much liquid remained in the tube undispensed after the valve had closed. I found out that the brand of plastic tubing I used does not maintain an appreciable amount of undispensed liquid. There is liquid in the elbow connector but this is negligible compared to the cost of a compressed air based CAMS. The volume in the elbow connector was found to vary by amount of liquid dispensed. When 10mL was dispensed then about 4mL of liquid remained in the elbow connector. When 20mL was dispensed then only 1mL of liquid remained in the connector. This effect is likely due to the higher volume of liquid providing more weight and force to push all liquid out of the connector. It is important to note that this is not a constant loss per dispensing. The first time a module is used, some of the liquid will remain in the connector. After that, the trapped liquid will be forced out by the new dispensing of liquid and a few mL will be left behind after that, but no net liquid will be lost. Also, it was found that the geometry of the valve and tubing assembly affected the amount of retained volume. Given that volume losses would be negligible, the CAMS design does not address this remaining liquid.

Design

Electrical Design

The most important design principals of the CAMS are high modularity, low cost, and ease of use. The design revolves around a central processing board that handles all user requests and communicates with a up to 64 number of self-contained modules, which each control the dispensing of one liquid cocktail ingredient. Ice, garnishes, and any other non-liquid cocktail components must be handled by the bartender before delivery to a customer. The central board is a single circuit board with a microcontroller, but the modules consist of a circuit board, a solenoid valve and associated driving circuitry, a 700mL water bottle with a hole cut in the top, an ultrasonic sensor mounted on top, a set of tubes and connectors to connect the bottle to the valve and then the valve to outgoing tubing, an a wooden housing to hold all of these components. Please refer to Appendix B: Schematics and Board Layout for detailed schematics and layout of the boards and circuitry used in the CAMS.

Prototyping Circuit Board

The main and module boards both use the same prototyping circuit board, known as a breakout board. This board is an existing design for prototyping systems using an ATMEGA1284P microcontroller and peripheral hardware for a wide range of prototypes. The power input on the board takes unregulated 7V to 35V and converts it to a regulated 5V to power the remainder of the board. The ATMEGA1284P is seated in a DIP socket on the board that allows the chip to be swapped out if it breaks or is damaged. The microcontroller is programmed using an on-board ISP header. An external 16MHz crystal provides the system clock through the XTAL pins on the microcontroller. The microcontroller communicates with other devices either through the single row pin header on the side of the board (for plugging into a breadboard) or through a USB connection enabled by the serial communication chip and female USB-B connector. A green LED on the board, useful for debugging, can be connected to pin D.2 on the microcontroller through a two pin male pin header and a small jumper. Similarly, the pins that ordinarily drive the communication chip can be disconnected by removing jumpers on similar headers. Serial communication jumpers were used on the CAMS for development purposes and for PC communication, but the jumper for the green LED was removed. On both the main and module boards, the row of pin headers is plugged directly into a breadboard for easily interfacing with other components. Please refer to Appendix B: Schematics and Board Layout for references to the prototyping board.

UART Bus

ATMEGA1284P microcontrollers have two separate UART channels. On the main microcontroller, UART0 is used to communicate with the PC to interact with the user, while UART1 is used for the UART bus to communicate with all the module boards. On the module boards, UART1 is used for the UART bus, and UART0 was left for PC communications for debugging purposes but is unused during normal operation. The PC connection to the microcontroller is made possible by the serial communication chip, an FTDI FT232RL chip, which converts TTL level UART communication from the microcontroller to the differential pair used in USB connections and vice versa. A female USB type B connector is connected to the chip and easily connects the UART0 on the microcontroller to a PC.

Given the hardware on the boards, PC to microcontroller communication is relatively simple. From the microcontroller message transmission, the standard fprintf function can be configured to output formatted strings on UART0. When the microcontroller receives a message, a flag indicates there is a byte waiting in the receive buffer, at which point either an ISR can be entered or the receive buffer can just be read by the main software. On the PC side, the USB connection is interpreted as a serial communications port (COM port), and any program that can handle serial I/O can bind to this COM port and interact with the microcontroller. PuTTY, a free Microsoft Windows serial terminal program, was used to communicate with the main board. A baud rate of 9600 was selected for the PC communications. The speed could have been increased, but slower baud is more reliable and the channel was not nearly saturated, so it was not necessary.

The UART1 channel on the microcontrollers was used to allow bus-based UART communication. The most efficient way for the main module to issue commands to anywhere from 1 to 64 modules was to use a bus. Because the current CAMS system is a prototype, the bus does not have a proper harness structure. The bus is implemented by using solid core wires and resistors plugged into breadboards and connected directly to the UART1 microcontroller pins. In the unlikely event that two microcontrollers are both trying to drive the bus at the same time, each transmitter pin has a 1kΩ resistor between the pin and the bus, so there is ample protection for both the transmitters and the receiver. It is also possible that, at some points in time, no module boards are driving the bus, so the receive line going to the main board is weakly pulled to 5V through a 10kΩ resistor. During development before the resistor was used, when there were no modules driving this line, any transmissions to the modules on the bus caused reflections on the receive line. This was likely due to the undriven receive line coupling to the transmit line, which caused anything that was transmitted to be reflected on the receiver. Adding this resistor eliminated this problem. For further noise prevention, the ground lines of each board are connected to avoid an issue with inconsistent grounds.

Long solid core wires are also not ideal for harnessing the CAMS. They are brittle and have poor electrical qualities for long lengths of wire. However, they were sufficient for the CAMS prototype. A low baud rate was used on the UART bus in case that the long wires had significant delay. The main board's transmit pin is connected to each of the modules individually, and each of the module board's transmitters are all connected to the main board's receiver. This is not scalable because attempting to plug in 64 wires (the maximum number of modules) into a single microcontroller pin would be physically difficult to implement and it is likely that the pin does not have the driving strength for the fan out of 64 microcontroller pins. However, not enough hardware was present in the development laboratory to allow for the testing of large scale buses.

Solenoid Valve Driving Circuit

Each module dispensed a single liquid ingredient from a bottle for storage to a cocktail glass or other container for consumption. The flow of the ingredient from the bottle to the glass was regulated by a solenoid valve located between the bottle and the tubing that carried the ingredient to the glass. This solenoid valve is driven by a small set of circuitry uses a different power source than the regular microcontroller power. This separate power source is necessary because the solenoid valve used is a heavy brass solenoid valve that takes a minimum of 6V and 1.6A, which is more voltage and current than can be provided by the regulator that powers the microcontroller.

The microcontroller is able to control the opening and closing of the valve even through turning off and on and opto-isolator which in turn allows for either no voltage or 6V to be dropped across the valve. A schematic of the opto-isolator circuit can be found in the bottom of the module board schematic, found in Figure C3 of Appendix C. An opto-isolator works by having an LED very near to a phototransistor, and isolating both from any other light source. This allows a microcontroller to control the voltage of the phototransistor in an analog manner, usually through pulse width modulation. The module board microcontroller is connected through a current limiting resistor to the anode of the LED inside the opto-isolator. The CAMS only needs the valve to be either turned "on" or "off", so instead of putting an analog voltage on the LED, the LED is either provided 5V or 0V. When the LED is not turned on the 10kΩ is weakly pulling the gate of the NFET to ground which causes it to not conduct. No voltage is being dropped across the solenoid, so the solenoid is turned off. When the LED is turned on, the phototransistor is conducting, which strongly pulls the gate of the NFET to valve power (minus threshold voltage of the phototransistor). This turns on the NFET, which causes the negative terminal of the solenoid to effectively be at ground, which results in a 6V drop across the solenoid, turning it on. There is a diode connected to the terminals of the solenoid used to prevent inductive spikes when the solenoid first turns on or off. There is also a capacitor located on the terminals of the solenoid to prevent high frequency noise spikes.

Mechanical Design

Due to limited development time, a complete mechanical design and structure had not been completed. Some preliminary mechanical designs had been drafted but were not implemented. Instead, a prototype was built using components and materials already available in the laboratory. The system design is not complex, but it still properly houses the three main components of the mechanical system: the container to hold the module ingredient, the solenoid valve for dispensing the ingredient, and the structure to hold all of the components. A picture of a module structure can be found in Figure B1, and a model from the mechanical CAD program, Solidworks, can be found in Figure B2.

The ingredient is contained within a 700mL Poland Spring water bottle. These plastic bottles were used because they are easy to obtain, they are cheap, and the bottle itself is not heavy which helps reduce the load requirements of the mechanical structure. Fortunately, the threaded cap size of the plastic bottle closely resembles the dimensions of a ¾" garden hose, so a standard connector can be used. The fit between the connector and the bottle is not perfect, but it is suitable for a prototype. The remainder of the valve, plastic bottle, and beverage-grade plastic tubing are connected together through a set of standard connectors. The plastic bottle is connected to a Brass ¾" female garden hose to ½" plastic tubing connector (called hose barb). The tube then extends 2.75" to an HDPE ½" hose barb to ½" male national pipe thread (NPT) converter which is screwed into the solenoid valve. The other side of the solenoid valve is connected to another hose barb to NPT connector, which is connected to 2" of tubing. The tubing is also connected to a polypropylene 90^{o} elbow with a hose barb connector on both ports. Tubing then connects this elbow to the dispensing area near the cocktail glass. Many of the connectors are sealed with Teflon pipe thread tape, to prevent leaking between threads. Additionally, the inside of the female garden hose to ½" hose barb connector is coated in a thin layer of silicon grease to prevent leaking and has an extra gasket to improve the seal between the connector and the ingredient bottle.

Accurately dispensing a certain volume of liquid out of the bottle requires holding the valve open for a specific duration, which is based on the flow rate. The flow rate out of the bottle is not constant - it is governed by Bernoulli's Principle. An ultrasonic sensor seated in a hole cut out at the top of the bottle can measure the height of the liquid in the bottle, which can in turn be used to determine flow rate. However, the flow rate can be difficult to calculate because the shape of the bottle makes modeling volume as a function of height difficult. Even if an accurate volume model could be determined and evaluate, determining flow rate would also be difficult. The exact material properties and physical dimensions of the connectors and valve are not known so the pressure drops (and thus the flow rate) in the system are difficult to model.

Instead, a scheme is used to directly map liquid height in the module to a duration of time to hold the valve open. This duration is calibrated to dispensing one "part" (22mL) of volume of the ingredient. The valve durations were determined through a set of experiments. Due to limited accuracy of laboratory equipment, the mapping from liquid height to valve duration only has a granularity of one half centimeter. Details on how the measurement and mapping are calculated can be found in the *Ultrasonic Sensor and Valve Duration* section of this report. The precision of the valve duration is discussed in the Results section of this design report.

The structure of the module is a large rectangular wooden box which stands upright on its smallest side, and the front face is open. This box also contains a set of sixteen small wooden guides along the long sides of the box to allow any configuration of shelves to be used. The shelves used in the prototype are made .5" Styrofoam, which was used due to availability in the laboratory. The valve assembly rests on the first shelf which sits on the 1st shelf slot, which is located 2.125" above the counter. The top shelf is 12" above the bottom shelf and is used to hold the ingredient bottle in place. This shelf has a rectangular hole cut out of the middle that is 2.5" across by 4" deep. The curve of the ingredient bottle rests in place on top of this shelf while the valve rests slightly above the bottom shelf, supported by a tube connector and held upright by the ingredient bottle assembly above. A model of this module design can be found in Figure B2.

This prototype is not the optimal structure for the CAMS, but it does have some key design points that a proper design should have. The current structure has an adequate way to hold the weight of the ingredient bottle and the valve, which constitute the majority of mass of the module and provide the greatest load on the structure. Additionally, it would be incredibly difficult to knock over or disturb this prototype, which makes it safer for a hectic bar environment. However, there are three major design flaws which need to be addressed. The first is that the box structure is not specifically designed for the CAMS so it is large and unwieldy. Constructing a fifteen module CAMS with this structure would require nearly seventeen feet of counter space. The boxes are easily stackable but it would be difficult to restock the ingredients inside. A good design would be much thinner and lighter so that many modules does not take up significant counter space while also being stackable to further consolidate linear space usage.

A better-designed structure would also have an easier way to attach the necessary wires. Either the structure would have a hole in the back or sides to string wires through, or would have connectors mounted on the side to easily plug in the module board to valve/sensor electronics. Depending on the connectors used, the wire scheme used to connect the microcontroller to the module electronics may also require updates to fit the chosen connector. A properly designed CAMS would have a better container system. The hole at the top of the plastic bottle is suitable for equalizing the pressure as liquid flows out of the container, but it does not ensure that the sensor is pointed directly downwards in the container. Tilting the sensor can impact the accuracy of results. Lastly, the method of refilling an empty module requires that the bottle be pulled out of the module and then filled up at the top. Pulling out the bottle also requires carefully removing the valve assembly. A better CAMS design would not require removing the internal assembly to restock the ingredient. These design improvements were not implemented due to development time constraints. Given additional time, these changes would have been pursued to improve the robustness, usability, and space usage of the CAMS.

Software Design

All of the CAMS software is written in C, developed on Atmel's AVR Studio, and run on the ATMEGA1284P microcontrollers either on the main board or on a module board. The software that runs on the main and module boards is slightly different. The main board software runs the code to initialize the system, runs the user interface, and coordinates communication on the UART bus to the modules using a protocol that leverages UART capabilities on the microcontrollers. The module board software sends and receives messages on the UART bus as well as reads the ultrasonic sensor to control the solenoid valve actuation. For a full software listing, please contact cig23@cornell.edu.

Data Structures

In order to efficiently store and reference cocktails that the CAMS can make, a complex data structure was created to store the cocktail information in a memory-efficient manner while also enabling fast lookup functions for the library of cocktails. At the lowest level of the structure, drinkEntryBits is a collection of two-bit fields in a C struct. Each two-bit field corresponds to a particular ingredient that the CAMS knows how to use and the number in the field represents how many parts of that ingredient go in a cocktail. For example, if a cocktail had zeros's in all of its drinkEntryBits entries except a one in the "Rum" field and a two in the "Coke" field then that cocktail would call for one parts Rum to two parts Coke. The drinkEntryBits structure currently only has seven entries defined, although this can be easily expanded to incorporate new drinks by adding additional entries in the struct definition and in all places where the struct is called.

