# Contents

1 Motivation .................................................................................................................. 6
2 Objectives .................................................................................................................. 6
3 Purpose ....................................................................................................................... 6

4 Mobile App ................................................................................................................ 6
   4.1 Mobile Application Requirements ................................................................. 6
   4.2 Log In/Register Account ..................................................................................... 7
       4.2.1 Register Account with Smartbar .............................................................. 7
       4.2.2 Register using Google Account ................................................................. 8
       4.2.3 Log In with Smartbar ................................................................................ 8
   4.3 Setup Payment ...................................................................................................... 9
   4.4 Drink Library ....................................................................................................... 10
   4.5 Additional Features ............................................................................................. 10
       4.5.1 Ride Sharing Applications ....................................................................... 10
       4.5.2 Reset Fingerprint ..................................................................................... 11

5 Server .......................................................................................................................... 11
   5.1 Requirements ...................................................................................................... 11
   5.2 Login / Register: ................................................................................................. 11
   5.3 User and Admin Accounts: ................................................................................ 11
       5.3.1 Smartbar Account Analytics: ................................................................. 12
   5.4 Database: ............................................................................................................ 12
   5.5 PHP Web Service ............................................................................................... 12
   5.6 SSL ....................................................................................................................... 13
   5.7 Figures: ............................................................................................................... 13

6 Sensors ......................................................................................................................... 17
   6.1.1 Approach ....................................................................................................... 17
   6.1.2 Process and Implementation ........................................................................ 21

7 Microcontroller ........................................................................................................... 26
   7.1.1 Background ................................................................................................... 26
   7.1.2 Process and Implementation ........................................................................ 26

8 User Interface .............................................................................................................. 31
   8.1 Approach ........................................................................................................... 31
   8.2 Process and Implementation .............................................................................. 32
9 Dispenser ......................................................................................................................... 41
  9.1 Dispensing System Requirements .............................................................. 41
  9.2 Dispensing System Research ................................................................. 41
  9.3 Dispensing System Design ...................................................................... 42
    9.3.1 Dispensing System Diagram .......................................................... 46
  9.4 Dispensing Control Hardware Requirements ...................................... 47
  9.5 Dispensing Control Hardware Component Selection ........................... 47
  9.6 Dispensing Control Hardware Design .................................................... 48
    9.6.1 Dispensing Software Requirements .............................................. 49
    9.6.2 Dispensing Software Design ......................................................... 49
    9.6.3 Software Flow ............................................................................ 50
    9.6.4 Communication and Functionality .............................................. 51
    9.6.5 Modules ............................................................................. 52
10 Power .......................................................................................................................... 53
  10.1 Requirements ......................................................................................... 53
  10.2 Research .................................................................................................. 54
    10.2.1 Building a Power Supply ............................................................. 54
  10.3 Buy - Pros and Cons ............................................................................ 54
  10.4 Build – Pros and Cons ........................................................................ 55
  10.5 Component Selection .......................................................................... 55
  10.6 Circuit Design ..................................................................................... 55
  10.7 Power Consumption Analysis .............................................................. 56
    10.7.1 Dispensing Machine Power consumption .................................. 56
    10.7.2 Bar Appliance Power Consumption .......................................... 56
    10.7.3 Restaurant Appliance Power Consumption ................................ 57
    10.7.4 Bar Power Consumption Estimates ......................................... 57
    10.7.5 Power Requirement and Analysis ........................................... 58
    10.7.6 Smartbar Component Power Consumption ................................ 59
  10.8 Dispensing Apparatus Concepts .............................................................. 59
    10.8.1 Static Mixing Tube Injector .......................................................... 60
    10.8.2 Vortex Funnel ........................................................................... 62
    10.8.3 Channeled Block ....................................................................... 64
    10.8.4 Pre-Dispense Drink Mixer ............................................................. 65
16.8 Confirm Drink ................................................................................................................................. 96
16.9 User Sign In ..................................................................................................................................... 97
1 Motivation
Our goal is to design a robotic bartending system with a regulation system that uses identity verification to ensure the correct person, who is not too intoxicated, will be receiving the drink. This robot will monitor users in detail using a combination of biometric recognition, BAC detection, and user limitations to support a safe drinking environment. Interfacing with the system will be done through a mobile application, allowing for easy ordering and constant access to personal drinking statistics.

2 Objectives
Our objective is to design a robotic bartending system with a regulation system that uses identity verification to ensure the correct person, who is not too intoxicated, will be receiving the drink. This robot will monitor users in detail using a combination of identity recognition, BAC detection, and user limitations to support a safe drinking environment. Interfacing with the system will be done mainly through a mobile application, allowing for easy ordering and constant access to personal drinking statistics. The system will be designed for minimal interaction between the user and robotic system to safeguard the user from operating machinery while intoxicated.

3 Purpose
The purpose of this project is to create a robotic bartending system that accurately monitors users to ensure their safety. In society today, high tech systems that allow users to complete tasks more efficiently are becoming increasingly common with the evolution of modern technology. These systems typically require some kind of user input to allow the system to function correctly, but what happens if user input becomes impaired? Warning labels advise against operating equipment while impaired, but what if the equipment directly causes impairment? Technically, the system would function correctly because it operated as designed, but the outcome may not be the originally intended result. This is an important safety issue that appears to be overlooked in the design process of robotic bartending systems. If it is unsafe to operate heavy machinery while intoxicated because of potential injury, it is also unsafe to use a machine that dispenses drinks, which can be harmful in large quantities.

4 Mobile App

4.1 Mobile Application Requirements

The Smartbar mobile application had few requirements.

