2019
Group 11
Eric Velez
Lance Adler
Ryan Burns
Parke Novak

Micro Manufacturing Beverage System

Senior Design 2 Documentation

Table of Contents

1.0 Executive Summary 1

2.0 Project Description 2

2.1 Motivation and Goals 2

2.2 Objectives 2

2.4 Hardware Diagram 3

2.5 Software Diagram 4

3.0 Research and Background Information 5

3.1 Similar Projects and Products 5

3.1.1 Drink Wizard 5

3.1.2 Under the Sun Drink Mixer 6

3.1.3 Smartender 6

3.1.4 KnightTime 7

3.1.5 Active Noise Cancelation Device 7

3.1.6 Conclusion 8

3.2 Stretch Goal 8

3.3 Components 9

3.3.1 Barrel Jack 9

3.3.2 Wall adapter power supply 9

3.3.3 Regulator 10

3.3.4 Capacitors 11

3.3.5 Touchscreen Lcd 11

3.3.6 UTFT Library 12

3.3.7 UrTouch Library 13

3.3.8 Real Time clock 14

3.4 Microcontroller 14

3.4.1 ATMEGA32U4 15

3.4.2 Cortex-A53 16

3.4.3 Msp430 16

3.4.4 SAM3X8E Arm Cortex M3 17

3.5 State Machine 17

3.6 FSM Library 18

3.7 Multithreading vs RTOS 18
3.8 Arduino Scheduler 19
3.9.1 CMSIS RTOS 19
3.9.1 Keil RTOS 20
3.9.1 FreeRTOS 20
3.10 Data Log 22
3.11 USB Port 22
3.11 Temperature Sensor 23
3.12 Fluid Control System Hardware 24
3.12.1 Types of valves 25
3.12.1.1 Solenoid Valve 25
3.12.1.2 Pneumatic Diaphragm Valve 25
3.12.1.3 Ball Valve 26
3.12.2 Types of motors 26
3.12.2.1 DC 26
3.12.2.2 Stepper 27
3.12.2.3 Servo 27
3.12.3 Choosing a Servo Motor 28
3.12.4 Types of Linear Actuators 29
3.12.5 Load Cell 30
3.13 Power Requirements 31
3.13.1 12 V Power Supply 32
3.13.2 5 V Supply 32
3.13.3 1.8 V Supply 33
3.12.6 Servo Control 35
3.12.6.1 Level Shifter 36
3.14 Other Fluid System Components 37
3.14.1 Kegs 37
3.14.2 Compressed Gas 37
3.14.3 Tubing 38
3.14.4 Housing 38
4.0 Standards 40
4.1 Health Standards 40

4.1.1 NSF/ANSI 61 40
4.1.2 FCC 40
5.0 Design 47
5.1 Implementation 47
5.2 Design Motivation 47
5.3 Presentation 48
6.0 Project Software Design Details 49
6.1 Threading and Multitasking 49
6.1.1 Conclusion 50
6.2 Memory Requirements 50
6.2.1 Arduino 50
6.2.2 SD Card 50
6.2.3 USB 51
6.2.4 Conclusion on Storage 51
6.3 Logging 52
6.4 Sensor Reading Time Configuration 53
6.5 Read-Write Collision 53
6.6 Logging Format 54
6.7 Remote Login 55
6.8 Conclusion on Remote Login 56
6.9 Local Login 56
6.10 Security 56
6.11 Network Login Security 57
6.12 Code Structure 57
6.13.1 UTFT 57
6.13.2 Initialize UTFT 58
6.13.2 Using UTFT 58
6.13.3 URTouch 59
6.13.4 URTouch Calibration 58
6.13.4 URTouch Initialization 60
6.13.4 URTouch Precision 61
6.13.5 URTouch Usage 61
6.13.6 Screen State 61

6.14 User Interface Pathing ... 69
6.14.0 UI Buttons ... 69
6.14.1 Login ... 70
6.14.2 Main/Start Page ... 71
6.14.3 Logs Page ... 71
6.14.4 Settings Page ... 72
6.14.6 Brew Preset Page ... 73
6.14.7 Brew New Options Page ... 73
6.15 Front/Back-End Memory Read/Write ... 74
6.15.1 Login ... 75
6.15.2 Activity Log Page ... 76
6.15.6 Brew Preset Page ... 77
6.16.0 MERN Stack ... 77
6.16.1.0 MongoDB ... 77
6.16.1.1 MongoDB Local Storage ... 78
6.16.1.2 MongoDB Remote Storage ... 79
6.16.1.3 Mongoose ... 79
6.16.1.4 Conclusion ... 81
6.16.2.0 Express ... 82
6.16.2.1 API ... 82
6.16.2.2 Routing ... 82
6.16.3.0 React ... 83
6.16.3.1 Components ... 84
6.16.3.2 App.js ... 85
6.16.3.3 render() ... 85
6.16.3.4 React state ... 86
6.16.3.5 React Redux ... 86
6.16.3.6 onComponentDidMount() ... 87
6.17.0 Axios ... 88
6.17.1 Node.js ... 88
6.18 Tentative Schematic Design ... 88
6.18.1 Microcontroller ... 89
6.18.2 12v to 5v Converter ... 90

6.18.3 5v to 1.8v Converter 90
6.18.4 16 Channel Servo Driver 91
6.18.5 Load Cell Amplifier 91
6.18.6 Touchscreen 92
6.18.7 Thermometer 93
6.19.0 Software Final Design 93
7.0 Testing 94
7.1 Hardware Testing 94
8.0 Administrative Content 111
8.1 Milestone Discussion 111
8.1.1 Senior Design 1 Milestone Discussion 111
8.1.2 Senior Design 2 Milestone Discussion 113
8.2 Budget 116
8.3 Work Division 117
8.4 Copyright 118
9.1 Appendix A: Works Cited 119

List of Figures

Figure 1: Basic Hardware Design Diagram..3
Figure 2: Basic Software design Diagram..4
Figure 3: Raspberry Pi 7" Touchscreen (with components)..................................11
Figure 4: PVC Ball Valve...25
Figure 5: Servo Motor..27
Figure 6: High-Speed Linear Actuator..28
Figure 7: Half-Bridge Wheatstone Bridge..30
Figure 8: 5V Power Supply Schematic..32
Figure 9: 1.8V Power Supply Schematic...33
Figure 10: 1.8V Power Supply PCB Layout...33
Figure 11: Servo Control Board – Servo Pinout..34
Figure 12: Servo Control Board..35
Figure 13: Level Shifter Schematic...36
Figure 14: SPI Master-Slave Connections...41
Figure 15: I2C Start/Stop protocol...42
Figure 16: I2C Data Transfer...42
Figure 17: 1-Wire Schematic..44
Figure 18: Lapped Lead Placement..45
Figure 19: Threading Explanation..48
Figure 20: SD Card Example Module..50
Figure 21: Page Flowchart for User Interface...65
Figure 22: Button Colors Used for UI..66
Figure 23: Login Used for UI..66
Figure 24: UI Design for Main Page...67
Figure 25: UI Design for Brew Page...68
Figure 26: UI Design for Brew Preset..69
Figure 27: Front-End and Back-End Relation for User Interface............................71
Figure 28: React Overview..79
Figure 29: Redux Data Flow...83
Figure 30: Tentative Schematic……………………………………………………………..84
Figure 31: Load Cell Wiring...86
Figure 32: RA8875 Driver Board Connection...93
Figure 33: Display Testing..94
Figure 34: Servo Testing Setup..95
Figure 35: Level Shifter Connections...96
Figure 36: Servo/Ball-Valve Connection...97
Figure 37: Load Cell Testing Setup..98
Figure 38: Load Cell/HX711 Connection..100
Figure 39: Load Cell Data Output and Clock...101
Figure 40: Level Shifter Connections...102

List of Tables

Table 1: Bluetooth Components……………………..9
Table 2: Regulators……………………………………………………………………10
Table 3: Voltage Regulators…………………………………………………….....10
Table 4: Touchscreens……………………………………………………………...11
Table 5: Real Time Clocks………………………………………………………....13
Table 6: FreeRTOS Items…………………………………………………………..20
Table 7: Supported Microcontrollers…………………………………………….21
Table 8: USB Ports………………………………………………………………….....22
Table 9: PTC Thermistor…………………………………………………………….23
Table 10: NTC Thermistor…………………………………………………………..23
Table 11: Servo Comparison………………………………………………………27
Table 12: Linear Actuator Comparison…………………………………………..29
Table 13: 12V Power Supply Comparison……………………………………….31
Table 14: I2C Modes………………………………………………………………….42
Table 15: C Versus KB……………………………………………………………….64
Table 16: Wire Colors…………………………………..……………………………...90
Table 17: SPI Pins……………………………………………………………..………91
Table 18: Senior Design 1 Project Milestones….……………………………….112
Table 19: Senior Design 2 Project Milestones………………………………….....112
Table 20: Part Cost Breakdown…………………………………………………117

1.0 Executive Summary

This project is to design a beverage manufacturing system catered for micro operations. With the small-batch beverage manufacturing space expanding rapidly, demand for automation in the space is growing. This system may be split into two parts for simplicity - beverage dispensing and business tracking. All aspects of the system are routed to the MCU, which will display parameters via a touchscreen interface.

This system will dispense liquid into cans or bottles proportionately. Each beverage is composed of two parts - the concentrate (which contains all functional ingredients) and water. The system will accurately and reliably dispense the correct water to concentrate ratio, creating a finished product. The system will dispense the liquids via solenoid valves controlled by the MCU. Precision is ensured via sensors underneath the cans. These sensors will ensure that the containers are filled to the specified weight. If the weight is outside of the tolerance margin, the unit is rejected.

The system will track various parameters of business operations. The MCU will automatically track batch logs and production logs. Batch logs will include date, quantity, and other details of concentrate production. Production logs will include number of units produced, number of rejected units, and other details. The system will also track critical temperatures using an array of temperature sensors. The temperatures are displayed on the touchscreen.

The project is intended for implementation in a small scale beverage manufacturing facility. Typically sensors and various other components would be spread out throughout the facility. In order to display this to the senior design panel, the project is compacted onto a small mobile unit.

2.0 Project Description

In this section why this project is doing done, some goals and the exact technical objectives are going to be answered.

2.1 Motivation and Goals

The main motivation of this project comes from need of having a machine that will mix the water and concentrate to create energy drinks to be sold to customers in Orlando/Destin area in the warehouse for SoFlo energy but without spending a large amount of money on an industrial mixing machine.

There are three things that are going to be presented to the judges as features and they are as follows:

- Temperature sensor accuracy (± 2%)
- Weight sensor accuracy (± 7%)
- Ability to use the touchscreen with relatively low latency

The goal is to successfully implement these 3 features into this project as a team.

2.2 Objectives

The main objective of this project is to eventually use it in the Flo Beverages warehouse to create energy drinks. To be able to do this, the three main features need to be focused on.

- To achieve high accuracy with the temperature sensor
- To achieve high accuracy with the weight sensor
- To achieve low latency with the touchscreen

2.3 Project Requirements and Specifications

- Mobile unit to mount project for display.
  - Must be able to hold 100 lbs of weight
  - Approximately 3' x 3' x 5'
  - Constructed out of metal and wood
  - Will contain compartment for keg storage
  - Filling station are mounted on top
  - Mobile unit are outfitted with wheels to allow ease of movement
- Interface for measurement display and control inputs
  - Touch screen display

    - MCU capable of handling multiple tasks at once
    - Temperature sensing
- Filling station
    - Station must be able to move up and down
        - Must utilize linear actuator
    - Must dispense three drinks at once
        - Will utilize 6 MCU controlled valves
    - Include sensors in base to weigh each drink
        - 3 sensors total
    - Valves will draw from pressurized kegs
    - Kegs are pressurized using a 5 lb CO_2 tank

2.4 Hardware Diagram

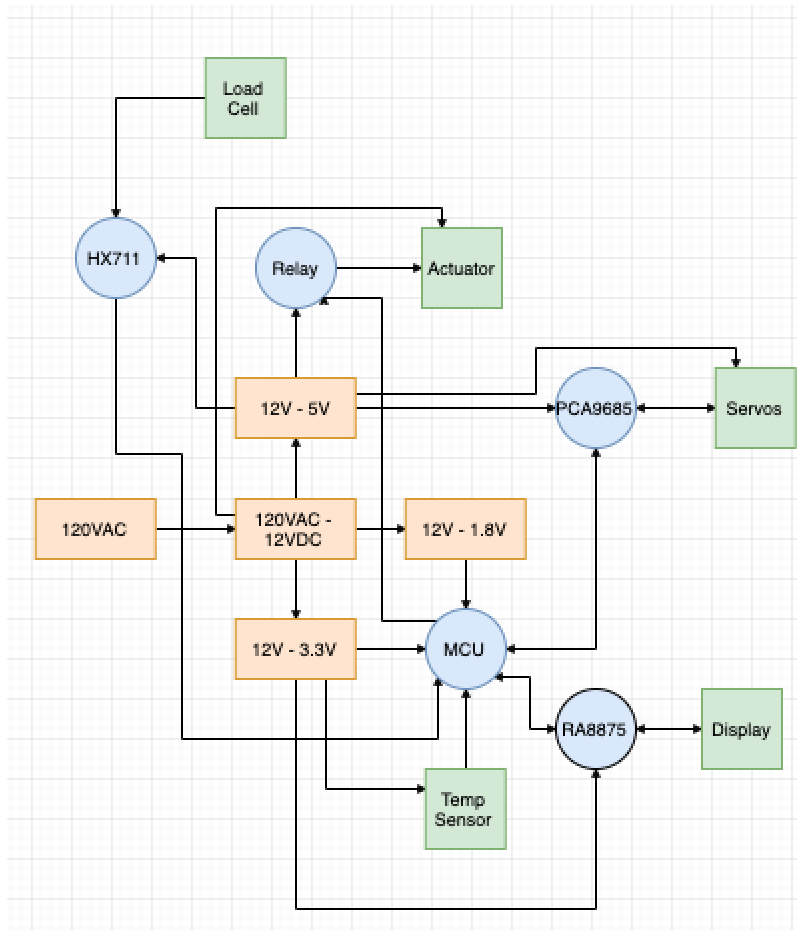


Figure 1: Basic Hardware Design Diagram

2.5 Software Diagram

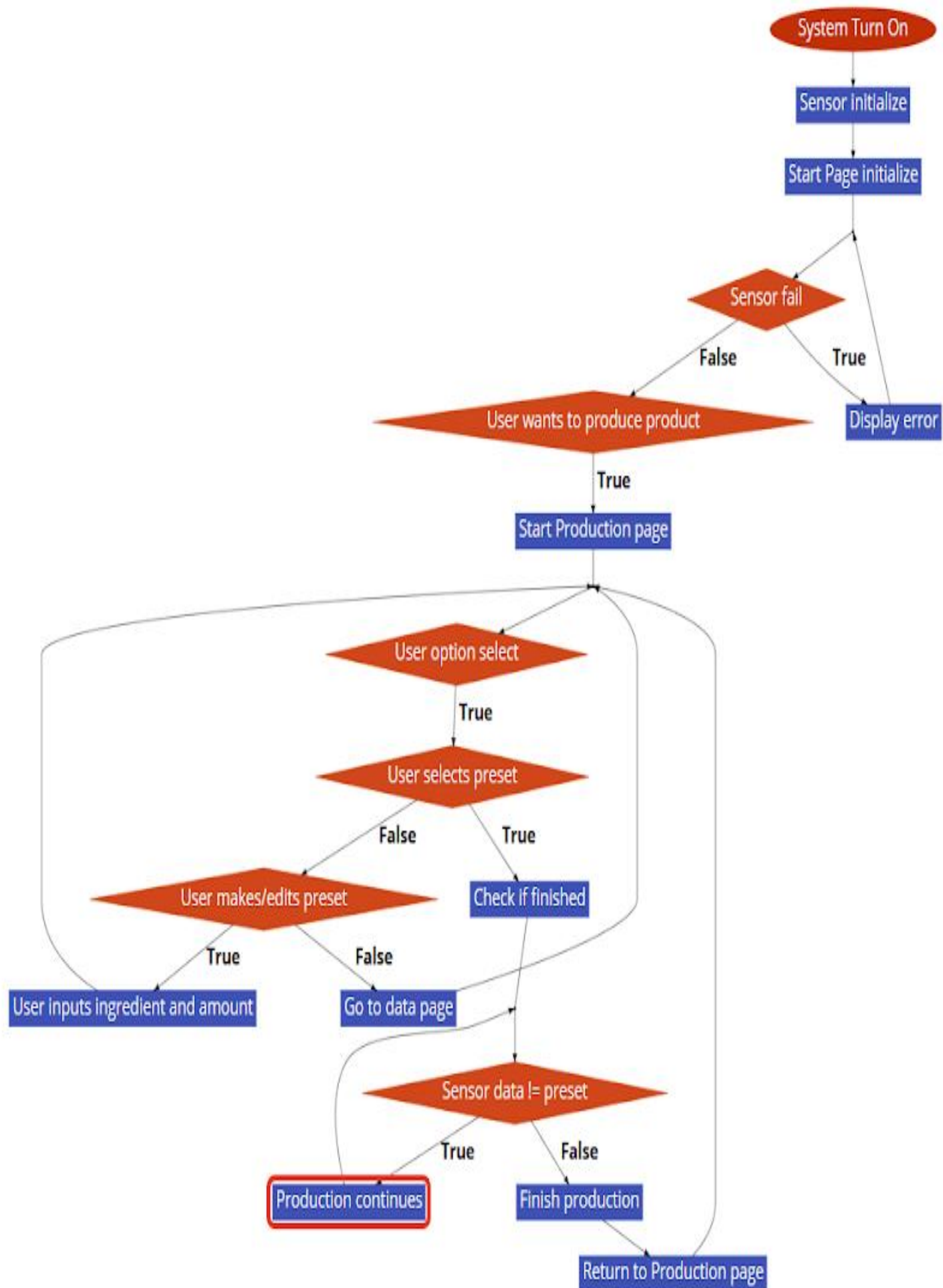


Figure 2: Basic Software Design Diagram

3.0 Research and Background Information

The major components need to be researched and this means to compare all possible reasonable options for that component. Price and other factors must be considered for each component. Other projects are going to be researched to see what they did to solve certain problems.

3.1 Similar Projects and Products

The following section contains projects that are similar to this project.

3.1.1 Drink Wizard

DrinkWizard was a previous UCF senior design project done in 2015. DrinkWizard project uses a smartphone that connects through Bluetooth to the vending subsystem to be able to mix drinks together. The idea of DrinkWizard is to be able to provide drinks to people even when your busy making food on the barbeque. DrinkWizard and this project are similar in the fact that they both mix drinks but there are two big difference between the two. The first one is the fact that the input for this project is mainly the touchscreen lcd and the main input for DrinkWizard is the application on the smartphone. Using a smartphone is a stretch goal for this project and would be very cool to implement since it'll make the mixing machine be able to be controlled remotely. The second difference is the fact that DrinkWizard is used in a much more casual setting, while this project is to be used in a more in industrial type setting (warehouse of an energy drink company).

The microcontroller chosen to be used for the Drink Wizard project was the MSP430G2553. The MSP430 is a good choice for DrinkWizard project since those students will already have experience working with a MSP430 from previous classes like embedded systems making it a bit easier for them to program the microcontroller. The msp430 requires a supply voltage ranging from 1.8v and 3.6v which is relatively low for a microcontroller. When the Msp430 is in active mode it draws 165uA which is also low for a microcontrollers. The DrinkWizard has the microcontroller asleep until an order is sent to the microcontroller for a drink order which will then wakeup up the microcontroller. The msp430 when in standby uses very small amount of power; it only draws 0.1uA of current. The time to wake up from standby mode to active mode is 5uS. This makes the Msp430 a good fit for Drinkwizard.

3.1.2 Under the Sun Drink Mixer

Under the Sun is another UCF senior design project that was done in 2013. This project is very similar to the DrinkWizard but the biggest difference is that the power comes from an array of solar cells. Getting the project powered by the sun makes the project more mobile since now it can be taken outside but the biggest downside of this is that if its overcast, cloudy, ect.. where the sun is obstructed the Under The Sun Drink Mixer can't function. Implementing solar cells into this project would be cool to do because of the added mobility, but is very unlikely to happen due to time constraints.

The microcontroller chosen to be used in the Under the Sun Drink Mixer was the Am3359 Arm Cortex A8. The Arm Cortex A8 is a fast microcontroller; it has a base clock speed of 720Mhz and can go all the way to 1Ghz. An operating system can be put onto the microcontroller such as Linux, Android, and Windows making it a very flexible microcontroller. It has a L1 and L2 cache instead of flash memory and also has a graphics accelerator making it very capable to be used with the LCD display chosen to be used in the Under The Sun Drink Mixer project.

3.1.3 Smartender

The Smartender is a portable self-contained bartender system that creates mixed drinks just like any drink that would get at a bar but is controlled by a touchscreen. The Smartender is very similar to this project since a touchscreen is being used, but the biggest difference is that the touchscreen is relatively large and that the user interface is very polished.

For this project a touchscreen of that size is not needed since this project won't be doing as many things as the Smartender does. To make the user interface for this project very polished similar to that of the Smartender is a stretch goal. The main goal of course is to make the user interface function well and work with all the hardware. The Smartender also implements a bar unlocked/locked feature where the bar can only be unlocked by someone who knows the password (the owner or an employee) or by someone who has a card that is programmed to unlocked the Smartender. This would be a cool idea to implement in this project but since it's mainly going to be only one person using it having it locked to only person is useless in this situation.

3.1.4 KnightTime

KnightTime is a sleep management system that provides data to the user conveniently as possible. It monitors pulse, movements, and temperature by using sensors that the user wears while asleep. One of the goals for KnightTime by the Ucf students is to have their senior design project to have a modular design. A modular design is one that can be broken down into separate pieces and this makes it so that the senior design project is easier to debug/troubleshoot and also makes it so it is scalable in the future (add more features).

Another important idea that was implemented in KnightTime is the system state machine. A state machine in general is a device where it can be in a set number of stable conditions based on the previous condition and the present inputs. When looking at the main state diagram for the user interface for KnightTime, it has a log in, settings, graphics, ect… which are all things that are hopefully going to be in the user interface for this project.

The state machine helps by grouping code to all function to do a certain task in a specific state in the state machine (i.e. settings, configure settings, ect…) by having the code organized in this manner it helps with the overall modularity/scalability of the project.

3.1.5 Active Noise Cancelation Device

The Active Noise Cancelation Device was a project done by a senior design group at UCF and their project is a headset that can cancel low frequency ambient noise. A big thing in the project that they considered was the price of the headset when it is finally complete to the consumer. Their logic is that it must be relatively cheap since it should be used as an alternative for other noise cancelling type of headphones.

The Active Noise Cancelation Device project used a real time operating system (RTOS) in its implementation. With a RTOS multitasking can be implemented into the project. Multitasking is of course being able to do multiple tasks at once, but the way it works in a RTOS is that multiple tasks are available to be executed but only one is processed at a time and the order in which those tasks are processed is determined by the scheduler.

There are 4 RTOSs that were considered for their project:

- CMSIS-RTOS

- FreeRTOS
- Keli RTX

CMSIS-RTOS is a RTOS that is made for ARM Cortex-M processors. FreeRTOS is a RTOS that has a scheduler that allows for multithreading, priority and deadline. The size of the RTOS is small and is compatible with many different microcontrollers. The Keli RTX supports ARM and Cortex-M series microcontrollers.

RTOS has some interesting advantages and should be in consideration for this project.

3.1.6 Conclusion

There are many similar mixing project but all of them bring different ideas to the table. The most interesting idea out of all of them comes from the DrinkWizard. The idea of using a smartphone and connecting to the mixing machine with bluetooth is very powerful and makes using the mixer much easier/convenient and is a stretch goal for this project.

The idea of making a senior design project more modular (like KnightTime) should be attempted to be implemented in this project to make it easier for the team to debug/troubleshoot and also to make it a bit easier to add other features in the future (stretch goals). Having a state machine for the user interface worked well for KnightTime and should be something that should be considered for this senior design project. If a state machine isn't implemented a RTOS should be considered. It has many advantages (multithreading, priority, deadline ect…) and may be a better choice than a state machine.

This project is being used in a more industrial setting but inspiration/ideas can be found by looking at mixing machines in a more casual setting.

3.2 Stretch Goal

The stretch goal for this project is to be able to use a smartphone to control the mixing machine. To be able to achieve this goal this project with the touchscreen needs to already be functional and there is enough time in the semester for this stretch goal to be worked on.

The program that is implemented on the touchscreen display will need to be ported to the android development environment to be used with an android smartphone. For this app to be able to communicate with the project it can be done in two ways; bluetooth or wifi. The advantage of using wifi is that the project can be controlled

within any distance as long as the smartphone with the ported app, and the project in the warehouse are both connected to wifi.

The smartphone with the ported app can connect to the project with bluetooth as well but the biggest drawback is if the user walks outside of the range of the bluetooth module, the smartphone will disconnect meaning controlling the project from a distance is impossible. Another difference to consider is the price of the two modules. The prices can be seen in the following table.

Component	**Price**
BLUETOOTH-SERIAL-HC-06	$9.94
ESP32-WROOM-32U	$4.00

Table 1: Bluetooth Components

Since the wi-fi module is more desired in this stretch goal (no constraints on how close/far you can be to the project) and isn't that much more expensive that the bluetooth module, the wi-fi module is used.

3.3 Components

The following section contains components considered and used for this project.

3.3.1 Barrel Jack

To provide power for everything in this project a DC barrel jack is going to be used. This is on the board (pcb) and is connected to al the components that need power; valves, relays, mcu, touchscreen lcd. The jack is 5.5mm in size with a 2.1mm center pole diameter.

Another option would be is to use this board on the pcb. This board takes a 6-12v input voltage and then it can either output a regulated 5v or 3.3v. The outputs can be changed from 5 and 3.3 with a simple switch on the board. Also there is an on and off switch that is very useful to completely cutoff all the power to the pcb and all its components.

3.3.2 Wall adapter power supply

The barrel jack that is going to be on the pcb, is providing power to everything in this project. The power is going to be split, high voltage (12v or so) from the wall adapter power supply is going to supply power to the replay which therefore is going to power the valves, the 12v will then be regulated to a lower voltage (5v or so) and this will power the microcontroller, sensors, and touchscreen.

To provide power into the barrel jack a wall adapter power supply is going to be used to which will convert AC to dc and regulate the voltage and current so that it is useable for this project.

There are multiple options for power supplies and they are as follows:

	Regulated Voltage	**Regulated Amperage**	**Cost**
XINKAITE	12V	2A	$8.98
Sparkfun	9V	650MA	$5.95

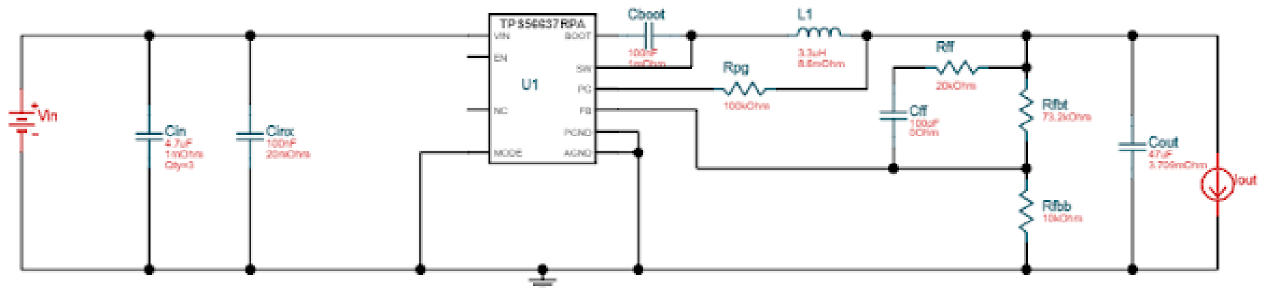
Table 2: Regulators

Since the valves picked for this project require 5V to work, the XINKAITE power supply is used. These power supplies have power cables built into them so they don’t need to be bought separately thus saving money.

3.3.3 Regulator

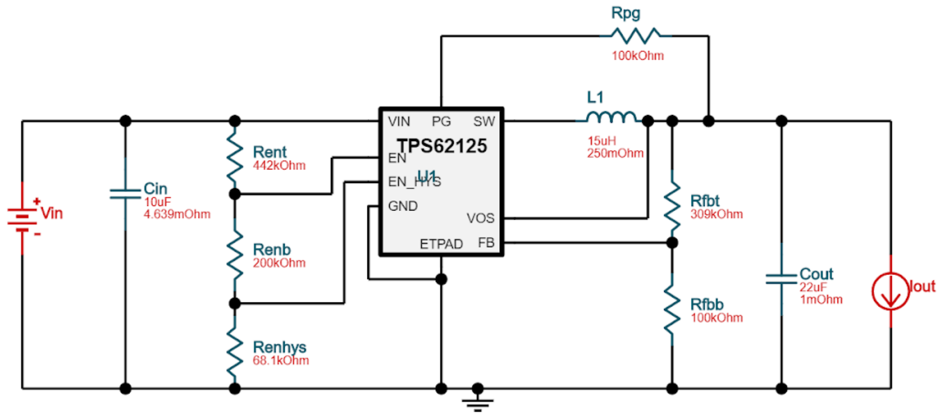
The 12v from the power supply needs to be stepped down to 5v and 3.3v. Linear regulators could work but they created too much heat so buck converters are a better option.

A 12v to 5v buck converter looks like the following:



Vin	8V-28V
Vout	5V
I out	6A

A 12v to 3.3v buck converter looks like the following:



Vin	4V-17V
Vout	3.3V
I out	0.1A

3.3.4 Capacitors

Adafruit recommends using a 10uF electrolytic capacitors on both the input and output of the regulator to help filter excess noise. This is implemented in this project.

3.3.5 Touchscreen Lcd

The touchscreen lcd for this project takes all the inputs from the user and performs a corresponding action.

There are multiple options for a touchscreen display that could be used in this project. Those options are displayed in the following table.