The drinkEntryBits struct is embedded in a C union called drinkEntryUnion with an unsigned 16-bit integer. The purpose of this union is to allow setting and clearing of all the fields in the entry at once without having to enumerate each individual entry. A 16-bit integer is used because at this time there are only seven 2-bit entries, but this can be easily expanded to any integer size to accommodate the size of drinkEntryBits. A drinkEntryUnion is embedded in a drinkEntry struct, which is the struct used to store cocktail entries in a library. There is a string field in a drinkEntry struct that holds the name of the cocktail and a drinkBitUnion. The syntax for referencing an individual bit field from the highest level structure is cumbersome, as a field within drinkEntryBits within drinkEntryUnion within drinkEntry must be referenced. However, the algorithmic performance on this data structure is fast. The algorithmic performance is discussed in the *Main Board Software – Initialization* section of this report. Memory space requirements increase logarithmically instead of linearly because every power of 2 number of bit fields requires one larger size unsigned integer in drinkBitUnion to be able to access all fields at once. Using an array of unsigned 8-bit integers instead of bit fields would make each library entry take up a significant amount of space. 16 allowable ingredients would only require 4 bytes of drinkEntryBits space, but using an array would take up 16 bytes. This may not seem significant for small numbers of drinks on the menu, but when scaling up to hundreds of drinks, the library can take up a large portion of available memory. Also, if a smaller microcontroller is used to save cost (see the *Reducing Electronics Cost* section of this design report), then having a smaller memory requirement would make it easier to find a cheaper microcontroller.

Main Board Software

Initialization

Initializing the CAMS is done in three steps. First, low level setup code runs to clear and turn off the watchdog timer, configure the on-chip UART hardware for eight-bit UART at 9600 baud. Hardware timer 0 is set up to provide a 1 millisecond time base used in the communication protocol. Next, the global library of drinks is populated from the file drinkLibrary.h by calling populateLibrary(). Each entry in the library is one cocktail that the CAMS can prepare, which is encoded by a drinkEntry struct. This library does not represent the cocktails that can be made at that time – that cannot be known until the contents of each module is determined in the next step.

Next, the function populateStock() is called to allow the user to set up the modules currently in use in the CAMS. The user is prompted to indicate what ingredients are located in each module. The CAMS sends the prompt “What is in slot 0?” to the serial terminal, and the user enters a number corresponding to an ingredient number, and then presses enter. This method requires that the user setting up the system knows the mapping from ingredient names to ingredient numbers. The table of ingredient numbers and corresponding names can be found in Table A1. Each successful ingredient entry increases a variable indicating the number of valid modules present and updates the beverageModules array indexed by ingredient number that indicates which module each ingredient is located in. After each ingredient is entered, the CAMS increases the module number on the prompt, populating the internal module list one by one. If ingredient number 99 is entered, that indicates that all modules have been listed and to move on to the next step in the process. If 100 is entered, that indicates the user is requesting a software reset. Software reset occurs by turning on the watchdog timer to the fastest setting and waiting in a while(1) loop until the timer expires. This forces a software reset, and the microcontroller with clear the watchdog timer when it reinitializes. After the stock is populated, printStock() is called to print the results of populateStock() in a readable format. This serves no functional purpose – it is just included for diagnostics.

After the modules have been configured, the CAMS determines, given what cocktails it knows how to make, which cocktails can be made with the given stock of ingredients. This is done through the checkTotalAvailability() function, which leverages the allBits field in a drinkEntryUnion (which is a concatenation of all of the bit fields combined) to quickly determines whether or not a cocktail can be made. First, for each drink in the global library, the allBits field is logically ANDed with the allBits field of the module stock that was populated in the previous step. If the result is non-zero, that means there are some bit fields in the current stock that are non-zero that are also non-zero in the same place in the library entry. This means that there is at least one ingredient in the library entry that the current modules have in stock. Next, this resulted is XORed with the library entry, and checked if the result is zero. If only some of the bit fields from the library entry matched the stock that means that there were some ingredients from the library entry that were present in the populated stock. If that is true, then the XOR will be nonzero because the ingredient fields that were not present in the current stock will be ones while the previous AND with the current stock would have resulted in zeroes in those bit positions. However, if the XOR result is zero, that means that there are no ingredients that were present in the library entry that were not present in the populated stock, so the CAMS can make the cocktail indicated by that library entry.

If this algorithm determines that the cocktail can be made, it populates the cocktail in the list of available cocktails. Instead of performing a deep copy of all the fields of the drinkEntry structure and copying them to a new array, the list of available cocktails (called the menu) is just an integer. This integer indexes into the global library and pulls ingredient information from this library. This algorithm is complex but it provides many benefits. First, using the allBits field in the drinkEntryUnion within the drinkEntry structure requires that only three operations are needed to determine if a cocktail can be made, regardless of the number of possible ingredients there may be. If each ingredient is checked individually, via a method such as iterating through each ingredient and checking if the library entry needs the ingredient and the ingredient is in stock, then the lookup time for each module would scale linearly with the number of possible ingredients. Similarly, performing a deep copy from one array of drinkEntry structures to another would also scale linearly with the number of ingredients. The time saved may not matter for small scale systems, but when a CAMS is trying to serve hundreds of patrons in a bar, then saving time only having to perform 3 operations total instead of 3 per ingredient plus the time to perform a memcpy may be relevant. Unfortunately, it is not possible to determine the time savings without programming the alternate method and creating a library of hundreds of drinks that supports many ingredients.

User Interface

Once the availability of cocktails is efficiently checked, the menu of available cocktails is printed to the serial terminal. The available drinks are displayed with the name and ingredients in each drink, and by menu item number. That number is bounds checked to ensure that it is less than the number of cocktails on the menu. Bounds checking is performed on the item number, and if it passes, then that number is used to index into the menu array that stores indexes into the global library. For example, if the cocktail "Gin and Tonic" is labeled menu item number 46 but is overall drink number 238, then a user attempting to order a Gin and Tonic would enter 46 on the terminal. Element 46 in the menu would be accessed, and that value would be 238. 238 would then be used to index into the global list of all cocktails, and that would indicate the ingredients in a Gin and Tonic, and the cocktail name and ingredients would be printed to the serial terminal for verification. The CAMS would then issue commands to the appropriate modules, and respond with a success message when the dispense commands have been acknowledged by all relevant modules. Similar to the module population, entering cocktail 100 forces a software reset of the CAMS through a watchdog timer overflow.

Communication Stack

The CAMS implements a basic communication stack, with UART at the physical level, to handle transmission of messages and receipt of acknowledges on the bus. This is a three-level stack, which means there are three levels of function calls before data is actually loaded into transmit buffers. This protocol is not knowingly modeled after any existing protocol – it was created to avoid having one monolithic communication function with a complicated error signaling scheme. Failures at different levels of the protocol indicate different messages to help diagnose issues.

When a cocktail is ordered, the orderDrink() function is called with the global drink number as the argument. This function iterates through the allBits field of the drinkEntryUnion to determine if the cocktail needs that ingredient. This is done by saving the allBits field into a variable called selectedDrinkBits, forming another variable called numParts that is an AND of the selectedDrinkBits and the number 3. If the AND is zero, that means that ingredient number 0 is not used in this cocktail. The ingredient number is incremented by 1 and selectedDrinkBits is right shifted by 2, to move to the next ingredient. If the AND is non-zero, that means that the cocktail needs the current ingredient, so a message should be formed to the relevant module. Again, the ingredient number is incremented and selectedDrinkBits is shifted right by 2.

This process is repeated for all of the ingredients in the ingredient list. This bit math could be avoided by spelling out the process directly for each ingredient, but this would cause the code size to inflate because there would be several lines of code for each module, instead of a slightly larger number of lines for all modules. Also, if each module was listed directly, adding any ingredients to the list would cause further code size increase.

For each ingredient that is needed by the cocktail, a command is attempted to be sent to the module to dispense the ingredient. This is achieved by calling the sendCommand function with arguments as the module number (as determined through the beverageModules array) and the number of parts, indicated by the numParts variable. The sendCommand function returns either true or false – if it returns false then sendCommand prints that the request to a specific module failed, which module failed, and which ingredient was not present. This information is not relevant to the user ordering the drink as they only care that their drink was not prepared. However, the bartender can see which module wasn't functioning properly and inspect it, and by knowing which ingredients weren't dispensed, he or she can manually add in the remaining ingredients.

The sendCommand function calls the function to send the specified data over the bus, but also implements retries to allow for noise and disturbance on the bus. The bus driving function, uartSendByte, is first called to send the specified number of parts to the specified module number. If the sending fails, it tries twice more before returning false. Each time sending the data fails, the message "No response" is printed to the terminal, and sending is retried. If the data is unsuccessfully sent over the bus three times, the sendCommand function returns false. If any of the three attempts succeeds, then the message "transaction complete" is printed to the terminal. These messages help indicate if there were any failures on the bus before success, or if the sending process never completes.

At the lowest level of the communication stack is the process of attempting to send an individual message to a module. A message is successfully sent if a short messaging handshake is completed. First, the uartSendByte function records how many seconds since system initialization have elapsed, as indicated by the variable from the millisecond time base ISR. Next, it waits until the UART1 transmit buffer is empty, and then it sends a byte, with a value of 0x30 (or 48) plus the module number, over the bus. The increment is added to the module number of avoid ambiguity between module addresses and commands. After the address is transmitted, the function then transitions to a state of waiting for an address acknowledge. If a message is received that is not an address acknowledge from the correct module, the function returns false. If the address acknowledge is correct, then the function waits until the transmit buffer is empty again, and then the number of parts to dispense is sent over the bus. One part of liquid refers to half-shots, or approximately 22mL.The function that transitions to a state of waiting for the command acknowledge. If the command acknowledge byte, which has value 128, is received, then the handshake is complete, the sending of the message was a success, and the function returns true. However, if the whole handshake takes more than 50 milliseconds before finishing (as indicated by the start time and the global time) then the function returns false regardless of the components of the handshake that have been completed.

Regardless of if the cocktail order was a success, once the orderDrink function is exited, the main board prompts the user to enter another menu item number to order and repeats the process. An example of a full drink ordering process starting from system initialization to successful dispensing of multiple ingredients can be found in Figure A1 in Appendix A.

Module Board Software

Initialization and UART ISR

Initialization of the module board proceeds in a similar manner to the main board initialization. First, the pins on port D are configured to allow proper transmission and reception on UART1. Next, the address of this module is copied in to a variable. Unfortunately there is currently no easy way to assign module numbers without slightly changing the code, recompiling, and reprogramming for each module. On a production design, the initialization code would include some way to indicate the module number, such as using pull up and pull down resistors on a port or using a keypad. After the module number is recorded, then the UART is initialized by calling uart_init. The UART is configured for 9600 baud rate and the receive ISR is enabled. The receive ISR will execute every time a byte is successfully received on UART1. The ISR copies the contents of the receive buffer into a variable used by the main software and sets the flag prtintByte1 high, which is also used by the software. The process of reading data out of the receive buffer clears the interrupt flag so multiple ISRs do not trigger from a single byte of data.

Communication

The main loop of the module board simply calls a communication function which executes the module side of the handshake. Inside this function, the printByte1 flag is polled. If the flag is high that means there is new data to evaluation in the readData1 variable. In that case the flag is cleared, and the data is checked against the stored address. If the data does not match the stored address then the module board turns off the UART transmitter to prevent it from driving the bus when it was not the module being addressed. If the data matches the stored address, the software first enables the UART1 transmitter, waits for the transmit buffer to be empty, and acknowledges the address by sending the address back on the bus. If the next byte received matches the address again, another acknowledge is sent. This allows the handshake to be reset without timeouts, retries, or signaling to the main board when a reset occurs. If the address doesn't match on a byte directly following a correct address, it is assumed that this is a command byte, so the number of parts of the ingredient to dispense is stripped out from the message by taking the lower order 6 bits and subtracting 0x20.

Once the number of parts has been successfully stripped out, an acknowledge message of the command is sent. The command acknowledge contains just the number 128, and once this is sent then the UART transmitter is turned off to prevent the module board from driving the bus unnecessarily. This concludes the handshake on the module board side. However, once the command has been acknowledged, the module board must open the solenoid valve for the appropriate amount of time to dispense the specified liquid. This is done by calling valveDuration, which will be discussed in the following paragraphs. Once valveDuration terminates, the module board communication function returns to a state where it waits for an address message from the main board.

Ultrasonic Sensor and Valve Control

Once a module receives a command to dispense a certain volume of liquid, then the module must then open a solenoid valve to dispense the liquid out of the plastic bottle. The flow rate of the ingredient out of the bottle is a function of the amount of liquid in the bottle, so an expression relating liquid volume and flow rate must be reached. However, the exact expression for determining flow rate as a function of volume is difficult for several reasons. First, it is difficult to determine the volume in the bottle because the shape of the bottle is not easily modeled. Next, the valve combined with the tubes and connectors makes the pressure drop calculations difficult due to my inexperience in fluid mechanics. A poor physical model of the bottle combined with a poor model of the pressure drops in the valve, tubes, and connectors would lead to a very inaccurate dispensing of ingredient from the bottle.

No analytical solution could be reached, so the problem of determining how long to hold the valve open for was empirically determined. Ultimately, the goal was to find a way to determine the volume in the bottle, which could then be used to find the flow rate, which in turn gives a duration of time to hold the solenoid valve open for the dispense the correct amount of liquid. However, since no accurate expressions could be found for any of the intermediate steps, an experimental mapping from the height of the liquid in the bottle to duration of time to hold open the valve to dispense a particular volume of liquid was developed. This was a very slow process, but a discussion of the experiment is found in the Results section of this design report. The mapping of liquid height to amount of time to hold the valve open was used instead of mapping height to flow rate because determining how long to hold the valve open as a function of flow rate is difficult. As the valve is held open, the height of the liquid changes, and so either the height is going to have to be dynamically calculated or reread from the sensor.

Both of these schemes would make the dispensing process slow, so a mapping of liquid height to duration to hold the valve open is used. Determining liquid height is done by an ultrasonic sensor mounted on top of the ingredient bottle. The ultrasonic sensor has a four-pin interface – it has 5V and ground connections, a trigger pin, and an echo pin. To save power, the ultrasonic sensor is not constantly sending out ultrasonic pulses and measuring the distance. It only performs a measurement when a 10 microsecond pulse is sent to the trigger pin. At that point, the sensor sends out a series of ultrasonic pulses and performs calculations to get a distance measurement.