- Run on a mobile device
- Allow login/register account for new/returning Smartbar users
- Allow users to setup payment information
- Allow users to browse through a library of drink options
- Users can order a drink
The first decision made was for which mobile device the application would run on. The two main options we had chosen were Android and iOS, since together they made up a large majority of the smartphone market. We wanted to develop our prototype application for a large user base, which both Android and iOS have. The main reason that we chose an Android mobile application was that since our User Interface was also going to be a mobile application on an Android tablet, it would allow for co-development of the two applications as well as make both the mobile app and the UI to be stylistically cohesive.

4.2 Log In/Register Account

Smartbar Mobile allows for a user to log in to the system via two methods: the first using an account created with Smartbar that lives on our web server, and the second is using an existing Google+ account.

4.2.1 Register Account with Smartbar

One option users have regarding account registration is to create an account with Smartbar. At this point in the application flow, the user will see the screen shown in the figure below. Once all the missing parameters have been filled out, the mobile application first checks whether all the parameters entered are valid. For example, the user must enter an age of at least twenty-one, and the case-sensitive passwords must match. Then if the mobile application finds no issues with the users’ information, it then queries the Smartbar web service to determine if the remaining parameters are valid. Each username and email address entered must be unique to the system, and the database will return an account creation failure if there is already an account with the specific username or email address.

If the Smartbar web service finds no issue with the new account information, then the mobile application receives an account creation success message and allows the user to continue into the normal application flow.

![Figure 1: Smartbar Register Account](image)
4.2.2 Register using Google Account

Another option for log in/register on the Smartbar mobile application is for a user to sign in using an existing Google account. For this feature, the Google API Client library was the only necessary addition to compile into the app. Adding this library allowed Smartbar Mobile to communicate with the Google servers and allowed access to the different accounts configured on the device. With the mobile application configured for particular permissions necessary for this interaction to complete, the Google servers used the OAuth2.0 protocol to securely transfer data to the mobile application regarding a Google account profile.

The figure on the left shows the screen displayed once a user chooses to sign in via a Google+ account. This particular device had three separate Google accounts configured, so the user must be prompted to choose which account he/she would like to be associated with their Smartbar account. Once a user chooses the account and clicks ‘OK’, he/she must then allow Smartbar permissions to their account profile. The application requires users to allow these permissions only at the first registration of the Google account or after a user disconnects their Google account from Smartbar.

Once the mobile application has used the Google API Client library to securely establish a connection with the Google server, Smartbar Mobile checks the account profile to ensure that the user has already configured the necessary parameters needed to create an account with Smartbar. If the user has their age and gender configured with their Google+ account, the mobile application proceeds to register the user with their Google+ profile information, provided their age is over 21. If any parameters required to create an account with Smartbar are missing from the user’s Google account, the user will be prompted to enter in any remaining parameters, and then will be directed to the normal application flow once the account has been created.

4.2.3 Log In with Smartbar

If a user already created an account with the Smartbar system, he/she will be able to simply sign into the mobile application using their username and password. The app will query the database with the entered information, and the web service will return a login success if the username and password are valid, and will return a failure if no account was found with that username/password combination.
4.3 Setup Payment

Every new user must setup payment information with the Smartbar system in order to proceed to the application flow. To process payment, Smartbar Mobile incorporated a service by PayPal called Braintree which allows the mobile application to use the Braintree APIs to verify and validate a credit card entered into the system. Since the Smartbar system is not receiving real payments at this time, we configured a service that PayPal Braintree offers called the Sandbox. The Sandbox allows our applications to simulate the transaction process without actually charging a credit card and without having an actual Smartbar account for charges to be made to. The Sandbox allows Smartbar to create a merchant ID associated with our system which essentially simulates a Smartbar account for charges to go to. It also has multiple credit cards which are validated by the system as well as multiple credit cards which are denied by the system. This way, we can ensure that the transaction process is functioning without actually charging real accounts.

In order for this process to begin, the mobile application first must communicate with the Smartbar web service to obtain the necessary parameters to communicate with the Braintree servers. The mobile application first requests a Braintree token (configured on the Smartbar web server using Braintree APIs as well) from the Smartbar web service which is only valid for a small period of time. Once the mobile application has received the Braintree token, the mobile app then calls a series of methods from the Braintree API library and uses the token to initiate a setup payment screen. The figure shown here displays the Braintree Drop-In UI which allows for a user to enter their payment information.

This screen is called from the Braintree server so that the Smartbar mobile application does not need access to the particular payment information. This way, the app can ensure that the user’s payment information is secure and only on the PayPal Braintree servers rather than Smartbar. Once Braintree has verified the credit card information, it returns a Payment Method Nonce to the Smartbar mobile application which tells the app that payment has successfully been configured for the user.

Figure 3: PayPal Braintree Setup Payment

Since the mobile application only authorizes payment information and does not actually charge the credit card, we needed to configure a way for the UI to access the payment information when a drink is dispensed. To do this, we once again used the Braintree APIs to create a Braintree customer ID which
lives on the Smartbar server, yet has no actual payment information. This way, the mobile application and the UI can use the Braintree customer ID to charge an account without knowing the credit card information.

After entering in credit card information in the payment screen shown above, the mobile application sends the Smartbar database the returned Payment Method Nonce which is necessary to create a unique Braintree customer ID. This ID is available to our UI as well so that the credit card is charged only when a drink is dispensed.

A user is required to setup payment information only once at the time of account creation.

4.4 Drink Library

After a user has logged into the system and setup their payment information, they are ready to begin ordering drinks from the Smartbar!

One of the main features of the mobile application is to allow users to browse through a list of drink options available with Smartbar. This ties in closely with all the possible options of drinks that the Smartbar database is configured for as well as the actual dispensing system inventory which is also stored in the database.

Before the screen shown here is displayed for the user, the mobile application queries the Smartbar database to obtain a real-time updated list of drink options which is currently available from the Smartbar inventory.