Touchscreen Display	Screen Size	Cost	Resolution
Raspberry Pi	7”	$74	800x480
Adafruit	3.2”	$29.95	240x320
Adafruit	4.3”	$62.44	?

Table 4: Touchscreens

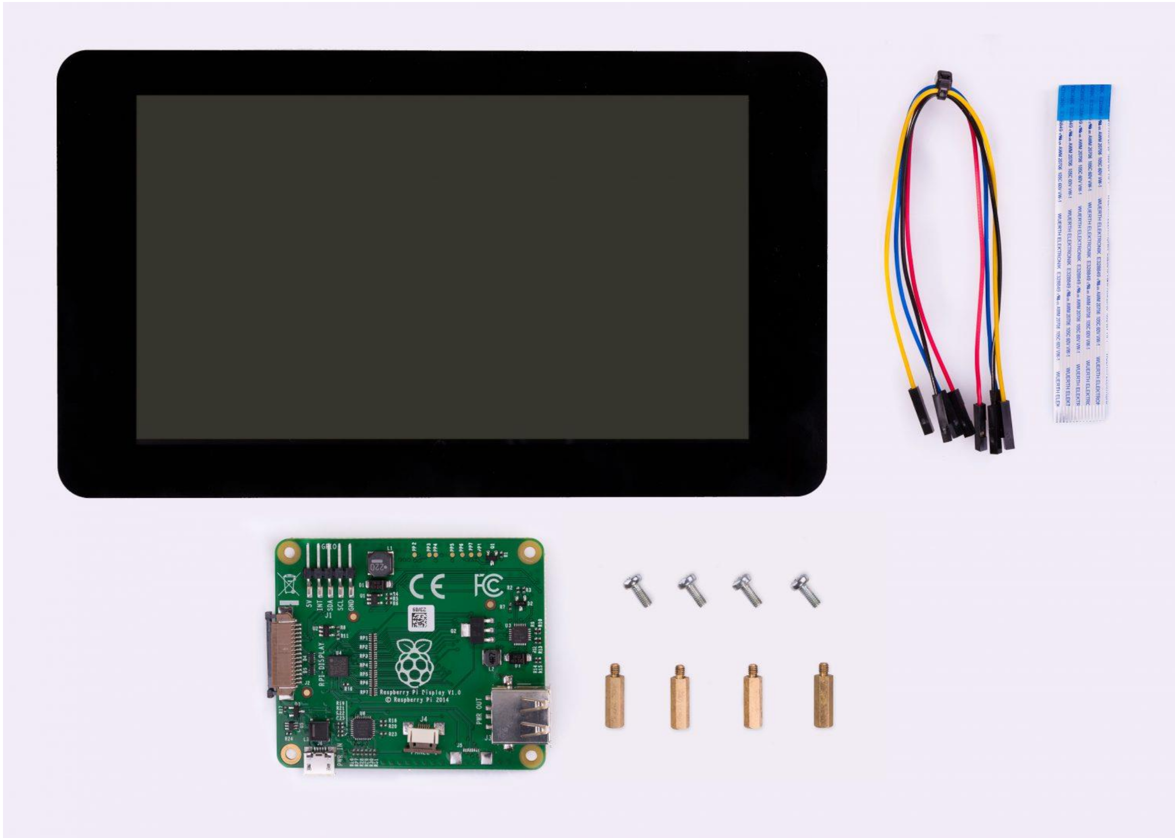


Figure 3: Raspberry Pi 7" Touchscreen (with components)
Image Courtesy of RaspberryPi.org

The Adafruit 3.2'' touchscreen supports both resistive and capacitive touch. The way this touchscreen is able to support both type of touches is because of the 4 resistive touch screen pads are glued on top of the capacitive touch. The resistive touch can be used by using the resistive touch pins on the touchscreen. Capacitive touch is the more desirable type of touch out of the two since it's the most accurate (modern phones use this type of touch method). The capacitive touch can be used by using the capacitive touch pins on the touchscreen.

The other two touchscreen displays only use capacitive touch but the biggest difference aside from that is the cost, and resolution. Since this project is actually going to be used in a warehouse it is important to make the touchscreen as practical as possible. A touchscreen that isn't too big would be harder to use since there be more scrolling involved to get to tasks, so a larger touchscreen would be more desirable. With that being said a raspberry pi touchscreen areused for this project.

3.3.6 UTFT Library

One of the microcontrollers being considered for this project is the SAM3X8E Arm Cortex M3. It is found on the arduino due and because it was used in the duo, it is compatible with the Utft library.

This library will make it possible to have a functioning graphical user interface that the user can use this project with. The following functions are important to point out:

- smallFont/BigFont/SevenSegNumFont
- print(st, x, y [, deg])
- setColor(r,g,b)
- fillRect(x1, y1, x2, y2)

The three fonts that are provided already in this library all look good and more fonts can actually be found by going to the resources section in the website that this library is found on. If none of the extra UTFT fonts are satisfactory the font maker on the website can be used.

The print function has 4 arguments in which only the first three are required. The st argument is the string that is going to be printed on the screen. The x argument is the x-coordinate of the left corner of the first character. The y argument is the y-coordinate of the upper left corner of the first character. The last optional argument, deg, rotates the text from 0-359 degrees and rotates the text by the upper left corner.

The setColor function has 3 arguments. The r argument is the red component of an rgb value (0-255). The g argument is the green component of the rgb value (0-255). The b argument is the value component of an rgb value (0-255). This function is used to set the color for all of the draw, fill and print commands.

The fillRect function has 4 arguments. The x1 and y1 arguments are the respective x and y coordinates of the start corner. The x2 and y2 arguments are the respective x and y coordinates of the end corner. This function can be used with the set color function to create a colored book anywhere on the lcd screen that the user can interact with.

3.3.7 UrTouch Library

The UrTouch Library is an arduino library that is meant to be used in conjunction with the UTFT library. This makes it so that the screen can be interacted with by the user and actually function. The following functions are important to point out:

- dataAvailable()
- read()
- getX()
- getY()

These four functions work very closely with each other. The dataAvailable function is used to check and see if new data from the touch screen is waiting. The read function reads waiting data from the touchscreen. If the dataAvailable function is true then the getX and getY functions can be used. The getX function will get the x-coordinate from the last position touched on the touchscreen and the getY function will get the y-coordinate.

3.3.8 Real Time clock

Since a log of temperature and weight readings from the respective sensors are going to be created, having a real time clock to see when a reading was made (time and date) is going to make the data log more practical/useful.

There are multiple options for a Real Time Clock for this project and they are displayed in the following table.

Real Time Clock	**Price**
BU9873F-GTE2	$0.97
RV-3149-C3 32.768kHz OPTION B TA QC	$2.53

Table 5: Real Time Clocks

The first option is very cheap and as able to keep time and date and is connected to a cpu via I^2C. It has automatic leap year recognition (up to year 2099) and also can be adjusted by ±30 seconds. The second option is very similar to the first option but the biggest difference is that it's made to be easily integrated with a development board. Even though a development board isn't being used for the final project (only for testing and debugging) since a microcontroller that is used in an Arduino is being used, it makes implementing the second real time clock much easier for this project. It's also important to note the second option is able to still keep time and date even if the whole project is off since it's powered with a small 3v button cell battery.

3.4 Microcontroller

The hardest thing to pick for this project is the microcontroller and a lot of things need to be considered when picking one that is the best fit for this project.

	ATMEGA32U4	**Cortex-A53**	**MSP430G2553**	**SAM3X8E Arm Cortex M3**
Price	$4.09	$134.62	$2.46	$9.97

Supply Voltage (Min)	2.7V	0.9V	1.8V	1.62V
Suuply Voltage (Max)	5.V	1.025V	3.6V	1.95V
Current Draw	1.2mA	50uA	165uA	700uA
I/O Pins	26	6	24	103
Clock Speed (Max)	16MHz	1600MHz	16MHz	84Mhz
Data Bus Width	8 Bit	64 Bit	16 Bit	32 Bit
Flash Memory	Yes	No	Yes	Yes
Flash Memory Size	32 Kb	N/A	16 Kb	512 Kb
L1 Cache	No	Yes	No	No
L1 Cache Size	N/A	32 Kb	N/A	N/A
L2 Cache	No	Yes	No	No
L2 Cache Size	N/A	1 Mb	N/A	N/A
SRAM	Yes	No	Yes	No
SRAM Size	1 Kb	N/A	512 B	N/A

3.4.1 ATMEGA32U4

The Atmega is a popular microcontroller found in many Arduino development boards such as the Arduino Uno, Arduino Leonardo ect… This particular atmega microcontroller is found in the Arduino Leonardo and what makes it different from the other arduino boards is the fact that it has a built in USB communication so a secondary processor isn't needed.

A lot of the Atmega microcontrollers are also used in Arduino development boards and therefore an Arduino bootloader can be used. Since the microcontroller is being bought separately and is not actually part of a development board a bootloader won't be installed onto the microcontroller already. A raspberry pi development board (the Arduino ide can be used with the raspberry pi) is being used to debug the project before the actual implementation with the pcb, if an Arduino bootloader is used with the microcontroller it'll make it easier to port the code written on the development board to be used with the final project.

There are two ways that the Arduino bootloader can be implemented with the Atmega microcontroller. The first way is using an Arduino board with an AVR

programmer and the second way is using a breadboard and an AVR programming adapter, but that way is much more complicated/time consuming so the Arduino bootloader is going to be installed the first way.

On the Arduino ISP the ATMega chip is placed, and then the ArdinoIsp is connected on top of an Arduino development board and then that board is connected to a computer through usb where the bootloader then can be uploaded to the microcontroller.

Since the Atmgega can use the arduino bootloader this means it can use the arduino scheduler multithreading library. The Atmega has a decent clock speed on at 16 Mhz, good flash memory at 32 Kb and a decent supply voltage range at 2.7v and 5v.

3.4.2 Cortex-A53

The Cortex-A53 is the microcontroller found in the raspberry pi 3 mode b+. This microcontroller is crazy powerful and is capable of a lot and can be thought of more of a mini cpu. An operating system can be installed such as Raspbian which is a variation of the Linux operating system. Using an operating system with a microcontroller is very useful in being able to do more complicated things such as running multiple processes or functions at the same time, schedule parallel tasks, file system management ect... This project is not going to be at that level of complexity so having an operating system would be overkill for this project. This microcontroller is very capable and there is a big price increase compared to the others that warrants that level of capability. As nice as this microcontroller is, it is out of the budget for this senior design project.

3.4.3 Msp430

The Msp430 is a microcontroller that everybody in this senior design group is familiar with because of taking embedded systems. This would make programming the microcontroller a bit easier due to the prior experience. The Msp430 is a very low power microcontroller. The min supply voltage is 1.8v (lowest out of all the microcontrollers in consideration for this project) and the max is 3.6v. When the msp430 is in standby mode it only draws 0.1uA of current and is able to switch to active mode from standby mode within 5uS.

This makes the msp430 a great option for this project due to the low power usage and also the familiarity on how to program/use the msp430.

3.4.4 SAM3X8E Arm Cortex M3

The SAM3X8E Arm Cortex M3 microcontroller is found in the Arduino Due. Since it is used in an arduino, this means it can use an arduino bootloader and therefore be used with the arduino ide. The arduino ide will make programming the microcontroller significantly easier to program. Also since for this senior design project multithreading is something that is going to be attempted, the arduino scheduler multithreading library can be used.

This microcontroller can have a USB port that allows for serial communication (CDC) over USB. It also makes it so the microcontroller can emulate a USB mouse or keyboard is attached. Another feature of this microcontroller is that it can act as a USB host for connected peripherals such as mice, keyboards, smartphones, ect…

This microcontroller is very powerful and affordable. This will make it be able to communicate with the relay, sensors and touchscreen all at once and be able to do it with very low delay since the microcontroller has a relatively high clock speed. If either the msp430 or atmega microcontroller were to be used instead, the project would still be able to function but there would be some kind of noticeable delay, making this microcontroller a much better choice.

With all things considered, the SAM3X8E Arm Cortex M3 microcontroller areused for this senior design project.

This microcontroller is programmed with the Atmel Ice through Atmel Studio, and requires a jtag port to communicate with the microcontroller.

3.5 State Machine

Implementation of state machines in modern day electronics is very common, specifically finite state machines. For this project a state machine would increase the modularity of the code since now the code is grouped up into the different states that are in the state machine. This increases the scalability of the project and will make it a lot easier to implement another into the code in the future.

A state machine is a device that can have a set of stable conditions and depends on its previous condition and present value of inputs to determine its state.

There are two popular types of finite state machines that can be considered for this project:

- Mealy Machine
- Moore Machine

A moore machine has its output values determine only by its current state. So for example there could be a system that dispenses cookies. The system will dispense cookies only after the purchase button is pressed (this is something that would never be found in the real world since the cookie could be obtained without inserting money).

A mealy machine has its output values determine both by its current state and the current inputs. So for example there could be a system that dispenses cookies and the system will only dispense cookies after the purchase button is pressed and money is inserted (this is a much more realistic to what happens in the real world).

3.6 FSM Library

The FSM library is a library for the arduino that makes a state machine relatively easy to be implemented in the arduino ide environment. The FSM acts as a manager that organizes a set of states and the transition between the states. The transition can occur because some conditions are met in the state either internally or externally and this transition can either occur right away or during the next cycle.

A finite state machine can be created with the following function:

FiniteStateMachine(State& current) or FSM(State & Current)

All the states that are going to be used within the finite machine must be initialized to a variable and can be done with the following function:

State(enterFunction, updateFunction, exitFunction)

To be able to transition between states the following function can be used:

void transitionTo(State& next)

These three basic functions that are the backbone to make the finite state machine work in the arduino ide environment.

3.7 Multithreading vs RTOS

For this project a microntroller with only a single core is going to be used. Since it only has one core it will only be able to process one task a time. To make the

program more efficient there are two different approaches that can be taken. The first is to have the program have a real time operating system (rtos) and the other is to have the program use multithreading.

For the multithreaded approach there can be a thread for all things related to the sensors another for the user interface, another for valves ect… In the thread all the programming related to that thing are in there. This will help to help organize the code a bit and also help to make the whole program more efficient since the program is rapidly switching between tasks. This is far more efficient than having one large loop where a lot of time is wasted where the decompiler is going through the same lines of code over and over again in the loop. Fortunately the microcontroller chosen to be used in this project is compatible with an arduino bootloader, and since the arduino ide can be used with it, the arduino scheduler library can be used to implement the multithreaded approach.

For the rtos approach it's similar to multithreading but the biggest difference is that there are priorities associated to the different tasks. This has the same effect of being able to switch between tasks fast therefore being faster than having a large loop.

3.8 Arduino Scheduler

The Arduino Scheduler is a library that allows SAM and SAMD architectures to run multiple functions at the same time. This makes it so that tasks can occur at the same time without interrupting each other.

The library consists of only two functions:

- startLoop()
- yield()

The startloop adds a function to the scheduler that will run concurrently with the loop function. The yield passes control to other tasks when called to in the code. This should be used on function that will take a decent amount of time to complete.

With this library it will make it relatively easy to have multiple functions occur at the same time on the microcontroller. It is possible to take input from the user on the lcd to tell which valve to open but at the same time also log what is happening into a data log that can be viewed later by the user.

3.9.1 CMSIS RTOS

CMSIS RTOS is a RTOS that is designed for Cortex-M processor devices so for the microcontroller that was chosen for this project it would be compatible with this RTOS. The CMSIS RTOS has the following features:

- Thread management
- ISR (Interrupt Service Routines)
- Mutexes and Semaphores
- OsDelay
- osWait

Threads can be defined and created within CMSIS making it relatively easy to control all threads used in the kernel. An Interrupt Service Routine is a process in which an active process is interrupted to then complete the task of the interrupt. Mutex and Semaphore and both types of data types that are common in operating systems. A Mutex makes a resource to only be able to available to one thread, whereas a Semaphore can be accessed by multiple.

The osDelay function takes a thread created by the user into a waiting state for an amount of time set by the user. The other function osWait waits for a certain event to happen for something to happen to a certain thread.

3.9.1 Keil RTOS

This RTOS has multiple version that each support different hardware. They are as follows:

- MDK-Arm Middleware
- RTX Real-Time Operating System
- RTX51 Tiny
- ARTX-166 Advanced RTOS
- RTX166 Tiny

Both the MDK-Arm Middleware and RTX Real-Time Operating System are both compatible with the Cortex-M processor chosen for this project.

Unlike CMSIS RTOS to have all the features available to the developer, a license needs to be bought.

3.9.1 FreeRTOS

FreeRTOS is a type of RTOS design specifically meant to be run on microcontrollers but not limited to. FreeRTOS provides real time scheduling

functionality, inter-task communication and synchronization. For this project is would be a good idea to have the user interface have a low priority and then all the sensor measurements a high priority.

FreeRTOS is very small and only requires 5 to 10Kbytes of ROM space.

Item	**Bytes**
Scheduler	236
Queue	76 Bytes + queue storage
Task	64

Table 6: FreeRTOS Items

Some of the supported microcontroller companies/families are as follows:

Company	**Family**
Altera	Cyclone Vsoc, Nios II
Armv8-m	Arm Cortex M33
Atmel	Arm Cortex M7, SAM3 (ARM Cortex M-3), Sam4 ARM Cortex M-4, …
Cadence	Tensilica Xtensa
Cortus	APS3
Cypress	PsoC 5 ARM Cortex-M3
Freescale	Kinetis ARM Cortex-M4
Infineon	TriCore, XMC4000 (ARM Cortex-M4F)
Fujitsu (Now Spansion)	FM3 ARM Cortex-M3, 32bit
Luminary Micro/ TI	All Luminary Micro ARM Cortex-M3 and ARM Cortex-M4
Microchip	PIC32MX, PIC32MZ, PIC32MZ EF, PIC24, PIC24EP, dsPIC
Microsemi	MiFive (RISC-V), SmartFusion, SmartFusion2
NEC (now Renesas)	V850 (32bit), 78K0R (16bit)
NXP	VEGAboard (RISC-V), LPC1500 (ARM Cortex-M3), LPC1700 (ARM Cortex-M3)
Renesas	RZ/A1 (ARM Cortex-A9), RX700 / RX71M, RX600 / RX64M
SiFive	RISC-V RV32
Silicon Labs	EFM32 Gecko (Cortex-M3 and Cortex-M4F), 8051 compatible microcontrollers
Spansion	FM3 ARM Cortex-M3, 32bi
St	STM32 (ARM Cortex-M0, ARM Cortex-M7, ARM Cortex-M3 and ARM Cortex-M4F), STR7 (ARM7), STR9 (ARM9)
Ti	RM48, TMS570, ARM Cortex-M4F MSP432, MSP430, MSP430X, SimpleLink, Stellaris (ARM Cortex-M3, ARM Cortex-M4F)
Xilinx	Zynq, Zynq UltraScale+ MPSoC (64-bit ARM Cortex-A53 and 32-bit ARM Cortex-R5), Microblaze, PPC405 running on a Virtex4 FPGA

Intel/x86	IA32 (32-bit flat memory model), Quark SoC X1000 (32-bit flat memory model), any x86 compatible running in Real mode only

Table 7: Supported Microcontrollers

Fortunately the microcontroller chosen to be used for this project is supported, so FreeRTOS can be used.

To install FreeRTOS onto the microcontroller Atmel Studio 7 must be used. FreeRTOS must first be downloaded off the FreeRTOS website and then added into the Atmel Studio project as a module.

3.10 Data Log

For the ARM Cortex M-3, one of the features that it has that it makes special in the arduino due, is the USB host capabilities. There are two main types of USB devices.

- USB slave
- USB host

A USB slave is a device that must plug into a USB host in order to function. Some examples of common USB slaves are flash drives, web cam, printer, scanner, ect... These devices are inserted into the host and respond to the host. The USB slave will never initiate communication between these two devices that is the job of the USB host.

A USB host will always initiate communication on the bus. If for example a flash drive is plugged into a usb port, a file will only be saved onto it if the USB host decides to save a file onto it. Since for this project a data log is going to be created that tracks certain parameters, it’s highly desired to have the ability to view this later on a computer. To be able to save the data report onto USB, a microcontroller that has USB host functionality is needed and an actual USB port. The microcontroller for this project supports USB host so that is covered all is needed now is an actual USB port.

3.11 USB Port

To be able to write the data report to a flash drive a USB port is going to be needed. The following USB ports can be used for this project:

USB	Price
BOB-12700	$4.50
USB Type-A Jack	$0.75

Table 8: USB Ports

For the first USB connected this is a breakout board. This is a usb on a board that is meant to be screwed down on a surface but since a custom pcb is going to be made this is not a good choice. The second USB is the same act USB as the first (type-A jack) but without the breakout board. This makes it so that the USB port can be part of the pcb without any issue.

The USB has 4 connections. GND, D+, D- and VCC. Ground is connected to the ground for the whole circuit involving the microcontroller and VCC is going to be the voltage that is used for the microcontroller so it should be connected. The D+ and D- are connected directly to the microcontroller for the USB to be effective in the project.

3.11 Temperature Sensor

There are different types of temperature sensors and they are as follows:

- Thermocouple
- Thermowells
- RTD (Resistance Temperature Detectors)
- Thermistor

A thermocouple is a type of temperature sensor that has two dissimilar metal wires that are joined together at a junction and temperature change causes a slight change in voltage which can then be interpreted after the fact as a certain temperature. Thermocouples are cheap, small, and have a wide temperature range. A thermocouple is flexible since it can be used for both air and water applications. This could work for this project since a temperature sensor is needed to be in water.

A thermowell allows RTD probes, thermocouple probes and thermometers to be inserted and removed to measured temperature without stopping the process. These are usually found in industrial environments. This wouldn’t be too useful for this project since the temperature is needed to be constantly regulated and not checked every so often manually. Not to mention thermowells are relatively expensive.

An RTD otherwise known as Resistance Temperature Detectors, are able to measure temperature due to platnium, copper, nickel ect… having a well-defined resistance versus temperature relationship (granted the RTD is within the operating temperature). The material for the RTD (platinum, copper, nickel, ect…) is going to be in a wire and wrapped around a ceramic or glass core and since all the parts of the RTD are fragile they are housed in protective probes. RTDs are

very accurate for temperatures below 600 degrees C which shouldn't be a problem for this problem, but they are expensive.

A thermistor is a resistor that is dependent on temperature. There are mainly two different types of thermistors:

- NTC (Negative Temperature Coefficient)
- PTC (Positive Temperature Coefficient)

A PTC also known as Positive Temperature Coefficient is a thermistor in which resistance increases as temperature rises. PTC thermistors are commonly used for circuit protection.

PTC Thermistor	**Resistance Min**	**Resistance Max**	**Min Operating Temp.**	**Max Operating Temp.**	**Price**
PTCSL03T081DT1E	20 Ohms	120 Ohms	-40C	165C	$0.61
PTCSL03T121DT1E	20 Ohms	120 Ohms	-40C	165C	$1.08

Table 9: PTC Thermistor

These two thermistors are very similar to each other and the only noticeable difference between the two is the price.

A NTC also known as Negative Temperature Coefficient is a thermistor in which its resistance decreases as temperature rises. NTC thermistors are commonly used for temperature measurement and are used for this project.

NTC Thermistor	**Tolerance**	**Resistance**	**Min Operating Temp.**	**Max Operating Temp.**	**Price**
NKI10NF103C1R5E	5%	10K Ohms	-40C	190C	$5.27
NKI100NF103C1R1E	1%	10k Ohms	-40C	190C	$5.93

Table 10: NTC Thermistor

Since high accuracy is highly desired for this project the NKI100NF103C1R1E thermistor are used.

3.12 Fluid Control System Hardware

For an industrial beverage filler, reliability and accuracy are crucial. Just a small deviation in the systems performance and the product are ruined. Therefore, much consideration was put into finding the correct components. The system dispenses fluids from kegs pressurized at 10 psi.

Since the fluids are under pressure, there is no need to use a displacement pump. Instead, valves used to control fluid portioning. The system dispenses into 3 containers at once. Each container is filled with carbonated water and concentrate from kegs, therefore 6 total valves are needed.

3.12.1 Types of valves

Valves are used to control when and what ingredients will flow to the output line. These are mostly controlled by the motors, but it is important to choose which valve will work best with what is expected for this project.

3.12.1.1 Solenoid Valve

The most common and most available type of electronically controlled valves are solenoid valves. The two most important factors to take into consideration when choosing a valve are reliability and price. Three of the six valves carry concentrate through them. This is where reliability truly becomes a concern. Valves can typically handle water without issues, but the concentrate is a more viscous and contaminated medium.

Undissolved solids in the concentrate can easily clog valves and render them useless. The undissolved solids will gradually clog the valve, and slowly alter the amount of fluid dispensed. Solenoid valves are especially susceptible to undissolved solids, since the valves operation depends on the suction that comes from a small hole in the rubber plunger. Solenoid valves may work well for a non-industrial application, given their low price point and availability, but low reliability will keep them out of serious contention for this project.

3.12.1.2 Pneumatic Diaphragm Valve

Diaphragm valves are controlled the same way as solenoid valves, and require a solenoid in order to controlled electronically. The main difference between diaphragm and solenoid valves is the diaphragm mechanism is actuated using compressed air from an external source. The external source is typically an industrial compressor. In order to activate the valve, a solenoid is excited which allows the compressed air to pass, opening the valve.

These valves are extremely reliable, and can handle all sorts of media including corrosive. The downside to these valves is that they are extremely expensive, costing approximately $500 per valve. The valves also require an industrial air compressor, which typically cost thousands of dollars. These valves would work well with this project, but unfortunately due to their high cost they cannot be used.

Figure 4: PVC Ball Valve
Image Courtesy of United States Plastic Corp.

3.12.1.3 Ball Valve

Ball valves are among the most basic plumbing switches. Ball valves themselves are purely mechanical devices that require relatively low torque to turn off/on. Ball valves are exceptional candidates for this project, they can handle almost and kind of media and have a low cost. However, the major issue with ball valves is that they are strictly mechanical devices.

There are motor controlled ball valves available, but their cost are comparable to that of the diaphragm valves. Motor controlled ball valves purchased on the market are typically bulky, which limits their use in a project such as this. For this project, ball valves are used, but in order for them to be used electronically they are coupled with an electrically controlled component.

3.12.2 Types of motors

The motors will act as the electronically controlled component responsible for controlling the ball valves. The motors used must be cheap, reliable, and produce enough torque to open/close the ball valve. There are three types of motors that are candidates for this project.

3.12.2.1 DC

The first candidate is a generic DC brush motor. These motors are very common and come at a low cost. They are capable of high RPM and can rotate 360 degrees.

The downside to these motors is that they do not have precise positioning, which is essential to this project. They also do not provide positioning feedback.

3.12.2.2 Stepper

Stepper motors are similar to brushless DC motors except that the motor stator has grooves which are locked to the motor poles. This ensures that the motor moves in discrete steps versus the brushless motor which rotates continuously. Stepper motors typically have high torque capability as well, which is needed in this project. Steppers have many teeth in the gears and this allows for extremely precise positioning.

The downside to these motors is that they are bulky and draw a large amount of power. They also do not provide feedback, so if the motor makes an error in rotation, the software will not be aware. The torque required for this project is low, so the chances that this motor will make an error while rotating is extremely small. These motors would be perfect for this project, but they also have a relatively high price and their bulkiness makes them difficult to work with.

3.12.2.3 Servo

Servos operate similar to steppers. They have many gears which allow them to generate a high torque output. They typically rotate slowly, and usually have an axis of rotation limited to 180 degrees or less. Due to the many gears, they can make precise movements similar to the stepper motor. Servo motors typically have three wires. One wire is GND, one is VCC, and one is for data.

The downside to servo motors is that the do not have positioning that is as precise as a stepper motor. They also do not provide as much torque as a stepper motor. However, a distinguished difference is that servos provide positioning feedback to the software, which make them ideal for this project. Since the torque and positioning required for this project are not substantial, servo motors areable to do the job.

Figure 5: Servo motor
Image Courtesy of Sparkfun Electronics

3.12.3 Choosing a Servo Motor

When choosing a servo, price, torque, voltage, and quality were taken into consideration. The relevant information for the servos considered is listed in Table 1 below.

Servo	FEETECH 6V 6kg.cm Analog Servo	HS-422 Standard Deluxe Servo	MG996R High Torque Metal Gear Dual Ball Bearing Servo
Price	$12.95	$18.95	$3.70
Torque	5 kg·cm - 6 kg·cm	3.3 kg·cm - 4.1 kg·cm	9.4 kg·cm - 11 kg·cm
Voltage	4.8 V - 6.0 V	4.8 V – 6.0 V	4.8 V – 7.2 V
Current	160 mA – 190 mA	8 mA – 150 mA	500 mA – 900 mA
Speed	0.18 s/60º - 0.16 s/60º	0.21 s/60º - 0.16 s/60º	0.17 s/60º - 0.14 s/60º
Gears	Plastic	Plastic	Metal
Dimensions	40.8mm x 20.1mm x 38mm	40.6 x 19.8 x 36.6mm	40.7 x 19.7 x 42.9 mm

Table 11: Servo Comparison

The chosen servo is highlighted in the table above. The MG996R beat the other servos in almost every category. The servo is tremendously cheaper than the

others. This is especially important considering there are6 servos used in this project. Also, the MG996R has almost double the torque than the others. Given that the servos will have a load attached (the ball valve), having a high torque output is important for reliability. Even if the servo is able to move the valve, if it does not have a suitable amount of torque it may prematurely fail. The operating voltage between the servos is relatively the same, so this category isn't of concern.

The operating current is the category where the MG996R falls behind. The MG996R draws nearly four times the amount of the other servos. If the system was being powered by a battery, then this would be a huge concern. Given that the power for this system comes from an outlet, this current draw disparity isn't very important. The MG996R's speed is higher than the other, even given the its high torque output. The metal gears of the MG996R give it an additional edge with regard to quality. The size of all the servos is relatively the same, so this category isn't of any concern. Given all this information, the MG996R is clearly the servo that should be used for this project.