The calculated distance is output from the sensor to the object on the echo pin in the form of a signal pulse ranging from a few hundreds of microseconds to tens of milliseconds. The sensor datasheet provides a formula for determining measured distance from the output pulse width, but an experiment was run to determine the validity of this model. The datasheet for the ultrasonic sensor can be found in Appendix G. The experiment, the results of which are discussed in the *Results* section of this design report, indicated that the model was not quite correct. The data had similar functional form to the model, so an experimentally determined model was created to be able to more accurately estimate distance based on the ultrasonic pulse width. A lookup table would have been the most accurate way to estimate distance based on pulse width, but the experimental setup in the laboratory did not allow for accurately determining a distance for each possible pulse width. Across the length of the plastic bottle, the experimental model was within ±3% accuracy. This model was computationally simple so it did not hinder performance.

Interfacing with the ultrasonic sensor presented an interesting challenge. The hardware timers on the microcontroller have the feature of being able to detect a logic level change on a pin and immediately stop the timer, called the Interrupt Capture Unit. This feature is useful for determining pulse width, but only when the start time of the pulse is known. To initiate the sensor reading, the microcontroller needs to send a pulse of at least 10 microseconds (called the trigger pulse) to the sensor's trigger pin. The sensor sends a series of ultrasonic pulses to determine distance, then at some point later, the output (also known as the echo) pulse starts. The datasheet does not provide any references for knowing when the echo pulse starts with respect to the trigger pulse. Either the Interrupt Capture Unit would have to be configured to trigger on pulse edges of the echoed pulse, or the interrupt would only be triggered on the falling edge and the sensor would have to be recalibrated.

Instead, Timer 1 was used in conjunction with an edge transition pin interrupt on pin C2, separate from the timer interrupts. Whenever a command is received to dispense an ingredient, and sensor reading was initiated by putting an 11 microsecond pulse on pin C3 by using the _delay_us function from util/delay.h. This is not as accurate as using a hardware timer because the delay function does not account for time to execute ISRs, but it was simpler than setting up a microsecond time base ISR. The sensor is triggered by anything larger than a 10 microsecond pulse, so attempting to send an 11 microsecond pulse that is elongated by an ISR is acceptable.

After the microsecond delay is finished, an edge-triggered interrupt on pin C2 is enabled and the software waits until pin C2 goes high and then low again, indicating a full pulse has occurred. On the rising edge of the echo pulse, the edge triggered ISR is entered and Timer 1 is cleared and started with a prescalar of eight, giving a 0.5 microsecond time base. At the falling edge of the pin interrupt, Timer 1 is stopped to get the time between when the start pulse was sent and when the echo was received. The resulting time is then converted to twice the distance in centimeters and rounded to get an effective rounding to the nearest 0.5 centimeters. This result is then used to index into a table of valve durations.

As discussed, a mapping from liquid height to number of milliseconds to hold the valve open was experimentally determined. Ultrasonic readings are rounded to the nearest 0.5 centimeters, so the table of valve durations must has an entry for each 0.5 centimeter. The ultrasonic sensor data sheet claims the sensor is accurate to 0.3 centimeters but it is safer to not approach the maximum accuracy bounds. The discussion of the valve experiments can be found in the Results section of this report. The end result of these experiments is that for each 0.5 centimeters increment, the CAMS module microcontroller knows how long to open the solenoid valve for in order to dispense one unit (22mL) of liquid.

Despite the experimentation and data analysis that was part of the design process, the final process for dispensing an ingredient is very simple. First, a reading of the ultrasonic sensor is initiated and read. Next, the value from the sensor reading is converted to a distance in centimeters, which is then rounded to the nearest .5 centimeter. This is stored as a number twice as big because indices of an array must be integers. This number is combined with the number of parts (units of volume) to dispense and indexed into a table to determine how long to hold the valve open for. Lastly, the microcontroller drives pin C0, the pin connected to the opto-isolator (discussed previously in *Electrical Design*) high for the specified number of milliseconds to open the valve and start dispensing liquid. Hardware timer 0 is used as a millisecond time base and after the specified amount of time, the pin is driven low to shut off the valve. If a cocktail requires multiple parts of the same component, the module board waits for 500 milliseconds for the valve to settle and repeats this process for each part of liquid to dispense.

Results

Sensor Experiments

As discussed in the Design section of this report, calibration of the ultrasonic sensor and ingredient dispensing required calibration to yield accurate data. For the ultrasonic sensor, I ran an experiment to determine the accuracy of the sensor and to check the accuracy of the model on the datasheet. I set up a microcontroller to initiate a sensor reading every 500ms and an oscilloscope was connected to the echo pin on the ultrasonic sensor so that the pulse width could be accurately determined. A tape measure was taped to the laboratory bench and the breadboard with the microcontroller and ultrasonic sensor was clamped to the table. A rectangular piece of aluminum was placed on the tape measure to attain a flat surface facing the ultrasonic sensor while being able to read the tape measure.

Basic initial experimentation indicated that the sensor was not accurate at distances closer than 4cm, despite the data sheet indicating that the sensor was accurate to 2cm. Between 4cm and 2cm, the sensor would jump between readings – holding the aluminum in one place would sometimes make the oscilloscope read one pulse width but would randomly jump between that reading and a few microseconds more. Instead of trying to handle this sporadic behavior in software, it is more reliable to impose a maximum fill line restriction on the bottle to ensure that the liquid is never closer than 4cm from the ultrasonic sensor.

After the lower bound on distance-to-sensor was determined, the ultrasonic sensor was tested over a range of 20cm, which was chosen because the length of the bottle from end to end (including the hose connector) was 24cm. Distance increments of one-half centimeter were used, so 40 data points were collected in total. The results of this experiment can be found in Table E1. The datasheet for the ultrasonic sensor provided a formula for determining measurement distance as a function, which is given in equation 1:

$$Distance\ (cm) = \frac{time\ (us)}{58} \tag{1}$$

However, my results found that the datasheet model was not quite accurate. The slope of the line of data points was correct, but at certain points the experimental data had slightly different constant offsets from the model. Instead of relying on the datasheet model, I developed a piecewise experimental model. At all points in the experimental model, the slope of the experimental model was the same as the datasheet model – an additional 58 microseconds per centimeter. However, between 4cm and 7cm the plot was shifted down by 40 microseconds, between 8cm and 21cm the plot was shifted down by 20 microseconds, and between 21.5cm and 24cm the plot was shifted down by 55 microseconds. The data for the experimental model can be found in Table E1, and a visual representation is found in Figure E1. As indicated in Table E2, the between percent error between the experimental data and the experimental model was very low. The experimentally determined model was within ±3% accuracy. The mean squared error was .021%, and the standard deviation was 1.33%. The piecewise linear model was easy to program on the microcontroller using a few conditional statements and floating point operations.

UART Bus Reliability

The design of the CAMS requires that modules disconnect themselves from transmitting on the bus when not in use. According to the ATEMGA1284P datasheet, the UART transmitter can be turned on and off at will. The datasheet does not specify after how many cycles the transmitter is enabled that it is available to send data, so the software polls the transmit buffer empty flag (UDRE1 in the UCSR1A register) to wait until the transmitter is ready. When the bus is weakly pulled to ground through a large resistor, the first byte that is sent after a UART transmitter is enabled is always corrupted. The root cause of the corrupt byte issue had not been determined, but it is expected that when the bus goes from passively being pulled low to being driven high when the UART transmitter turns on, then the receiver confuses this transition with a transition associated with a UART byte in a message.

This issue is resolved by weakly pulling the bus to 5V so that when the transmitter is turned on there are no voltage level transitions. Pulling the bus to 5V eliminated all reliability issues – all transactions were successful without any retries. However, the communication stack was able to handle the errors with pulling the bus to ground with low level software. If a handshake with a module fails, then the main board retries twice more. Testing the bus reliability while pulling to ground revealed that the UART bus timing is able to correct itself without any software intervention. The byte transmission is successful on either the second or the third try. A successful full drink ordering process on a system where the bus is pulled to ground can be found in Figure A2.

Valve Calibration Tests

As discussed in the Mechanical Design section of this report, a direct mapping from liquid height in the container to duration to hold the valve open was created instead of a mathematical model of the flow rate. Determining this mapping was done in two steps. First, a characterization experiment was run to obtain a rough estimate on the flow rate at each height. The results of this experiment were analyzed to estimate a duration to hold the valve open for at each liquid height. Another experiment was run in which the valves were held open for the estimated amount of time and the number of mL of water dispensed was recorded. Based on the data from the first two experiments, another estimate was determined and the second experiment was repeated. This process continued for 1 additional run until the correct valve duration was determined. The results of the characterization experiment can be found in Table E3, and the results of the estimation experiments can be found in Tables E4. The final mapping of liquid height to valve duration can be found in Table E4.

The first experiment run was the characterization experiment. In this set up, the ingredient bottle was filled to 4cm below the top of the bottle with water, and the valve was periodically held open for a constant amount of time and the volume of liquid, measured in mL was measured. On each measurement, the starting height of the liquid was measured and the liquid was dispensed into a graduated cylinder. This experiment was run three times, and the results can be found in Table E4. Originally, I intended to average over three runs and use that average as a basis for my estimation. However, the first two data sets were inconsistent while the third data set was self-consistent and consistent with the expected results from Bernoulli's Principle that a taller height of volume gives a higher flow rate at constant pressure. In each of the data sets, a few points had to be thrown out due to fidelity of data. Either some of the water had spilled and so an accurate volume could not be determined, or the distance was not measured properly prior to dispensing. Regardless of these omitted data points, plotting the first two data sets with a linear trend gives very poor correlation with the data whereas plotting a linear trend line from the third data gives strong correlation.

Instead of taking an average over all three of these data sets, a trend line is formed using the last data set. The measurement of 4.5cm on the third trial is clearly an outlier, so it is removed to improve accuracy of the trend line but included in the graph in the same color as the rest of the data series. The graph of all three runs with the trend line from the third data set is found in Figure E2. The trend line was useful for characterization of the valve and very useful for determining a first pass estimation for valve duration for 22mL of liquid. The equation for the trend line is:

$$Volume\ Dispensed\ in\ 100\ ms\ (mL) = -0.3767 * Distance\ to\ Sensor(cm) + 26.72(cm)$$

Dividing this by the total time (100 ms) gives an average flow rate which can be used to estimate time to dispense 22mL. The flow rate is not constant over the duration because of the switching time of the valve, but the average is a good first-order approximation without analyzing the mechanical properties of the solenoid. Linearity of the data provided an easy first-pass estimation for determining how long to hold the valve open. The first estimation is given in Table E4, which lists a distance, valve duration (in ms), and measured volume.

On the estimation experiments, the estimated values are rounded to the nearest millisecond, so only a millisecond timer was used to coordinate the valve. The maximum possible error on this scheme is one-half of a millisecond. The internal counter that sets the millisecond time base is cleared right before the valve timing starts to ensure that that the number of milliseconds the valve is held open is as accurate as possible. Even with this rounding scheme, the estimated values are within $\pm.5\%$ accuracy. Based on the results of the first run, a second run was conducted to get a better estimate for a 22mL valve duration. To arrive at the second estimates, the difference between the dispensed and desired volumes was divided by the slope of the flow rate graph from the characterization experiment and was added to the valve durations from the first run. The goal of this was to try to add the exact amount of time needed to compensate for the undershoot of the first run. This was a valid method because no volumes dispensed in the first run were greater than 22 mL. Estimates for the first data run generally were less than 22mL but were sporadic. Only 10 out of 27 dispensed volumes were within 1 mL of 22 mL for the first run, whereas 21 out of 26 volumes were between 21 mL and 23 mL. An even more accurate mapping could be determined by further by iterating the process of determining an average flow rate, determining the difference between desired and measured volume dispensed, and adjusting the duration for each distance. These estimates were carefully formulated and demonstrated that a linear equation could not be used to determine valve duration.

However, further iterations were not pursued because the limited accuracy of the measurement system would eclipse the inaccuracy of the valve durations. During the estimation experiments, the bottle was filled to very near each half centimeter marking. Normal operation of the CAMS does not result in the liquid levels falling exactly at these intervals. In some sections of the ingredient bottle dispensing 22mL drops the liquid height .5cm, but in other sections it drops the height by a different amount. The ultrasonic sensing software rounds the liquid height to the nearest .5cm, so error will be introduced if the actual height is not a multiple of .5cm.

There was one last issue with the test set up and the success of this prototype of the CAMS. The ingredient bottle is too thin for the ultrasonic sensor to measure properly. When the top of the liquid is very close to the top of the bottle, the ultrasonic measurement is very accurate and allows the valve to be held open for the appropriate number of milliseconds. However the sensor effectively saturates at 10cm, as it reads all further distances as 10cm. To get an accurate distance throughout the entire length of the ingredient bottle, a bottle of adequate thickness throughout needs to be selected. Such a bottle could not be found within the time constraints of this design project, but it is a major design concern for the future to ensure accurate operation at all times.

Future Work

Reducing Electronics Cost

The CAMS prototype is not optimized for cost – most components were selected because they were either readily available or because I already have development experience with those or similar components. As a result, there are many cost and complexity optimizations to be made to help further reduce the price.

Without the PCB and breadboard costs, the electrical components for the module cost $24.40. The easiest way to cut costs of circuitry is to shrink the MCU. Even within the ATMEGA series, there are several other microcontrollers that are cheaper with enough functionality to fulfil the needs of the CAMS. The currently used ATMEGA1284P has a base price of $7.75, whereas the ATMEGA324P and ATMEGA48P have base prices of $6.10 and $2.75, respectively. Using the ATMEGA48 would save $5 on board costs and would require minimal development cost because the microcontrollers are in the same product family. For reference, purchasing 100 ATMEGA48 microcontrollers in bulk would cost $2.0034 per unit, whereas the same number of ATMEGA1284P would cost $5.73 per unit. Switching to the ATMEGA 48 and purchasing in bulk would reduce the cost of each module board by $5.7466.