Once Smartbar Mobile receives the list of drink options, the application parses through the list to display it in a way such that the user is able to easily scroll through the drink options as well as manually type into the text box to search for a particular drink. After a user has chosen a drink, he/she will be prompted to confirm their choice before the drink is added to the queue.

Figure 4: Drink Library

4.5 Additional Features

4.5.1 Ride Sharing Applications

Since Smartbar aims to reduce unsafe drinking habits, it was important to include ride sharing applications into the mobile app. Uber and Lyft applications will be available on every screen of Smartbar Mobile once logged into the application. If a user has the app installed on their Android device, then the button will open either the Uber app or the Lyft app. If the user does not have the
application installed on their device, then clicking the button will open the respective page in the Google Play Store to ensure the user has a ride-sharing option.

4.5.2 Reset Fingerprint
The mobile application also incorporates a reset fingerprint feature. In the unlikely event that the user forgets which finger that they enrolled with their account, or if the user suspects that another person enrolled their fingerprint with an account that does not belong to them, the user will be able to reset their fingerprint on the database so that a user can re-enroll a correct finger into the system.

5 Server

5.1 Requirements:
- Login / Register SmartBar Account
- User and Admin accounts on website
  - View analytics on user and admin accounts
- Database accessible by pi, tablet, and mobile app
- Web Service for cross communication
- SSL on server

Our web server is a consists of the core services needed knows as LAMP(Linux Apache MySQL PHP). This was custom installed on Smartbars UCSC Linux virtual box. Having our custom server enabled us to host our own website, web service, and database.

5.2 Login / Register:
There are two places where the user can login or register. The first is from our website where a user who can’t install our android app can go and create an account. This account is a custom Smartbar account that is stored securely in our database. A user can login to our website by entering the previous credentials when registered. The login and register both use hashing functions for the password for added security and so we don’t store plain text passwords on our database. The login and register functions were written using PHP and stored with MySQL queries.

We also require that a user can login and register from our Android application. The app uses google plus login so our user can use an already existing google account they own to login. Alternatively a user can create a new account with Smartbar by accessing the script used on the website but a translated version for the application. The login for Google plus works from Googles servers but when you login with a Smartbar created account the login is through our website script similar to that of register.

5.3 User and Admin Accounts:
Smartbars website has the ability to create two different types of accounts. First we have a user account which is for a customer of the Smartbar to be able to view their account information, update it as necessary, and view statistics about their ordered drinks, and enter payment information. Second we have an admin account which would belong to the Smartbar owner or operator. This account lets you
login to our website and view statistics on drinks poured, available drinks on the library based on current
inventory, and live updating liquid levels as drinks are being poured.

5.3.1 Smartbar Account Analytics:
When a user logs into the website they are able to view data on their drinks. Currently the platform is
setup to be able to view any data stored in the database. At the moment the user can see how many
drinks the have ordered and which type. Potentially it could be expanded to show how much a user has
spent, their BAC over the night, and when they get their drinks are just a few examples.

When a Smartbar owner logs into an admin account the are able to view data an the drinks the
Smartbar is currently pouring. Similar to the user accounts the platform is setup to be able to view any
statistic from the data in the database. Currently an owner can see the total amount of registered
customers, average customer age, and how many drinks have been poured. Potentially an owner could
also see who ordered drinks and break them up into age groups, what time of the day drinks were
popular, what time the most drinks get poured and countless other statistics from the data being
collected in the database.

5.4 Database:
The database we required was to be accessible by the website, mobile app, micro controller, and the
tablet. We needed an externally accessible database and after much research MySQL met all our
requirements. It was already installed on our custom web server thus making it an even more solid
choice. We opted to use the onboard android database but quickly learned it is all stored on the android
device so there would be synchronization issues as well as size limitation on the phone. A MySQL
database can be accessed remotely and is scalable up to millions of users with still reasonable execution
times. The database was structured into multiple tables to store all our data. A users table was used for
the login and register scripts on both the website and mobile application that contains all their
information such as their fingerprints and whether they have a drink on the queue. An inventory table to
store all the current liquid leve

5.5 PHP Web Service
We created a PHP web service because there is no direct way for android to communicate with an
external database or MySQL in our case. Also we needed a way to have a local Smartbar account from
the mobile app for users who did not wish to use Google log in. The web service uses PHP and
JSON(JavaScript Object Notation) to bridge the gap of communication between the database and
android app. The android app makes a call to a script requesting information such as if a user with a
certain login and password exists in the database. The script then queries the database to check and
returns a JSON message which the app can interpret as a successful login in this example. The service
does tasks from login, register, add drink to queue, check for fingerprint, add fingerprint, add recipe, add
inventory, update inventory, check for user pin, check to see if a user is 21 and many more tasks that
can be seen in the scripts on our Github. The microcontroller connects to the database with a native
Python library for MySQI access, similarly the website uses a native PHP MySQL library to communicate.
Since the database is built with SQL technology our system is very easily replicable for many Smartbar systems and can accommodate millions of customers with scalability.

5.6 SSL
In order to process payments securely or even login and register a user SSL was implemented on our custom server. A certificate from a reputable company can cost from 50-100$ or year we decided to employ a self-signed certificate to add that encryption without needed to use a big company service and to show proof of concept. SSL creates a secure connection between the server of the current website and the user’s web browser. Having this connection encrypted as well as hashing passwords and sensitive data create a very secure method of communication with sensitive data. This was also necessary for having a payment option on our website that a user could trust and use with confidence. For the future a Smartbar owner could easily purchase a certificate and our web server would continue to work as intended.