Figure 6: High-Speed Linear Actuator
Image Courtesy of Progressive Automations

3.12.4 Types of Linear Actuators

The linear actuator in this project is used to raise and lower the filling station. This is required in order to fill the beverage from bottom to top – while being submerged in the liquid. The main purpose of this is to reduce foam and also to conserve carbon dioxide. If the beverage was not filled in this way, the liquid would make more contact with the surrounding air. This leads to a higher air content in the fluid (which leads to foaming) and more carbon dioxide loss.

The filling station will weigh a considerable amount {ENTER WEIGHT HERE} , and the actuator will need to handle this weight. Also, the station must be able between the UP and DOWN positions quickly.

Lastly, the actuator will need to be able to extend at least seven inches. This is because the height of most 16 ounce cans is approximately six inches. 16 ounce cans are the largest variety of container that this station is able to operate with.

The table below compares different types of linear actuators that are considered.

Actuator	PA-15-8-33	FA-RA-22-12-XX	AM-N-TGF12V300-1
Price	$145.00	$149.00	$66.99
Speed	3.15"/sec	4.5"/sec	0.39"/sec
Stroke	8"	8"	12"
Voltage	12 V	12 V	12 V
Force	33 lbs	22 lbs	225 lbs
Warranty	18 Months	None	None
Current	9 A	5 A	4.6 A
Limit Switch	Built-in	Built-in	Built-in

Table 12: Linear Actuator Comparison

The PA-15-8-33 was chosen for this project. The deciding factors for this actuator were the warranty and the force. The PA-15-8-33 is made by Progressive Automations, which is a highly reputable company. They offer a great warranty, which is very important given that this actuator is subject to intense use and will likely have issues before the end of the warranty period. Also, in order to avoid having the actuator prematurely stop working, it is important to make sure that it can output sufficient force. 33 pounds should cover the weight of the filling station and give us a margin of error to work with. The 22 pounds of force that the FA-RA-22-12-XX outputs may be too close to the weight of the filling station, and may potentially cause it to fail early.

Using the PA-15-8-33 will help avoid part replacement in the future. The tradeoff for force in this case is speed, as the FA-RA-22-12-XX can complete its cycle considerably quicker. In an industrial setting even a few seconds per cycle can make a huge difference in production in the long run. However, potential downtime would cause a delay that would far outweigh that of quicker FA-RA-22-12-XX actuation speed. Therefore, the PA-15-8-33 was chosen.

The AM-N-TGF12V300-1 was not a serious contender. Although the AM-N-TGF12V300-1 produces considerable force and comes at a highly discounted price, the actuation speed is much too slow. The PA-15-8-33 would be able to complete approximately eight cycles before the AM-N-TGF12V300-1 would be able to complete just one.

3.12.5 Load Cell

A load cell is a device that detects a change in force by incorporating a micro-electromechanical sensor (MEMS) called a strain gauge. A strain gauge is basically a resistor that is contained in some sort of plastic enclosure. As the device is bent, the resistance changes (either increases or decreases depending on doping). This can be seen from the equation below. As the stain (ε) is modified, the resistance changes.

$$GF = \frac{\Delta R/R}{\Delta L/L} = \frac{\Delta R/R}{\varepsilon}$$

In order to detect the change in resistance, a Wheatstone bridge is used. The Wheatstone bridge effectively turns the change in resistance to a change in voltage. If no force is applied, then 0V is detected. If there is a force applied, then a voltage proportional to that force is detected. The Wheatstone bridge also eliminates the effect of thermal change, which the resistance of the stain gauge is heavily dependent.

By placing two stain gauges in the Wheatstone bridge (half-bridge) or four strain gauges (full-bridge) the effect of thermal change is canceled out since it effects all strain gauges equally. The detected voltage change is fed into an amplifier and then into the MCU via an analog input. Lastly, the MCU converts the analog signal into a digital signal via the internal ADC. The Wheatstone half-bridge configuration is in Figure 7. In Figure 7, R3 and R4 are strain gauges.

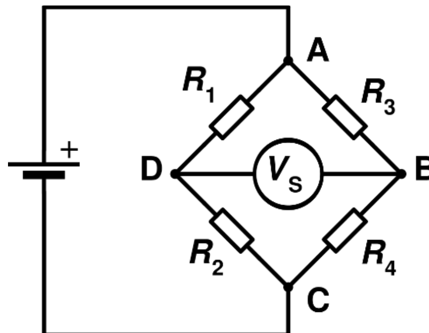


Figure 7: Half-Bridge Wheatstone Bridge

3.13 Power Requirements

This system requires three different voltage levels and relatively high current output. The MCU runs on 3.3V and draws very little current. The servo motors run on 5V and consume almost 6A altogether. This high current output must be kept in mind when choosing a regulator. The linear actuator runs on 12V and consumes a max current of 9A. This is also a high current, and altogether the system may pull as much as 16A at once. All power in the system will come from a 12V power source that is plugged directly into a wall outlet. A 12V to 5V regulator and a 5V to 3.3V regulator are added to the system in order to meet the power requirements.

3.13.1 12 V Power Supply

The 12V power supply in this project are the main source of power for all the components. Given that this system will draw up to 16A at one time, a power supply with a high current capacity is needed. The most important aspects of the supply that are evaluated are current capacity and price. The table below compares various 12V power supplies.

Power Supply	SUPERNIGHT 12V 30A	BMOUO 12V 30A	eTopxizu 12V 30A
Voltage	12 V	12 V	12 V
Max Current	30 A	29.2 A	30 A
Price	$20.99	$18.88	$18.95
Overload Protection	Yes	Yes	Yes
Weight	1.45 lbs	1.46 lbs	1.8 lbs
Dimensions	8.7 x 4.7 x 2.2 inches	8.5 x 4.6 x 2.1 inches	8.5 x 4.5 x 2.1 inches
Material	Aluminum	Aluminum	Aluminum

Table 13: 12V Power Supply Comparison

All of the power supplies met the standards of this project and they were all almost exactly alike in terms of specifications. Since none of the power supplies had a specification that stood out amongst the others, the chosen supply was simply the cheapest one.

3.13.2 5 V Supply

The 5V supply in this project is used to power the servo motors. The power supply will also be connected to a 1.8V supply in order to power the board. The servo motors in this project consume a lot of power – up to 900mA per motor. There are six motors, so when all the servos are operating they will pull up to 5.4A. This was the biggest factor that was taken into consideration when choosing a supply. The second most important consideration was cost.

Almost all of the regulators were within one dollar of each other, so this was not a concern. Other factors that were considered were footprint and efficacy. Footprint was also not much of a concern, considering there is plenty of board space. Also, since the project is supplied with power from an outlet and not a battery, efficacy was also not an issue. The supply that was chosen had a balance of all of the parameters considered.

A schematic of the supply is shown below. The designs were created using the Texas Instruments WEBENCH Designer.

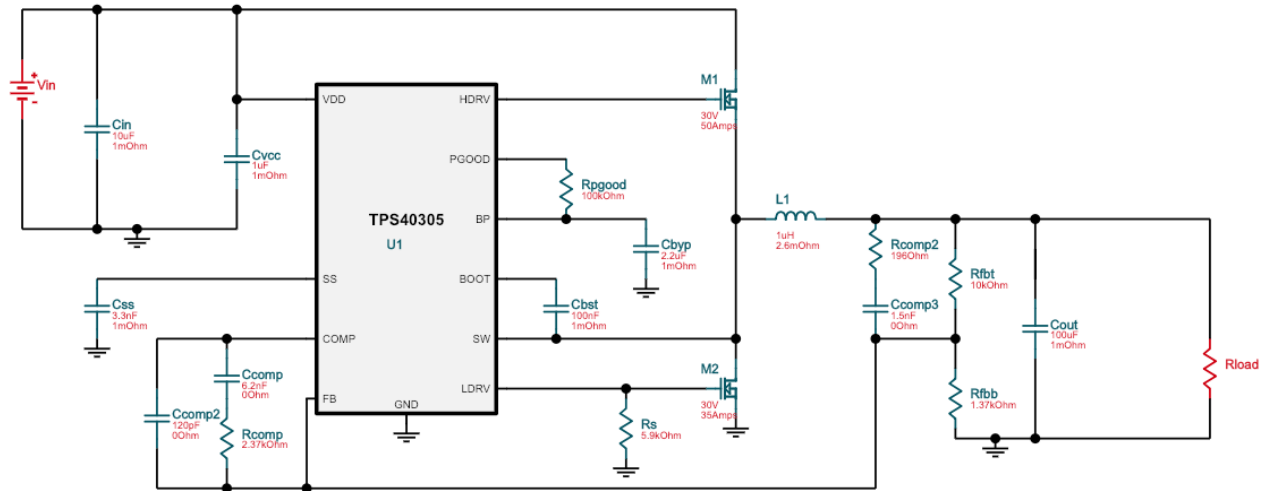


Figure 8: 5V Power Supply Schematic
Image Courtesy of Texas Instruments

3.13.3 1.8 V Supply

The SAM3X8E Arm Cortex M3 requires a 1.8V supply. This supply is needed solely for the purpose of powering the MCU. The schematic diagram and PCB layouts are shown below. The Vin min is 4.5V and the Vin max is 5.5V. The output voltage is 1.8V with a max output current of 2A. The controller being used is the TPS6208818YFP.

All of the inductor, capacitor, and resistor values shown in the schematic are used with the controller to achieve the parameters previously listed. The designs were created using the Texas Instruments WEBENCH Designer. To merge all schematic designs onto one board, CadSoft Eagle are used. This software allows for allows for all of the component schematics to be connected together into one schematic.

The PCB layout can also be done using Eagle. Footprints from all of the components are placed in optimal locations on the PCB board. After the footprints are laid out, traces are drawn between footprints to connect the devices. The footprints are connected in a manner that matches the schematic. Once the PCB design is complete, the file are sent to a PCB manufacturer. After the PCB is received, the components will need to be mounted. Some of the components are extremely small and some have many pins. In order to reduce error in mounting the components, the components and the PCB are sent to a third party for mounting.

The 1.8V power supply was not necessary for the PCB, and on the final PCB design the 1.8V regulator was not used. The internal voltage regulator on the MCU was used instead to generate 1.8V.

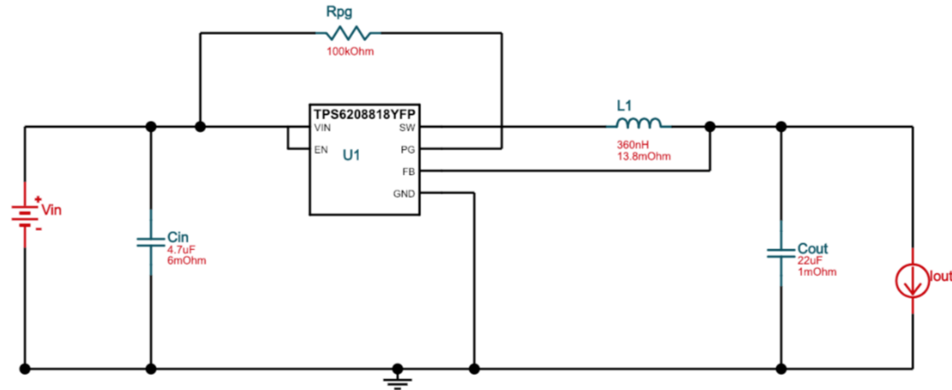


Figure 9: 1.8V Power Supply Schematic
Image Courtesy of Texas Instruments

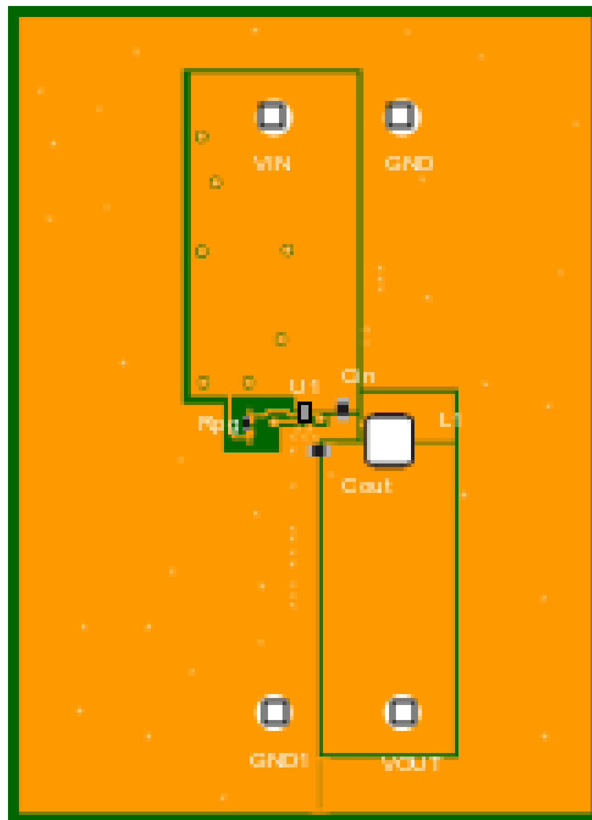


Figure 10: 1.8V Power Supply PCB Layout
Image Courtesy of Texas Instruments

3.12.6 Servo Control

To control a servo, a pulse width modulation signal (PWM) is used. The MCU sends a PWM signal through the control wire to precisely control servo movement. To safely and efficiently control all six servos, a motor controller is used. The servo controller that was chosen for the project is the Adafruit 16-Channel 12-bit PWM/Servo Driver.

This board are initially just used for testing the project. If there are no issues during testing, then the schematic of this board are imported into Eagle and are merged with the main PCB design. If the servo controller is unable to merge with the rest of the design, then the Adafruit board are used in the final project. To control the servos, all the servos must be connected to the servo control board.

Each servo must be connected to three pins on the servo control board, which are labeled PWM, GND, and V+. PWM is the servo control signal, which indirectly comes from the MCU. The MCU communicates with the servo control board using I2C protocol. From this I2C communication, the servo control board generates a PWM signal to the designated servo motor. The servo control board requires a logic signal that is at least 3.3V. The MCU outputs logic at 1.8V, so a logic level shifter are used.

This are described in the next section. The GND and V+ designations on the servo control board are connected to the 5V power supply. A schematic of the servo controller is shown below. The first portion of the schematic shows the servo pinout and the second shows the controller and the I2C input.

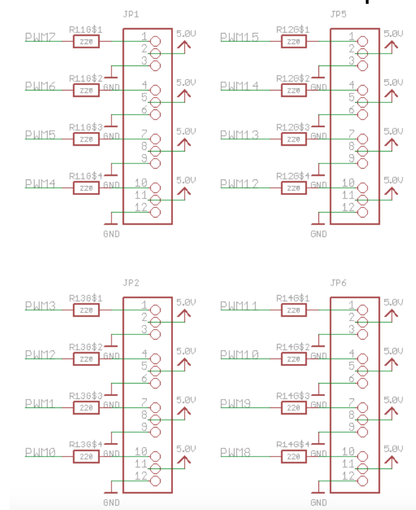


Figure 11: Servo Control Board – Servo Pinout
Image Courtesy of Adafruit Industries

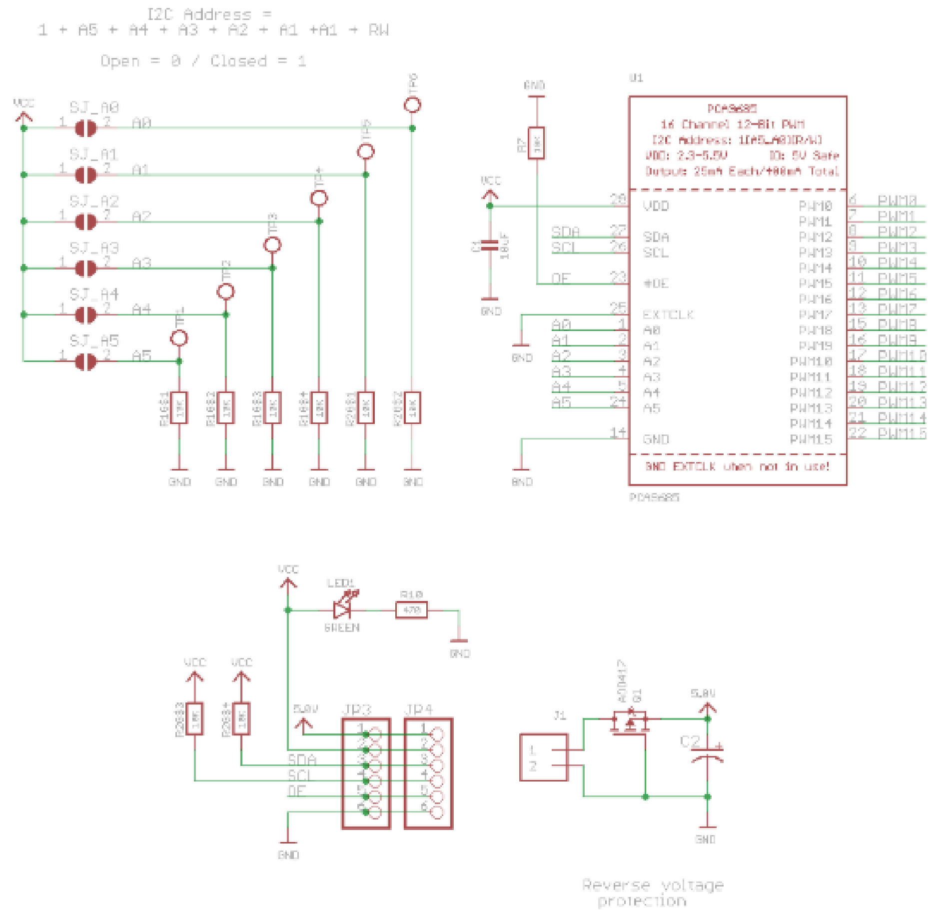


Figure 12: Servo Control Board
Image Courtesy of Adafruit Industries

3.12.6.1 Level Shifter

The level shifter in this project is used to assure communication is possible between the 1.8V MCU and 5V servos. The level shifter chosen to be used is the SparkFun Logic Level Converter - Bi-Directional. The SparkFun Logic Level Converter will initially be used just for testing. If there are no issues with testing, then the schematic are merged with the main schematic in Eagle. The level shifter are placed onto the main PCB. If issues arise during the testing of the final PCB, then the SparkFun Logic Level Converter may be used in the final project.

The bi-directional level shifter simply converts 1.8V logic to 5V logic, and 5V logic back to 1.8V logic. If a level shifter was not used, the incoming 5V logic from the servos would likely damage the MCU. The outgoing I2C or PWM signals from the MCU would likely be treated as ‘0’ values to the servo or servo control board, since these components operate at 5V. The schematic diagram of the

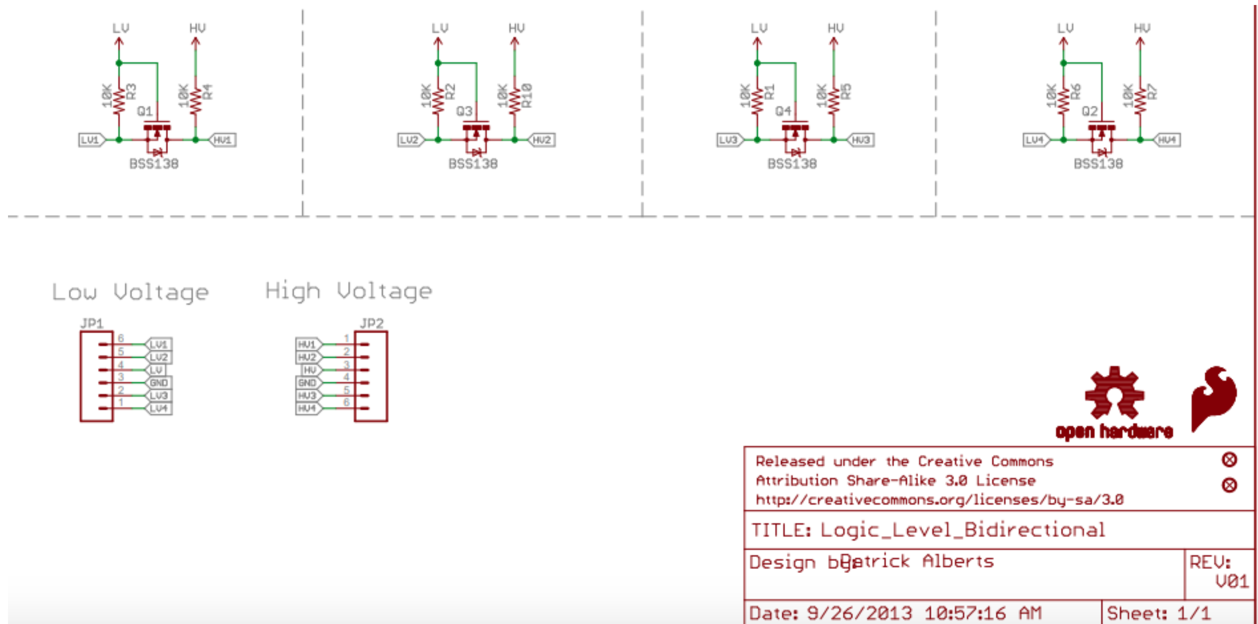


Figure 13: Level Shifter Schematic
Image Courtesy of Sparkfun Electronics

3.14 Other Fluid System Components

3.14.1 Kegs

For this project we're going to use two kegs to hold the fluid to be dispensed. One keg will hold carbonated water and the other will hold a concentrate (a concentrated solution of active ingredients that hold all the components of the final product). The kegs used are Cornelius kegs. This particular type of keg is perfect for small scale operations. The keg doesn't require any type of expensive equipment to fill, clean, or handle. To fill the keg, the pressure valve must first be opened (to alleviate pressure) and then a large portion of the top part of the keg may be removed.

The liquid to be dispensed can then be dumped into the keg. In order to dispense the fluid from the keg, there are two connections attached to the top. One connection is for the outflow of fluid. There is a ¼ barbed connection that will feed to the rest of the system. The other connection is also a ¼ barbed connection, but this connection is an inlet and are hooked up to a compressed gas source. The gas is attached to a regulator that is set at 30psi. This regulator will ensure that the gas, and therefore the fluid, will flow at constant speed.

3.14.2 Compressed Gas

Both kegs in the system are pressurized using food grade carbon dioxide. This pressure will propel the fluid through the entire system. Many industrial beverage operations propel fluid using this method, which is one reason this method was chosen. If the system was not pressurized, then a displacement pump would need to be used. Displacement pumps are a common point of error especially with tougher media like the ones being used in this project. The impurities in the concentrate can easily clog most displacement pumps.

3.14.3 Tubing

The tubing in this project are almost entirely braided ¼ inch braided nylon. Stainless steel and braided nylon are the two most common types of tubing used in industrial food and beverage operations. Stainless steel is much too costly to implement in this project. Typically, the stainless steel is custom made and welded for the project. Also, stainless steel usually requires clamped connections. Although using stainless steel and clamped connections is the most sanitary method to distribute food though lines, it is far too expensive for this project. Braided nylon is the second best option.

This type of tubing is very rigid and resistant to puncture. The nylon also grips barbed connections very well due to its rigidity. A common source of contamination in beverage lines is between the lines and the barbed fittings. Bacteria can form in the small crevices of the barbed fitting. The braided nylon fits the barbed fittings very snugly, therefore reducing potential bacteria content and increasing sanitation.

3.14.4 Housing

The entire housing are able to move up and down – in order to fill the drinks from the bottom of the container. In order to move, the PA-15-8-33 linear actuator are connected to the top panel of the housing. The other end of the linear actuator are connected to the table or counter, which the unit is mounted on. In order to ensure that the housing moves with ease, guides are connected to either end of the housing and mounted to the lower surface. The entire housing are made of either stainless steel or aluminum in order to be sanitary and resistant to the environment and frequent use. For demonstration purposes, the housing will be built using PVC.

The housing will contain the servo motors, ball valves, and tubing. The main water line will feed into the housing where it are split into three sperate lines via a manifold. The main concentrate line will also feed into the housing and split into three lines. The servo connections will run out of the housing to the servo control board. The power from the servo are connected to a relay on the PCB where it are controlled by pins on the MCU.

4.0 Standards

Standards are designed such that a product that follows them will not suffer problems later on when interaction occurs with either users or other products. Through standards, products can work together rather than against each other; it is through standards that products can be trusted not to damage themselves, other products, or the users of those products.

By following standards, a product becomes more accepted, and therefore more widely used. It is in the best interest of the individual/group to ensure that a product follows all standards for both economic and safety reasons.

4.1 Health Standards

This project must strictly follow health standards. This beverage filler is designed for relatively high volume industrial use, and therefore the beverages produced by this machine are far reaching. If all health standards are not followed closely, many people could be affected. Most of these standards deal with food contact with the equipment that we use.

4.1.1 NSF/ANSI 61

This standard covers components that make contact with drinking water. According to NSF/ANSI 61, if someone sells, manufactures, or distributes drinking water they must comply. This project is designed for manufacturing beverages using drinking water, thus it is applicable. Every aspect of our design must comply with this standard. Fluid makes contact with storage kegs, tubing, and valves before it is dispensed into the final container. The biggest concern with contamination is lead leaching from metals such as brass and contamination from certain schedules of PVC.

The valves that are used are made from a safe type of plastic that is NSF certified. The tubing used is made from braided vinyl, which is food-safe and highest recommended for use with food and beverage. The kegs are made from a food-safe stainless steel and are constructed with the intent of beverage storage. If possible, all components and surfaces should be made from a food-safe metal or plastic.

4.1.2 FCC

The FCC governs communications technology in the United States. FCC certification means that the product is safe for humans. The FCC evaluates electromagnetic fields radiated by the device as well as radio frequency exposure. All components in this are FCC certified where applicable. For example, the 12 V power supply emits a significant electromagnetic field when regulating power, and is therefore accompanied with an FCC certification. The following are classifications of radio frequency devices.

4.1.2.1 Incidental Radiators

Incidental radiators are not intended to produce radio frequency energy over 9kHz. These devices are not required to obtain equipment authorization. Incidental radiators may occasionally produce frequency energy over 9kHz. If there is an interference due to operation above the allowed 9kHz, then the user must remedy the interference. The FCC recommends that engineers and manufacturers use good judgement when choosing appropriate designation. Some examples of incidental radiators are AC and DC motors, light switches, and power tools that do not contain digital logic.

4.1.2.2 Unintentional Radiators

Unintentional radiators operate in the radio frequency band and contain digital logic. These device typically operate between 9 kHz to 3000 GHz and are additionally regulated under 47 CFR Part 15 Subpart B. Although these devices may operate at high frequency, they are not intended to radiate wirelessly. Examples of this type of radiator include coffee pots, wrist watches, and power tools that use digital logic.

4.1.2.3 Intentional Radiators

Intentional radiators intentionally generate RF and may be operated without an individual license. Examples of this type of radiator include cellphones and other wireless communication devices.

4.1.2.4 Industrial, Scientific and Medical Equipment

These devices include radiators that produce RF energy for purposes other than telecommunication. Uses may include heating, ionization of gases, mechanical vibrations, and acceleration of charged particles. Examples of this type of radiation include microwave ovens, arc welders, and fluorescent lighting.

4.2 Communication Standards

4.2.1 Serial Peripheral Interface

Serial Peripheral Interface protocol is a standard of communication between two or more devices, where one device is the "master" and one or more devices are the "slaves". In order to communicate, four pins are used on each slave device. The pins are designated as serial clock (SCK), master in slave out (MISO), master out slave in (MOSI), and slave select (SS). The master must add an additional slave select line for each additional slave. For example, if there are 5 slaves, then the master must have 5 pins dedicated to slave select. This configuration is shown in the diagram below.

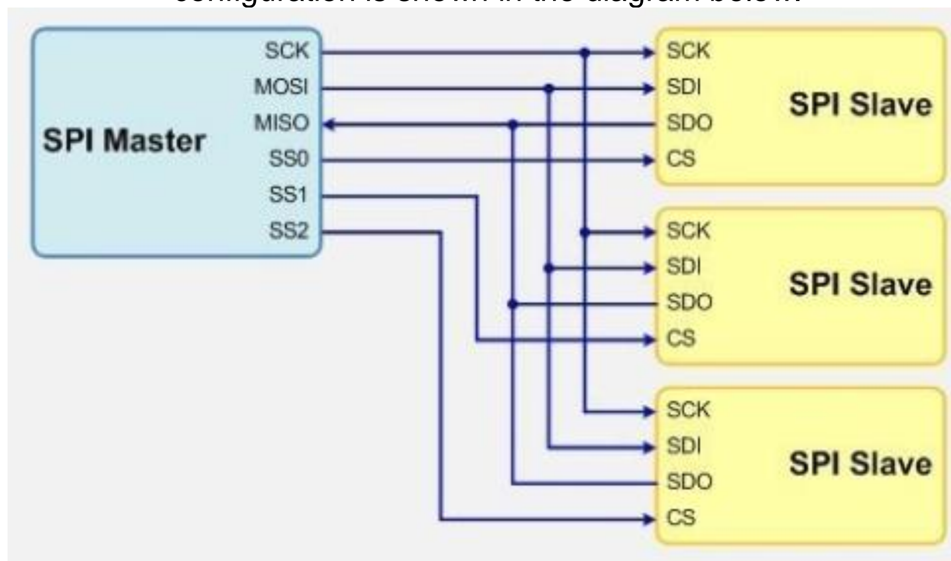


Figure 14: SPI Master-Slave connections
Image courtesy of Corelis

There are four different modes of communication which set bus timing. These modes depend on the devices being used. In mode 0, data is sampled during the leading rising edge of the clock (this is the most common mode). In mode 1, data is sampled in the trailing falling edge. In mode 2, data is sampled in the leading falling edge of the clock. Lastly, in mode 3 data is sampled in the trailing rising edge.

4.2.2 I2C

I2C is a communication protocol between two or more devices. I2C utilizes two busses, a serial data line (SDA) and serial clock line (SCL). Each device connected to the bus is addressable by a unique address and master/slave relationships exist

at all times. I2C has four modes of bidirectional data transfer. These four modes are included in the table below.