Another way to save on board cost is to update the circuit that drives the solenoid valve. Currently, the valve is driven by an opto-isolator circuit, which costs $1.93 in resistors, transistors, and an IC. Switching to a circuit that is based on a single power NFET could reduce the cost of circuitry because it requires fewer components. One such example transistor is the Fairchild Semiconductor RFD3055LESM9A NMOSFET. This transistor can handle a drain current of 11A (the CAMS runs between 1.6A and 2A), it can handle a V_{ds} of 60V (CAMS runs between 6V and 7V), and it can handle up to 38W of power (CAMS uses at most 14W). This NFET costs $0.78 for individual units (a savings of $1.15), and purchasing 100 units costs $0.524 per unit, which is a savings of $1.406 per module. Combining the savings of the updated solenoid driver with the updated microcontroller can save $7.1526 per board, which gives a total module savings of 6.9%.

Designing the CAMS was done with a prototyping board for the ATMEGA1284P and used several breadboards and DIP components. The prototyping board has some extra hardware on it that is not needed for a production board, and combining the DIP components on the breadboard on to the same board as the microcontroller would eliminate the need for the $5.95 breadboard entirely. To promote reusability of the prototyping boards, the ATMEGA1284P chips used are DIP components that fit into sockets mounted on the boards. This way, if a $7.75 chip breaks, the chip can be replaced without having to replace the entire $57.92 board. However, for a production board, the $0.51 DIP socket is not needed because the chips are not expected to break. Also located on the development board was an FTDI chip and associated components used to communicate with a PC over a serial connection. The CAMS prototype uses the Port D UART pins, and does not need any PC connection, so all of that hardware can be removed. All of the serial connection hardware costs $5.40. Removing other unused components such as the debug LED, unused port pins, and other assorted breakout pins and jumpers saves only about $0.25. Removing all debug hardware and the whiteboard in total would save $12.11 on total module costs. Both removing debug hardware and updating the circuitry/buying in bulk saves $19.26 per module, which is an overall savings of 18.60% per module.

Aside from reducing circuitry on the boards, purchasing circuit boards in bulk can significantly save on the cost. It is impossible to know exactly how much purchasing a circuit board in bulk would cost until the entire board is designed. A board redesign cannot begin until a new set of parts is selected. After that, the debug hardware would be removed from the prototype board, the new parts would be added, and additional utility would be added to make the product more robust. Instead of mounting the board on a whiteboard, screw holes could be added to physically mount the board to the structure of the module. Additionally, the existing UART bus is a set of wires plugged into breadboards, so a proper method of constructing an arbitrarily sized bus must be designed, then incorporated into the new circuit board design.

After the board design is updated to be used on a proper product, the circuit boards can be ordered in bulk to reduce costs. Advanced Circuits, a PCB manufacturer, offers a calculator for estimating board price for a simple two-layer board based on board dimensions (Advanced Circuits, 2014). Using the dimensions of the existing prototyping boards, purchasing 100 boards would cost $4.02 per board, which is a savings of $28.98, or 29.6% over the existing module costs. Combined with previously discussed methods of reducing electronics cost on the boards, I estimate that each module could be made for $55.35, which is 46.6% cheaper than the prototypes. This calculation does not include purchasing all components in bulk, or conducting full trade studies to determine the cheapest components that can be used while still maintaining proper operation of the CAMS.

Mechanical Design Updates

Analyzing cost changes to the mechanical design is more complicated. From the parts list, Table D5, it is clear that the best way to reduce mechanical cost is to use a different solenoid valve and to use a different connector to go from the plastic bottle to the tubing. The cost of the tubing and connectors can be reduced by purchasing components in bulk, but the cost of the valve and the hose connector dominates the mechanical cost. It may be possible to find a cheaper solenoid valve of the same size, but careful attention must be paid so that the new valve is also beverage safe. Similarly, another connector may be found to connect the ingredient bottle to the tubing, but it also must be of a safe material. However, to properly optimize costs of the mechanical subsystem, a proper trade study needs to be conducted across all solenoid valves, and possibly other types of electrically controlled valves to determine what the cheapest solution is. A basic search seems to indicate that valves of a smaller diameter are cheaper, which is expected, but a full study must be conducted to validate these claims. Not only should a trade study be conducted for the plastic bottle connectors, but other connection schemes should be investigated to see if there is a more efficient way to connect the bottle to the tubing.

However, as previously discussed, the mechanical design of the CAMS was not completed before the submission of this design report. A 20oz soda bottle was used to hold ingredients because it was readily available, but it did not lead to the most robust design. The cost of this bottle was not taken into account because it is difficult to estimate the cost, although it is expected that the cost of the plastic is low. However, for a more robust design, another container type should be selected. This container must be safe to hold alcohols for long periods of time and must allow a rectangular hole in the top to house the ultrasonic sensor. It is likely that this container will cost more than the cheap plastic soda bottle, but it is also possible that a new container would have an NPT or other standard connection, which would allow for the use of a cheap polypropylene or polyethylene connector instead of the expensive brass hose connector.

Unfortunately, it is impossible to know exactly how the cost will change until other container designs are investigated and evaluated for cost and applicability to the system.

The largest mechanical component that needs a proper redesign in the structure used to hold the valve, ingredient container, and module board. The existing design uses wooden and Styrofoam components that were available in the laboratory during design. Not only are those designs not reproducible because the wooden structure is a custom structure left over from old lab equipment, but it is not desirable to reproduce those designs because they are not well suited for the CAMS. A proper design would have a more stable way to hold the ingredient bottle in place and a sturdier method of holding the valve based on the exact weight and dimensions of the valve. Because the ultimate goal of the CAMS is to be a consumer product, it is important for the structural design to be able to handle getting knocked in to or otherwise battered by an ordinary user. Several structural designs must be prototyped and tested to determine cost and "user friendliness" of each design to read the optimal design.

User Interface Updates

Another aspect of the CAMS design that was not fully completed was the user interface for ordering cocktails. As discussed, the existing design has the framework for ordering any cocktail off of the menu, but it has to be done from a plain serial terminal. The purpose of the basic serial communications was to demonstrate the method that the menu information and selection could be efficiently transmitted to and from the user. Now that the basic infrastructure is in place, there are several ways to update the user interface without having to redesign way information is encoded. The most cost effective way to do this is to use a Bluetooth module and a smartphone app to take drink orders from customers. A user would download an application for their smartphone that would perform the same tasks as the current menu and ordering scheme but would not require the orders to be placed one by one at a terminal.

When the application is first started, it attempts to establish communication with the main module. Once communication is established, the main module would then send the menu over the Bluetooth connection. However, instead of a user simply choosing a number from a list, the application would display just the name and ingredients in a more user friendly way than just plain text. It would also be easy to have options to sort and search through drinks in a number of ways, such as alphabetically, popularity, by ingredient list, or by alcohol content. The Android development environment could be used to quickly write and test such an application.

The system cost would be increased by the cost of the Bluetooth module used for communication, although the serial chip would not be necessary, which would counteract some of the cost. However, there are two main challenges associated with using a smartphone application. The first is coming up with a secure way to run set up and diagnostics on the system. The current implementation requires that the bartender populate the modules in stock from the same terminal that a user uses. The bartender would either have to still use a serial connection to set up the modules and receive status messages about the system, or he would have to be running a different application that had only utilities for populating modules and receiving messages. This would require additional software development to write a second application and to allow one application to run set up the system and let other applications to order cocktails regularly. To service all beverage requests in parallel, the main module would also likely need to implement a queue of cocktails to be ordered and keep track of which devices ordered which beverages. When a beverage reaches the front of the queue and is prepared, a notification would be sent to the appropriate device to let the right user know to come to the bar and pay for their beverage.

The second challenge associated with using a Bluetooth framework is that a Bluetooth master can only actively communicate with up to 7 slaves. There has been some research into constructing ad-hoc networks of Bluetooth devices, but Bluetooth is not a common means of communication, so there are very few resources on how to build an ad hoc network on smartphones. The ad hoc network would replace the Smartphone applications can be turned on and off at any time, so the Bluetooth network needs to be able to handle many users coming online and going offline frequently. Other communication technologies can potentially be used but Bluetooth is very attractive for the CAMS because only a single Bluetooth module needs to be used for the entire system and most smartphones come packaged with a Bluetooth module. Also, because there aren't many other Bluetooth devices that would be used in a bar, it is likely that there wouldn't be any competition for resources with the Bluetooth module. A user might be trying to send data over the cellular network or use WiFi on their phones at any time, but Bluetooth is only used for a few types of uses, and those uses are generally orthogonal to the CAMS, so the CAMS would have sole use of the Bluetooth module.

Conclusions

At the time of submission of this design report, a small scale CAMS prototype can been successfully constructed and tested in the laboratory. This prototype contains one fully constructed module with the ingredient bottle, valve assembly, and ultrasonic sensor mounted and functioning together. A second development module is also in use that does not have a structure built but also has a valve and ultrasonic sensor connected properly that function in the same manner as the first module. Extensive experiments were conducted to characterize the ultrasonic sensor and the solenoid valve so that they could be used as accurately as possible during CAMS operation to consistently deliver cocktails to a user in a timely manner.

There are several shortcomings with the prototype which have been identified throughout this report. The most notable is that a proper mechanical structure has not been designed and implemented, and the ingredient bottle gets too narrow at points for the ultrasonic sensor to function properly. These are not particularly challenging design constraints, but due to the limited development time allowed by this project they could not be addressed at this time. Aside from the noted shortcomings, there are also several ways to improve the existing CAMS design. Many of these ways were discussed in this report, and these improvements range from an improved user interface to optimizing the bill of materials used to reduce costs without sacrificing functionality.

All of this being said, the CAMS prototype was successful. The major design subsystems either function correctly, or specific issues have been identified to fix the design. The existing system allows for a large range of functionality across many modules. A user must program the system to indicate which ingredients are in which modules, but different ingredients can be programmed each time at start up and the CAMS will determine which cocktails it can prepare. When a user orders a beverage, the main board sends a command over the UART bus to all module boards, and the appropriate module board responds to the command over the UART bus and utilizes and ultrasonic sensor to determine how long to hold a solenoid valve open for to dispense the correct amount of liquid.

Most importantly, the prototype meets all of its core design premises. First, the CAMS is less expensive than all existing systems, even though the prototype has additional hardware needed for development and debugging. A full mechanical design is going to add cost per module, but there are many cost optimizations that can be done for the electrical design to drive down the cost per module so that the CAMS is still more cost effective than all other systems. Second, the CAMS is scalable. The current initialization process and the UART bus scheme allow a wide range of modules to be used to prepare many different cocktails. Minor software updates will be required to provide an even greater range of ingredients and cocktails, but the software infrastructure for these additions already exists. The software architecture will not have to be altered to accommodate for additional modules. All of the system components function properly together and the prototype design easily allows the expansion from a few modules up to a maximum of 64. The constructed modules, cost analysis, and system structure clearly demonstrate a strong proof-of-concept for the Cocktail Automation Management System to be an accurate, cheap, and scalable bartending automation system.

References

Adafruit Industries. (2014). *Full Sized Breadboard*. Retrieved from adafruit.com: http://www.adafruit.com/products/239

Advanced Circuits. (2014). *2-Layer PCB Designs*. Retrieved from 4pcb.com: http://www.4pcb.com/33-each-pcbs/index.html

Advanced Circuits. (2014). *BareBones Printed Circuit Boards*. Retrieved from 4pcb.com: http://www.4pcb.com/bare-bones-pcbs/index.html

All Electronics Corporation. (2014). *2 x 40 Pins Snappable Headers*. Retrieved from allelectronics.com: http://www.allelectronics.com/make-a-store/item/dhs-40/2-x-40-pins-snappable-headers/1.html

Digi-Key Electronics. (2014). *Electronic Components Listing*. Retrieved from digikey.com: http://www.digikey.com/product-search/en

HC-SR04 Ultrasonic Sensor. (2011, May 10). Retrieved from Elec Freaks: http://www.elecfreaks.com/store/hcsr04-ultrasonic-sensor-distance-measuring-module-ultra01-p-91.html

Hobby King. (2014). *Ultrasonic Module HC-SR04 Arduino*. Retrieved from hobbyking.com: http://www.hobbyking.com/hobbyking/store/__31136__ultrasonic_module_hc_sr04_arduino.html

Ultrasonic Ranging Module HC-SR04. (n.d.). Retrieved from micropik.com - The Art of Components: http://www.micropik.com/PDF/HCSR04.pdf

United States Plastics Corporation. (2014). *Tubing, Hose, and Fittings*. Retrieved from usplastic.com: http://www.usplastic.com/catalog/default.aspx?catid=856&parentcatid=-1

Lowes Home Improvement. (2014). *Genova 1/2-in Dia Insert Elbow*. Retrieved from lowes.com: http://www.lowes.com/pd_22520-322-350705_0__?productId=3455100&Ntt=

Lowes Home Improvement. (2014). *Watts 3/4-in x 1/2-in Threaded Adapter Fitting*. Retrieved from lowes.com: http://www.lowes.com/pd_416836-104-LFA-684_0__?Ntt=lfa-684&UserSearch=lfa-684&productId=3824549&rpp=32

Adafruit Industries. (2014). *Brass Liquid Solenoid Valve*. Retrieved from adafruit.com: http://www.adafruit.com/products/996

Atmel Corporation. (2009, November). *ATmega1284P Datasheet.* Retrieved from atmel.com: http://www.atmel.com/images/doc8059.pdf

Land, B. (2013). *ECE 4760 Lab 4*. Retrieved from Cornell University ECE 4760 - Designing with Microcontrollers: http://people.ece.cornell.edu/land/courses/ece4760/labs/f2013/lab4.html

Chun, N. I. (2010, May). *Electrical and Computer Engineering Projects: Masters of Engineering and Undergraduate Independent Study.* Retrieved from Next Generation Atmel ATMEGA644 Prototype Board: http://people.ece.cornell.edu/land/courses/eceprojectsland/STUDENTPROJ/2009to2010/nic4/Writeupv5.pdf

BoozeBots. (2014). *BoozeBot - The Automated Cocktail Dispenser*. Retrieved from kickstarter.com: https://www.kickstarter.com/projects/boozebots/boozebot-the-automated-cocktail-dispenser

Bosch. (1991, September). *CAN Specification.* Retrieved from bosch-semiconductors.de: http://www.bosch-semiconductors.de/media/pdf_1/canliteratur/can2spec.pdf

Monsieur. (2014). Retrieved from monsieur.com: http://monsieur.co/

Party Robotics. (2014). Retrieved from partyrobotics.com: http://partyrobotics.com/
The Inebriator. (2014). *The Inebriator: Arduino Powered Cocktail Machine*. Retrieved from theinebriator.com: http://www.theinebriator.com/

Appendix A: Software and User Interface

This Appendix contains a reference for the user interface of the CAMS. Table A1 shows the mapping from ingredient number to ingredient name which is used during initialization to specify what ingredients are in each module. Only a limited number of cocktails were used for prototyping the CAMS, but a listing of the cocktails used and their respective recipes is contained in Table A2. Figure A1 shows an example sequence for a successful initialization of the CAMS and the ordering of a single cocktail. Figure A2 shows a successful ordering process where the system uses a bus that is pulled to ground instead of to 5V. For a complete listing of the software of the CAMS, please contact cig23@cornell.edu.