5.7 Figures:
User Account Information:

User Account Information update:
Admin Account Statistics:
Admin Account Stock Information:

Drinks in Stock:
- Cherry Italian Soda
- Vanilla Italian Soda
- Bubbly Water
- Vanilla Cherry Italian Soda
- Mild Cherry Italian Soda

Admin Account Liquid Levels:
### Smart Bar – Final Report

#### My SmartBar

**My SmartBar Admin Account**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bar Info</td>
<td>Stock Info</td>
<td>Liquid Levels</td>
</tr>
</tbody>
</table>

**Container Levels:**

#### Drinks

- **Container #0**
  - Spirit: Cherry Italian Soda
  - Brand: 0
  - Current Volume (Oz): 48
  - Total Volume (Oz): 50
  - 96% full

- **Container #9**
  - Spirit: Vanilla Italian Soda
  - Brand: 0
  - Current Volume (Oz): 100
  - Total Volume (Oz): 100
  - 100% full

#### Mixers

- **Container #5**
  - Mixer: Water
  - Brand: 1
  - Current Volume (Oz): 97.3
  - Total Volume (Oz): 100
  - 97% full

- **Container #6**
  - Mixer: Water
  - Brand: 0
  - Current Volume (Oz): 86.7
  - Total Volume (Oz): 100
  - 87% full

---

**PHP Web Service**
6 Sensors

6.1.1 Approach

6.1.1.1 Initial Requirements

For the sensor system, the requirements mostly focused on making the system customer friendly and gathering information to be utilized by the remainder of the system. The initial requirements were as follows:

- Uniquely identify users without login information. By using a sensing technology which could detect a trait unique to each individual, it would prevent people from using somebody else’s account and allow customers to login without having to remember their information and fussing with typing.
- Detect alcohol on users’ breath. We wanted to implement breath alcohol detection so that customer BAC could be monitored to log drinking habits and cut off users a bartender would after a certain threshold is reached.
- Determine whether a cup is in the dispensing area. This would help users determine if they are placing their cup directly below the dispensing nozzle to minimize loss of beverages when the system automatically dispenses.
- Keep track of the liquid level in dispensing containers. By tracking liquid level, whoever is maintaining the bar can be alerted to when containers are running low.
6.1.1.2 Sensor Considerations

- **User Identification:** facial recognition, biometric fingerprint
- **Blood Alcohol Content (BAC):** ethanol gas (semiconductor oxide, fuel cell)
- **Liquid Level:** software, ultrasonic, laser, and float
- **Cup Detection:** infrared, laser diodes and load

6.1.1.3 Theories of Operation

6.1.1.3.1 User Identification

One of the first design decisions in regards to the sensor system was which method of user identification we wanted to pursue. Early on, we decided against facial recognition as we felt it would likely require significant processing power and would limit our microcontroller selection greatly, and fingerprint identification would be more unique and therefore sufficient.

There are two prominent types of biometric fingerprint sensors which were considered for the project: capacitance and optical. Capacitive fingerprint sensors use semiconductor chips built out of tiny conductor plates covered in an insulating layer. The capacitance in a plate is greater under a ridge (a protruding characteristic of a fingerprint) in comparison to a valley. They are difficult to trick as they need heat from a finger in order to sense, meaning they cannot be deceived by an image of a fingerprint unlike optical. However, they are more susceptible to contamination and electrostatic discharge, and require significantly more maintenance than optical sensors, making them more expensive over time. The resolution of these sensors also tends to be relatively low.

Optical fingerprint sensors utilize charge coupled device (CCD) technology. As with digital cameras, this uses a series of light sensitive diodes (photosites), which create an image by capturing single pixels based on how the sensor responds to light photons. The image captured is usually inverted, with lighter areas representing valleys and darker areas defining ridges. Most sensing modules have a glass plate to place the finger on and have similar functions. There are enrollment and identification stages, as well as image quality checking and determining whether a finger is placed on the plate or not. To check if a finger is still on the plate and determine image quality, the average pixel darkness is checked to see if the image is too light or too dark. The enrollment process takes multiple images of the finger (2-3 depending on the sensor), and analyzes it for all the defining traits (arches, loops, whorls, bifurcations, crossovers and endings). The algorithm on the module measures the angles and distances between these traits and creates a template which can later be compared to for identification. We chose this type of sensor as it tends to create higher resolution images than capacitive, as well has longer life and are a more developed and readily available technology.

We surveyed three sensors of this type (GT5C113, ZFM20, TFSD40), all with very similar specs which made it difficult to decide. The decision to use the GT5C113 was largely based on the quality of the datasheet compared to the other options, as well as the abundance of resources available including an incomplete python library (it was missing the get and set fingerprint functions needed to remove and add fingerprint templates to/from the module). The specifications for the sensor are as follows:
The type of alcohol found in alcoholic beverages is ethanol (C₂H₆O), which can be detected in both blood (the most accurate measuring method) and breath. Unfortunately, while blood is more accurate it would be far too unsanitary and hazardous to have an autonomous system which would request a customer’s blood sample. Thus, we had to find a sensor which would be able to detect the presence of ethanol gas in the surrounding air. BAC refers to blood alcohol content, whereas BrAC refers to breath alcohol content, both of which are representative of a percentage of body mass per volume. The ratio of BrAC to BAC is 2100:1, which means BAC=(breath mg/L)*.21. This equation theoretically can be used to estimate BAC from BrAC when used along with the following equation used by the US Department of Transportation for BAC calculations:

\[
BAC = \frac{[(\text{# drinks consumed}) \times (\text{ounces of alcohol consumed}) \times (23.36 \text{ grams of alcohol/oz})]}{[(\text{body weight in pounds}/2.2046 \text{ lb/kg}) \times (\text{total body water volume}) \times (1000\text{t}/\text{kt})] + 100 - \text{time}}
\]

This measurement requires extremely regulated testing conditions and a known volume to be accurate. This would be useful information in future implementations of the project in which more accuracy would be required, but was discovered through several sources to be a challenge to accomplish within the scope of the class as it has been known to be difficult to achieve for the sensors readily available to the public. Additional factors which limit the accuracy of BAC sensors include user gender, whether a person has diabetes, if one has smoked close to the time of the measurement, whether residual mouth alcohol (RMA) is present, or if other gases are present depending on the sensing mechanism.