Mode	Bit Rate
Standard-Mode (Sm)	100 kbit/s
Fast-Mode (Fm)	400 kbit/s
Fast-Mode Plus (Fm+)	1 Mbit/s
High-Speed Mode (Hs-mode)	3.4 Mbit/s

Table 14: I2C Modes

When the bus is not busy, both the SDA and SCL lines are HIGH. All data exchanges begin with a START and end with a STOP signal. A START occurs when the SDA is pulled to LOW while the SCL is HIGH. The SCL must be HIGH in order to avoid conflicting with another data transfer. A STOP occurs when the SDA is pulled HIGH (signaling that the data line is idol) and the SCL is HIGH. This is demonstrated in the diagram below.

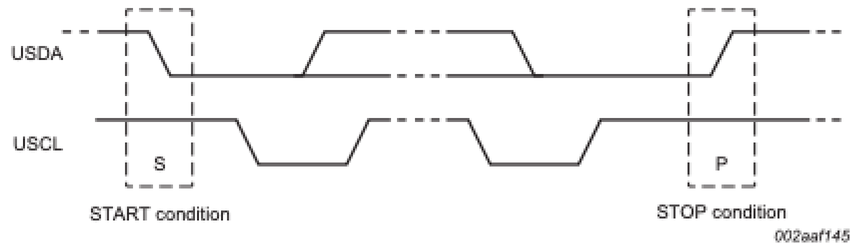


Figure 15: I2C Start/Stop protocol
Image courtesy of NXP Semiconductors

After a START signal is given, the slave address is sent by the master. The master sends the 7-bit slave address along with a read/write bit as the LSB. A ‘0’ signifies a write and a ‘1’ signifies a read. If the master would like to change which device (slave) to communicate with, then the master must repeat the START signal. The master is not required to send a STOP command first. To speak with a device, the master must send a START and the new slave address along with the read/write bit. Once a connection with the slave or slaves is established, data is transferred in 8-bit bytes, with the MSB transferred first. After each byte, the master drives the SDA line high during the acknowledge (ACK) cycle. The data transmission cycle is shown in the figure below.

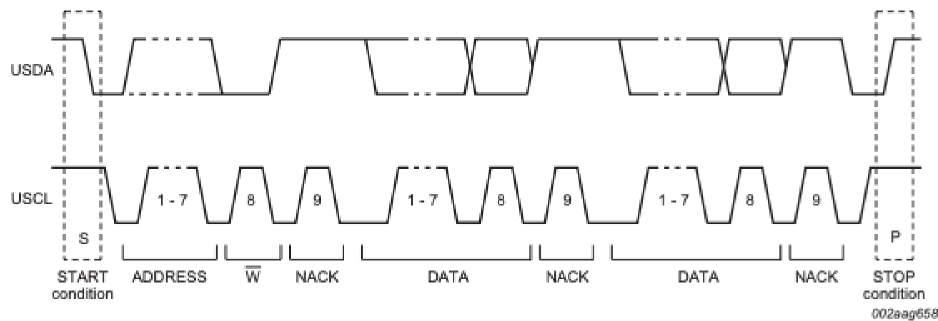


Figure 16: I2C data transfer
Image courtesy of NXP Semiconductors

4.3 Sensor Standards

Sensor standards exist so that sensors may be used universally – this is the same reason standards are set for other parameters in the industry. The bottom line is that sensor standards ensure consistency across designs.

4.3.1 Temperature Sensor Standards

The governing body for temperature sensor standards is the American Society for Testing Materials (ASTM). The ASTM is a guide that includes a tremendous amount of standards. The ASTM standards guide of temperature measurement is split into ten sections. The sections are explained below.

4.3.1.1 Digital Contact Thermometers

In this section, ASTM cites standard E2877 – 12e1 which is the standard guide for digital contact thermometers. This standard provides an overview of digital contact thermometers and also describes the nine different accuracy classes. These classes cover temperatures ranging from –500 °C to 200 °C. Although there are temperature sensors that can operate outside this range, they are much less sought after and much less common. In order for a thermometer to qualify for a certain class, it must accurately measure temperature in the respective class within a certain tolerance. The guide also specifies what is considered a “digital contact thermometer”. Thermometers that lie in this class are platinum resistance sensors, thermistors, and thermocouples. The guide goes on to specify recommended manufacturing practices and also establishes SI units as the standard units.

4.3.1.2 Editorial and Terminology

Standards E344 – 18 and E1594 – 16 are included in this section. E344 – 18 outlines terminology to ensure understanding of future documents. E1594 – 16 further describes terminology and also outlines methods for expressing temperature, temperature values, and temperature differences.

4.3.1.3 Fundamentals in Thermometry

This section outlines calibration – preparation and reporting. The standard method of calibration is ice-bath submersion. The standards in this section, notably E563 - 11(2016), go into detail about ice-bath preparation and requirements.

4.3.1.4 Types of Thermometers

The remaining sections of the ASTM measurement standard guide go into depth about the various types of thermometers and respective standards. The types of thermometers mentioned are liquid-in-glass thermometers, resistance thermometers, radiation thermometry, and thermocouples.

4.3.2 1-Wire Technology

1-Wire technology is basically a connection that uses serial protocol that operates between 2.8 and 5.25V. The technology is called 1-Wire because typically there is no pin used for power supply – the sensor draws power from the data line. This is known as parasitic supply. A schematic diagram of this is shown below.

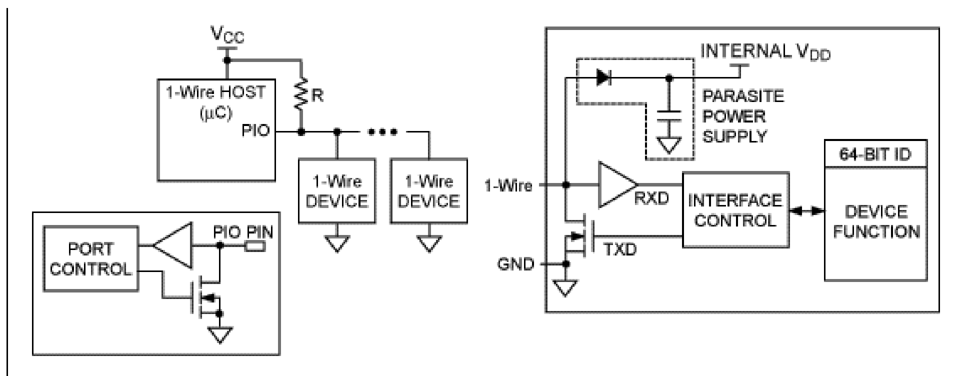


Figure 17: 1-Wire Schematic
Image Courtesy of Maxim Integrated

Multiple 1-Wire devices may be connected to single I/O pin on a microcontroller. This is possible because each device has a unit 64-bit identification code that is used for distinguishing. This 64-bit code is factory programmed and may not be

changed. Each 64-bit product identification code contains an 8-bit family code which identifies the device type and functionality.

4.3.2 Force Sensor Standards

The ASTM has also put forth standards for force measuring instruments. The standards clarify necessary procedures for instrument calibration. It is important to note that this standard only applies to force measuring devices that are elastic or force-multiplying systems. Some of the applicable standards are listed below.

- E74 – 13 “Standard Practice of Calibration of Force-Measuring Instruments for Verifying the Force Indication of Testing Machines”
- E74 – 18 “Standard Practices for Calibration and Verification for Force-Measuring Instruments”

4.4 Soldering Standards

Soldering standards for this project were taken from Nasa’s technical standard document. The standard is NASA-STD-8739.3. All components mounted on the PCB are surface mounted (SMD). The connections that come out of the device to be mounted are called part leads. Part leads can come be lapped, straight-through, or clinched. All parts being mounted on the PCB for this project will have lapped leads. Lapped leads can either be round or flat. In order to achieve an acceptable connection between the lead and the pad, the lead must be placed flush with the pad. To maintain a stable connection, there must be no torque on between the pad and lead. Acceptable connections are shown in the diagram below. In order to minimize torque and achieve a flush connection, the lapped terminations must not hover more than 0.25mm above pad when static.

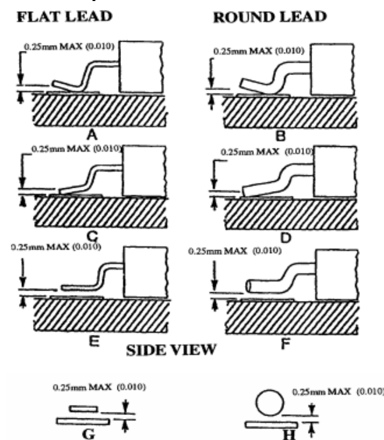


Figure 18: Lapped Lead Placement
Image Courtesy of NASA

5.0 Design

In this section more details on the design and the presentation are discussed.

5.1 Implementation

This system is meant to be implemented into a small scale beverage manufacturing plant. The filler should be placed in the main processing line. The kegs from which the filler draws concentrate may be placed at some remote area. During actual implementation, carbonated water will not be drawn from kegs. Instead, water are drawn from the main water line, filtered, and carbonated. Typically, the water is filtered using at least a carbon filter to remove chlorine and sediment. The water is carbonated using a carbonator. After carbonation the water are passed through a chilling unit in order to retain the carbon dioxide in the water. The chilling unit operates by chilling a water bath in which coil of drinking water pass through. The water bath chills the coils to the temperature of the bath. The temperature of the water are carefully monitored here. A temperature sensor are place into the water bath of the chilling unit (the temperature of this water provides a near estimate of the water temperature in the line). When the final product is dispensed at the filling station, the water must be between 0 and 5 degrees Celsius.

The concentrate that is held in kegs must also be chilled. The concentrate must be similar in temperature to the water when mixed in the final product at the dispensing stage. The concentrate will also pass through the chiller in order to be brought to dispensing temperature. All lines after the chiller including the water lines are insulated to reduce heating up before dispensing. At the filling station, three cans are placed under the filling heads. Once the user tells the filler to “fill”, the actuator will lower the fill heads into the containers. The valves will then dispense the appropriate amount of water and concentrate into the beverage container. Once the fill is finished, the fill heads will move up into starting position. Once the heads are removed the operator is notified that he/she may remove the filled containers. The operator will then seal the container. Once the product is finished, the user will place the final product on the load cell which will take a reading. If the weight of the container is not within the range, then the MCU will notify the operator via the screen. If the fill was done correctly, the monitor will read “Pass”.

5.2 Design Motivation

The main motivation of this design is to reduce the amount of defective product that is released into the market place. The load cells help mitigate this by ensuring that that the amount of fluid is correct. This is important for three reasons. The first

reason is that if the product is being overfilled, product is being wasted. The second reason is that if the incorrect amount of concentrate is being dispensed, the product will not taste correctly to the consumer. This can be potentially dangerous if the beverage contains a hazardous chemical such as alcohol or caffeine. The last reason is that underfilling the beverage may cause an air gap to form. Leftover air in the container will interact with the beverage and will potentially oxidize the drink. This can cause a drastic change in the taste of the beverage, especially if the drink is juice based. It may also encourage the formation of mold or bacteria which can be hazardous to consumers. A large air gap will also cause a pressure loss. Most of the pressure in the beverage comes from the liquid itself and the carbon dioxide gas. If the gap is large, the carbon dioxide gas will have a larger volume to occupy and thus have a smaller pressure. This will lead to cans with a deflated look that is not attractive to consumers.

Problems will also arise if the water is too warm. The beverage will not have enough carbonation. This may be detrimental to the product. Although many consumers may not be able to taste the difference in carbonation, the product may feel flimsy and easily collapsible when a consumer grabs it on the shelf – similar to the effect of not filling the container up entirely. This is especially important if the beverage is shelved on a cooled rack. The free carbon dioxide will dissolve into the beverage under cold temperatures and thus cause the internal pressure of the container to drop dramatically. Beverages with this defect often cause manufacturers a lot of money. Not only do they not get sold in stores, but they also hold up product that was stocked behind them on the shelf. They can leave a bad impression on a consumer, especially a first time buyer or potential buyer.

5.3 Presentation

In order to make this project presentable, some adjustments must be made. The filling station is mounted on top of a table. The keg, carbon dioxide canister, and other components that would typically be found throughout the warehouse are placed below the table. There was no carbonator or chilling unit used during presentation. The beverage is dispensed at room temperature. Also, the lines are not insulated.

6.0 Project Software Design Details

In this section the details on how the software implemented for this project.

6.1 Threading and Multitasking

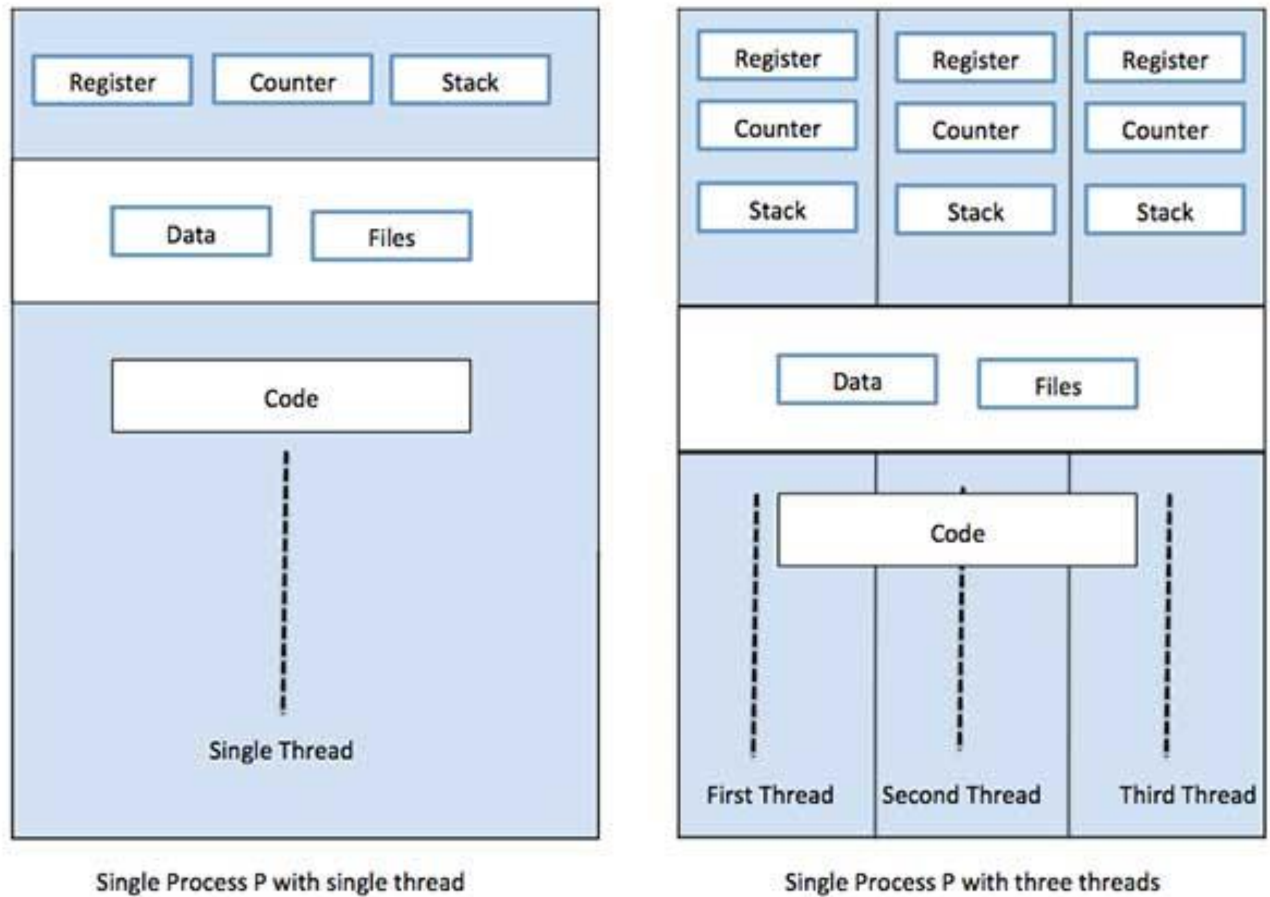


Figure 19: Threading Explanation
Image Courtesy of TutorialsPoint

First we formally consider the problem of multitasking and define it in terms of our application. We consider two cases: using an internal clock to simulate multitasking, and using the multiple cores of the chosen CPU to fully realize multithreading.

We define our problem from the standpoint of the software. We consider an arbitrary number of valves that need control, sensors that require monitoring, and a screen that needs to display and read input. Considering the control required on the valves: it is only setting a single bit on one of the pins. Thus we can assume minimal delay in processing this operation regardless of our timing architecture. However a problem does occur when we need to make a choice on what to do with a valve – that is, what signals for a valve to open or shut. That requires the

input from the sensors. Which means that if the sensors reading is delayed, the valve critical valve shut-off or open are delayed. Thus if a sensor reading is going to be delayed, as it might be in the case that use the internal clock to simulate multitasking, we need to guarantee that the delay is bounded by an acceptable amount. So we must know at most what our bound is.

All of the CPUs being considered have maximum clock speeds on the order of MHz so we can expect a clock delay on the order of microseconds if we really need. That is reasonably small. So if we place a high priority on the interrupts for the sensors to read we should be achieve feasible performance.

On the subject of actually threading the core, it obviously possible. And the performance of that system isn't worth going into detail on. It will work. But, it aremore than what the system requires.

6.1.1 Conclusion

Aiming for an architecture which using the internal clock and interrupt service routines to simulate multitasking is reasonable enough for acting as the kick-off for valve-shut off and open. Thus this system opts in favor of it.

6.2 Memory Requirements

Considering the requirement of logging sensor data, then no form of read-only memory will work. Additionally, we want to preserve the data beyond a loss of power. So solely using RAM will not work. We need some form of external storage. And we need some way of traversing the file system of the disk we implement. Thus we need some something to handle this for us. It is possible to implement our own file system management. But why reinvent the wheel when it's free?

6.2.1 Arduino

As already mentioned, we use the Arduino IDE. This gives us access to libraries that are capable of saving to external memory. One solution is to mount an SD card onto the board. There exists an SD library for interfacing with the SD. A USB is also possible.

6.2.2 SD Card

We assume that we mount an SD card with some form of SPI communication to our board. An existing module exists as a reference, the Arduino SD module seen in the figure below,

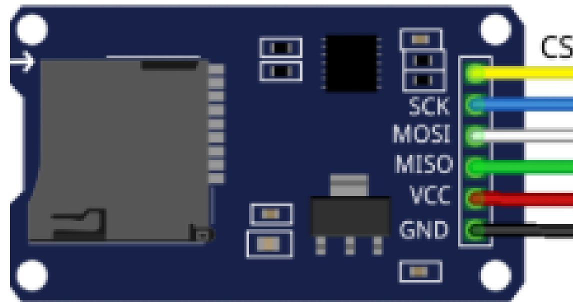


Figure 20: SD card example module
Courtesy of https://www.mschoeffler.de

Note that it isn't particularly important the CLK speed for writing. Reasonable, we wouldn't want to write more than one time a second, and even that might be too much. The reason is that much logging could quickly cause the log to begin wrapping back around. So we would want the logger to be sparse with its actions.

Then we would need to form an SPI connection with the SD card. The Arduino IDE does provide a library, SPI.h, for forming SPI connections.

6.2.3 USB

Alternatively a USB is possible. However there is a greater tradeoff in complexity since there are no libraries I was able to find for this. We would need to write our own library and the reward simply does not exists. An SD card can be hosted by a USB with an SD port for transferring the logs from the SD onto any computer with a USB.

6.2.4 Conclusion on Storage

An SD card is feasible, simple and quite effective for what is required in the project, with the limiting factor of the clock being a non-issue due to the relative small amount of read/write commands that will occur on average.

A USB does not have defined libraries, resulting in significantly more time and effort needed to make it effective for the purposes of this project.

As a result of the above, an SD is opted for.

6.2.5 User Storage

We have an object of the user and all of their presets in main memory and seek long term storage in the SD card. We construct a function capable of storing the user and preset objects as shown in Figure 5.

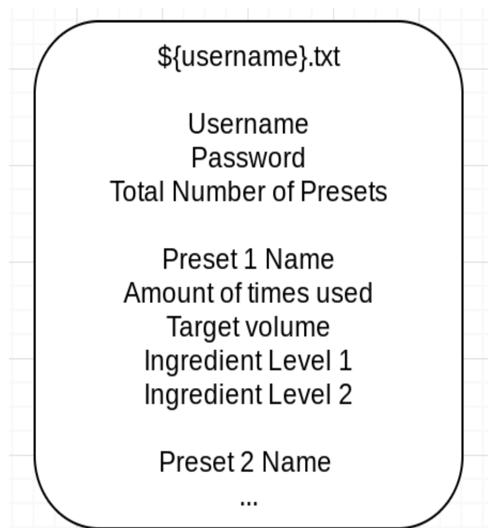


Figure 5: User File Layout

Once the data is stored, we can retrieve it line by line to instantiate a new object for use on the next bootup.

6.3 Logging

As mentioned, we don’t want to over log. So we take a reserved approach. Given some discrete time over what we wish to measure, suppose, 10ms, we average the readings taken within that time and record the low as well as the high of

whatever we are measuring (pressure, weight, etc...). Then we only need to record the average over that time and the extremes. This preserves the peaks that might happen during that time and still gives a fair reading of what happened over that time without logging every sub-division.

The user should be given some form where they can decide how long they want this time to be. A logging configuration page should be allowed. Note that there exists a reading time for the sensors as well. That is a periodic interval for reading in between the logs. Thus,

$$t_r \leq t_l$$

Where the first term is the period for reading from the sensors and the second term is the period for logging. So the questions is whether to allow for a sensor configuration option.

6.4 Sensor Reading Time Configuration

It's possible to allow the user to configure the time for reads however this creates unnecessary complications. Consider the data currently logged in a report. The report was created using some sensor readings at some period reading. If we allow the periodicity to be changed, what happens to the previous report. It no longer reflects what we want. But this is really a small complication. It would be fine to just allow the old readings to remain the same. And it's not necessary to assume we need the previous readings to reflect the same periodicity we use now.

A better questions is the necessity of configuring the periodic readings. Would it be better to find a periodic reading which we are comfortable with for accuracy? Or would we rather the user be allowed to later decide how often they want a reading to occur? If we don't allow the configuration, then we need to allow a small enough period for reading that covers all possible cases of what a user might want. This might achievable if we could think of every conceivable case and be sure that were right about all of them. But it'd perhaps be better to save the mental exercise and just allow the user to decide. This would help with optimizing the system so that only the reading times we want are being used, and not more than we want. Of course there would need to be a bound on how small the periodicity can be, which is limited by the CLK.

6.5 Read-Write Collision

Next, consider the problem of a live logger and a live reporter. If this were a parallel programming task, mutexes would need to be used as both the reporter and the

logger would be reading/writing from the same file. However since we are simulating multitasking using clock interrupts, we do not allow the file ever to be opened and read at the same time.

Suppose ISR1 handles the task of reading from a sensor and writing data to a log. And ISR2 handles the task of reading from the log periodically and updating the reporting software. Since they are both ISRs handled by a single CPU, it is impossible for either to occur at exactly the same time. We can't have the File opened by both functions at the same time, or else their read/write pointers would not exactly coincide. We could have two global read/write pointers and alter them on call but that's messy and uncalled for.

Instead, when ISR1 begins, it opens the data file. Writes what it needs to and closes the file. Then ISR2 may begin at any time after. Open the file. Read what it needs. Close and continue.

But this creates an additional complication. Supposed that ISR1 writes an arbitrary number of times before ISR2 reads. ISR2 has no way of knowing the number of times that ISR1 wrote and how many new values it needs to read. Additionally, it doesn't know what values are even new. It has no way to distinguish between values it has read before and values it now needs to update. Thus we need a way for it identify the next value to be read.

6.6 Logging Format

A simple solution to the problem of the ISR2 needing to some reference of what data to read next is that have each data piece contain some identification number. And then have the data piece contain some identification number. Next we need the ISR2 to know what the last ID was that it read. So if it sees an ordered list, once it sees the most recent ID, it knows all new values exist on one side of the list. So we write to the file in an ordered format. Either specifically at the top, or the bottom. Multiple complications arise from this solution.

One is that the file is constantly growing and needing to be opened and closed and saved. As the file grows, these operations become more expensive. And could cause delays in the system.

Two: the system needs to read all the way until the it finds the portion of data it needs to read. Additional time is lost for this search.

The benefit of this solution is that it is simple. And that's pretty much it.

If we were to implement this solution, the data could be formatted as such

Data:

<key> <value> <date> <time>,

```
<key> <value> <date> <time>,
…
```

An alternative solution is to maintain separate file pointers. That is to open the file once for both reading and writing. Create a copy of the file pointer. One pointer for reading, one for writing. Now each ISR only uses the pointer that it needs. The minor complication with this solution is that it needs to guarantee writing will happen before writing to avoid trying to read no change. If this is accomplished, then the overhead of opening and closing a file many times is gone. Also we no longer need to find where the new data is when we read, it will always be the next increment of the pointer. And as a cherry on top, the key element of our data no longer needs to be there since a consistent read pointer eliminates the need for us to distinguish old and new data.

Thus our new data object is,

```
struct LogData {
        float value;
        char[] date;
        char[] time;
} typedef LogData
```

6.7 Remote Login

Login is a bit of a problem. If we do only have a source of code running at a time and it's listening for changes on the screen, how does it listen to network requests. The answer is that it can't do both. Again we rely on sequential nature of the code to make it appear as if it does both. Assume our device is currently idle. It is not generating drinks, nor producing reports, nor even logged in. It sits at the home page. Calm and alert.

Since only one process runs, we can't run a server separately from our application. So the option is to implement our own server or use the code provided by someone else. However even if we get that code, we can't thread based on our design.

An alternative solution, is to install an Operating System onto disk, boot from there and then run a server and our application separately using the OS's scheduler. The approach would be simple. Run Node.js as the server. Then we could locally store the login info. On request from the Web Application built no the Node stack, we can verify a user login and present them with a UI that allows them to remotely navigate and control out system.

A problem with this solution is that it solves the problem by using a javascript server. This means that if we want to make the application the user sees once he logs in fully functioning, the server needs to make calls to our embedded program

which would require the program to listen for calls from the server. That's just nonsense. The original reason we considered this solution is because the code can't listen for those external calls. Better would be to implement the code that runs the platform on the server so that it only needs to receive the HTTP request from the remote user and then begin processing the request. This would ultimately mean we write the entire code Java. So although we've simplified the task of hosting a server to receive user logins, we complicate the embedded task. We also now require the JRE which uses much more memory than just running our original C code. And we've also appended the need for an Operating System. However perhaps the largest drawback is the speed performance lost to using the JRE instead of C. A question then is raised about is the loss in speed too great? Or is it manageable? Is the question even worth exploring? Perhaps a reconsideration of the feature's value is better.

6.8 Conclusion on Remote Login

Requiring remote user login adds too much complexity to the C embedded code. And if we opted for an OS with a Node stack, it adds system complexity and cost. We should consider further whether this feature is worth the costs associated with the current explored solutions.

6.9 Local Login

This is feasible. Since we don't require to listen to network calls, a user could easily use the touch screen and an onscreen keyboard to enter their login information.

Login information could easily be stored locally on the disk. A simple file containing objects,

```
struct User {
        char[] userName;
        char[] password;
        Presets[] presets;
        } typedef User
```

6.10 Security

Since these usernames and passwords are being stored locally, on the reasoning that there is no network connection, it is fine to store these in plain text. The only way areread is if someone has acquired the device physically. In that instance

there are greater causes for alarm than someone stealing login credentials and getting access to another person's presets!

6.11 Network Login Security

If remote login is decided to be used, we don't want to store the username and password in plain text. An option would be receive passwords as regular text and run them through an algorithm that expands a password into an incomprehensible jumble of fixed size. We store the corrupted password instead of the normal password. As long as the function for corruption is one-to-one, we can guarantee that no password except the one which is correct is accepted. Then when we receive a password from the server and the username, we first check that the username exists in the repository. Assuming it does exists, corrupt the given password. If the corrupted password matches the one stored, we accept the login. Else reject. Thus we never store the password in plain text.

For example, suppose we require passwords to be up to length n. Then for each character in the password consider the following process,

1. Compute ASCII value for each character.
2. Let each digit of the ASCII value correspond to a letter from A-J (69 -> FI)
3. Then append the two corresponding letters as part of the corrupted password
    1. Then append the two corresponding letters as part of the corrupted password.

   Continue appending until the there are no more character in the password.

6.12 Code

For this project, we coded in the Arduino IDE and made use of its libraries. We also used it to compile and receive our .hex file. As a result, we coded the whole project in C++. So we were able to use OOP concepts to maintain visibility over the moving parts of the project without getting clouded by a mishmashed overcast of cluttered code.

6.13.1 UTFT

UTFT (Universal Thin-Film-Transistor) is a library for configuring and supporting the use of a number display modules. It'll allow us to create a display object in

memory and use the libraries functions to drawn the object we want to display on the screen. The more general explanation of the motivation to use this library is found in section 3.3.6

UTFT uses approximately 80KB of flash memory.

6.13.2 Initialize UTFT

First we need to create a display module object using the UTFT constructor. The constructor we can use is

- UTFT(Model, RS, WR, CS, RST, ALE)

- Model is the a number ID associated with the type of display module we use. The display modules and their associated IDs can be found on the supported display modules PDF of the UTFT documentation. The ID we will use arespecified once we decide the display to use.
- RS specifies the Register Select which is what enables us to select between instruction mode and character mode. Instruction mode is what we will use for this project. It allows us to send discrete commands to the display module. Contrast this with the alternative Character Mode which only allows us to send character data over the 8bit data bus of the display module. The UTFT library allows us to send the character data we want through the print(...) functions which arediscussed later. Thus there's no need to use the limited character mode. To select Instruction Mode typically requires us to the set the pin to low. However this will need to be verified on the equipment we will use.
- WR specifies the pin we can write to.
- CS specifies the chip select.
- RST specifies the reset pin for clearing data.
- ALE is an optional parameter for latched 16bit shields. It's not necessary for what we are building so we won't discuss it.

Once the UTFT object is created we can initialize the values of the object with,

InitLCD([Orientation])

- Where Orientation is an object with PORTRAIT and LANDSCAPE attributes.

This would reset the colors of the display and set the font to none.