Table A1 – List of Ingredient Numbers and Names

Ingredient Number	Ingredient Name
0	Vodka
1	Rum
2	Gin
3	Peach Schnapps
4	Orange Juice
5	Grenadine
6	Coke

Table A2 – List of Cocktails

Cocktail Name	Number of Parts Vodka	Number of Parts Rum	Number of Parts Gin	Number of Parts Peach Schnapps	Number of Parts Orange Juice	Number of Parts Grenadine	Number of Parts Coke
Screwdriver	1	0	0	0	2	0	0
Rum and Coke	0	2	0	0	0	0	1
Potpourri	1	1	1	1	1	1	1
Gin and Juice	0	0	1	0	1	0	0
Cherry Schnapps	0	0	0	1	0	2	0

```
Starting...
What is in slot 0?
4
What is in slot 1?
6
What is in slot 2?
2
What is in slot 3?
5
What is in slot 4?
0
What is in slot 5?
99

Have vodka in slot 4
gin in slot 2
orange juice in slot 0
grenadine in slot 3
coke in slot 1

Current Menu:

Item 0, Screwdriver:
  1 parts vodka
  2 parts oj

Item 1, Gin and Juice:
  1 parts gin
  1 parts oj

What drink would you like to order?
0
You have selected 0
Screwdriver:
  1 parts vodka
  2 parts oj

This drink needs 1 parts of vodka, which is located in module 4
Attempting to request 1 parts from module 4
success!
transaction complete
This drink needs 2 parts of oj, which is located in module 0
Attempting to request 2 parts from module 0
success!
transaction complete
What drink would you like to order?
```

Figure A1 – Example Successful Serial Console Output

```
What drink would you like to order?
0
You have selected 0
Screwdriver:
  1 parts vodka
  2 parts oj

This drink needs 1 parts of vodka, which is located in module 4
Attempting to request 1 parts from module 4
no response
Attempting to request 1 parts from module 4
success!
transaction complete
This drink needs 2 parts of oj, which is located in module 0
Attempting to request 2 parts from module 0
no response
Attempting to request 2 parts from module 0
success!
transaction complete
What drink would you like to order?
```

Figure A2 – Example Successful Ordering Sequence Using a Grounded Bus

Appendix B: Mechanical Design

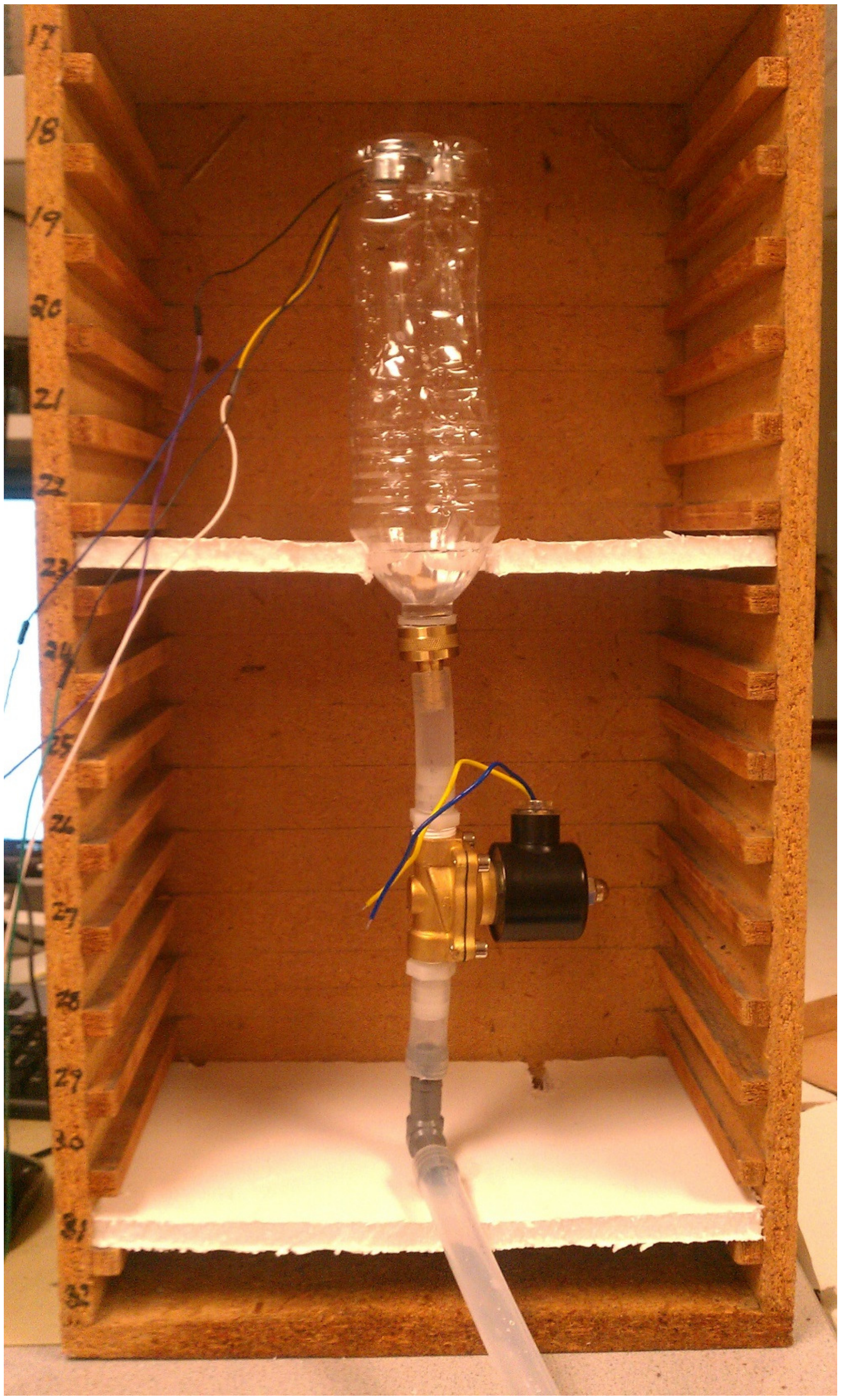


Figure B1 – Front Mechanical View of a CAMS module

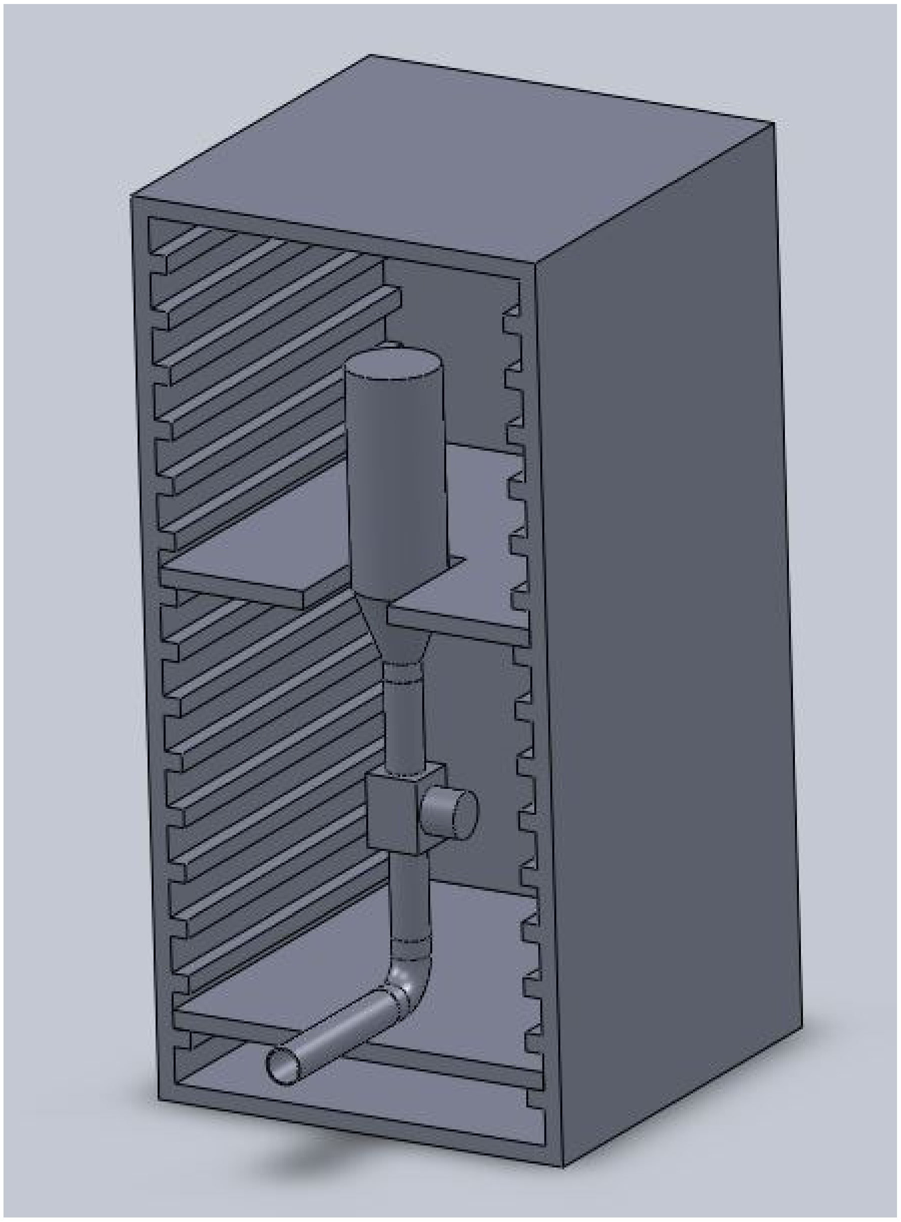


Figure B2 – SolidWorks Model of Mechanical Structure

Appendix C: Circuit Board Schematics and Design

The design and layout for the ATMEGA1284P breakout board is taken with permission from Nathan Chun (nic4) and Bruce Land (brl4). The board schematic and design can be found in the design report for the board (Chun, 2010).

The schematics found below are the full schematics for both the main and module boards. The schematics do not indicate which components are on the breakout board and which are external. Please refer to the breakout board design schematics for a reference for on board components. Tables D2 and D3 in Appendix D contain a listing of all parts used on each board. Figures C1 and C3 contain schematics of the main and module boards and Figures C2 and C4 contain pictures of the constructed circuits, respectively. Figure C5 contains the schematics of the CAN Bus breakout board that was designed but not used in the prototype.