There are several types of sensing mechanisms which are used to detect BrAC: infrared spectroscopy, fuel cell and semiconductor oxide. Infrared spectroscopy is a process which
requires a strong understanding of chemistry is easily the most expensive option of the options. Fuel cell tends to be the most accurate for BAC estimations, but it also tends to be tricked by other types of gases, such as isopropanol and methanol, more easily than the other two sensing mechanisms and commercially available sensors of this type had little in the way of datasheets and any additional resources. Of the options, semiconductor oxide is the least accurate for BAC estimation, but picks up only ethanol gas and is the most commercially available and inexpensive, with the most research obtainable.

With everything in mind, we selected the MQ-3 semiconductor oxide ethanol sensor. This particular sensor requires a 48 hour break-in period but afterward tends to be stable with a long life span. The sensing element is made of tin dioxide, which changes conductivity in relation to the amount of ethanol gas in the surrounding air, which in turn changes the amount of current flowing through the device which can be read as a voltage difference between two nodes. Higher voltages imply higher presence of ethanol near the sensing element and lower voltages imply lower amounts of ethanol. The specifications of the sensor are as follows:

### A. Standard work condition

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter name</th>
<th>Technical condition</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vc</td>
<td>Circuit voltage</td>
<td>5V±0.1</td>
<td>AC OR DC</td>
</tr>
<tr>
<td>Vh</td>
<td>Heating voltage</td>
<td>5V±0.1</td>
<td>ACOR DC</td>
</tr>
<tr>
<td>Rl</td>
<td>Load resistance</td>
<td>200K Ω</td>
<td></td>
</tr>
<tr>
<td>Rh</td>
<td>Heater resistance</td>
<td>33 %5%</td>
<td>Room Tem</td>
</tr>
<tr>
<td>Ph</td>
<td>Heating consumption</td>
<td>less than 750mW</td>
<td></td>
</tr>
</tbody>
</table>

### B. Environment condition

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter name</th>
<th>Technical condition</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tao</td>
<td>Using Tem</td>
<td>-10°C-50°C</td>
<td></td>
</tr>
<tr>
<td>Tsa</td>
<td>Storage Tem</td>
<td>-20°C-70°C</td>
<td></td>
</tr>
<tr>
<td>Rh</td>
<td>Related humidity</td>
<td>less than 95%RH</td>
<td></td>
</tr>
<tr>
<td>O2</td>
<td>Oxygen concentration</td>
<td>21%(standard condition)/Oxygen concentration can affect sensitivity</td>
<td>minimum value is over 2%</td>
</tr>
</tbody>
</table>

### C. Sensitivity characteristic

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter name</th>
<th>Technical parameter</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rs</td>
<td>Sensing Resistance</td>
<td>1MΩ - 3 MΩ (0.4mg/L alcohol)</td>
<td>Detecting concentration scope: 0.05mg/L—10mg/L Alcohol</td>
</tr>
<tr>
<td>α</td>
<td>Concentration slope rate</td>
<td>≤0.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Standard detecting condition</td>
<td>Temp: 20°C±2°C Humidity: 65%±5% Vc: 5V±0.1 Vh: 5V±0.1</td>
<td></td>
</tr>
</tbody>
</table>

#### 6.1.1.3.3 Liquid Level

For this aspect of the project, we surveyed several sensors, including laser, float and ultrasonic. We opted out of float first as, while it can be very accurate, the container restraints of our system would make it difficult to get an accurate measurement from this type of sensor. Secondly we considered laser sensors, which would send a beam of light and use the return time as a measurement of distance; this was the most accurate option but by far the most expensive. Finally, we considered ultrasonic sensors, which use a ping and the time it takes to return to the device to determine distance. While this was our most hopeful option, they are highly susceptible to residue and unpredictable conditions. After considering other aspects of our system design combined with the information gathered on these sensors, we decided that the software
calculations handled by the dispensing system were adequate for liquid level checking, as well as the most time and cost efficient option.

6.1.1.3.4 Cup Detection

Cup detecting was perhaps the most vague requirement to work from, but the sensors we considered for this application were laser, infrared and load sensors. The first sensor we considered and decided against were load sensors; while they could be used to help figure out how much loss there is from the dispensing system to the cup, one of our restraints was the ability to sense Solo cups, which proved to be too light (0.56 oz) for most of these types of sensors to detect accurately. The second type of sensor we considered and eliminated were infrared. An infrared beam would have been poised across from an IR sensor and the change output voltage could be used to determine if contact between the beam and sensor are broken, given some signal processing. However, after some research into optical safety, it was found that IR is on the more dangerous end of lasers when compared to visible light as they do not stimulate the blinking reflex and they have an unsafe power output.

Laser diodes satisfy all the limitations of our system with adjustable power output which can bring them into a safer range for the human eye. The implementation and adjustment of this sensor will be discussed later, but the operation of this aspect of the system is based on a laser being pointed at photoresistors. Photoresistors are very simple sensors, consisting of cadmium sulphide which changes the resistance of the sensor depending on the amount of light it is exposed to. As they are exposed to more light, the resistance decreases and thus the voltage between two nodes is proportional to the amount of light present.