6.13.2 Using TFT

Suppose we intend to create a clickable button. The functions mentioned int 3.3.6 would be used and here we describe their interaction with a little more clarity. The fillRoundRect(x1, y1, x2, y2) function fills a rounded rectangle in at the specified location with the currently set color. The UTFT display module object created by the constructor always creates its new object using the current set color when calling on the drawn functions. So if you wanted the rectangle to be a specific color we would call the setColor(r,g,b) function from our UTFT object. The r,g,b parameters are unsigned short integers (0-255). Thus if we wanted the rectangle we just drew to be red we would first call setColor(255, 0, 0) and only then call the function create the round rectangle.

6.13.3 URTouch

Like UTFT, this library was first mentioned above is section 3.3.7. Here we explain its implementation details and usage with more clarity. Recall the URTouch library supports receiving the input on the LCD screen and stores it in a URTouch object. Additionally it has a pin for signaling interrupts.

6.13.4 Touchscreen Calibration

Due to electrical faults from components being too close together, or mechanical misalignment caused during manufacturing, a touchscreen's input is often faulty and needs calibration. An example of the difference between the reported input and the true, intended input is shown in the figure below.

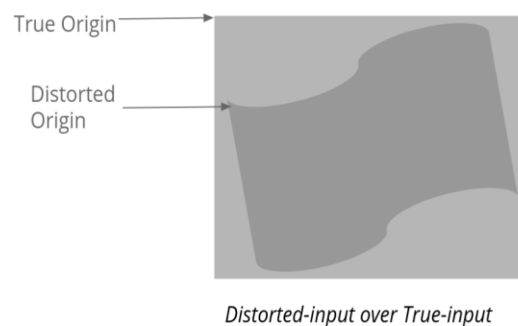


Figure 2: Touchscreen Distortion

So the aim of calibration is given some distorted input, transform it to the true origin. The algorithm we went with for calibration is the Classical Three Point Algorithm.

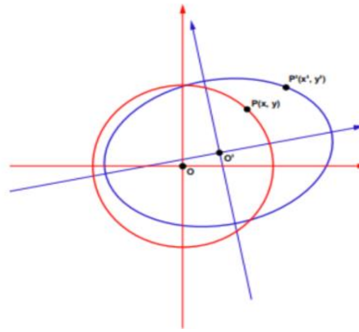


Figure 3: Input Map

The aim is to approximate the shape of the distortion map by a circle which is assumed to have been rotated, scaled, and shifted from the true origin.

Let there be three known points P1, P2, and P3. Displaying their positions on the touchscreen and and allowing the user to touch those points generates distorted inputs P1', P2' and P3'. We can then generate values for the following equations

$$\begin{cases} x = KX_1x' + KX_2y' + KX_3 \\ y = KY_1x' + KY_2y' + KY_3 \end{cases} \quad (1)$$

Where x and y are the true values, x' and y' are the read values and the coefficients are the calibration constants and are six unknowns, K being a value we provide. Using the data of the three points, we generate six equations with six unknowns and solve for the coefficients. Now given any future inputs we can use this equation to approximate where the users intended click is located.

6.13.4 URTouch Initialization

Like the UTFT, there exists URTouch constrcutor for setting the required pins,

URTouch(TCLK, TCS, TDIN, TDOUT, IRQ)

- TCLK is the Touch clock
- TCS is the Touch Chip Select
- TDIN is the Touch input.
- TDOUT is the Touch output.
- IRQ the pin for signaling Touch interrupts.

Once the function object created, calling InitTouch([Orientation]) on that object will initialize the touch screen for use.

6.13.4 URTouch Precision

The URTouch library offers a means for lowering and raising precision for a tradeoff in performance. The function is,

setPrecision(precision)

Which uses precision enum types of PREC_LOW, PREC_MEDIUM, PREC_HI, PREC_EXTREME. Note that it will take longer to higher precisions. Considering that the application of this doesn't require fast input, and we may well opt for a small screen, it would be best to use one of the higher options. Testing can be done to ensure we choose the smallest one that maximizes performance and minimizes delay to our liking.

6.13.5 URTouch Usage

Once the screen is touched, data is stored in a buffer for the URTouch object. Using a built in method from the library, we could periodically check if data is ready to be read (assuming we don't use interrupts). The method dataAvailable() will return true if data is ready to be read. We then use the class method read() to convert the raw data to URTouch fields x, and y. Then we can use getX() and getY() to get the coordinates of the touch.

So if we had made a call earlier to UTFT of fillRoundRect(50,50,100,100) and then when we read the data we get coordinate (75,75) we know that the rectangle created above was clicked. So supposing this rectangle was a button for going to Reports, we could then make a call to another function we'd written drawReport().

6.13.6 Screen State

Consider that we are using either CLK or click interrupts for checking if the screen has been touched. Now suppose that it has been touched at some coordinate K. Now we need to check if a button existed at K so we can process the request the user has. However there exists multiple pages. So how will we know which screen it is we're on for checking if a button exists?

A simple solution is to use a global screen state. Then we can use enumerated types for each page and we only need to check which page we're on. Once we know the page, we know what boxes should exist and where they exists.

6.13.7.0 Virtual Keyboard

Since we have a touchscreen and we require input from the user, we've implemented a virtual keyboard. We use the TFT library for our draw functions which includes the rounded rectangle shape. Next we create a series of constants to scale our keyboard into different sizes and allow for key detection.

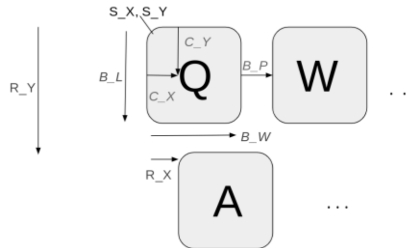


Figure 4: Button Layout

Then given some input we can detect the index of key that was pressed.

$$row = \frac{y - S_y}{R_y} \qquad column = \frac{x - S_x - row \cdot R_x}{B_w + B_s}$$

$$index = \sum_{i=0}^{row-1} (CharactersInRow_i) + column \tag{2}$$

Note that the order of characters on the keyboard is stored in a character array so the index is all we need to know which character was pressed.

6.13.7.1 Input Page

Whenever we want to input something we know we'll at least need to draw the keyboard. But there is second consideration, do we draw the keyboard over the existing page or load a new page for input?

If we load a new page for input, it seems a bit counterintuitive from what we've come to expect from most programs. Typically when input is being requested on a touch screen, a key board is popped up from seemingly under the page, and

overplayed onto the current page. The keyboard, once done, is dropped down and it's context exited. The previous page never needs to be reloaded.

It is possible to do this and we can consider another possibility. We could load an entirely different page for keyboard input. This might seems strange but consider that we're not drawing a context on top of another context. We don't need to keep track of that complexity and we use less memory as result.

A third option is to draw the keyboard within the context of the page. To the user, it'll still seem like the keyboard is its own context, but actually we have written over the previous context within that segment of the page. Then when the keyboard is done, we would simply need to reload the page which called the keyboard.

The downside of that method is sheer ugliness. The page would appear to blip, because it's being reloaded, once the keyboard is done being used. The benefit would be that like the second option, it doesn't rely on concurrent contexts and does require the complexity of conserving those contexts. And as an aesthetic bonus, it appears to operate as modern programs do with virtual keyboards.

6.13.8 Preset Modeling

If we're going to used store presets as data to be retrieved and interpreted by our program as timing for valves and sensor data requirements, then we need to be sure we understand the needs of a beverage at production time. Once we know what a beverage making process needs to keep track of during production, we can set that as the data to be stored in the Preset Schema.

So first we consider the process of making a beverage. It might be that there are multiple valves, each controlling some flow of ingredient. There can be two cases for determining how long to keep a valve open.

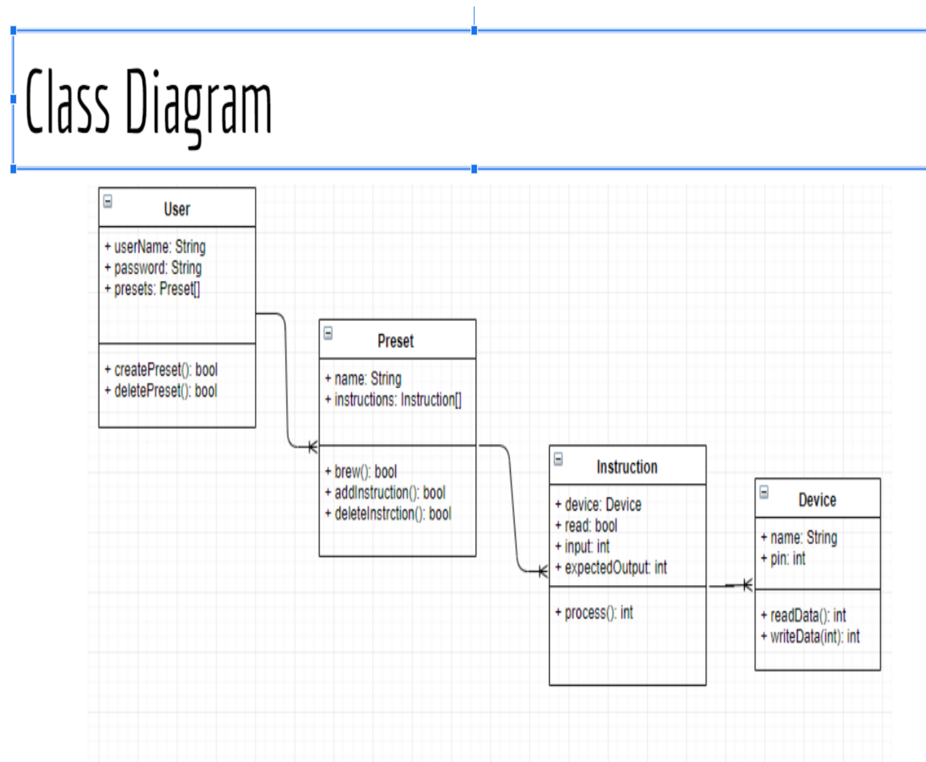
If we want to release a specific amount for a period of time where we assume flow rate to be consistent, we can just designate an instruction to have a time-period for keeping the valve open. Again this works only under the assumption that flow-rate is consistent. The error is that they may not be. We can consider an alternative approach, but we will see even that has it's issues.

Suppose then that we decide to take measurements of the weight of the container as we dispense a fluid into it. Then we simply need to set a small enough periodic interval for checking the fluid that the system areable to hit the target within a reasonable margin of error. However there is also a drawback if we were to only rely on this method. Consider if the amount of a concentrate being added is smaller than the weight sensor is able to measure.

The idea is that if we can't rely on the weight sensor, use a periodic value for setting the dispense time.

If the change in weight is easily measurable, it would be better to rely on the weight sensor than setting the periodic interval.

Recall that this is only a single instruction we're considering. A preset is made based off a series of instructions. So a preset must contain an array of instructions, each with a type and a target (valve).



Next, we need to define an instruction.

6.13.9 Instruction

As noted before, instructions can come with two types, either a measurement of time or a measurement of the beverages weight. So an instruction needs some way of determining which functionality to implement. We can have an enumerated type for determining the how to measure the success of the dispensing. Then an instruction will have a targetWeight and a timeToWait value. Only one areused depending on the type.

The last component of an instruction is the valve to operate on. Suppose that we have N valves. We could use an enum as well to declare which valve the instruction should use. So the Instruction will have a targetValve that is equal to the enum of the valve it corresponds to.

Thus we have,

```
enum instructionType { Time, Weight }
enum valves { valve0, valve1, ... , valveN }
struct Instruction {
        enum instructionType type;
        enum valves targetValve;
        float targetWeight;
        float timeToWait;
} typedef Instruction;
```

The value that targetWeight should take is simply the reading we want to see on the weight sensor. However the time to timeToWait is a little less obvious. Is it seconds? Is it in milliseconds? Is it even good practice to have it in seconds? Yes, for abstraction purposes it would make the most sense to store it in seconds. The reason is that this data type is stored from what the user enters. It's normal to think in terms of seconds instead of clock cycles. However our program must convert that into clock cycles when it's creating the ISR for scheduling the valves to turn off. So we can assume that the function that calls this makes use of this data type will need to be aware that the time is in seconds and adjust accordingly.

6.13.10 Storing Presets

There are two possibilities: we could store all presets in a single file or create a file per preset. If they're stored in a single file, the search times would be small for our preset we want to delete or edit since we're allowing a maximum number of 32 presets. There really is no gain from complicating this by using a file per presets so we will opt for a single file.

It's worth noting that we want to know how many presets exist in a file before add or lookup information from it (considering the case where there are no presets to load). So the first byte of the presets file should just be a number that lets us know how many presets we have stored.

Then we look at a file, we can look at only the first byte and know exactly how many presets exist. Then we don't have to count all the presets when we add a new preset. We just look at the first byte, increment the count and append the new preset.

We should also include another header byte. The default index. This tells the software on bootup which preset to begin using for manufacturing by default.

The user should be prompted each time before they begin brewing, by seeing some type of selection of presets. But the first preset that is already in the selection box arethe one that is marked as the default.

6.13.11 Instruction Data Size

Now we aim to calculate the total memory that's going to be used by the presets. First we look at the size used by Instructions.

- InstructionType enum has two enums so it is 8 Bytes (Not dynamic)
- Valves enum is 4N Bytes where N is the total valves (Not dynamic)
- Type is 4 bytes
- TimeToWait is 4 bytes
- TargetValve is 4 bytes
- TargetWeight is 4 bytes

So the total number of bytes per Instruction is 16 Bytes. Instructions are by in large the biggest part of the preset schema. Since there could be many instructions per preset.

We can optimize this by choosing to not use an enum for the any of the valves or the instruction type. An alternative solution is to define the valves and types at the header space of the program using a number on the range of a byte. Then we could set the values using those predefined expressions and change the data type to a char. This would reduce the bytes per Instruction to 10 bytes.

Assuming that the references to the pins can be extracted from only one byte is safe.

6.13.12 Preset Data Size

Now we will consider the size of a Preset

- First there is the name. We will restrict preset names to be 16 characters. So name has at worst 16 bytes (1 for each char) and 4 bytes for the pointer. A total of 20 bytes.
- The Instruction pointer is 4 bytes and the array of instructions will by C16 bytes. Where C is the total number of Instructions. Thus 4 + C16 bytes.
- Lastly, there is only 1 Byte for the numberOfInstrucitons.

Thus the total number of bytes per Presets is 25 + C16. It has been previously expressed that the maximum number of presets is going to be 32. If we allowed this we would be using 800 bytes for constant space and 512C bytes dynamically. That means for only one instruction, we are looking at a worst case of over 1KB.

We can see a table of the approximations below:

Total Instructions Allowed (C)	Kilo-Bytes

1	1
2	2
4	3
8	5
16	9
32	17
64	33

Table 15: C versus KB

16 Instructions seems like a we'd be giving more than needed. For our purposes, no drink arethat complicated. Our system doesn't have the many valves to even consider. And since each instruction maps to a unique valve, and there would be no need to use the same valve twice, we only need to allow for as many instructions as there are valves.

Thus the correct approximation for this file is to use C = 8, which concludes 5KB for our presets.

6.13.12 Revisiting Logging

We had previously mentioned the aspects of logging but now we will visit them with a more granularity.

Given the chip selected for this project, we can expect about half a megabyte of megabyte of flash memory if we decide not to use some external storage. Then we need to consider that logging should be restricted.

In general, we know that we want to log approximately 2 different types of data. Logging the temperature of the water bath, and the weights of the beverages produced. It is possible to allow these measurements to go on indefinitely but that's obviously reckless and asking for trouble. So we know we need to limit the amount of data being logged per tracking target.

To know how much we should restrict logging, we need to know the size of each data point. Before we considered marking the time of the data point so that would be able to tell the old data from the new data. Marking data like this is intuitive but costly in memory.

An alternative is to remove the need for timestamping and keep an index of the most recent value appended to the file.

Imagine the data file organized like so

<dataPoint1>
<dataPoint2>
<...>

<dataPointN>

Note that the X in dataPointX does not define it's relation to the other data points in time. Only in index. When we open the file, if there was an index marker that told us the most recent dataPoint added was index k, and we wanted to append a new piece of data, we would only need to replace the data value of index (k + 1) % N. Then we increment the index and mod it by the maximum index + 1. Note that N is being used to also mark the maximum number of data points we allow for storage.

If we always use this method for replacement, we can always be certain the index k + 1 is the oldest entry in the file.

6.13.13 Restricting Logging

We have now secured a way so that we only need to store the value of each data point and not an additional timestamp. We only need to append and index and maximumIndex datapoint at the start of the file to be used each time the file is navigated.

We can assume each data point in the file is 4 bytes. This is because the water bath is measured in temperature, so it's best to use a float to store its data.

It would be possible to store the data in a single byte by truncating the float to an integer and storing that value in an unsigned byte. We are allowing ourselves not to worry about the case where the temperature exceeds 255 or dips into the negative. But we avoid that case altogether and allow for greater precision by simply using a float.

So the total logging file will by approximately 4N bytes where N is the total logs we want to keep track of. Even a modest 200 data points allowed keeps this log under a Kilobyte, so issues with logging and memory overflow will not exist.

6.13.14 Logging Headers

A final note on a logging file is the headers required.

- CurrentIndex – the index of the most recent data point
- Max – the maximum value seen by this log
- Min – the minimum value see by this log
- Average – the average value of all the values seen by this log.

6.14 User Interface Pathing

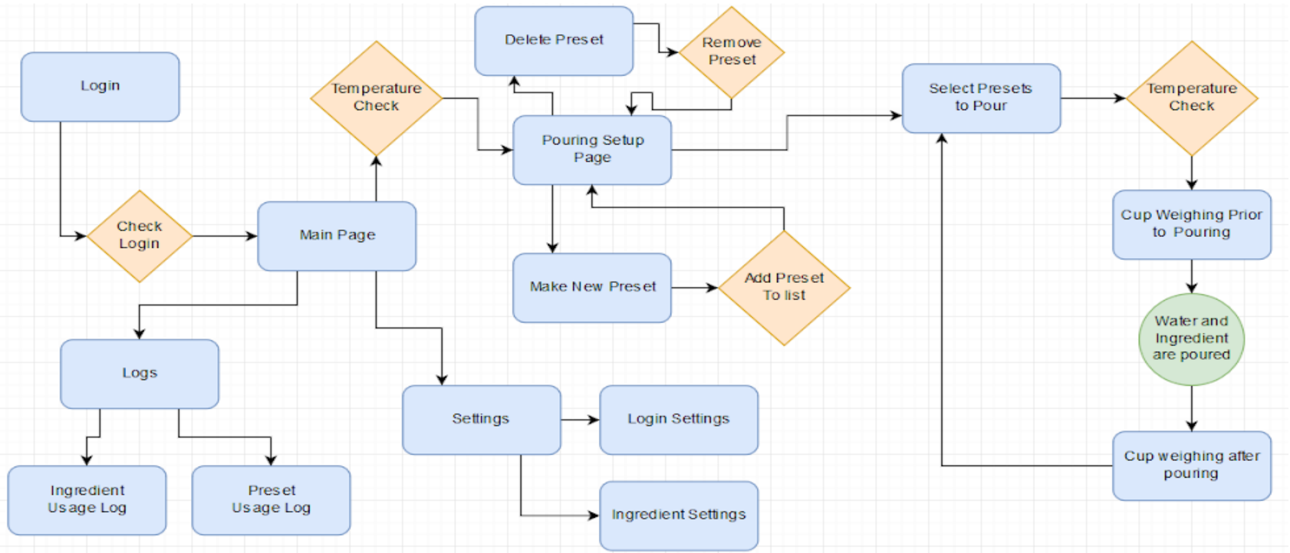


Figure 21: Page Flowchart for User Interface
Created using draw.io

The user interface will follow a set of screens, with connections between each connecting screen. These connections are visualized in the graphic above.

6.14.0 UI Buttons

The User Interface will make use of buttons to move between different screens. Each button will have three potential colors. The first color is the "standard/idle" color, which areused for show that the button can be pressed but is currently inactive. The second color is the "active/pressed" color, which are either green in the case of "forward" movement (for example, moving between the Main Page to the Brewing page) or white in the case of "backwards" movement (Logging out from the Main Page). The final color is the "restricted" color, which will appear as a grayscale version of the button; this is only possible on certain buttons when specific requirements are not met (i.e. The BREW button on the Main Page will not work if the temperature is not within the correct range).

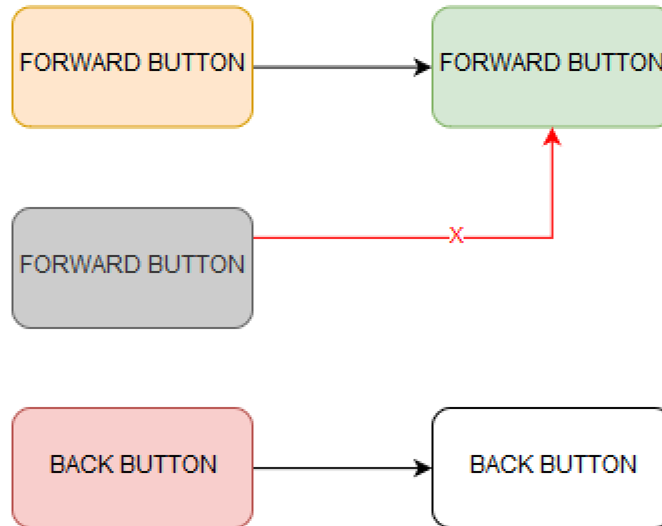


**Figure 22: Button Colors used for UI
Created using draw.io**

6.14.1 Login

**Figure 23: Login Screen used for UI
Created using draw.io**

The User Interface will always start at the Login screen. Here, the users are asked to provide a Username and Password. This are checked against a locally stored information system, which will ensure that the person attempting to access the Micro Manufacturing Beverage System has the proper authority to do so. Username and Password length are between 4-characters and 8-characters; the username and password are alphanumeric with capitalization and punctuation being critically checked. Should either the username or password fail, an error screen will appear detailing the mistake; the system will assume that the username

is incorrect if both username and password are incorrect. After logging in successfully, the user will move to the Main/Start Page.

6.14.2 Main/Start Page

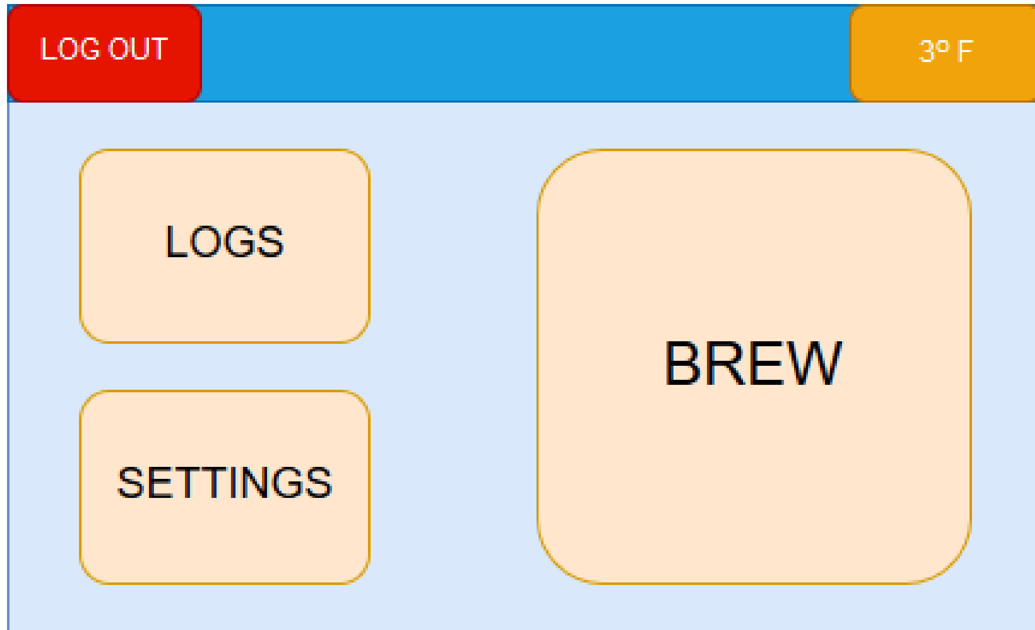


Figure 24:UI Design for Main Page
Created using draw.io

In the Main/Start Page, the user can select between three options to select what they wish to do: Logs, Settings, and Brew. However, the user may not be able to access the brew option if under improper settings. When the user has logged in, the system begins a sensor check. If any of the sensors fail to match their specified requirements, the option to access the Brew Page are greyed out and have a red triangle warning next to it. When this red triangle is chosen, an error message will appear; this error message will explaining which sensor (or sensors, in the case of multiple failures) has failed. The system preforms this check every few seconds to ensure that all sensors are up-to-date. Should the user wish to log out, they may do so from the Main/Start Page. At the top right of the Main Page, the temperature should be displayed.

6.14.3 Logs Page

In the Logs Page, the user can access information about the operations that have taken place at the Micro Manufacturing Beverage System. The user may choose between options to view what information they are searching for. If the user chooses Ingredient Usage Log, the user may view the current amount of both ingredient and water usage based on the current run, with the option to reset the

amounts if needed. If the user chooses the Preset Usage Log, the user are given information on how often a preset was used, again with the option of reseting the log if necessary. The user may leave these options and return to the Main/Start Page via a return button.

6.14.4 Settings Page

In the Settings Page, the user is given options for changing system options. If the user chooses to Login Settings, they can change the username and password for a user account. If the user chooses the Ingredient Settings, the user can change the name of ingredient, its density, and the safe temperature of the ingredient such that the safety checks for brewing are passible. The user may leave these options and return to the Main/Start Page via a return button.

6.14.5 Brew Page

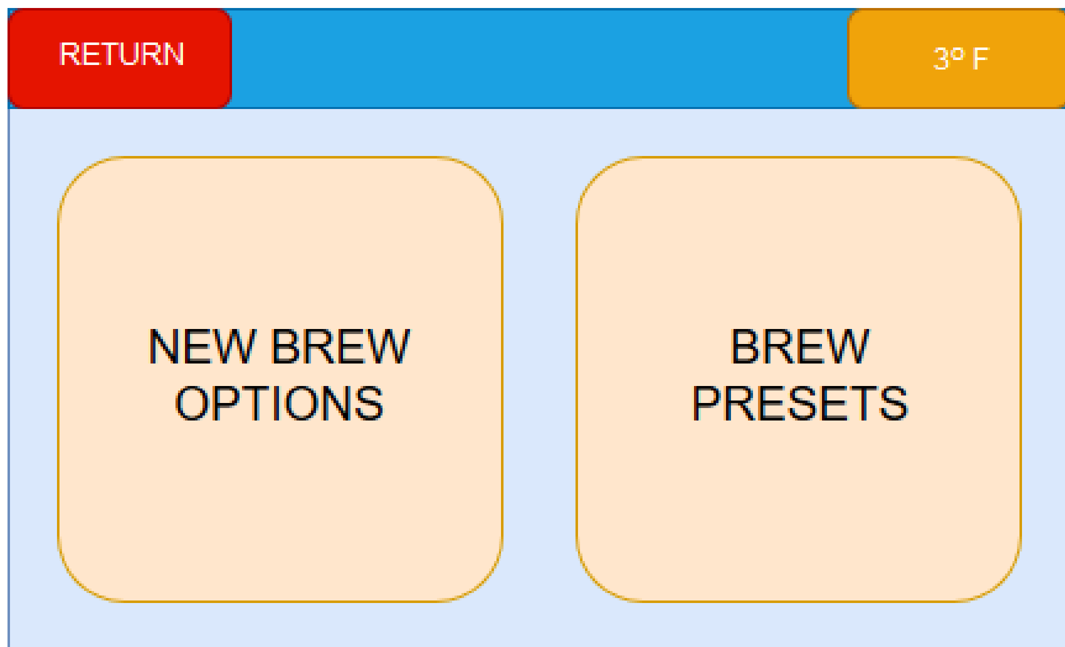


Figure 25:UI Design for Brew Page
Created using draw.io

In the Brew Page, the user is given two options: the user may create a new brew, which will send them to the New Brew Options Page, or the user can choose from a preset that had already been made via the Brew Preset Page. If the sensors checking the system find an error (i.e. the temperature sensor finds an incorrect temperature for brewing), an error message will appear displaying the error and send the user back to the Start/Main Page. At the top right of the page, the temperature is displayed so as to ensure that the system is operating appropriately.

6.14.6 Brew Preset Page

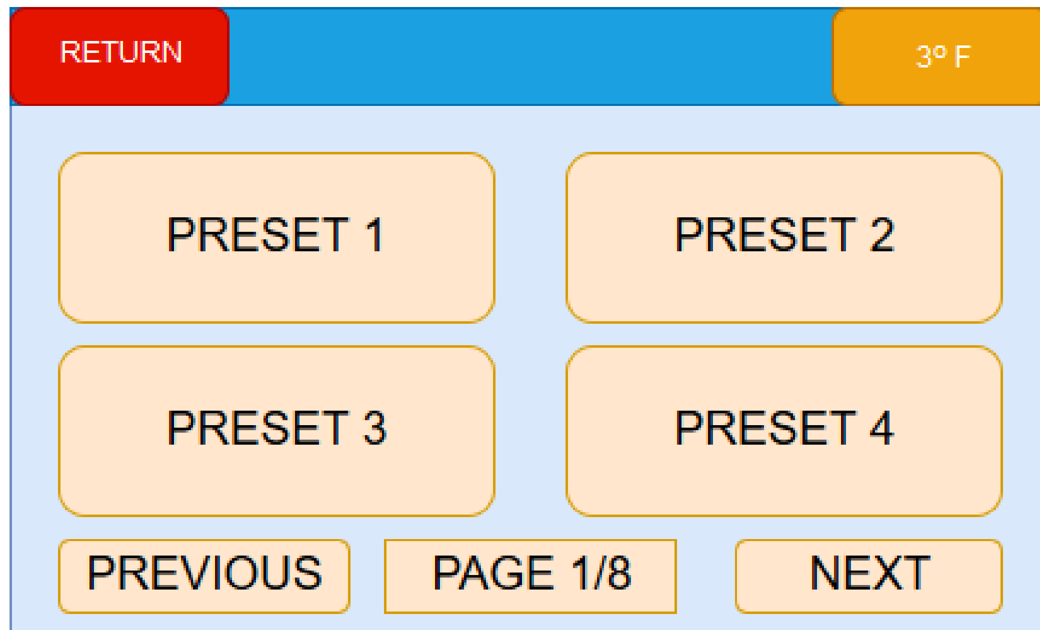


Figure 26: UI Design for Brew Preset Page
Created using draw.io

In the Brew Preset Page, the user can select between the presets that have already been made and saved to the system. There are 4 options per page, with the option to switch pages if the preset the user is looking for does not appear. When clicking on a preset, the user can choose to modify the preset or simply select it; if modify was chosen, the user is taken to a page displaying information on the preset, including the preset name, the amount of preset in percentage, and the total volume of the preset. The user may choose to delete a preset from this screen as well. The total number of presets possible are limited to 12 to prevent potential memory issues.