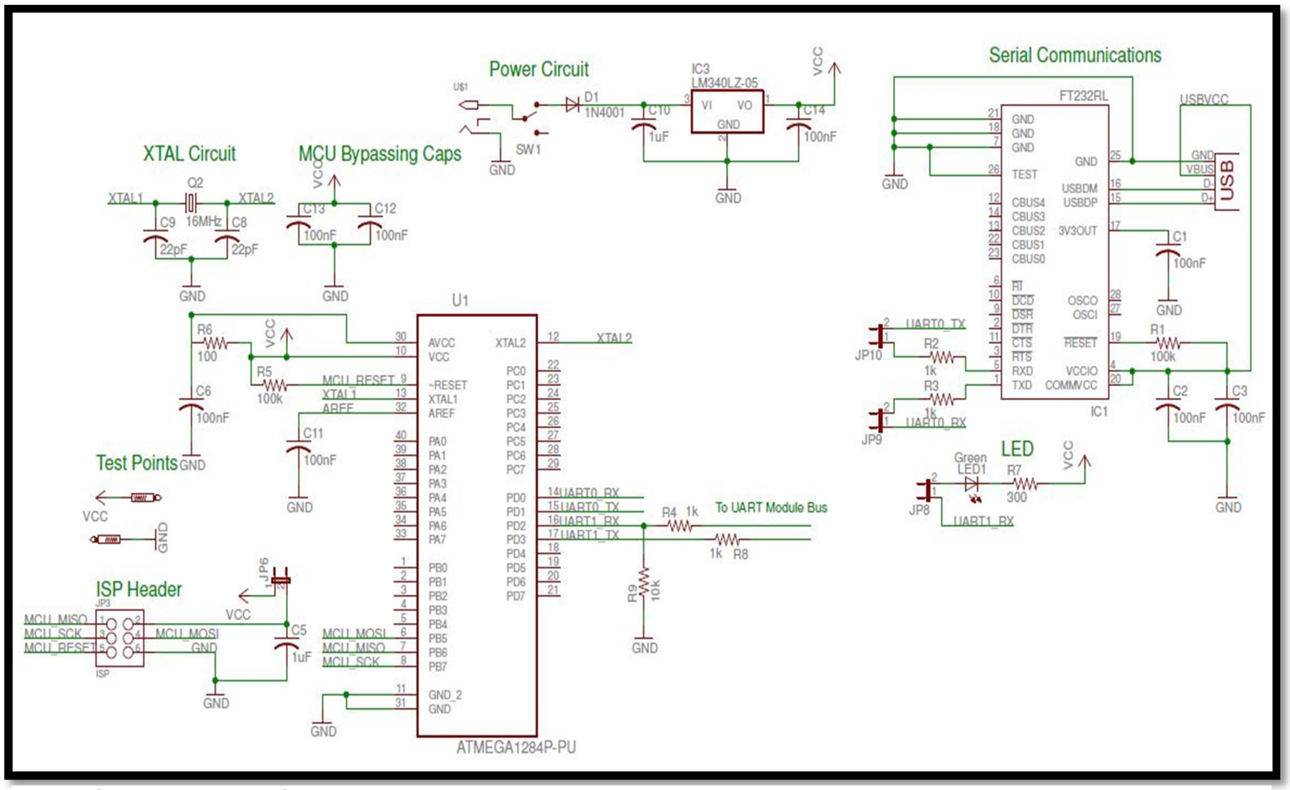


Figure C1 – Main Board Schematic

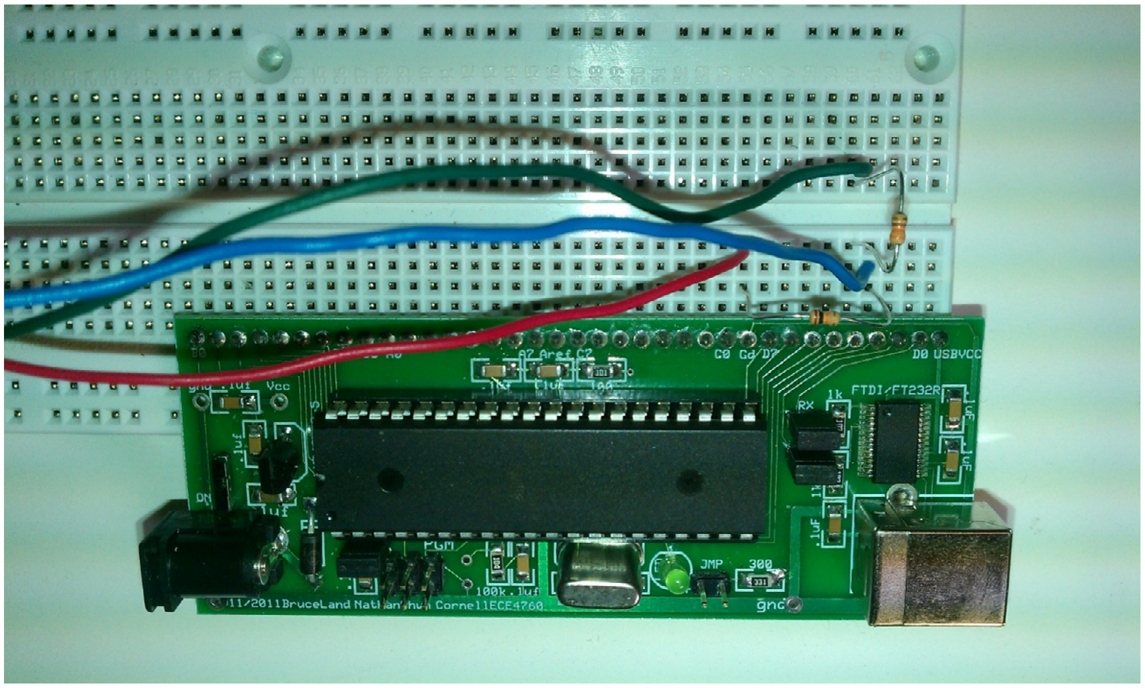


Figure C2 – Constructed Main Board

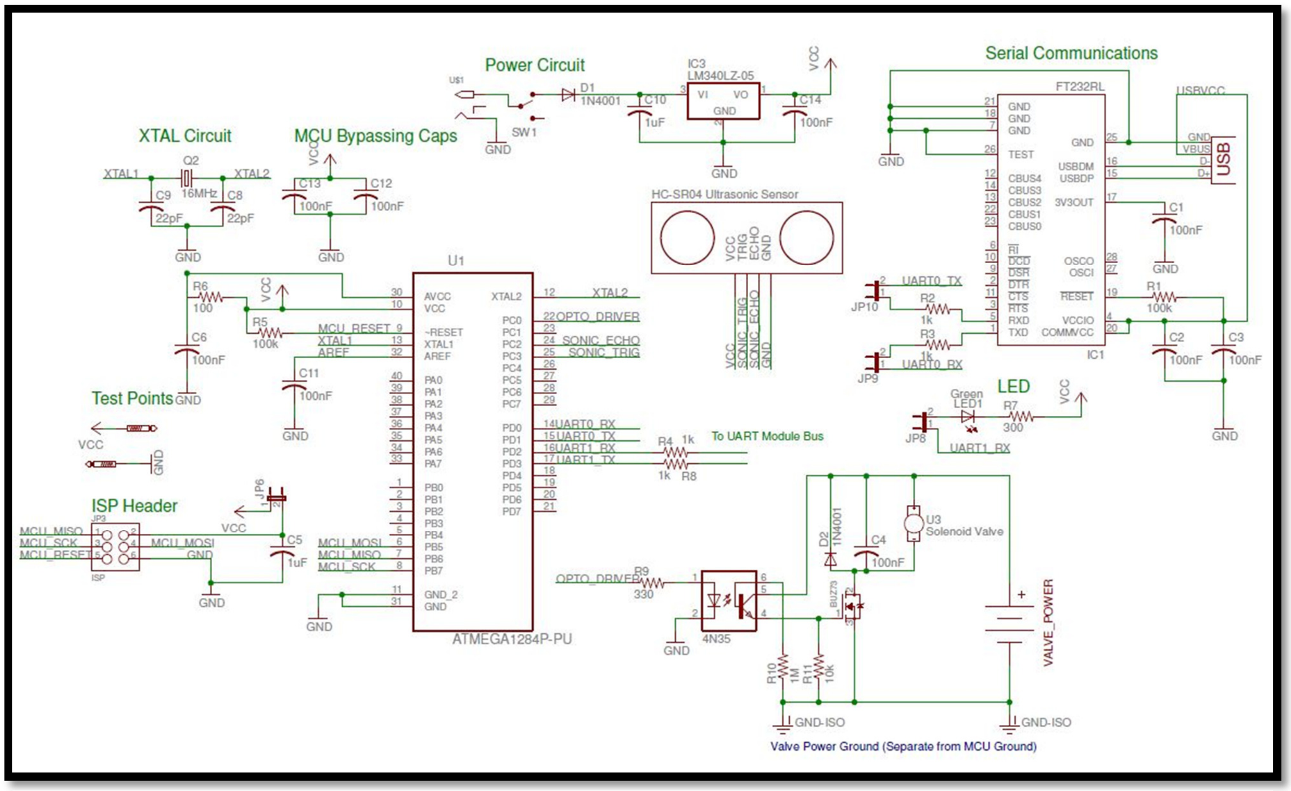


Figure C3 – Module Board Schematic

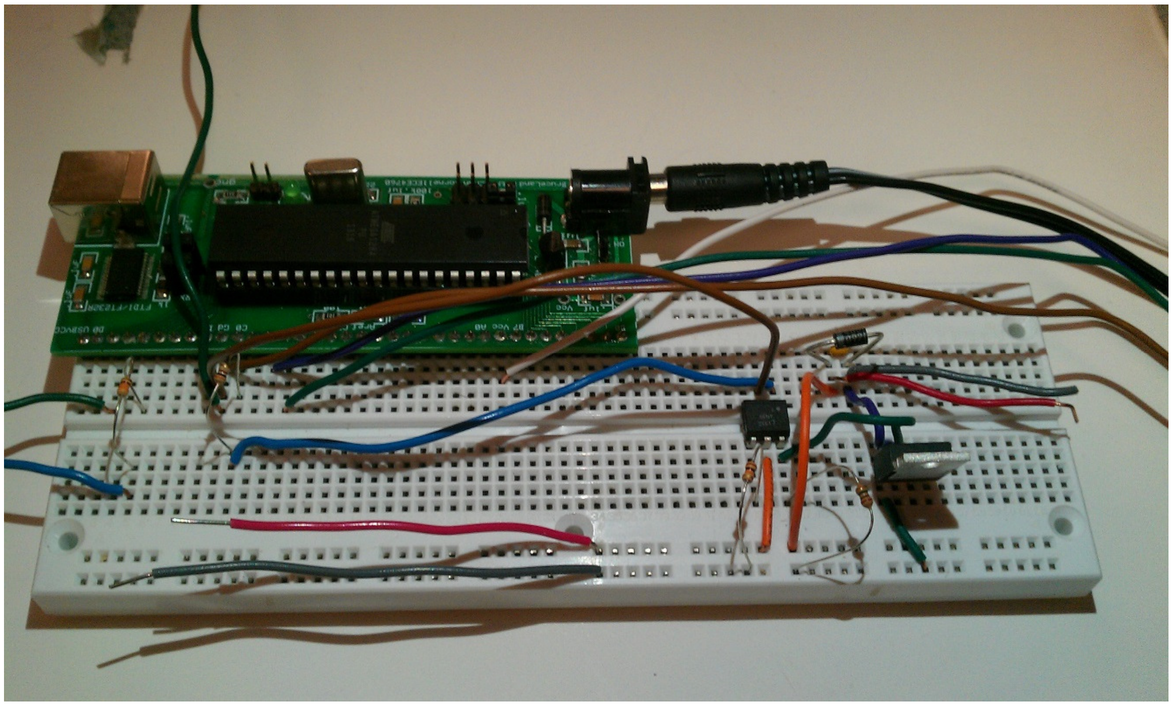


Figure C4 – Constructed Module Board (Solenoid Valve and Valve Power Rail Connections Disconnected)

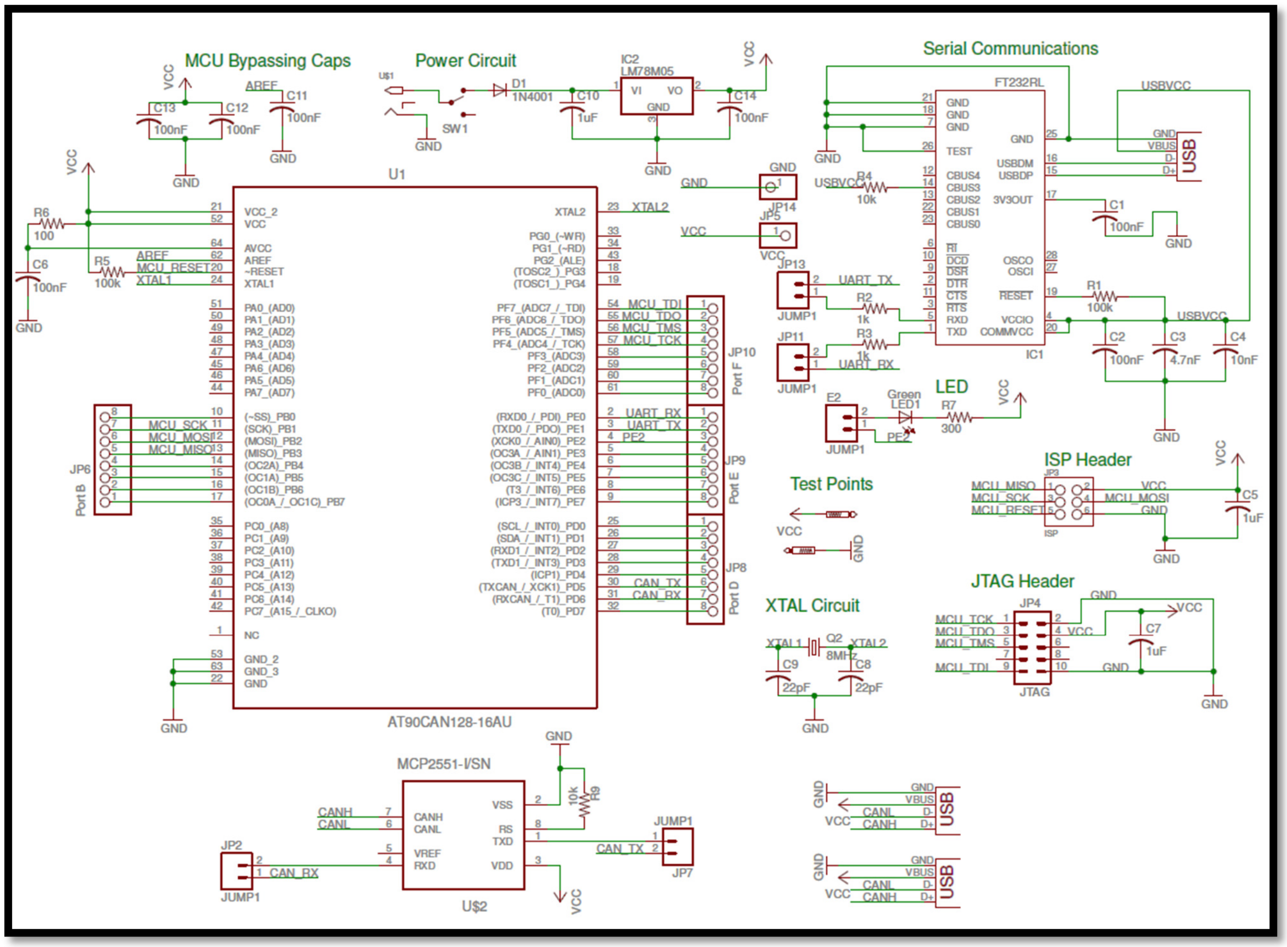


Figure C5 – AT90CAN128 Breakout Board

Appendix D: Component Costs

This Appendix contains a reference for the part numbers and costs used to estimate a price for each CAMS module. Please note that the values listed in these tables are for reference only, the actual price and availability of electrical and mechanical components may vary significantly. The electrical component costs here are all taken from digikey.com (Digi-Key Electronics, 2014), unless otherwise listed. The mechanical component details are all taken from usplastics.com (United States Plastics Corporation, 2014). Price comparisons for the existing systems are taken from their respective product websites (Party Robotics, 2014), (BoozeBots, 2014), (Monsieur, 2014).