6.1.1.4 Final Requirements

- Read fingerprint accurately enough to associate with each user
- Fingerprints can be taken from the module and stored on the database and vice versa
- Convert gas input for BAC detection
- Gas reading accurate enough to set cutoff threshold
- Check for cup placement on device directly below the dispensing nozzle
- All sensors must be able to communicate with a microcontroller

6.1.2 Process and Implementation

6.1.2.1 High Level Operation

6.1.2.1.1 Individual Sensor Testing

- BAC: Waft into sensor at varying states of alcohol dilution and use a program which can assist in determining an appropriate threshold for our application. We compared measured values to known values based on a commercially available, relatively accurate BAC sensing device. When characterizing the output of the alcohol sensor, the sensor output was read and logged via microcontroller whilst performing a series of regulated tests.
- Fingerprint: Put thumb on sensor and have a program store value temporarily at first. Change user and/or lighting before next measurement and showed that the print either
matched up or did not based on condition changes. Further testing would include getting and setting the fingerprint template on the module.

- Cup Placement: Had a circuit which turns an LED on or off based on whether cups of varying sizes and weight were placed in the desired location (between the emitters and receivers). Eventually connected to microcontroller and set thresholds in software.

### 6.1.2.1.2 Integrated Sensor System Testing

The user would order a drink remotely, and then approach the machine. At this stage the user would have their fingerprint scanned and compared to the user’s known fingerprint. Then the user would be prompted to breathe into the alcohol sensor to determine whether they were eligible to continue drinking. The system would check the cup placement sensors to determine if there was a container which can be poured into.

### 6.1.2.2 Sensor Application

#### 6.1.2.2.1 User Identification

This fingerprint scanner uses UART communication (default baud rate: 9600bps), thus there are four pins to connect to the module: TX (3.3V), RX (3.3V), GND, and VIN (5V). The only additional circuitry needed for this sensor was a voltage divider to prevent RX from exceeding 3.3V, and a reverse power diode to prevent breaking the module. The response, command and data packets are all included in the module datasheet, as well as a list of responses and commands and their respective codes. Both C++ - which we first used for testing on arduino pending our microcontroller selection - and Python libraries were available for use with the sensor we selected, so the command and responses were mostly available as functions. We made get and set template functions for the Python library for use with our system. We tested all of the functions thoroughly, starting with blinking the light in the scanner, then proceeding to enrollment and identification, and the get/set template functions once we wrote them. Additional functionality we needed to test included deleting a specific fingerprint ID (as well as delete all) and checking if that
ID still has an associated fingerprint. When enrolling, the module is told to look for the next available ID number to associate with a fingerprint. We also tested both the verify 1:1 functionality, which takes an ID and checks for a match, and 1:N verification which compares the print to all of the prints in the module. After testing the module thoroughly we were able to start testing it within the context of the larger system, which was discussed in more detail in the microcontroller section along with the get/set template functions.

6.1.2.2.2 BAC

The MQ-3 sensor has six pins with a few redundancies, but all breakout boards for this sensor simplifies it to 3-4 pins. Either A or B is used for VIN, with the other being used as the output node. H is a heating element and can be tied to whichever pin is being used for VIN. Finally, there is usually a ground pin as well. Aside from that, the output pin’s voltage is divided using a load resistor, which we use 10kΩ. A breakout board we used later simplified this even further with VIN, GND, analog out and digital out pins, which removed the need for additional circuitry. There is a required burn-off stage of at least 48 hours for the sensing material to be ready for use, but at least once we found that it took longer for the material to burn in before we could see much difference in the output when alcohol was present. For early testing, we used hand sanitizer to test functionality, later developed an off-campus procedure for testing to collect data (the results of which will be discussed more in depth in a later section), and finally we would use gum and mouthwash to simulate alcohol consumption. In order to test, we felt it was necessary to have a BAC chamber to direct breath over the sensor, as all breath based BAC tests used a chamber of some sort. In future implementations, it would be beneficial to have the chamber regulate the exact amount of air in the chamber to simplify the math for estimating BAC (which would also require us to collect user gender and weight), but for this project it proved sufficient to simply direct the air over the sensor.
6.1.2.2.3 Cup Detection

When we initially set up the cup detecting test, we used a comparator circuit with hysteresis to turn on and off an LED indicating whether a cup was placed between the transmitter and the receiver. The trip points we used were 4.0 and 4.3V at the time of testing (this would have to have its voltage dropped in order to be used by the microcontroller which is limited to 3.3V). The original plan was to also use this comparator circuit to send a distinct high/low into the microcontroller, but the trip points change depending on lighting condition, so we opted for collecting raw sensor values from the photoresistors and having adjustable thresholds in code, which was discussed in detail in the microcontroller section. We started with a single laser diode/photoresistor pair, but needed at least two pairs in order to have a point where they intersect which would give a more precise location under the dispensing nozzle. Additionally, we wanted to have a grate so that spilled beverages could feed into a drain, so we designed a grate which had holes cut out of the sides where the lasers and resistors would be located. The placement
of these sensors can be seen in the image above. Finally, we had to consider optical safety when designing this system. Initially, we ran the sensors at the recommended 3V which gives a power output of 5mW. While 1-5mW is considered a fairly safe output, it can still temporarily damage the eye, and in dark rooms where people's reflexes are diminished this could be dangerous. Thus, we calculated and tested dropping the power output below 1mW, which corresponded to dropping the input voltage below 2V, which we immediately implemented.