6.14.7 Brew New Options Page

In the Brew New Options Page, the user attaches a name to the new brew and specifies the amount of ingredient used in percentage and the total volume of the beverage. The amount used will translate to the following formula:

$$A_{total} = A_{activate} + (F * t) + A_{deactivate}$$

Where A_{total} is the amount of an ingredient, $A_{activate}$ is the amount added to the brew as the valve is opened, F is the flowrate under normal circumstances, t is the

time the valve areopen, and $A_{deactivate}$ is the amount added to the brew as the valve is closed. The reason that A_activate needs to be considered is due to how valves may leak on start-up and A_deactivate needs to be considered because of lingering/remaining ingredient in the tube-line after the valve has been closed. When the options have been decided, the preset are saved and the user may move to the Pour Drink Page. If the user does not wish to do this, the user can return to the Brew Page.

6.14.8 Pour Drink Page

The Pour Drink Page has little user interaction and is more of an information page, detailing which ingredient is being poured into the brew at the current time, as well as giving a brief estimation of the time remaining for the brew. At the beginning of this page and in between ingredients, it will ask the user to weigh the container that will be/is holding the drink so as to ensure that all parts are being measured correctly. If any of these parts are not correct (i.e. too little of an ingredient was poured), it will inform the user that the batch was bad and that the user should get rid of the drink; in the event that it is too little, there will be a red negative sign, while too much will show a red positive sign. Afterwards, it will allow the user to return to the Brew Preset Page (where they can choose their next option), create the same brew again, or go to the Main/Start Page.

6.15 Front/Back-End Memory Read/Write

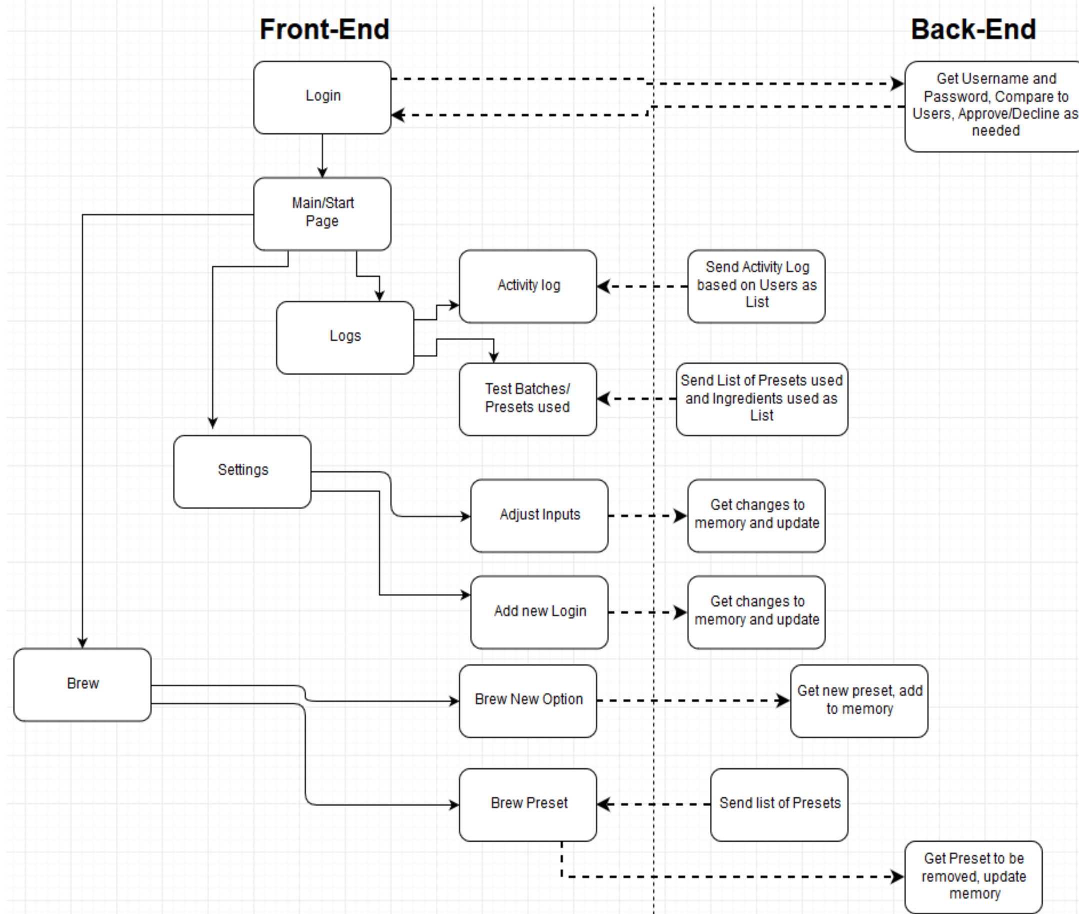


Figure 27: Front-End and Back-end Relation for User Interface
Created using draw.io

The relationship between the front-end (what the user will see) and the back-end (what the user will not see) is an important part of any project. When dealing with a device that are used for testing different drink combinations, memorizing certain pieces of information carries significant weight, and it is important if not tantamount that there is a clear understanding of how and when memory is affected in this project.

6.15.1 Login

The first memory access (regardless of the intention of the user) occurs when the user logs in. The user inputs their username and password into the front-end, which is sent off to the back-end. From here, the username and password are compared with the information stored in the memory. If the usernames and passwords are correct, then a “pass” is sent back to the front end and the user can enter the Main/Start Page. However, if either the username or password do not match with the recorded memory, an error screen displaying this is sent to the

user, preventing them from advancing; if both the username and password are failures, the system will assume that the username is incorrect.

6.15.2 Activity Log Page

When accessing the Activity Log page, the back-end sends the actions committed by each user as a list ordered based on most recent user activity and what actions they took. This allows for security checks to occur if any unauthorized access occurs.

6.15.3 Test Batches/Presets Used Page

When accessing the Test Batches/Presets Used page, the back-end sends a list of usages for each preset (listed in order of most recent usage). It will then list the usage of ingredients. From here, users can view which combinations prove popular for testing, and adjust inventory management as needed.

6.15.4 Adjust Inputs Page

When accessing the Adjust Inputs page, the front-end allows the user to change values for the inputs available. When the user has finished making changes, the front-end sends the changes to the back-end. The new information is written to the memory, updating the system with the new information.

6.15.4 Add New Login Page

When accessing the Add New Login page, the front-end allows the user to add an additional user. When the user has finished setting up the new account, the front-end sends the changes to the back-end, where the new user is written into memory.

6.15.5 Brew New Options Page

When accessing the Brew New Options page, the front-end allows the user to create a new mixture. When creating a new mixture, they can add a name to the mix while deciding on the proper ingredient amounts. When the user has brewed the beverage, the user can choose to save the mixture as a preset for next time; if the user chooses to do this, the front-end sends the recipe along with the name to the back-end, where the preset is written into memory. It should be noted that this cannot occur if there are currently 32 active presets already written into the system;

this is to limit the data changes and ensure that the user doesn't stockpile recipes that will not be used.

6.15.6 Brew Preset Page

When accessing the Brew Preset Page, the back-end sends a list of presets to the front-end upon page entrance. The list is limited, only showing 4 presets per page; to list all 32 presets on one page would be impossible with the size of the touchscreen available. To access other presets, the user will need to choose the next/previous page option, which will load up the next presets on the list. Should the user feel the need to delete any preset, the front-end will send the back-end a deletion notice, which will lead to the preset being removed from the list and the memory being updated via writing the new list. For the purposes of operation, this removes the preset at the current position, and moves all other presets below it up by one.

6.16.0 MERN Stack

In a previous section, we noted the dangers of using an embedded approach with Javascript as the core language and reliance on the Java Virtual Machine. If the resource expenditure is deemed appropriate, then we can now consider the solution methods allowed by that framework.

Considering the problem of login, we need the ability to host a server that has internet connection and is able to write to pins of the board on command. As noted before, this means that we need some level of multithreading or at least the appearance of it. Add in the complication of producing a UI as well, and the idea of a web-application may come to the mind of some experienced software designers. However we are dealing with an embedded environment. Knowing our limitation on memory and speed might cause us to disregard the idea of a MERN stack outright. However given the strength of the CPUs being considered and the disk space provided by the SD card, our only limitation would be the need for an Operating System and the memory available to support the Java Virtual Machine. Let us define the components of a MERN stack and it's feasibility.

The MERN Stack consists of MongoDB, Express, React, and Node.js.

6.16.1.0 MongoDB

MongoDB is a non-relational database that stores data in groups known as Collections. Each collection is defined by a Mongoose Schema and are JSON (Java Script Object Notation). A JSON object can be thought of as a HashMap where each variable name is replaced by a key and the value is just an object. Since it's just an object, it can be anything. A string, an integer, or another JSON. For the purpose of our project, we could create a Schema just for authentication. An example would be,

```
const Schema Authentication = new Schema ({
        loginName: String,
        password: String
        userID: String
}
```

Of course these would not be stored in plain text with their true values but would be stored in their encrypted form by some algorithm similar to the one described in the above sections. We could also easily create similar schemas for our logging. Using the struct defined before but in Mongoose Schema form.

The userID variable is what maps the authentication object to the user object which holds all the user's presets. Assuming login is successful, the user object can be passed to the instantiation of the session formed with the device.

6.16.1.1 MongoDB Local Storage

There are two options for where we can store the database. One is external on some third-party hosting. This would mean connecting to them through the internet and sending the queries via HTTP. The other is host a Mongo database on the Pi. Ultimately the implementation of this is largely the same as if we were to have it hosted externally.

So what are the benefits? Well for one if the design is decided to not allow wireless connection to the internet, the database would still be available. The downside if that our own machine needs to process lookups and writes instead of using the machine of the hosting server. So although we gain some degree of independence, we suffer a loss in speed/performance.

A large downside of this is consider when the system is offline. If we wanted to review the system's data remotely, we only have the connection to the system to

do so. However that connection no longer exists. Thus our ability to check the data around the clock a direct result of the system being online around the clock.

6.16.1.2 MongoDB Remote Storage

There are many third-parties that offer the ability to do this. A free one with reasonable space is Atlas. It offers about 500MB for free which is much more than what we'll need. Connection is easy to be made via the Mongoose library and has prewritten functions for writing queries to the database. All that's needed is the key and login information of the database which is made upon creation. Similarly, a key is used for local database management as well.

The most glaring benefit of using a third party for storage is that the only processing we do for retrieving and saving data is the request. This request is made through an HTTP call. So in reality, the only processing we do it an HTTP request. We leave the actual navigation of the database, storing, and uniqueness validation to the remote host.

The second bonus of this is that the data stored there is always available to us. This of course that the third-party will always be available which is actually one of the faults of this approach. Placing our data on a remote server means relying on them to always have it available and their servers up. This might not always be the case. So it would be wise to backup our data on one of our own servers. Preserving the data and the functionality of it in the event that the remote database is temporarily or permanently lost.

The overhead of this is that the WiFi of our device will need to be reasonably fast if we're going to making perhaps a couple hundred write/read requests a second when we consider logging. The exact amount of times we may want to do these operation may be bottlenecked by our connection if we decide on using an external storage.

6.16.1.3 Mongoose

Mongoose is an open-source library that is used to communicate with a MongoDB. It is able to form the connection and then make queries. It provides a compact approach of this through methods of a mongoose object.

If you recall, a userID was placed in the AuthenticationSchema. The reasoning was to be able to directly reference a user object containing a set of presets designated by that user. So once a user logs in, how would I know who that login information belonged to if I did not have a pointer to it? Would it have made more sense to place the Authentication information inside the User Schema?

Consider that you wanted to enter a Rome and you required a pass. Every resident and traveler to that city has a pass to get in. The pass is itself a unique object that defines who can use. But does the pass identify anything about the person using it? Certainly not. It is the person who can use it that defines the pass. So it follows that since a user defines a pass, the pass should not contain information about the user other than a reference to them. Similarly, the user doesn't need to contain information about the pass. So Authentication receives it's own Schema that contains a reference to the user. And the user contains no references to the authentication.

Now consider how this containment would work for our application. Once a login is received, we would need to search the database for entries matching the username given. The username must be unique and this is enforced at the time a Schema is created with

```
loginName: {
        type: String,
        unique: true
}
```

Mongoose then provide find() function for the schema. You can pass a JSON object containing the parameters that need to match the object being returned. Note that this is not a promise that there is only one object in the collection matching that description. Thus find() returns an array of object regardless of the number of object found. Once we receive the object, since the schema guaranteed the username was unique, we know the array areof size one and the authentication object we are concerned with areat the zero index.

Now that we have the object, we confirm that the passwords match by first running the given password to the encryptor. If the two encrypted strings match, we can then allow for authentication and create a session.

Recall in the session, we need to include the user object. Now we make another call using mongoose, findById(). In MongoDB, when an object is added to a

collection, it is automatically added with a unique id under a _id field. So when we create an authentication, we would simultaneously have created a user object. That user object would have a unique _id and that is what we would store in the Authentication userID field.

With authentication being accepted, we use the findByID function and the userID field returned from the login request, to search for the target user and stores their information and preferences into the session created for the current user.

Additionally, there are other functions that will serve us well for updating, and deleting data from the database.

Lastly, a note about using Mongoose to delete values from a JSON object is that it is not simply enough to rewrite the object by saving a cloned version with the intended object removed. MongoDB favors , and rightfully so, explicit deletion. That is that simply because the field was not placed into the copied object, does not necessarily mean that you want to delete that object. It may have been the intent of the user to update the series of fields included and preserve the ones ignored. Thus to remove a field and its value from the object, you must save it with field included but with the value set to undefined. The undefined value is considered a flag for deletion and willfully expresses the intent of the user to delete that field.

6.16.1.4 Conclusion

Recall the concerns faced by using files as our pseudo-database from the C only approach. There were problems considered about using Global pointers, opening and closing files too often, and the problem of synchronizing data between tasks due to not know what data is new or old.

A remote MongoDB does not remedy the issue of what data is new and what is old for logging. If a call is made to the database to retrieve logging information, that data still needs to be parsed to update the frontend for the new values.

But it does remedy the worry of opening/closing files and/or an unwanted reliance on global values. This means to we avoid the computation and complication overhead of the solutions offered to those problems for the tradeoff on relying on an internet connection.

6.16.2.0 Express

Express arethe framework for Node.js. It is provided under the MIT license and areour main access point from the internet.

At initialization time, we designate to the express objects what ports it should listen to. Thus we must guarantee that a port is open on our device.

6.16.2.1 API

Consider a user a remote user is attempting to make a call to the device to begin brewing beverages. It must have some way of communicating exactly that intention. Or consider that the user wants to change the polling rate of the sensors, or delete old data, or set new preferences. For every option the user has, there must be a clear protocol between the device and the user's application of what they are allowed to do and what information they must send over to achieve that result. Thus the device will need some Application Protocol Interface. Express offers a simple way of implementing this via routing.

6.16.2.2 Routing

The idea is that now that we have an endpoint open for our device, we can use a similar idea to subnetting for specifying intentions. So suppose that we have opened our devices' server on http://localhost:8080. Now we know we have a collection of authentications and a user wants to create a new authentication (create a new account). We specify an endpoint for them do this on through express. In particular we can route all actions regarding authentication to a single endpoint. http://localhost:8080/authentication.

This is handled by Expresse's router object. Once we instantiate a router, we can specify it's function by using

```
router.<HttpRequest>("/function", function( … ))
```

- router – is the router object
- HttpRequest – is the HTTP GET/PUT/… request made by the application
- "/function" – is the endpoint appended on our host domain
- Function() – is the actual logic to be followed.

So for authentication it might an authenticate function which would could be a GET request.

Now that we have defined our routing, we simply include it in original express object specifying the /authentication maps to routes we created for it.

6.16.3.0 React

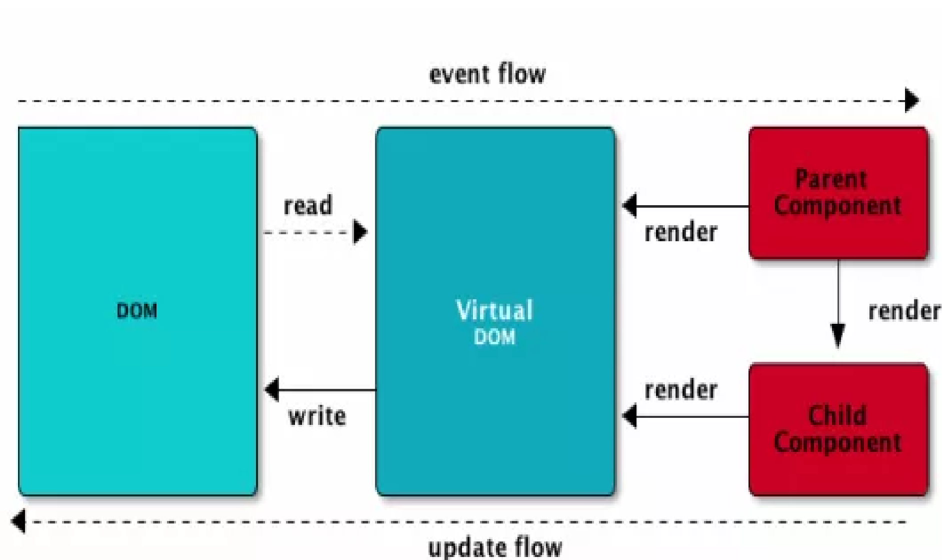


Figure 28: React Overview
Courtesy of https://blog.frevvo.com/reacting-to-react/

Now recall before that we were using the UTFT library to create our User Interface. In the MERN stack we have React. React is a set of libraries that places the HTML code normally used for front-end development into more modern looking versions of itself with some added capabilities. Overall React is essential to the project and could be easily replaced with just standard HTML without any noticeable loss in function. And perhaps even a slight increase in response time since. The reason for this is that the libraries used by React are ultimately still HTML in some form with added CSS to make it looks a better. Thus we'd be using the base HTML with a flavoring on top for added aesthetic.

A downside of React is that it requires a considerable bit of memory to use. The library alone is approximately 300MB. So the rationale is that if we're going to use a large SD card, it might be useful to take advantage of the tools given to create the most visually pleasing User Interface.

There's no reason to sacrifice a chance to make our application look better if we have the resources for it.

6.16.3.1 Components

In React the HTML code is divided into components. Components are the like the blueprint of the HTML but written in a similar syntax. Like a house being built, React provides the language with which to measure and specify how the house should look and feel. The HTML code being the cemented establishment derived from the React component specification.

Because of React's popularity, it has gained a wide spread code support from its users. As a result, most things you can conceive of wanting to do on web page, React has some component already made to help with that.

Consider you want to make a button on your page. In HTML a simple <button> would work. But there's a limitation to the original HTML button design. They were intended to be used submitting forms. So the attributes you can specify in only HTML directly follow only from that functionality. You can tell the button where to send the data (URL) and what method to use to send it (POST, GET). However you cannot specify code to be done before submitting or just to run some set of code independently.

That is, suppose you want a button that updates a current list of items on screen. But the screen is quite dense with information and you're typing up some more information on other parts of the current page. Thus you don't want to reload the whole page. You only want the data that needs updating to be reloaded.

This is where React starts to show its value. Because React was designed with the knowledge that it would run alongside Javascript, it implemented ways of deffering variable name calls and even entire processes to Javascript equivalent. So suppose you had some function written called updateItems(). A React Button has a property onClick which expects to receive a Javascript function. Then when the button is clicked, it's that function that is called. So no promise is made to reload a page, or direct input somewhere else.

This is liberating because now we can allow a button to have arbitrary functionality.

6.16.3.2 App.js

This is the manifest file of a React application. When Node.js is started, a script is ran, react-start. This script looks at the /src directory of a project and looks for the App.js file.

This file is used to specify to the application all the Routes to be used by the application. Similar to how we specified the router for Express, we specify which endpoints correspond to which Javascript files we've created. That is supposed we have written a page for Logging called Logging.js. When a user is on our domain, we want the path there to be http://localhost:3000/logging. Note the change in port as opposed to the routing API we specified earlier. That's because it's good practice to decouple your API from your frontend application. The reason being if you want to work on one portion and it starts misbehaving as a result, you haven't lost the functionality of the other portion.

In order to map the Logging.js to that endpoint we simply add a route to it in App.js

```
<Route exact path="/logging" component={Logging} />
```

Using this for all the pages needed by the application provides an easy to maintain, top level view of the application's components.

6.16.3.3 render()

When a component page is being loaded, it looks for a render() function. This function is the page but serves return the components and their functionality to the called which is the main process running the application, Node.js.

The render function is called when new changes happen to the screen. But Node.js is clever. It doesn't simply need to redraw all the components. By default, when the state of the page is changed, render is called, but it can be specified what to do on update so that the entire page isn't reloaded which was that problem we wanted to avoid with the original html button.

shouldComponentUpdate() is the function we can overwrite on each component to tell the component on a change of state exactly what we want it to reload. Thus

with some care, we can specify only the portion that requires reloading and not the entire page.

6.16.3.4 React state

Consider a user logging into a system. Once the user logs in, his information needs to get passed from the login page once authentication is confirmed, to the next page. And the next page, and so on. It's a set of data that needs to be preserved and maintained throughout the use of the application.

Enter the state. A state is a variable that exists at component level that holds all the relevant data for that component. Thus all functions being used within that component have the scope to view the state. It's a common approach to pass states between pages. So whenever there is a link being used by one component, some state data is preserved and passed in to the next component. Notice how this is different from global state. Instead of existing at a higher level of visibility than the current component, it exists at component visibility. And a copy is passed to the next component visibility when a call to change components is made.

But there is an alternative. State passing may be useful, but having a global state has the benefit of allowing for multiple pages to be interacting at the same time. That is suppose that a user has a session on our application. On page they have opened is to the logging information. They are currently tracking changes in the production. Simultaneously they have a page open to the presets page. They are in the process of changing the presets. Deleting or creating new ones. In particular they are changing the presets of the current log they are tracking on the other page. So once the changes are made on the one presets page, the logging page should immediately start reflecting the change. If the data was only being preserved in local state to the components, this would not be possible. The new updated state from the presets page would need to be passed to the logging page and the logging page reloaded. However we circumvent this problem entirely by providing a global store of data that is within scope to both components simultaneously.

6.16.3.5 React Redux

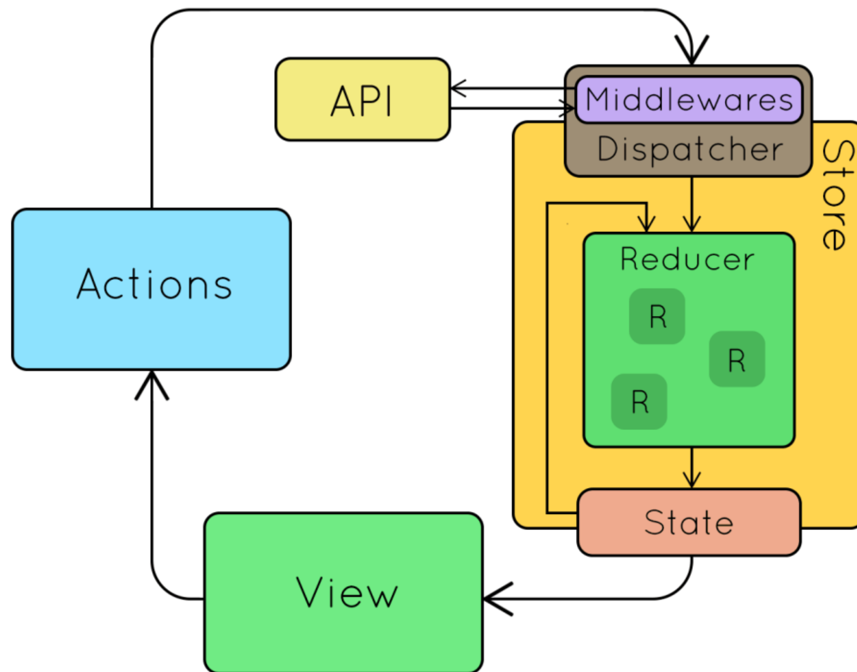


Figure 29: Redux Data Flow
Image courtesy of Github Open-Source reduxjs Project

Redux is another open-source library to aid in the development of React projects. It’s function is to create, maintain and secure the protocol for a global store that other components can share a live feed to.

Given some Action by the user, Redux uses a dispatcher function specified by the developer to update the necessary data. It can use middleware to connect with the API routes written from Express. And then uses a reduces to merge the data into the new data into the store. The View, or the Component in our case, is constantly monitoring the store’s state and on a change is able to use the new data.

6.16.3.6 onComponentDidMount()

The next key function when a Component page is loaded it the onComponentDidMount(). It is called before the render function and serves to do any preprocessing the page requires before displaying the information to a user. An example might be updating visits, or verifying the state of the Redux store.

6.17.0 Axios

Axios is another open-source library that is useful for our project. As mentioned we are hosting our API on a Node.js server. So it areopen to the internet and the API is accessed by making calls to the endpoints of the server. So if our components are running on Javascript code, we need code that can hit those endpoints. This is how Axios comes in.

Axios allows Javascript to make HTTP requests to any endpoint desired and include a body or set of parameters in the requests.

6.17.1 Node.js

This arewhat actually runs our server. Express and React setup our logic and Node.js opens to ports on our device to be open to the internet.

With this, we've completed our MERN stack.

6.18 Tentative Schematic Design

The following is the tentative schematic design for the project:

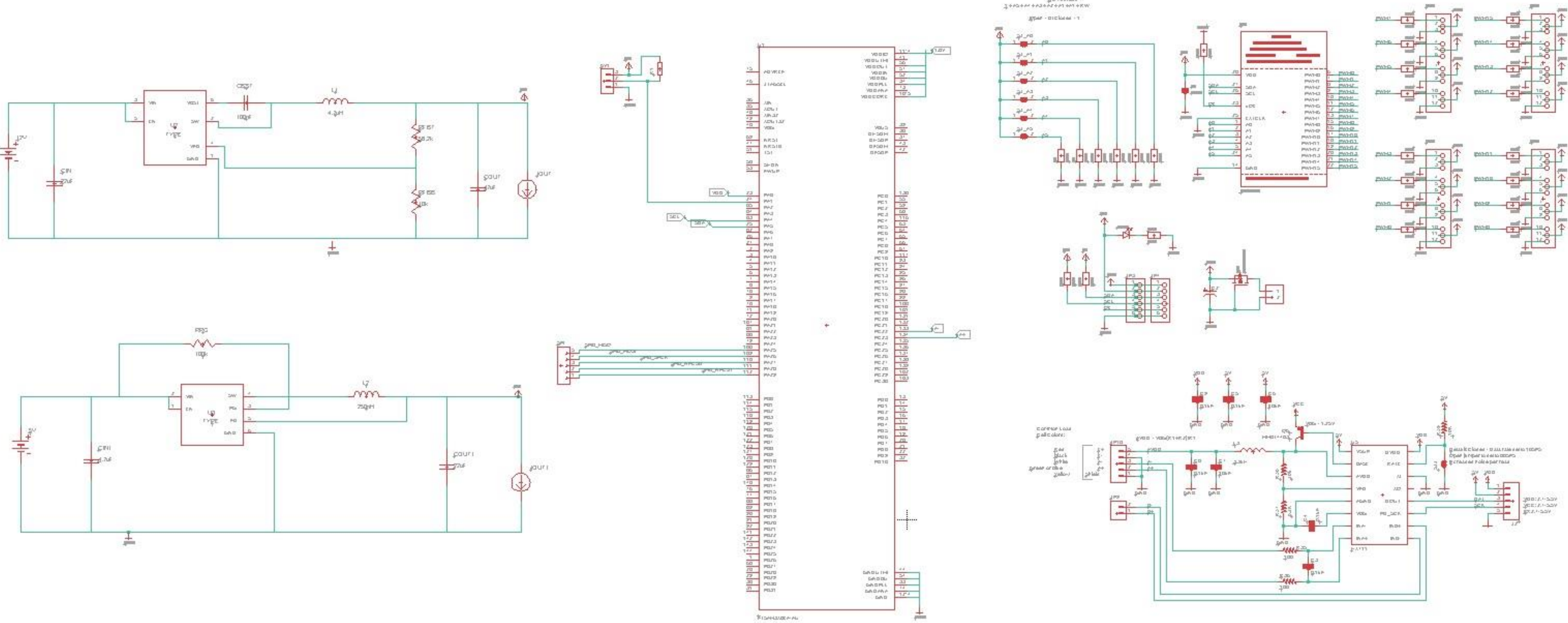


Figure 30: Tentative Schematic

In this schematic it consists of 7 main parts:

- Microcontroller
- 12v to 5v converter
- 5v to 1.8 converter
- 16 Channel Servo Driver
- Load cell amplifier
- Touchscreen
- Thermometer

These 5 parts are all going to be part of the PCB that's going to be used in this project. In senior design 2 this schematic design is more than likely going to be changed either due to a design change or to make the schematic more efficient ect…

6.18.1 Microcontroller

The microcontroller is the central part of the schematic design. Everything is connected to it. The important thing to note about the microcontroller is the power and gnd pins.