Table D1 - Electrical Components

Component Name	Manufacturer	Part Number	Unit Cost
Diode	Comchip Technology	1N4001-G	\$0.11
NFET	Infineon Technologies	BUZ73 H3046	\$1.34
1MΩ Resistor	Stackpole Electronics	CF14JT1M00	\$0.08
100kΩ Resistor	Stackpole Electronics	CF14JT100K	\$0.08
10kΩ Resistor	Stackpole Electronics	CF14JT10K0	\$0.08
330Ω Resistor	Stackpole Electronics	CF14JT330R	\$0.08
100Ω Resistor	Stackpole Electronics	CF14JT100R	\$0.08
1uF Capacitor	Kemet	C1206C104K3RACTU	\$0.20
100nF Capacitor	Kemet	C1206C104K3RACTU	\$0.10
10nF Capacitor	Kemet	C1206C103JARACTU	\$0.12
4.7nF Capacitor	Kemet	C1206C472K5RACTU	\$0.14
22pF Capacitor	Kemet	C1206C220K5GACTU	\$0.16
16MHz Crystal	CTS-Frequency Controls	MP160	\$0.66
Green LED	Lumex, Inc	SSL-LX5093PGD	\$.055
Opto-isolator	Lite-On Inc	4N35	\$0.43
DIP Socket	Assmann WSW Components	A40-LC-TT	\$0.51
UART-USB Converter	Future Technology Devices Limited	FT232RL-REEL	\$4.50
5V Regulator	Texas Instruments	LM340LAZ-5.0/NOPB	\$0.95
USB-B Socket	On-Shore Technology, Inc	USB-B1HSB6	\$0.54
Single Row Male Pin Header	Molex Inc	0022284360	\$0.00356/pin
Dual Row Male Pin Header	All Electronics	DHS-40	\$0.0375/row[1]
Microcontroller	Atmel	ATMEGA1284P	\$7.75
2.1mm Power Jack	CUI Inc	PJ-002A	\$0.93
Ultrasonic Sensor	Hobby King	HC-SR04	\$2.89[2]
SPDT Switch	C&K Components	AYZ010AGRLC	\$0.91
White Breadboard	Adafruit	239	\$5.95[3]
Custom PCB	Advanced Circuits	N/A	\$33[4]

[1] (All Electronics Corporation, 2014)
[2] (Hobby King, 2014)
[3] (Adafruit Industries, 2014)
[4] (Advanced Circuits, 2014)

Table D2 – Main Board Components

Component Name	Unit Cost	Quantity	Total Price
Diode	$0.11	1	$0.11
10kΩ Resistor	$0.08	1	$0.08
330Ω Resistor	$0.08	2	$0.16
100Ω Resistor	$0.08	1	$0.08
1uF Capacitor	$0.20	2	$0.40
100nF Capacitor	$0.10	5	$0.50
10nF Capacitor	$0.12	1	$0.12
4.7nF Capacitor	$0.14	1	$0.14
22pF Capacitor	$0.16	2	$0.32
16MHz Crystal	$0.66	1	$0.66
Green LED	$.055	1	$.055
DIP Socket	$0.51	1	$0.51
UART-USB Converter	$4.50	1	$4.50
5V Regulator	$0.95	1	$0.95
USB-B Socket	$0.54	1	$0.54
Single Row Male Pin Header	$0.00356/pin	42 pins	$0.14952
Dual Row Male Pin Header	$0.0375/row	3 rows	$0.1125
Microcontroller	$7.75	1	$7.75
2.1mm Power Jack	$0.93	1	$0.93
SPDT Switch	$0.91	1	$0.91
White Breadboard	$5.95	1	$5.95
Custom PCB	$33	1	$33
Total Cost			$57.92

Table D3 – Module Board Components

Component Name	Unit Cost	Quantity	Total Price
Diode	$0.11	2	$0.22
10kΩ Resistor	$0.08	1	$0.08
1kΩ Resistor	$0.08	2	$0.16
330Ω Resistor	$0.08	2	$0.16
100Ω Resistor	$0.08	1	$0.08
1uF Capacitor	$0.20	2	$0.40
100nF Capacitor	$0.10	9	$0.90
22pF Capacitor	$0.16	2	$0.32
16MHz Crystal	$0.66	1	$0.66
Green LED	$0.06	1	$0.06
DIP Socket	$0.51	1	$0.51
UART-USB Converter	$4.50	1	$4.50
5V Regulator	$0.95	1	$0.95
Opto-isolator	$0.43	1	$0.43
USB-B Socket	$0.54	1	$0.54
Single Row Male Pin Header	$0.00356/pin	42 pins	$0.01
Dual Row Male Pin Header	$0.0375/row	3 rows	$0.11
Microcontroller	$7.75	1	$7.75
2.1mm Power Jack	$0.93	1	$0.93
SPDT Switch	$0.91	1	$0.91
White Breadboard	$5.95	1	$5.95
Custom PCB	$33	1	$33.00
1MΩ Resistor	$0.08	1	$0.08
10kΩ Resistor	$0.08	1	$0.08
330Ω Resistor	$0.08	2	$0.16
NFET	$1.34	1	$1.34
Ultrasonic Sensor	$3.13	1	$2.89
Total Electrical Cost			$63.42

Table D4– Mechanical Components

Component Name	Manufacturer	Part Number	Unit Cost
Brass ¾" Garden Hose to ½" Hose Barb	Watts	LFA-684	$9.99[5]
HDPE ½" Male NPT to ½" Hose Barb Converter	United States Plastic Corporation	62017	$0.36
Brass Solenoid Valve	Geerte	2W-160-15	$24.95[6]
Bev-A-Line Plastic Tubing	United States Plastic Corporation	56282	$2.62/foot
HDPE ½" Male NPT to ½" Hose Barb 90^o elbow	United States Plastic Corporation	62043	$0.55
HDPE ½" Hose Barb T Connector	United States Plastic Corporation	62067	$0.55
Polypropylene ½" Hose Barb 90^o elbow	Genova	22520	$0.58[7]
Polypropylene ½" Hose Barb Y Splitter	United States Plastic Corporation	62256	$0.91

Table D5 – Single Valve Module Mechanical Components

Component Name	Unit Cost	Quantity	Total Price
Brass ¾" Garden Hose to ½" Hose Barb	$9.99	1	$9.99
HDPE ½" Male NPT to ½" Hose Barb Converter	$0.36	2	$0.72
Brass Solenoid Valve	$24.95	1	$24.95
Bev-A-Line Plastic Tubing	$2.62/foot	1.5 feet	$3.93
Plastic ½" Male NPT to ½" Hose Barb 90^o elbow	$0.58	1	$0.58
Total Mechanical Cost			$40.17

Table D6 – Compressed Air Module Mechanical Components

Component Name	Unit Cost	Quantity	Total Price
Brass ¾" Garden Hose to ½" Hose Barb	$9.99	1	$9.99
Plastic ½" Male NPT to ½" Hose Barb Converter	$0.36	2	$0.72
Brass Solenoid Valve	$24.95	2	$49.90
Bev-A-Line Plastic Tubing	$2.62/foot	3 feet	$7.86
Plastic ½" Male NPT to ½" Hose Barb 90^o elbow	$0.55	2	$1.10
Polypropylene ½" Hose Barb Y Splitter	$0.91	1	$0.91
Total Mechanical Cost			$70.48

5 (Lowes Home Improvement, 2014)
6 (Adafruit Industries, 2014)
7 (Lowes Home Improvement, 2014)

Table D7 – Compressed Air System Module Costs

Component Name	Price
Total Mechanical Cost	$70.48
Total Electrical Cost	$63.18
Total Module Cost	$133.66

Table D8 – Single Valve Module Costs

Component Name	Price
Total Mechanical Cost	$40.17
Total Electrical Cost	$63.18
Total Module Cost	$103.35

Table D9 – Price Comparisons

Product	Number of Modules	Total Cost	Effective Cost Per Module	Price Relative to CAMS Cost
CAMS	2	$265.10	$132.55	-
BoozeBot	2	$683.81	$341.91	257.95%
CAMS	4	$471.80	$117.95	-
BoozeBot	4	$1112.26	$278.07	235.75%
CAMS	7	$781.86	$111.69	-
Bartendro	7	$2499.99	$357.14	319.75%
CAMS	8	$885.21	$110.65	-
BoozeBot	8	$1283.64	$160.46	145.01%
Monsieur	8	$3,999 .99	$499.88	451.76%
CAMS	15	$1608.68	$107.25	-
Bartendro	15	$3699.99	$246.67	230.00%
CAMS	16	$1712.03	$107.00	-
BoozeBot	16	$1883.48	$117.72	110.01%

Appendix E: Experimental Results

This Appendix contains the tables and graphs of raw data gathered from the experiments used to determine the accuracy of components used the CAMS. Table E1 contains the data from the test to determine the mapping from ultrasonic sensor output pulse width to distance in centimeters, along with the model from the datasheet and experimentally determined model. Figure E1 graphs these results for a visual representation of the models. Table E2 gives the calculated percent error between the two models and the data gathered from the experiment. Table E3 contains the data from the first valve experiment, where the valve was held open for a constant duration and the dispensed volume was measured as function of liquid height in the ingredient bottle. Figure E2 graphs these results and plots the trend line of the most accurate data set. Table E4 contains the data from the two runs of the experiment to determine the duration of time to hold the valve open to dispense 22mL.

Table E1 – Results of Ultrasonic Sensor Test

Measured Sensor Distance (cm)	Measured Pulse Width	Datasheet Model	Experimental Model
4	194	232	192
4.5	222	261	221
5	248	290	250
5.5	278	319	279
6	306	348	308
6.5	330	377	337
7	360	406	366
7.5	408	435	415
8	432	464	444
8.5	464	493	473
9	492	522	502
9.5	532	551	531
10	572	580	560
10.5	604	609	589
11	636	638	618
11.5	644	667	647
12	692	696	676
12.5	696	725	705
13	728	754	734
13.5	756	783	763
14	784	812	792
14.5	812	841	821
15	840	870	850
15.5	868	899	879
16	890	928	908

16.5	920	957	937
17	950	986	966
17.5	980	1015	995
18	1000	1044	1024
18.5	1040	1073	1053
19	1070	1102	1082
19.5	1100	1131	1111
20	1140	1160	1140
20.5	1170	1189	1169
21	1200	1218	1198
21.5	1200	1247	1197
22	1220	1276	1226
22.5	1250	1305	1255
23	1280	1334	1284
23.5	1310	1363	1313
24	1330	1392	1342

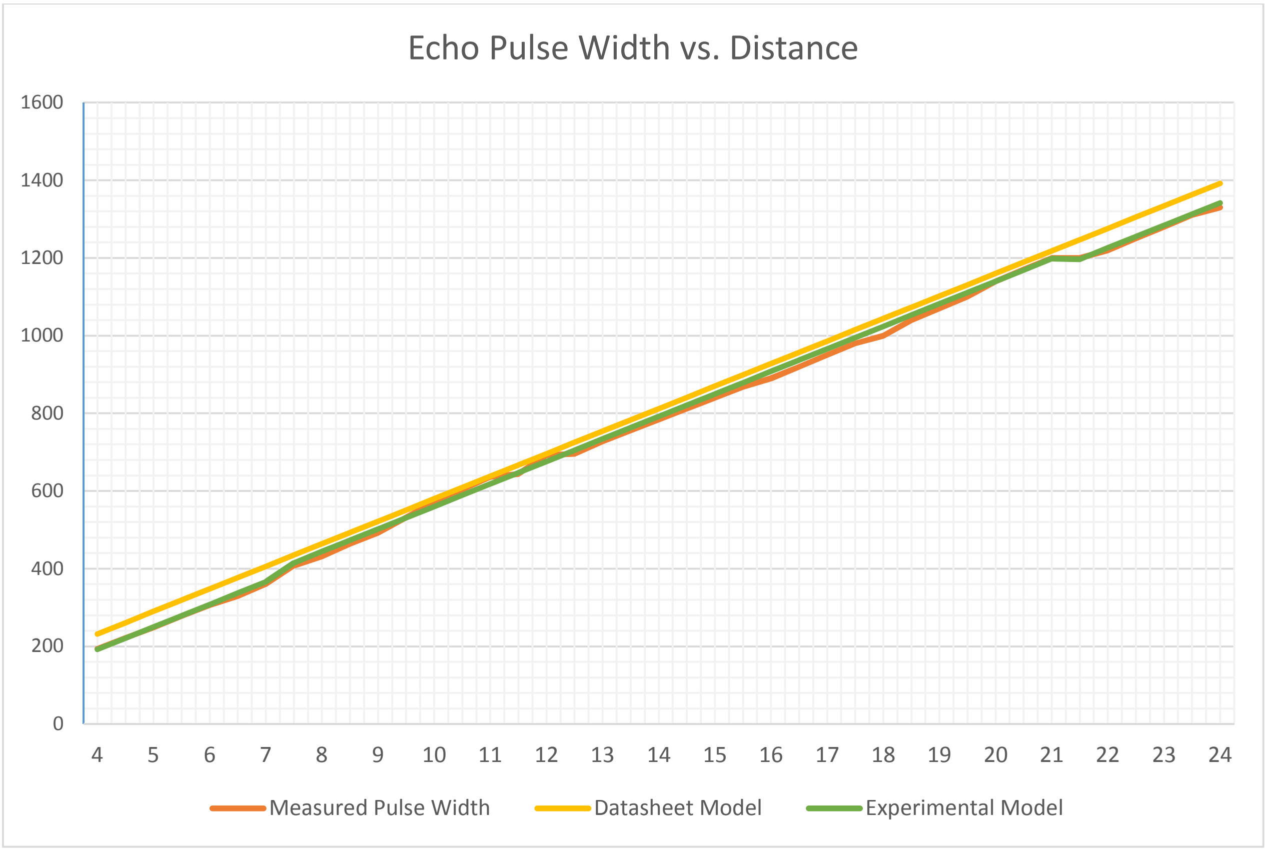


Figure E1 – Graph of Ultrasonic Sensor Test

Table E2 – Error of Ultrasonic Sensor Models

Measured Sensor Distance (cm)	Datasheet Model Error	Experimental Model Error
4	19.59%	-1.03%
4.5	17.57%	-0.45%
5	16.94%	0.81%
5.5	14.75%	0.36%
6	13.73%	0.65%
6.5	14.24%	2.12%
7	12.78%	1.67%
7.5	6.62%	1.72%
8	7.41%	2.78%
8.5	6.25%	1.94%
9	6.10%	2.03%
9.5	3.57%	-0.19%
10	1.40%	-2.10%
10.5	0.83%	-2.48%
11	0.31%	-2.83%
11.5	3.57%	0.47%
12	0.58%	-2.31%
12.5	4.17%	1.29%
13	3.57%	0.82%
13.5	3.57%	0.93%
14	3.57%	1.02%
14.5	3.57%	1.11%
15	3.57%	1.19%
15.5	3.57%	1.27%
16	4.27%	2.02%
16.5	4.02%	1.85%
17	3.79%	1.68%
17.5	3.57%	1.53%
18	4.40%	2.40%
18.5	3.17%	1.25%
19	2.99%	1.12%
19.5	2.82%	1.00%
20	1.75%	0.00%
20.5	1.62%	-0.09%
21	1.50%	-0.17%
21.5	3.92%	-0.67%
22	4.59%	0.08%
22.5	4.40%	0.00%
23	4.22%	-0.08%
23.5	4.05%	-0.15%
24	4.66%	0.53%