6.1.2.3 BAC Results

After designing a test procedure and software test harness for logging BAC, we found that the MQ-3 was accurate in ways we weren’t anticipating, but inaccurate in expected ways. The base values for the output didn’t necessarily have to be consistent as we saw that sometimes when the sensor didn’t have time to cool down between measurements, the output would jump up to consistent values when users of the same BAC blew into the sensor. However, sensor output varied slightly with temperature conditions and when there wasn’t enough time for the sensor to warm up before first taking measurements. The data was collected during testing to be analyzed to determine how an ADC value correlates to BAC. The graphs below show how, while there are numerous outliers, especially in the second experiment, there is a visible trend in the data points collected during the experiments held on different days. Based on this data and observations collected during the testing phase, we decided that around .06-.07 would be suitable cutoff limits. Since an additional drink would likely send an individual over .08 - the legal limit - this was intended to prevent people from exceeding that value. Using this BAC selection and looking at the graphs, we chose an ADC value of 350 as the cutoff threshold, tending a little lower than the trend seen from the second experiment as a precaution due to the results from the first experiment which were slightly lower in value.
7 Microcontroller

7.1.1 Background
It was decided early on that the dispensing systems, sensor systems and other low level hardware would be a separate entity from the user interface itself. Because of this, there were decisions that needed to be made about first, which microcontroller to use, and it had to be decided how the hardware and software would flow. The microcontroller would be the hub that would control and interface the following peripherals: breathalyze, fingerprint scanner, cup detection, dispensing system, and tablet. Beyond this, and as a result of the aforementioned peripherals, there were some lower level requirements as well. The microcontroller required an analog to digital converter (ADC) for interfacing the MQ3 ethanol sensor and the two laser cup detectors, USB for interfacing the tablet, UART for the GT511C3 fingerprint scanner, and internet connectivity for receiving data from the server. With these requirements in mind, we decided to use a Linux based microcontroller because of the computing power and operating system environment that made it efficient to program both embedded software and achieve the networking requirements. Ultimately, we had a choice between the Beaglebone Black and the Raspberry Pi B+. The advantages of the Beaglebone over the Raspberry Pi is that it has 69 GPIO pins whereas the Raspberry Pi has only 40, the BeagleBone has 7 onboard analog pins where the Raspberry Pi has none, and has more power at 1GHz vs the Raspberry Pi’s 700MHz. The Raspberry Pi, however, offers its community which includes support and a bigger user base, 4 USB inputs over the Beaglebone’s 1 and at the time of consideration was 20 dollars cheaper at 29.99 to the Beaglebone’s $50 price tag. At that price, and considering we would be using multiple USB devices including a mouse, keyboard, Wi-Fi USB dongle, and tablet, the Raspberry Pi was decided. Since we decided that the Beaglebone’s 69 GPIO was overkill due to most external systems being driven serially using shift registers. Also, with respect to ADC, an external chip ADC chip was used.

Once the microcontroller was decided, we needed to decide on how to implement the different modules. For the breathalyzer and cup detection inputs, the Microchip MCP3008 10-bit 8 channel ADC was used. The advantage of this ADC is the fact that it communicates over SPI which is a well understood and well documented communication interface. This particular ADC met our requirements. At 10-bits, the best resolution is $2^{10}$, so 1 in 1024 which is .0977% of the input voltage of 3.3V. The fingerprint scanner and tablet communications needed less hardware to get to run but more software.

7.1.2 Process and Implementation

7.1.2.1 USB Communication

“Everybody get on the bus”

USB was chosen for communication between the tablet and the microcontroller. Android Open Accessory (AOA) protocol is the Android developers’ solution to communication between Android devices and products over USB or Bluetooth. Bluetooth was considered momentarily but was quickly dropped because of connection robustness concerns as well as security concerns. Universal Serial Bus or USB is a standard that is widely used and its protocol has been long developed so there are plenty of resources for it. There is a process set up for peripherals to connect to Android via Android Open Accessory. There are three things that can happen when one connects an Android device to the accessory.

- the connected device is AOA compatible and in accessory mode
- the connected device is AOA compatible and not in accessory mode
- the connected device is not AOA compatible
Since we purchased an AOA 2.0 compatible device, the third case would not apply. The process used to establish connection with the accessory is as follows:

1. get accessory()
   Finds USB device with the respective vendor and product ids of 0x18D1 and 0x2D01 which describe an Android device in Accessory mode. This is used at the beginning and end of the connection process. Returns device descriptor, dev, if device is found. If not, it return NULL and the rest of the following functions are run.

2. get_rca(dev)
   Finds USB device with the respective vendor and product id’s of 0x0414 and 0x0C02 which describe the RCA tablet, originally found using lsdev.

3. set_protocol(dev)
   This function does a control transfer to the USB device and depending on the return value, determines which protocol is used.

4. set_strings(dev)
   This function sends strings describing the device connecting to the Android. The strings describe the manufacturer, model, description, version number, url for the website to download the appropriate app, and the serial number. These strings will prompt the device to spawn the appropriate app.

5. set_accessory_mode(dev)
   Control transfer which actually tells the Android to go into accessory mode. Once this finishes, getAccessory() is called once more to find the device descriptor.

Once the device is in accessory mode, it is ready to send and receive messages from the tablet. A thread is started that reads the input buffer at the device descriptor every .02 seconds if a message is received. The message is then concatenated and sent to Parse_Message() which lives in Control_IO.py. Parse_Message() parses the message received by the tablet and calls the appropriate subsystem to deal with the parameters. Messages received from the tablet are in the format: ${system},{subsystem or parameters}. For example, when the tablet sends a $FPENROLL,18318675309, the fingerprint enrolling function is called with that user ID as the parameter. Messages are sent back to the tablet by writing to the output buffer at the device descriptor.

One of the persisting issues with the Android Open Accessory protocol using the tablet as the accessory is an issue with reconnection. When the USB cable is physically disconnected, the tablet sees a USB disconnect, an intent is raised and the tablet changes back to its normal mode. When the USB is still connected and the app is closed, the Pi still sees it in accessory mode and so to it, nothing is wrong. Unfortunately, the tablet does not know to be in the app and it is the Pi that puts it in the app normally. When the USB is disconnected and connected again, the connection is reestablished and the tablet is put into accessory mode once more. Unfortunately, the app could crash, or be exited for some reason and both systems would be none the wiser. One solution we came up with was implementing watchdog timers between the tablet and the Pi.