The following power pins are connected to the 1.8v power source:

- VDDIO
- VDDOUT
- VDDIN
- VDDBU
- VDDPLL
- VDDCORE

The following ground pins are grounded:

- GNDUTMI
- GNDBU
- GNDPLL
- GNDANA
- GND

6.18.2 12v to 5v Converter

This 12v to 5v buck converter steps down the voltage from the power source which is 12v to 5v. The 5v power source is used to power the servo driver and the load cell amplifier.

The 5v voltage that has gone through this converter is then used in another buck converter in this schematic.

6.18.3 5v to 1.8v Converter

The 5v to 1.8v buck converter steps down the voltage from the 12v to 5v buck converter so that microcontroller can be powered. The microcontroller has an operating supply voltage of 1.8V so using 5v or 12v would destroy the microcontroller and is the reason why the voltage has the brought down twice for it to be used in the microcontroller.

6.18.4 16 Channel Servo Driver

To make the 6 servos work in this project a servo driver must be used. The pins on the board are separated into power pins and controls pins. There are 3 types of power pins:

- GND
- VCC
- V+

The GND pin is of course connected to the ground. VCC is the supply voltage for the servo driver and must be between 3-5V and for this project all the VCCs are 5v. The V+ is an optional power pin that distributes power to the servos and isn't going to be used in this project.
There are 3 control pins:

- SCL
- SDA
- OE

The SCL pin is an I2C clock pin that connects to an I2C clock line on the microcontroller. The SDA pin is the same as the SCL but it would be connected to a different I2C clock line. The OE pin is an output enable and is used to disable all the outputs when high and enables all the outputs when low. This is an optional pin and won't be used in this project.

6.18.5 Load Cell Amplifier

For the chosen Load Cell the wiring will look as follows:

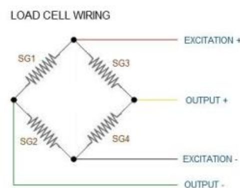


Figure 31: Load Cell Wiring

On the load cell board there are colors that are associated with each node on the Wheatstone bridge and the following table shows which color is associated with which node on the bridge:

Wheatstone Bridge Node	Wire Color
Excition E+ or Vcc	Red
Excition E- or Gnd	Black or Yellow
Output+ O+, Signal S+, or Amplifier + A+	White
O-, S-, A-	Green or Blue

Table 16: Wire Colors

These connections are found on the left side of the load cell board and connects to the amplifier. On the right side of the board there are 5 connections that need to be made to the controller for the load cell to function.

- VDD
- VCC
- DAT
- CLK
- GND

The VCC is the analog voltage that is going to actually power the load cell and the VDD is the voltage that is used to set the logic level. So for this case the VCC is 5v and the logic of the microcontroller 1.8 so VDD is 1.8V.

The GND pin is connected to the ground and the DAT AND CLK pins needs to be connected to the microcontroller. A GPIO pin on the microcontroller will work with the DAT and CLK pins.

6.18.6 Touchscreen

The touchscreen connects to the microcontroller through SPI. The pins that use SPI on the microcontroller are as follows:

I/0 Pin	Peripheral A
PA25	SPI0_MISO
PA26	SPI0_MOSI
PA27	SPI0_SPCK

PA28	SPI0_NPCS0
PA29	SPI0_NPCS1

Table 17: SPI Pins

A pin header is used to connect the pins on the touchscreen to the pins on the microcontroller.

6.18.7 Thermometer

A thermometer is going to be connected to a 3 pin header.

- VCC
- I/O
- GND

The VCC pin is going to be the 5v power that is used across the board. The I/O pin connects to the microcontroller. GND of course connects to ground. A resistor is connected in between the VCC and I/0 pin.

6.19.0 Software Final Design

Before we define the final design we acknowledge our hardware to work with. We are working with half a megabyte of flash memory, 64KB of RAM. Our libraries and expected data places us about 111KB of memory being needed just for our code to start. Thus we have a significant code space. But there is a complication. These libraries need to either be in the flash memory or in programmed to them. We know we can program the chip to run our code, but how can we load a library onto it before we program our code to run there.

Assuming that we can transfer the files to our chip, we're well within our memory restrictions. So there's no cause to be alarmed.

- We opt for the URTouch and UTFT libraries discussed in 6.13.
- The Cyclic Logging index technique mention in 6.13 Logging Revisited
- The system will not support wifi, or remote login
- And the we opt for the data types derived in 6.13

7.0 Testing

To ensure that the product is safe for use and effective, proper testing must be done on all levels; an individual part lagging may cause a cascading failure later. Through testing, a product can be refined to allow for more accurate measurements and smoother running of software.

7.1 Hardware Testing

Hardware testing aredone both on an individual component level, then with combinations of the components. Almost all testing will occur in room-temperature lab room, to help establish a medium through which all parts are equally tested.

Testing of the hardware begins with all components individually; this is both to ensure that all of the hardware that was ordered is operating correctly and to give a clearer example as range and speed of operation. Component testing beings with valves; allowing the team to know what makes the valve move and the limit of that movement set the foundation for future work on the project, as the valves are critical for the actual operation of the Micro Manufacturing Beverage System.

The next major set of components tested focus on the sensors. Without the sensors, there would be no clear way of understanding if the temperature, pressure, and weights for each ingredient are in a stable condition; if the ingredients are in an unstable condition, problems associated with brewing become much more likely to occur. As a result, the sensors are nearly as important to the b Micro Manufacturing Beverage System as the actual brewing components of the Micro Manufacturing Beverage System. Each sensor is tested individually and put through a small but noticeable variety of circumstance to help identify the best possible locations for each sensor. With the sensors in the right place, the chances of potential problems getting caught early before brewing increases greatly.

The last major set of components boils down to the actual brewing equipment, such as tubes are moving the ingredients or the pressurized gas for actual brewing. This is less about stress testing and more so about ensuring all parts are operating correctly (i.e. the tubes do not leak).

After individual parts have been tested, the system is built, during which additional testing occurs when several components interact with each other. It is during this stage that the timing used for the valve system during the brewing arefound; after running several tests, the group areable to narrow down the timing offsets required to account for turning the valve on and off with regards to ingredient management.

Upon full construction, the Micro Manufacturing Beverage System aretested to ensure that no parts were improperly installed. At this point, the testing that went into individual parts has made all members of the group intimately familiar with each component that they can identify any problems that may come up with the installation.

7.2 Hardware Specific Testing

1. Power Operations
    a. User turns power off for Micro Manufacturing Beverage System
        i. Expected Result: the Micro Manufacturing Beverage System should turned off almost immediately and becomes inoperable as a result of power loss

    b. User turns power on for Micro Manufacturing Beverage System
        i. Expected Result: the Micro Manufacturing Beverage System turns on quickly and begins loading OS for use within 10 seconds of power-on.

        ii. Practical Alternative: the Micro Manufacturing Beverage System may wait up to 30 seconds between power-on to operational state.

2. Temperature Sensors
    a. Temperature sensor is held in controlled environment at X degrees
        i. Expected Result: Temperature Sensor will have display X degrees as an output

        ii. Practical Alternative: Temperature sensor displays a temperature with 1 degree of the controlled X temperature.

    b. Temperature sensor is held in correct place, with value checked against a thermometer
        i. Expected Result: the temperature read by the thermometer is the same as the temperature found by the temperature sensor

        ii. Practical Alternative: the difference in temperature values between the thermometer and the temperature sensor are roughly 1 degree.

The temperature sensor is allowed to be off by a degree since the Micro Manufacturing Beverage System is not allowed to operate unless it is at least 4 degrees below the upper “spoil” point and 4 degrees above the lower “spoil” point. As a result, the temperature sensor does not need to be exact; the temperature sensor just needs to be able to tell if the temperature is significantly off balance for operational purposes.

3. Weight Sensors
    a. Weight sensor is tested with an object that weighs X grams
        i. Expected Result: the weight sensor will display X grams as an output
    b. Weight sensor is used in common operation and compared to another weighing device
        i. Expected Result: the weight sensor's output will match the output given by the other weighing device

Unlike temperature sensors, the weight sensor cannot have a sizeable level of tolerance due to how a slightly different amount of ingredient input can significantly change both the output that was requested as well as the level of leftover input ingredients. The weight sensors have to be significantly accurate; its values have to be within a gram of the expected amount at most, and even then this may be cause for an error.

4. Valves
    a. The system is given an input amount (via software calculation), and the valve is then sent a signal to open. After a calculated time (again via software), a signal is sent to close the valve.
        i. Expected Result: the Micro Manufacturing Beverage System will fill the testing container with the expected input amount.

Additional test may be necessary as the project moves from the conceptual phase and onto the building/coding phase.

7.2.0 Display Testing

The display used in this project is a 800x480 TFT touchscreen. In order to properly drive the display using an ARM Cortex M3, the RA8875 driver board from Adafruit was used. This board handles all of the parameters of the touchscreen including memory, refresh speed, and pin requirement. If the ARM Cortex M3 was used to drive the touchscreen, more than 40 pins would have been required. The RA8875 breakout board communicates with the MCU using SPI, and only needs 6 pins for communication and 3 for power.

The 6 pins used for communication are INT, RST, CS, MOSI, MISO, and SCK. These pins are used to transmit packets of data using SPI protocol to simply control of the board from an external MCU. The connections are shown in figure 25.

Since the RA88775 is a 5V chip, high-speed level shifters are used to translate logic between the 3.3V ARM chip. The on-board level shifter used is the CD74HC4050 from Texas Instruments. This level shifter has a very small propagation delay (approximately 10ns) which makes it ideal for driving a display.

The 3 pins used for power are GND, 3Vo, and VIN. The GND pin must be the same for all boards and power sources. This is a common ground between all components. The 3Vo pin is used in level shifting, since there is a voltage discrepancy in the logic. The 3Vo pin is used as a reference for the level shifters. The VIN pin is connected to the 5V power source. The display as well as the RA8875 IC are powered by 5V. This configuration is shown in figure 32.

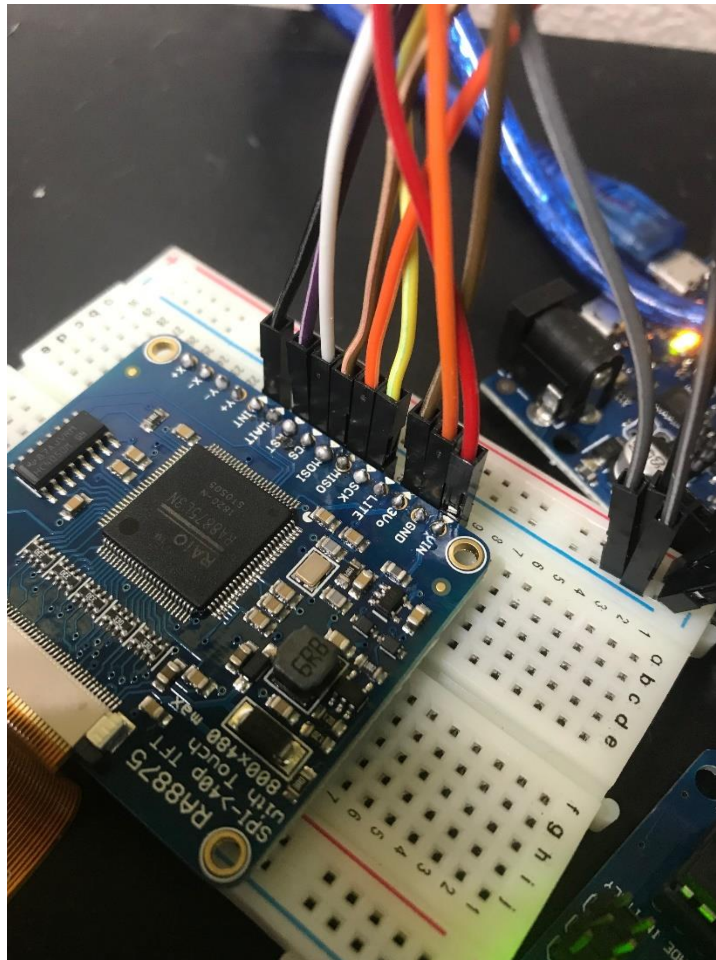


Figure 32: RA8875 Driver Board Connection

To test the display, the Adafruit RA8875 library was downloaded and example sketch “buildtest” was uploaded to the Arduino Due using the Arduino IDE. The code iterated through different display colors and finally allowed the user to draw on the screen. This is shown in figure 33, where the display is colored green. The Arduino Due is at the top of the image (circled in blue). The Arduino Due is connected to the RA8875 board (circled in pink) using 6 jumper wires for communication and 2 jumper wires for power. The power lines are GND and 3Vo. The Arduino UNO (circled in green) is used as a 5V power source. During actual operation, the Arduino UNO arereplaced by the TPS40305 5V regulator. The regulator is connected to the GND and VIN pins on the RA8875 board.

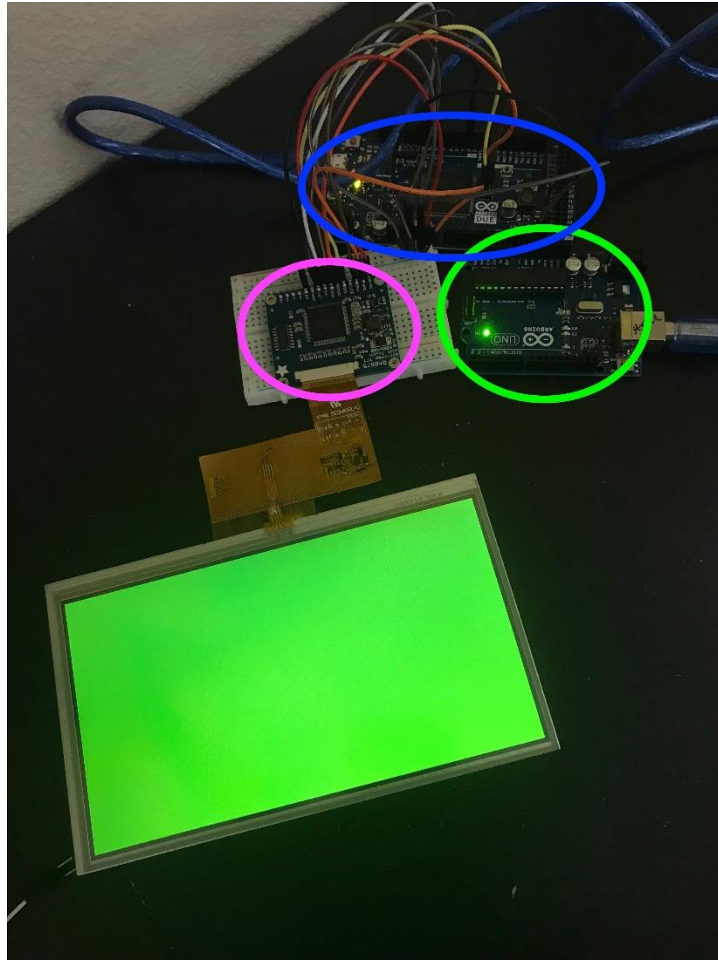


Figure 33: Display Testing

7.2.1 Servo Testing

The servos used in this project were chosen for primarily for the torque they could output. This is extremely important in order to ensure the servo can rotate the ball valve consistently. This was an area of interest during testing. Two other primary interests during testing were making sure that the level shifters worked and also that the servo could accurately and consistently rotate 90 degrees.

The entire setup that was used in testing the servos is shown in figure 34. The Arduino Due (circled in blue) is connected to the level shifter (circled in pink) using three jumper wires. The PWM which drives the servo motor is outputted from digital pin 9 on the Arduino Due. The 5V power which drives the servo power pin and converts the 3.3V ARM logic is outputted from the Arduino Uno (circled in red). The Arduino Uno was used as a 5V power supply for testing, but during implementation all 5V power will come from the TPS40305 5V regulator. The servo motor is circled in yellow and the ball valve is circled in orange.

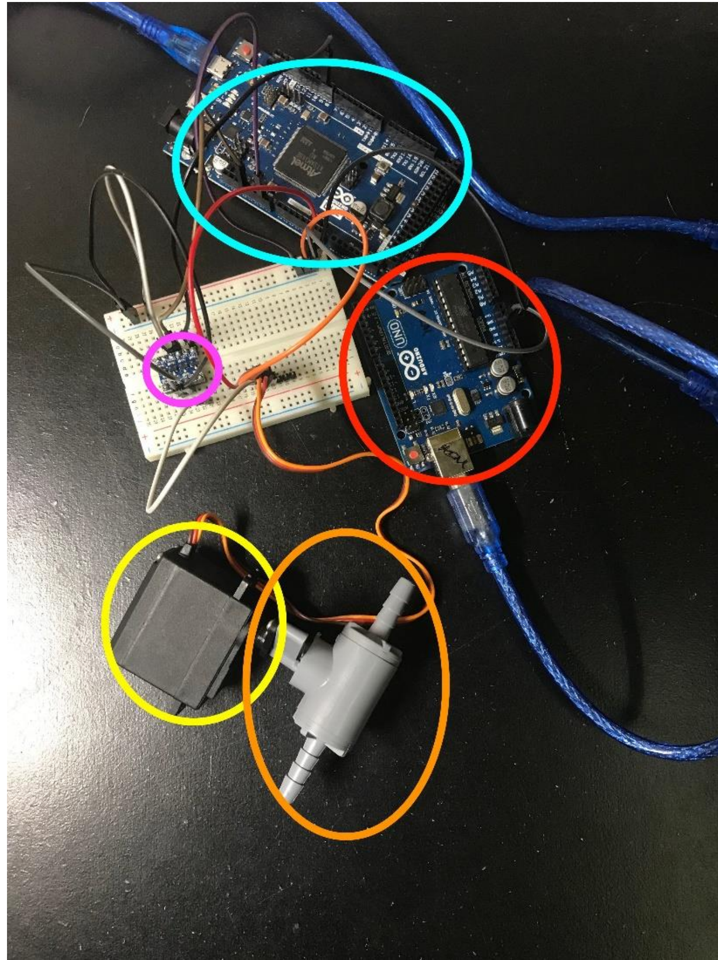


Figure 34: Servo Testing Setup

The servo control line requires 5V PWM signals to operate properly. The motor will except a 3.3V PWM signal, however the torque output aresignificantly lower. In order to properly operate the servo motor, a bi-directional level shifter was used to step the 3.3V PWM signal to a 5V PWM signal. The connection of the level shifter is shown in figure 35. The left side of the level shifter is for the lower voltage level (3.3V). The right side is for the higher voltage level (5V). Pin 9 on the Arduino due (black jumper wire on the left side) is connected to the LV1 on the level shifter. The 3.3V power pin on the Arduino Due is connected to the LV pin on the level shifter (brown jumper on left side). The GND pin on the Arduino Due is connected to the GND pin on the level shifter (white jumper on the left side). The servo data pin is connected to the HV1 pin on the level shifter (white jumper on the right side). The 5V power is connected to the HV pin on the level shifter (black jumper on the right side) and the GND pin on the level shifter is connected to the common ground between all devices (grey jumper on the right side).

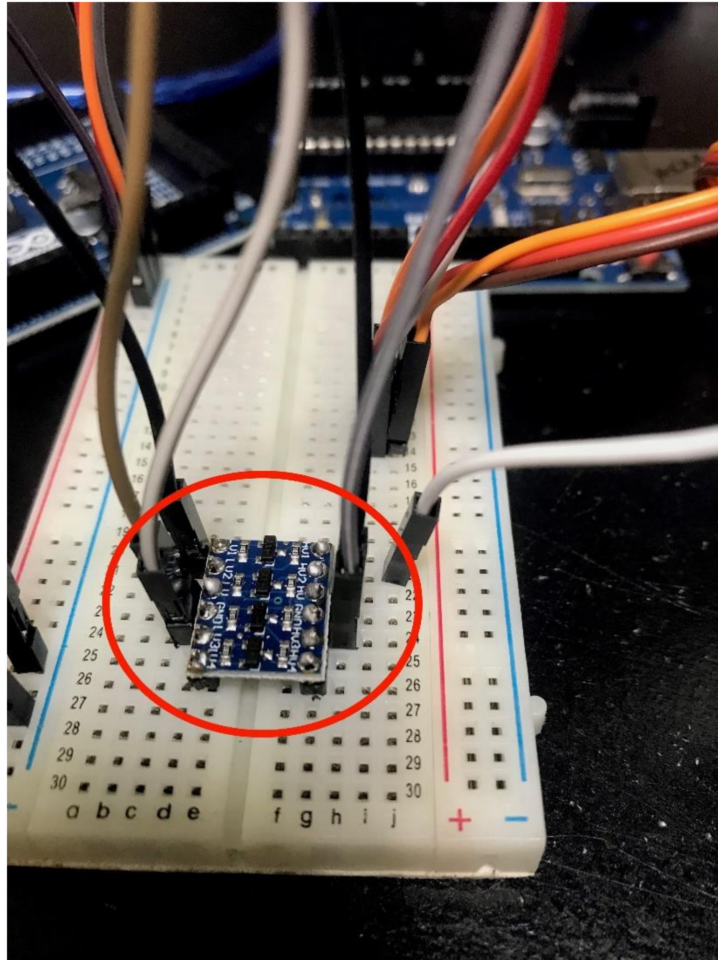


Figure 35: Level Shifter Connections

An important aspect of testing the servo motors was making sure that the servo produced enough torque to rotate the ball valve. The servo was connected to the ball valve using M2 machine screws and a servo horn. To produce maximum torque on the ball valve, the servo horn must be as long as possible and be mounted as close to the tip of the lever on the ball valve. The servo horn must also be mounted snugly in order to reduce vibration, which will reduce induced torque. The connection between the servo and the ball valve (circled in red) is shown in figure 36 To test the servo motor, an example sketch was used from the Adafruit servo library. The sketch “servo_test” was uploaded to the Arduino Due using the Arduino IDE. The test code will rotate to servo 90 degrees in and then –90 degrees to return to the starting position. The code used to test is shown below.

```
#include <Servo.h>

Servo myservo;  // create servo object to control a servo
// twelve servo objects can be created on most boards

int pos = 0;    // variable to store the servo position
```

```
void setup() {
  myservo.attach(9);  // attaches the servo on pin 9 to the servo object
}

void loop() {
  for (pos = 0; pos <= 90; pos += 1) { // goes from 0 degrees to 180 degrees
    // in steps of 1 degree
    myservo.write(pos);              // tell servo to go to position in variable 'pos'
    delay(15);                       // waits 15ms for the servo to reach the position
  }
  for (pos = 90; pos >= 0; pos -= 1) { // goes from 180 degrees to 0 degrees
    myservo.write(pos);              // tell servo to go to position in variable 'pos'
    delay(15);                       // waits 15ms for the servo to reach the position
  }
}
```

To test if the servo was able to handle the load. I observed the servo rotate under no load to see if the servo was able to rotate properly and position properly. I then attached the servo to the ball valve (as shown in figure 36) and observed how the servo functioned under load. The servo was able to rotate the lever on the ball valve (shutting it on and off). The servo also appeared to be able to rotate the full 90 degrees and also appeared to center properly by visual inspection.

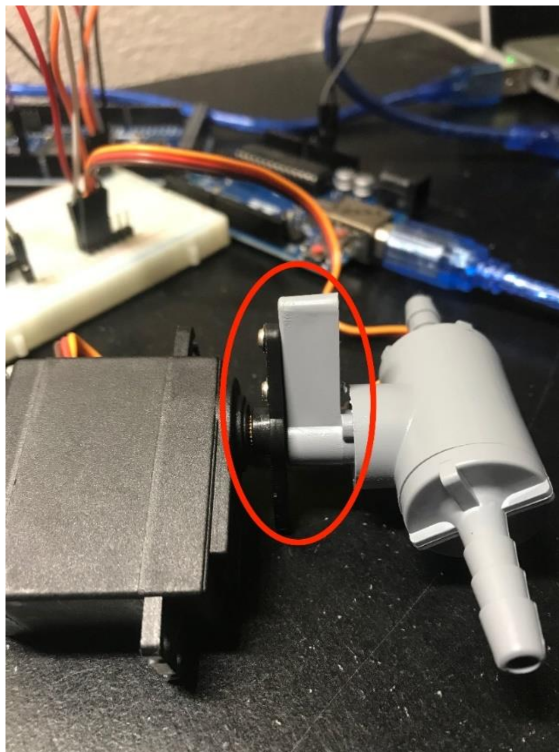


Figure 36: Servo/Ball-Valve Connection

7.2.2 Load Cell Testing

To test the load cell, the setup shown in figure 37 was used. The Arduino Due (circled in yellow) is connected to the load cell using pin 2 for load cell data and pin 3 for load cell clock. The 3V and GND pins are also used for level shifting. The Arduino Uno (circled in red) is used as a 5V power source. The HX711 board is used as an amplifier and ADC (circled in blue). Since the MCU outputs 3.3V logic, level shifters (circled in pink) were used. The load cell (circled in green) is connected to the HX711 board.

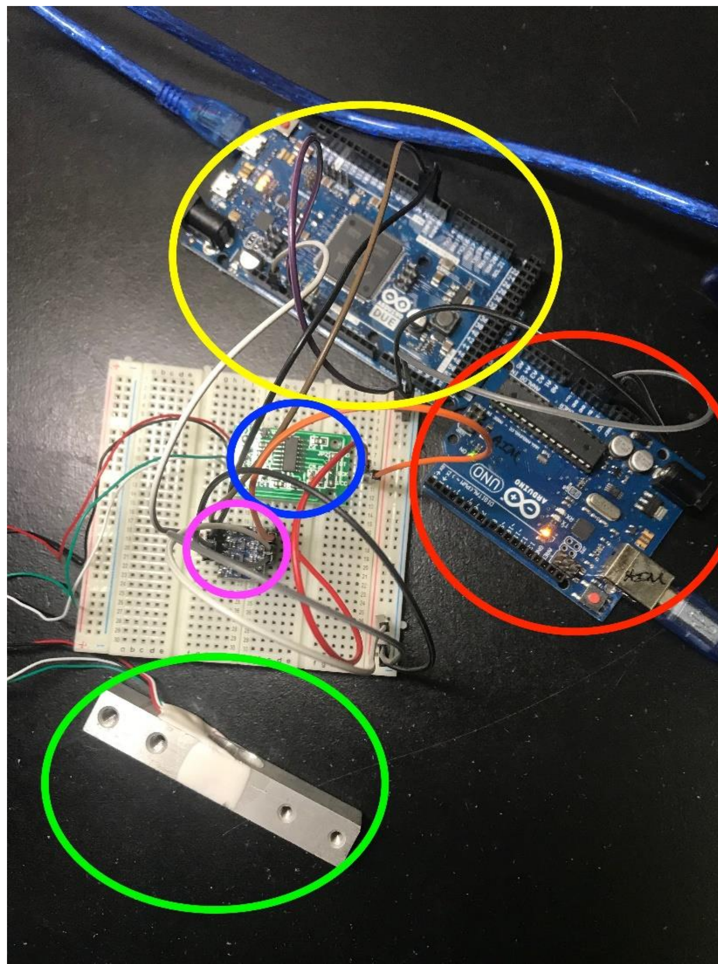


Figure 37: Load Cell Testing Setup

The loadcell was configured and was tested using the following code from "HX711_basic_example" in the HX711 library on the Arduino IDE.

```
#include "HX711.h"
// HX711 circuit wiring
const int LOADCELL_DOUT_PIN = 2;
```

```
const int LOADCELL_SCK_PIN = 3;
HX711 scale;
void setup() {
 Serial.begin(57600);
 scale.begin(LOADCELL_DOUT_PIN, LOADCELL_SCK_PIN);
}
void loop() {
 if (scale.is_ready()) {
   long reading = scale.read();
   Serial.print("HX711 reading: ");
   Serial.println(reading);
 } else {
   Serial.println("HX711 not found.");
 }
 delay(1000);
}
```

The code displays readings from the load cell to a serial monitor at a 57600 baud rate. The load cell continued to spit out a range of values even when no force was applied. To make sure that the load cell was working properly, a force was applied to the load cell and the change in values was observed. To test the accuracy of the load cell was, a 100g weight was used for calibration. Before calibrating, an appropriate offset value had to be chosen since the load cell was sending a range of readings to the serial monitor even when no force was applied. To find this offset value, an average of the values was taken. The offset value chosen was 8334655. After the offset was entered, the load cell values ranged between –30 to 30 under no force. When the 100g weight was placed on the load cell, the values hovered around 40844. To convert the output to grams, the entire reading and offset were divided by 408.44. This effectively converted the readings to grams. After entering the line of code shown below, the output displayed values that ranged between 99.87 - 100.03.

```
Serial.println(reading – 8334655 / 408.44f);
```

The load cell in this project contains two strain gauges in a half-bridge configuration. One strain gauge is located at the top of the cell and one is located at the bottom. The output of the device has four connections, E-, E+, A-, and A+. The ‘E’ lines are input (reference) voltage. The ‘A’ lines are the analog outputs.

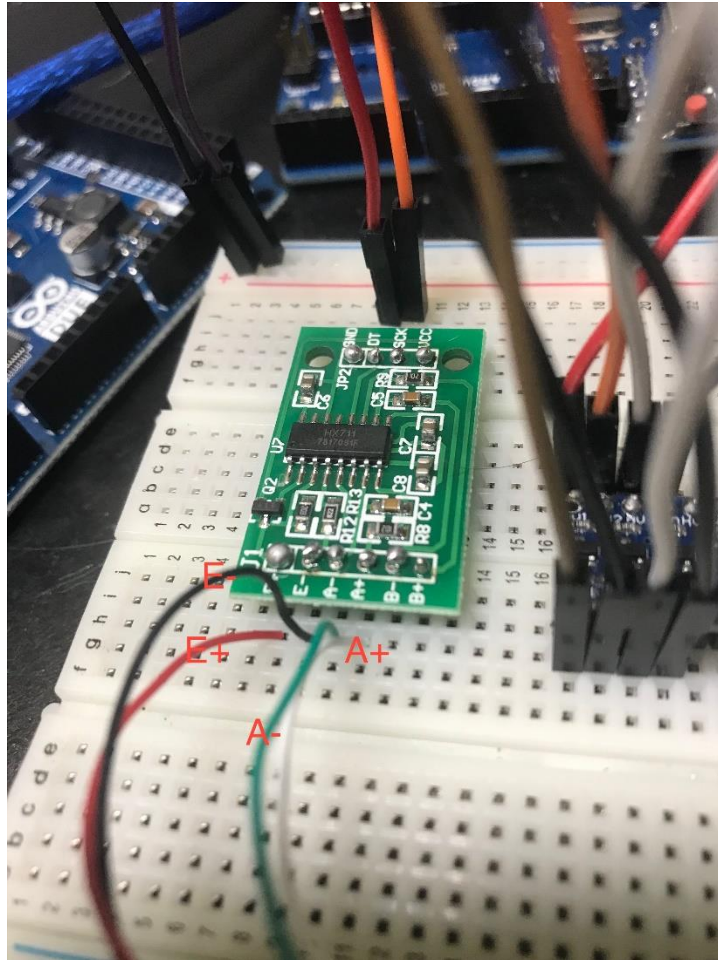


Figure 38: Load Cell/HX711 connection

The output of the HX711 board is the load cell data line (DT). The SCK line is used to sync the clocks of the devices. The DT connection on the HX711 board is fed into pin 2 on the Arduino Due. The SCK connection is fed into pin 3 on the Arduino Due. This connection is shown in figure 39.