Table E3 – Results of Constant Duration Valve Test

	Volume Dispensed in 100 ms (mL)		
Distance from Sensor to Liquid (cm)	First Run (mL)	Second Run (mL)	Third Run (mL)
4	23	24	25
4.5	23	18	32
5	23		24
5.5	18	30	-
6	23	-	25
6.5	-	24	-
7	29	-	24
7.5	14	23	24
8	23	-	-
8.5	30	22	24
9	21	-	23
9.5	-	17	23
10	-	20	23
10.5	21	23	22
11	18	29	23
11.5	25	21	22
12	21	-	23
12.5	21	15	23
13	14	20	22
13.5	21	28	22
14	20	12	22
14.5	26	23	21
15	19	20	21
15.5	15	28	21
16	19	16	20
16.5	25	22	20
17	19	15	20

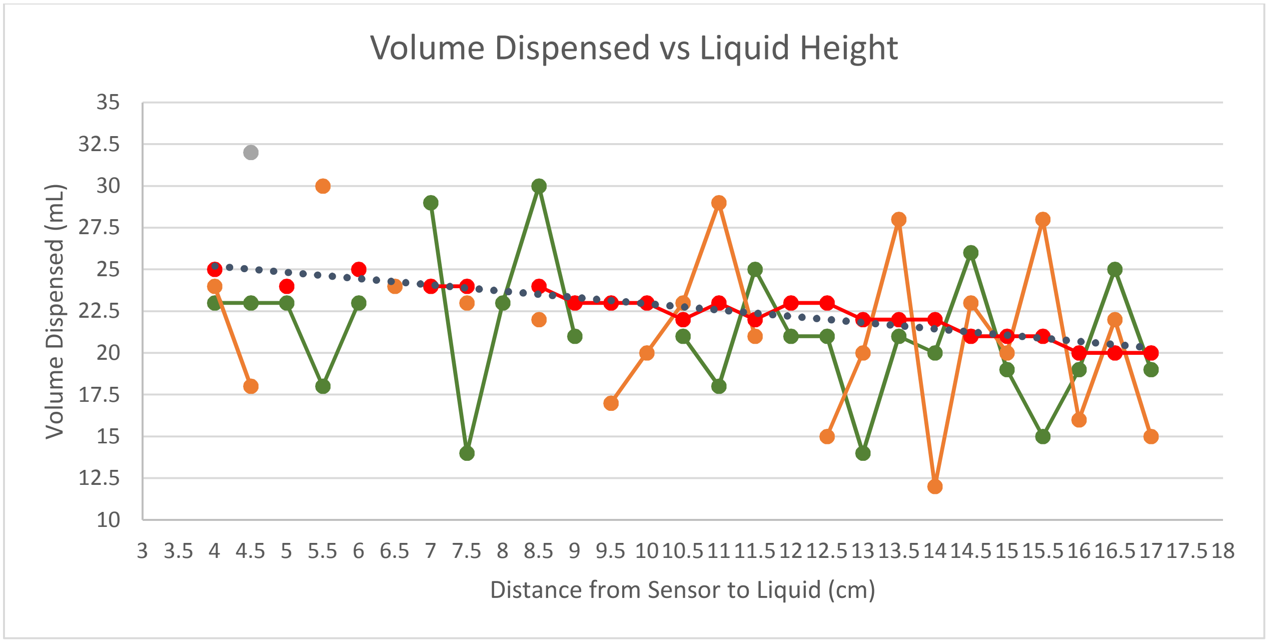


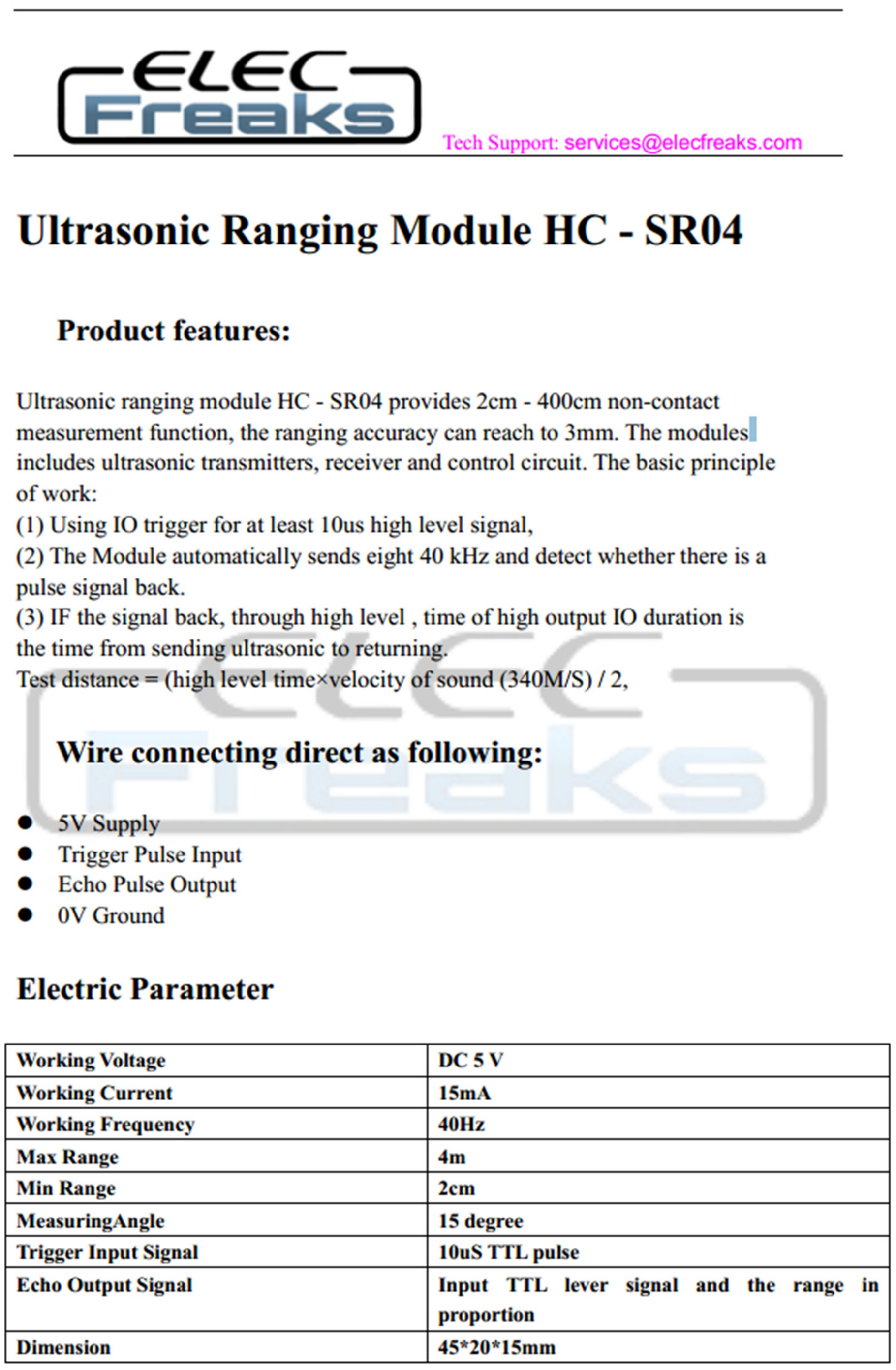
Figure E2 – Volume Dispensed in 100ms vs Distance from Ultrasonic Sensor to Top of Liquid

Table E4 – Calibration Data for Valve Durations

	Run 1		Run 2	
Distance from Sensor to Liquid (cm)	Valve Duration (ms)	Volume Dispensed (mL)	Valve Duration (ms)	Volume Dispensed (mL)
4	87	20	93	24.5
4.5	88	20.5	92	23
5	89	20	94	24
5.5	89	20.5	93	23
6	90	21	93	23
6.5	91	20.5	95	23
7	91	20	97	23.5
7.5	92	21	95	23
8	93	20.5	97	21
8.5	94	21	96	22
9	94	20	100	23.5
9.5	95	21	98	22
10	96	20	101	23
10.5	97	20	102	23.5
11	97	20	103	22.5
11.5	98	20	104	24
12	99	20	104	23
12.5	100	22	100	21.5
13	101	20.5	105	22
13.5	102	22.5	100	21.5
14	103	19.5	109	22
14.5	103	21	106	21.5
15	104	21	107	21
15.5	105	21	108	22
16	106	20	112	21
16.5	107	21	110	21
17	108	20	114	22

Appendix G: Ultrasonic Sensor Datasheet

The datasheet for the ultrasonic sensor is not hosted on the manufacturer's site, but is hosted instead at micropik.com (Ultrasonic Ranging Module HC-SR04). However, the manufacturer's product page has several other resources listed, including a User's Manual and Software Library for integration with an Arduino (HC-SR04 Ultrasonic Sensor, 2011).



Tech Support: services@elecfreaks.com

Ultrasonic Ranging Module HC - SR04

Product features:

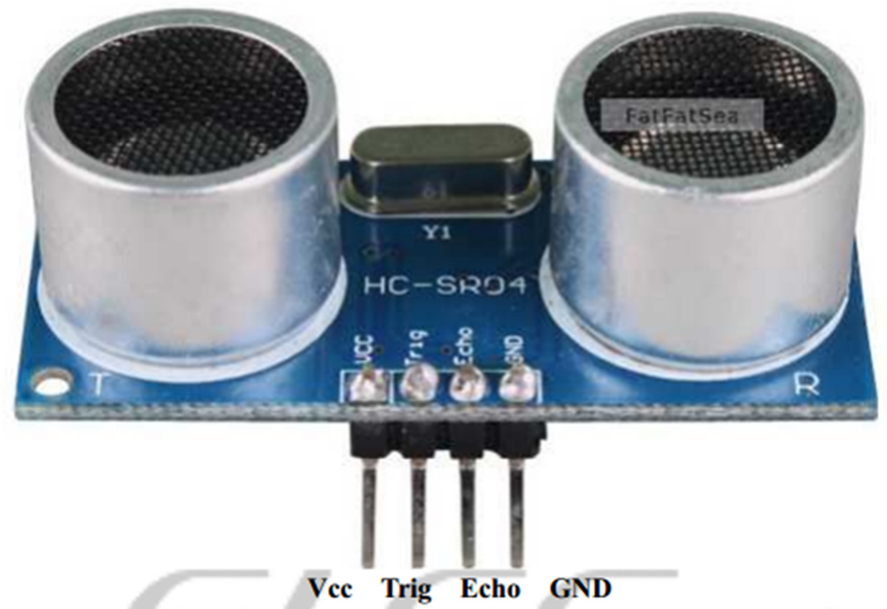
Ultrasonic ranging module HC - SR04 provides 2cm - 400cm non-contact measurement function, the ranging accuracy can reach to 3mm. The modules includes ultrasonic transmitters, receiver and control circuit. The basic principle of work:
(1) Using IO trigger for at least 10us high level signal,
(2) The Module automatically sends eight 40 kHz and detect whether there is a pulse signal back.
(3) IF the signal back, through high level , time of high output IO duration is the time from sending ultrasonic to returning.
Test distance = (high level time×velocity of sound (340M/S) / 2,

Wire connecting direct as following:

- 5V Supply
- Trigger Pulse Input
- Echo Pulse Output
- 0V Ground

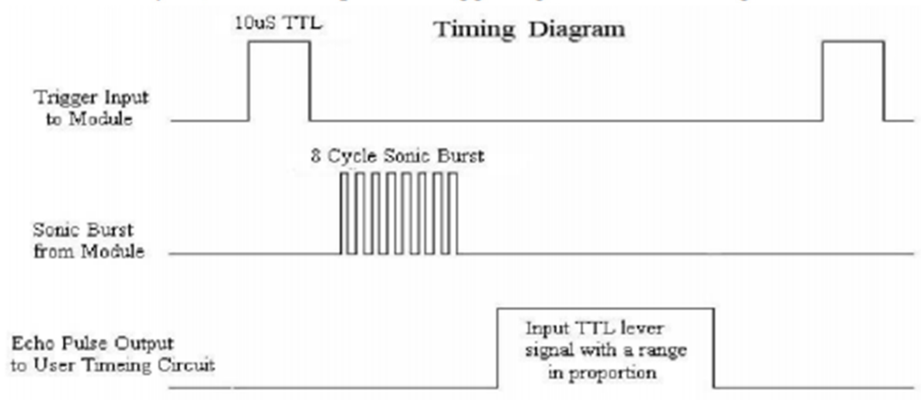
Electric Parameter

Working Voltage	**DC 5 V**
Working Current	**15mA**
Working Frequency	**40Hz**
Max Range	**4m**
Min Range	**2cm**
MeasuringAngle	**15 degree**
Trigger Input Signal	**10uS TTL pulse**
Echo Output Signal	**Input TTL lever signal and the range in proportion**
Dimension	**45*20*15mm**



Timing diagram

The Timing diagram is shown below. You only need to supply a short 10uS pulse to the trigger input to start the ranging, and then the module will send out an 8 cycle burst of ultrasound at 40 kHz and raise its echo. The Echo is a distance object that is pulse width and the range in proportion .You can calculate the range through the time interval between sending trigger signal and receiving echo signal. Formula: uS / 58 = centimeters or uS / 148 =inch; or: the range = high level time * velocity (340M/S) / 2; we suggest to use over 60ms measurement cycle, in order to prevent trigger signal to the echo signal.


10uS TTL
Timing Diagram
Trigger Input to Module
8 Cycle Sonic Burst
Sonic Burst from Module
Echo Pulse Output to User Timeing Circuit
Input TTL lever signal with a range in proportion