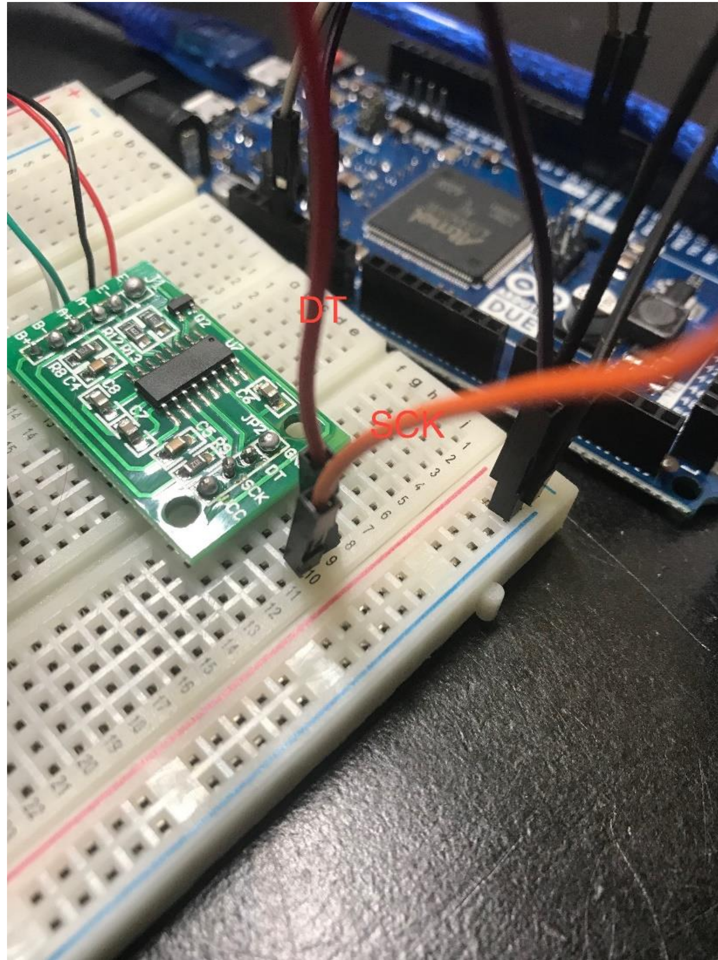


Figure 39: Load Cell Data Output and Clock

In order to properly connect the HX711 board to the Arduino Due, level shifters were used. The only connections that needed to be level shifter were the connections to the Arduino Due I/O pins. The connections were the DT to pin 2 connection and the SCK to pin 3 connection. The configuration used to properly level shift the logic is shown in figure 40. Line 1 is connected to the DT pin on the HX711 board. Line 5 on the low voltage side of the shifters is connected to pin 2 on the Arduino Due. Line 2 is connected to the SCK pin on the HX711 board. Line 6 on the low voltage side is connected to pin 3 on the Arduino Due. Line 3 is connected to 5V and Line 4 is connected to common GND. Line 7 is connected to 3V and line 8 is connected to common GND.

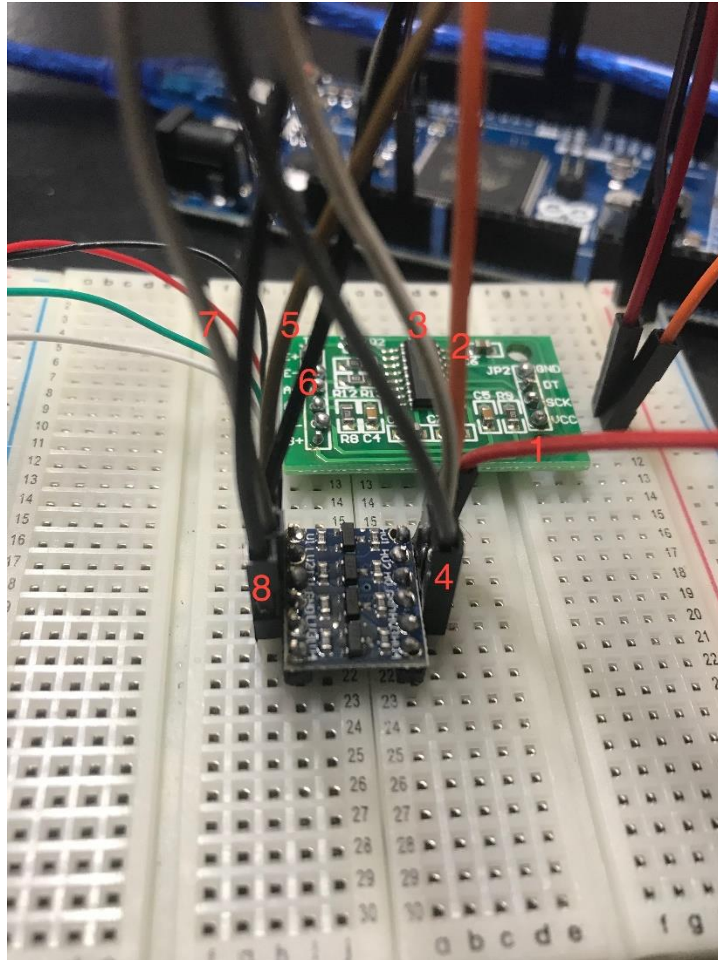


Figure 40: Level Shifter Connections

7.3 Software Testing Environment

Software testing are completed with tested inputs in the system and will occur over the course of coding the software for the project as well as when the project is completed.

Software testing will occur in twice: a) when writing the code, to ensure that the code is working properly and will achieve the proper result (or error in the case of error testing), and b) when the device is fully built, to ensure that the final product is operating smoothly and correctly. The reason for this break-up in testing is that scenario A testing is much easier to accomplish than scenario B, especially since scenario B will occur during a time when it aredifficult to re-code the system; when everything is in place, it is not desirable to have to remove the any components.

In scenario A, testing will occur via the IDE used for creating the code. In Scenario B, it area visual test of the system and its outputs.

7.4 Software Specific Testing

1. Login system
    a. User inputs false username and password
        i. Expected result: User receives error stating incorrect username

    b. User inputs correct username but false password
        i. Expected result: User receives error stating incorrect password

    c. User inputs incorrect username but correct password
        i. Expected result: user receives error stating incorrect username (system assumes username is wrong before password is wrong)

    d. User inputs correct username and password
        i. Expected result: user is logged in, sees Start/Main Page

    e. Under the Settings page, select the Add New Login page, and create a new username and password, then log out and attempt log in using the new username and password
        i. Expected result: user is logged in, sees Start/Main Page

Arguably one of the most important security features of the product is the login system; without it, anyone would be able to use the product. As such, it is tantamount that the login system works properly so that only authorized individual may use the product. A username and password are used to determine if the user attempting to log in is authorized for use; the system assumes that a username is wrong before a password is wrong, so that an unauthorized user cannot simply try passwords as well.

2. Page movement
    a. User will move between all pages, moving forward and back as necessary to ensure all paths are working appropriately
        i. Expected result: no difficulty moving between pages

    b. User will attempt to move between pages whenever possible while sensors feed false data designed to display errors
        i. Expected result: user areunable to access the Brew Page, and therefore unable to brew anything at this time.

Page movement is obviously important; if you cannot move between UI pages, there isn't a whole that can be accomplished with the Micro Manufacturing Beverage System. The tests are mostly to ensure that the system is working

properly, rather than acting a stress test for the Micro Manufacturing Beverage System.

3. Sensor and Sensor Errors
    a. Input false temperature sensor data
        i. Expected result: user receives error stating which temperature sensor is failing

    b. Input false weight sensor data during a “brewing” cycle (only for software testing)
        i. Expected result: user receives error stating weight is incorrect and to return container to drink output tray

    c. Attempt to brew when there is not enough ingredient available
        i. Expected result: user receives error stating that they cannot brew due to a lack of ingredients

    d. Load up either the Main Page or the Brewing Page, and compare the temperature found there to the temperature measured via a standard thermometer in the same location as the sensor
        i. Expected result: the two values should be equal, or at least a close match

    e. Go to the Logs page and go to the Freezer Levels Page. Compare the displayed value for sensors a simulated value input into the code
        i. Expected result: the values should be a match

Because the Micro Manufacturing Beverage System is designed to create beverages, a major part of the project revolves around sensors that are supposed to ensure that all ingredients are in good condition (even if the Micro Manufacturing Beverage System is designed for test purposes rather than large-scale automation). If the sensor can’t detect that the ingredients are non-optimal, it can an effect on the drink that the Micro Manufacturing Beverage System produces; at incorrect temperatures, it the ingredients may be spoiled or frozen, and at incorrect pressures, the valves might close too early or too late to properly create the combination of ingredients that were requested.

4. Memory checks
    a. Input string of data for user to view in Activity Log
        i. Expected result: user views data correctly as a list; this is less of an error check and more of a translation check for the system

    b. Input string of data for user to view in Test Batches/Presets Used
        i. Expected result: user views data correctly as a list; as with the previous check, this is less of an error check and more of a translation check for the system

c. Make a new Brew, then save the data as a preset
    i. Expected Result: new preset is made

d. Go to Brew Presets, select the preset saved in the previous test and brew the new preset
    i. Expected Result: User should be able to view the information on the preset prior to brewing, then the preset should be successfully brewed when it has reached that point

e. Go to Brew Presets, select the preset used in the previous two tests, and delete it
    i. Expected Result: user should be able to delete the preset from memory. Any presets listed after this preset should be incremented down (i.e. preset 4 becomes preset 3 for the page) as needed.

f. Make a new Brew, then attempt to save the data as a preset while there are already 32 presets available
    i. Expected Result: user should receive an error screen stating that they cannot save any more presets, while suggesting that they remove presets they no longer use

g. Turn off the Micro Manufacturing Beverage System after ensuring that there are several presets saved, then turn the Micro Manufacturing Beverage System back on. Go to the page that displays presets.
    i. Expected Result: user should be able to access all of the presets that were saved prior to turning the power off. All presets should have maintained previous positions.

Due to the nature of the Micro Manufacturing Beverage System, memory has to be maintained throughout the usage of the Micro Manufacturing Beverage System. If the Micro Manufacturing Beverage System cannot be trusted to keep the presets it used, this can cause problems when going between daily operations; the presets contain information that can be easily compared within the system, and allow for rapid testing with only minor tweaks as needed. It also allows new brews to be tested against older brews.

5. UI Buttons
    a. Upon loading into the Main Page while all sensors are within the correct range of operation, the user should view the colors of the buttons
        i. Expected result: All buttons should bin the “idle” color.

b. Upon loading into the Main Page while at least one of the sensors are not within the correct range of operation, the user should view the colors of the buttons
    i. Expected result: the Brew Buttons should not be available for usage
c. User clicks on a “forward” button (i.e. the BREW button on the Main Page)
    i. Expected result: user should see the BREW button change color when pressed, and then move to the next page (in this case, the Brew page)

d. User clicks on a “backward” button (i.e. the red Log Out button on the Main Page)
    i. Expected result: user should see the button change color when pressed before returning to the previous page

e. User clicks on a “forward” button while that button is in “restricted” mode
    i. Expected result: No change should occur; the button should not change color to better reflect the lack of action.

Compared to many other parts, ensuring the UI buttons work properly is not a significant part of the operation of the Micro Manufacturing Beverage System. However, without ensuring that the UI buttons work properly, this can result in confusion on the part of the User, which should be avoided whenever possible.

Additional test may be necessary as the project moves from the conceptual phase and onto the building/coding phase.

8.0 Administrative Content

This section of the document are to show how well the team can manage their time and budget on this project. The timeframe and due dates for each milestone are shown from the initial Project Idea being decided upon to the Project Presentation time. The budget are listed to show each expense that makes up the Micro Manufacturing Beverage System with the approval of all members of the group.

8.1 Milestone Discussion

This section will break down the milestone completion of the project. It will show the project process from the initial idea stage in Senior Design 1 in the Spring of 2019 semester down to the final presentation at the end of Senior Design 2 in the Summer of 2019 semester.

At the current time, all milestones are currently on schedule, with sufficient time to complete other problems. It should be noted that many of our milestones often have similar due dates; this is usually because testing of physical components occur prior full integration into the system, allowing the software team to set up initial foundations for the hardware-software integration. This in turn allows each member of the group to work a task with minimal waiting between members. It is interesting to note that the majority of actual project work occurs during Senior Design 2 rather than senior design 1; this means that group are focused on how to properly build the project prior to actually building it, but it also assumes that no unexpected problems will arise as testing occurs.

8.1.1 Senior Design 1 Milestone Discussion

Senior Design 1 acts as a way to establish a group, determine a project idea to work on, and generally plan out the entire process from research to ordering of components to designing and building the project, followed of course by testing and presentation of the project. Because of the importance of the process, as well as the limited time frame students will have to work with to fully produce a project, it is tantamount for a plan of action be designed and made ready for Senior Design 2 to flow smoothly.

Senior Design 1 offers the team a basic start-up timeframe to begin compiling ideas for what the team will work on for at least two semesters. That being stated, there is little reason to grant the team too great of a time for this period. A period of a week was considered by the professors to be sufficient time to form a team and begin the idea formulation process, and the team for this project saw little reason to increase the time. The idea was agreed upon by the group on the

twentieth of January after comparing the inputs of all team members and their respective knowledge for the project.

	Milestone	Start Date	End Date	Status
Senior Design 1	Project Idea Decided	1/13/2019	1/20/2019	Completed
	Initial Project Documentation	1/21/2019	2/1/2019	Completed
	Project Research	2/1/2019	3/1/2019	Completed
	Project Draft Document	3/1/2019	3/29/2019	Completed
	Project Components Received	3/1/2019	3/29/2019	Completed
	Project PCB Designed	3/30/2019	4/20/2019	Completed
	Project Final Document	3/30/2019	4/20/2019	Completed

Table 18: Senior Design 1 Project Milestones

The second task, after determining which project to work on, was to begin the most basic of documentation for the idea. A time frame of one and a half weeks (11 days) was given for the task; while this was sufficient for the most basic of documentation, the group found that little of the project was finalized at this time. The group realized that while they had help narrow down potential solutions to their problem, the solutions would require additional research to help determine which would be selected for the final product. The initial project documentation was completed on the first of February with little difficulty.

The third task was the preform general project research. This task proved to be incredibly broad, and caused many minor frustrations as a result of finding that certain parts were unable to work together properly. It proved to be an illuminating experience for the group, as few members had experience with hardware being outright incompatible in their time. It also shed light on the topic of budget and the slow but steady increases in prices; the process of determining whether a more expensive sensor or valve was actually necessary was a major talking point in the group. While most major research had finished on the twenty-eighth of February (a day ahead of schedule), it was agreed within the group that the full research required for the project may not finish until after Senior Design 1 had finished.

The fourth task was the project draft, a basic 60 page write-up on what research had occurred. Nearly an entire month was set aside for this task due to the sheer amount of work necessary to complete it. The group was able to finish it on the twenty-ninth of March, with minimal difficulty finishing the task despite completing it on the final day. As a whole, the paper was mostly limited to formatting errors

rather than serious issues, and was considered by the group to be a success as a result.

The fifth task was gather the necessary components and supplies for the project. This task was to be accomplished at the same time as the writing of the sixty page paper, so as to have multiple problems addressed at the same time. A majority of the components arrived before the twenty-ninth of March, as agreed upon by the deadline set by the group. Unfortunately, not all components were able to arrive at the deadline, with the final parts arriving on the ninth of April due to shipping problems. This set the group back as a result, due to the fact that testing on components was impossible without all the parts necessary to run them. While time is being made up elsewhere, this delay presents a cascading problem for the next two reports due during the Senior Design 1 semester.

The sixth task was to design the PCB. This task aregiven less time than originally intended due to the lateness of the component arrival. While the team is prepared to make up for the lost time and aremore than able to finish the PCB prior the final report, this has forced the team to delay major actions and ideas for the second report, specifically on certain tests for the hardware and how the outputs can be affected in certain conditions. The team was able to complete this task on the assigned date, and will test the design at the beginning of Senior Design 2.

The seventh task was finish the final document of Senior Design 1. This document will need to be around one hundred and twenty pages long, detailing all findings as a result of the research and component testing, as well as demonstrating the PCB design agreed upon by the group prior to creating and testing it on the final product. This area challenge for the team due to the lateness of the equipment arrivals as well as the other aspects of the group members lives taking potential priority. While less time is available than what would be desired, the team sees no potential problem completing the final project report for Senior Design 1 within the deadline.

As a whole, Senior Design 1’s milestones were being followed in a timely and efficient manner, but are now being subjected to a slipped schedule as a result of the component delay. While regrettable, this provided a wonderful learning experience for the group; a schedule can be changed greatly changed by the most insignificant of delays.

8.1.2 Senior Design 2 Milestone Discussion

Senior Design 2, unlike Senior Design 1, operates under the assumption that the students already have a plan for the entire semester and can begin almost immediately on building and coding their project. The expected result is that most of the actual project building and design should be finished relatively quickly, so as

to allow for rigorous testing to occur and allow the team to adjust and fix any problems found with extreme haste.

The first major task of Senior Design 2 is to create the outer casing (i.e. the frame/box) of the project and build the PCB that was designed for the project. These tasks, despite being fundamentally opposite in nature, are done together to allow for all members to work on a task at the same time. Building the PCB designed in Senior Design 1 was a task done in patience, due to the nature of project. By moving too fast, it was possible to miss an important piece of information that would have crippled the project early on.

The second task of Senior Design 2, which will run concurrently with the PCB building, is the creation of the code that are used for determining sensor values and the alerts needed in such a case that the sensors are incorrect. This included individual component testing to ensure the accuracy of these values for a later date. As the nearly all components arrived at the time of writing this report, no problems or delays were expected at this time; the team was able to complete this task rather quickly to help with other areas.

Senior Design 2	**Project Outer Casing and PCB Built**	**4/30/2019**	**6/10/2019**	**Completed**
	Project Sensor Software Written	**4/30/2019**	**6/10/2019**	**Completed**
	Project Sensors/Valves Installed and PCB Testing	**4/30/2019**	**6/15/2019**	**Completed**
	Project All Hardware Installed / PCB Finalized	**5/15/2019**	**6/25/2019**	**Completed**
	Project All Software/UI Code Written	**5/15/2019**	**6/25/2019**	**Completed**
	Project Testing	**5/25/2019**	**7/15/2019**	**Completed**
	Project Product Finalized	**6/15/2019**	**7/20/2019**	**Completed**
	Project Presentation	**6/20/2019**	**7/21/2019**	**Completed**

Table 19: Senior Design 2 Project Milestones

The third task involves the installing of the sensors and the testing of the PCB to ensure that it is capable of working correctly. While the PCB design created during the end of Senior Design 1 should have worked, it is important to recognize the differences in theory and reality. The PCB failed to account for the Arduino Due's inherent difficulty with programming over JTAG. This left the group in a major problem that would cascade into the final product.

The fourth task was to install all the hardware and finalize the PCB. As the PCB was failing due to the difficulty of the Arduino Due, this was not easily done. To counter balance this, the group focused on making sure everything was perfect on the development board and attempted to get another PCB.

The fifth task was to finish writing the code for the project. This task was mostly completed by the same time as it was expected to. As a result, every page in the UI was finished and all of the basic formulas and actions were added. However, numerous technicalities prevented a truly finished product, such as accuracy and aesthetics; while not to detract from the precision of the code, testing at a future date would allow for accuracy improvements while also ensuring that the entire system worked.

The sixth task was to test the final product. While testing should have occurred over the course of Senior Design 2, this was more encompassing and designed to be testing that interacted with all parts of the Micro Manufacturing Beverage System together. This should occur on many levels, with systematic testing designed to identify shortcomings in the design. Despite what should be a straightforward task, the group set aside a significant amount of time for this, as a high accuracy was desired. As such, while the team has agreed that reaching this task early was imperative to a more fluid and functional final product, there was no rush to finish this task ahead of time.

The seventh task was to finalize the product. Based on information gained from testing in the previous task, changes were made in order to fine tune the final product in ways not previously expected or encountered. Theoretically, this task should not be necessary; practically, nothing goes according to plan, and it was important to set aside a specific time for address and making changes to finalize the product. This was accomplished to the best possible degree given the time allowed, as well as the difficulties encountered via hardware.

The eighth and final task was to present the final product. Presenting the final product occured on two seperate days (Friday July 26 and Monday July 29), but there was plenty of time to prepare for this given the schedule that was kept. As the presentations were based entirely on what was accomplished in the project, it only makes sense that the project would work as a conclusion to all of the groups hard work.

As a whole, Senior Design 2 depended heavily upon the work that was planned out in Senior Design 1. Scheduling was given more leeway, with major parts given a chance to be built, tested, and rebuilt as needed in this later semester when compared to Senior Design 1. As far as timing goes, the team agreed that there was plenty for each section, as many of these tasks could have been easily completed within a few days at most; the extra time was for redundancy purposes, to allow a member to not rush through building and unnecessarily damage the product.

8.2 Budget

This section will include a brief description of how the project are financed and other factors that effected are decisions to purchase particular parts. This section will also include a table that shows displays parts and their respectable costs.

8.2.1 Financing Decisions

The project are mostly financed by one member of the group. One member of the group are keeping the project. The person that are keeping the project after it is completed are financing 80% of the project. The other members will finance the remaining 20%. Since the remaining 20% are divided three ways, the other members will each be accountable for 6.66% of the entire project. Decisions on buying parts was a group effort. We deliberated as a group about which particular parts were suitable for the project and arrived at a unanimous decisions on whether to purchase each part.

8.2.2 Purchased Parts

The table below breaks down each part cost, quantity, and part name.

Part	**Quantity**	**Total Cost**
Arduino Due	1	38.97
Linear Actuator	1	154.69
HX711 Board	1	3.49
Load Cell	1	6.78
TFT Display	1	39.88
RA8875 Driver Board	1	27.56
Temperature Sensors	2	6.00
Valves	6	30.00
Servos	6	22.20

Servo Driver Board	1	19.67
Housing	1	75.00
12V Supply	1	15.46
PCB	2	50.00
Buck Converters	3	12.00
	Total Cost:	501.70

Table 20: Part Cost Breakdown

8.3 Work Division

The Micro Manufacturing Beverage System has four individuals tasked with the building and coding of the Micro Manufacturing Beverage System. To divide the work in an effective manner, two teams were created to ensure both parts of the Micro Manufacturing Beverage System are built in a timely manner.

The first team is the Hardware team. The Hardware team is tasked with ensuring that all parts and equipment are in the correct positions and are operating appropriately. This team will have a predominate focus on the PCB design and building. After the PCB is built and the Micro Manufacturing Beverage System has had multiple components successfully installed, the Hardware team will begin to focus of testing the components such that it can output a more accurate beverage prototype.

The second team is the Software team. The software team is tasked with making sure that the appropriate information is being exchanged between the input sensors and the output valves such that the Micro Manufacturing Beverage System can work without minimal error. The Software team will focus predominately on creating the UI for the Micro Manufacturing Beverage System, as well as saving the appropriate information such that presets and user data can survive a power outage/down.

These two teams were split because the nature of the Micro Manufacturing Beverage System requires both extensive hardware and software work. To ensure that the Micro Manufacturing Beverage System is built and coded in a timely manner, both the Hardware and the Software teams areworking at the same time; in this manner, the project can be rapidly tested after making a simplistic UI and backend. While this means that more time might be spent on testing rather than designing, it allows the teams to work alongside each other rather than waiting until one team has finished for another team to start.

To counterbalance the problem stated above, both teams will assist in the testing phase of the project. This is done to ensure that both teams collectively agree on the level of output from the Micro Manufacturing Beverage System.

The Hardware team are Eric Velez and Lance Adler, who are both Electrical Engineers. The Software team are Parke Novak and Ryan Burns, who are both Computer Engineers. This division was chosen based on the respective strengths of each team member.

Adding to this, should any team begin to fall behind on their section, enough time in the schedule permits the other team to assist them in finishing the tasks required. This is to help deal with potential problems found while building the Micro Manufacturing Beverage System, as well as keep all members up to date on any changes required for operation to begin again.

8.4 Copyright

Due to the fact that no patents are created for this project, the niche nature of the Micro Manufacturing Beverage System (being that it is specifically designed for testing), and the general lack of specialized hardware, the threat of copyright is minimal.

9.1 Appendix A: Works Cited

Works Cited

1743. (n.d.). From Mouser Electronics: https://www.mouser.com/ProductDetail/Adafruit/1743?qs=sGAEpiMZZMve4%2FbfQkoj%252bI1W3rujPEMIZvIvtqfpWks%3D

2V DC 2A Wall Power Supply Adapter with 2.1mm x 5.5 Plug 2A(2000MA) AC 100-240V to DC 12Volt Transformers, Switching Power Source Adaptor for 12V 3528/5050 LED Strip Lights. (n.d.). From Amazon: https://www.amazon.com/Adapter-100-240V-Transformers-Switching-Adaptor/dp/B07GRZB5Y9/ref=asc_df_B07GRZB5Y9/?tag=hyprod-20&linkCode=df0&hvadid=242082743678&hvpos=1o2&hvnetw=g&hvrand=12521102941941367459&hvpone=&hvptwo=&hvqmt=&hvdev=c&hvdvcmdl=&hvlocint=&hv

8-channel Bi-directional Logic Level Converter - TXB0108. (n.d.). From Adafruit: https://www.adafruit.com/product/395

Ada, L. (n.d.). *Pinouts*. From Adafruit: https://learn.adafruit.com/16-channel-pwm-servo-driver/pinouts

Arduino. (n.d.). From Arduino: https://www.arduino.cc/en/uploads/Main/arduino-Due-schematic.pdf

Arduino and DS3231 Real Time Clock Tutorial. (n.d.). From How To Mechatronics: https://howtomechatronics.com/tutorials/arduino/arduino-ds3231-real-time-clock-tutorial/

ATMEGA32U4-MU. (n.d.). From Mouser Electronics: https://www.mouser.com/ProductDetail/Microchip-Technology-Atmel/ATMEGA32U4-MU?qs=JV7lzlMm3yJYRpi0cY3cKw%3D%3D&gclid=EAIaIQobChMI1InS54764AIVjFcNCh0UrQmCEAAYAiAAEgLD6PD_BwE

Atmel. (n.d.). From MouserElectronics: https://www.mouser.com/datasheet/2/268/doc11057s-1369003.pdf

ATSAM3X8EA-AU. (n.d.). From MouserElectronics: https://www.mouser.com/ProductDetail/Microchip-Technology-Atmel/ATSAM3X8EA-AU?qs=sGAEpiMZZMtQuSbTnHsVtm6dgCW59dFS

Building an Arduino on a Breadboard. (n.d.). From Arduino: https://www.arduino.cc/en/Main/Standalone

FreeRTOS FAQ. (n.d.). From FreeRtos: https://www.freertos.org/FAQMem.html

Library: URTouch. (n.d.). From Rinky-Dink Electronics: http://www.rinkydinkelectronics.com/library.php?id=92

Library: UTFT. (n.d.). From Rinky-Dink Electronics: http://www.rinkydinkelectronics.com/library.php?id=51

MSP430G2553IRHB32R. (n.d.). From Mouser Electronics: https://www.mouser.com/ProductDetail/Texas-Instruments/MSP430G2553IRHB32R?qs=sGAEpiMZZMsuBfEaN9EhVfuKxkcA8SoY

Official FreeRTOS Ports. (n.d.). From FreeRTOS: https://www.freertos.org/RTOS_ports.html

SEMICONDUCTOR, A. (n.d.). *24-Bit Analog-to-Digital Converter (ADC) for Weigh Scales*. From SparkFun: https://cdn.sparkfun.com/datasheets/Sensors/ForceFlex/hx711_english.pdf

SparkFun. (n.d.). From SparkFun: https://learn.sparkfun.com/tutorials/load-cell-amplifier-hx711-breakout-hookup-guide?_ga=2.62641641.4373212.1555694583-1780506403.1554854212

SparkFun Load Cell Amplifier - HX711. (n.d.). From SparkFun: https://www.sparkfun.com/products/13879

STM32F103TBU6TR. (n.d.). From Mouser Electronics: https://www.mouser.com/ProductDetail/STMicroelectronics/STM32F103TBU6TR?qs=sGAEpiMZZMuoKKEcg8mMKJ29ob8kKavQ%252B9bS9Z%252BTzJkgoUg2ATmo2A%3D%3D

The Arduino Playground. (n.d.). From Arduino: https://playground.arduino.cc/code/FiniteStateMachine/

TLV76050DBZR. (n.d.). From Mouser Electronics: https://www.mouser.com/ProductDetail/Texas-Instruments/TLV76050DBZR?qs=sGAEpiMZZMsGz1a6aV8DcLm6%2Fe7CQV8Ic%2Feb5Yk0Rt0%3D

Webench Power Designer. (n.d.). From Texas Instruments: https://webench.ti.com/power-designer/switching-regulator