A module named Watchdog holds the system time every time the tablet sends a message to the Pi. When a new message comes in, the WatchDog gets the new system time. If this time ever gets over a certain value, chosen at 2 minutes, the function usbreset is called. Usbreset does a system call on a perl script that finds the USB device by its descriptors in lsusb and resets that device’s connection in the usb subsystem. This reset only works if the app is closed so on crash, it has to close.
7.1.2.2 Sensors

The microcontroller is what handles all of the sensors and relays information back to the tablet for continued program flow. There are three sensors in the Smartbar kiosk, two of which are interfaced via an analog SPI chip and the second over TTL serial using the UART on the Pi. The MCP3008 uses only 4 pins on the Pi’s header: clock, input, output, and chip select. These are connected to the Pi’s SPI_CLK, SPI_MISI, SPI_MISO, and SPI_CE0_N pins, respectively. Analogspi is the module that initializes the SPI device on the Pi’s SPI0. Readchannel(channel) is also in the module and returns 0-1023 depending on the parameter, channel (0-7).

CupDetect is the module which deals with the cup detection sensors. Due to varying lighting conditions, the cup sensors may return a number of different values but generally, when a solid object interrupts the photoresistors, the value will drip by at least 5 ADC values. Finding the high and low threshold is done by Calibrate() which returns the maximum, minimum, and average value in 100 readings 0.2s apart. Calibrate is done when a cup is placed and also when a cup is removed to find the appropriate thresholds for CupDetect(). CupDetect simply samples the ADC values and when they dip below or above a threshold. Hysteresis was originally implemented but deemed unnecessary due to the signal from the sensors being stable enough. CupDetect is never run on its own, however. The tablet calls a $CUPDETECT,START on the Pi and a thread is spawned, CupMonitor(), which sends either $CUP, PLACED or $CUP, REMOVED back to the tablet. It also resends the aforementioned messages every second after in case the tablet missed the message. The CupMonitor thread is terminated when a global variable is changed to false.

BACDetector is the module which handles the BAC sensor. When the tablet sends the $BAC,START command, a number of steps happen. First, BACDetector takes a baseline of the sensor in ambient air. Next, a moving average of the sensor value is run in a loop until it reaches a certain threshold above the baseline. This is used to actually detect if a person is blowing. Whether or not a person is intoxicated, the sensor value will increase slightly when air moves past it. Once the threshold is reached, CollectBACSample() is called which collects 400 samples .005s apart. The BAC module sends $BAC,PASSED,\{x\}$ or $BAC,FAILED,\{x\}$ where \{x\} is the BAC value detected. Unfortunately, this particular sensor is not the best or most accurate so the threshold value has to be constantly adjusted depending on several factors including the temperature of the sensor.

Fingerprint

The GT511C3 fingerprint scanner is an integral part of the system. It allows users to forego the long process of logging in with their personal information which could take a long time, resulting in impatient users skipping the Smartbar. It is also a security consideration which prevents peering eyes from stealing a user’s login information. The goal we had in mind for the fingerprint scanner was for a user to be able to walk up to the Smartbar, put their finger on the scanner and pick up their drink without having to press more than two buttons. There were several hurdles to overcome before this could be done but in the end, it turned out to be a remarkably robust and stable means of identification.

First, getting the information in and out of the sensor. The library we used for the GT511C3 scanner did not have the GetTemplate and SetTemplate function implemented. This means that we were limited to the 200 fingerprints that could be stored on the scanner itself. This would pose several issues. Should the particular Smartbar have more than 200 users, new users would not be able to get a drink. If we deleted the fingerprint templates periodically, users would have to re-register and that would be an annoyance which could prevent them from using our service. The only solution would be to implement the Get/SetTemplate functions. To do this, we studied the underlying code and datasheet and found that the scanner will send and receive messages with the following packet structure:
We ended up printing a lot of the data to try and visualize what data the scanner was returning at a given time. Using the function hexlify, which turns binary data to human readable hex, we were able to see the specific pieces of the packets and distinguish the different parts of the messages sent from the scanner. We were finally able to enroll a fingerprint, and then print out the raw data from the fingerprint template. Once this was achieved, SetTemplate came relatively easy. The next step was storing the fingerprints on the system.

### Command Packet (Command)

<table>
<thead>
<tr>
<th>OFFSET</th>
<th>ITEM</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x55</td>
<td>BYTE</td>
<td>Command start code 1</td>
</tr>
<tr>
<td>1</td>
<td>0xAA</td>
<td>BYTE</td>
<td>Command start code 2</td>
</tr>
<tr>
<td>2</td>
<td>Device ID</td>
<td>WORD</td>
<td>Device ID: default is 0x0001, always fixed</td>
</tr>
<tr>
<td>4</td>
<td>Parameter</td>
<td>DWORD</td>
<td>Input parameter</td>
</tr>
<tr>
<td>8</td>
<td>Command</td>
<td>WORD</td>
<td>Command code</td>
</tr>
<tr>
<td>10</td>
<td>Check Sum</td>
<td>WORD</td>
<td>Check Sum (byte addition)</td>
</tr>
</tbody>
</table>

### Response Packet (Acknowledge)

<table>
<thead>
<tr>
<th>OFFSET</th>
<th>ITEM</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x55</td>
<td>BYTE</td>
<td>Response start code 1</td>
</tr>
<tr>
<td>1</td>
<td>0xAA</td>
<td>BYTE</td>
<td>Response start code 2</td>
</tr>
<tr>
<td>2</td>
<td>Device ID</td>
<td>WORD</td>
<td>Device ID: default is 0x0001, always fixed</td>
</tr>
<tr>
<td>4</td>
<td>Parameter</td>
<td>DWORD</td>
<td>Response == 0x30: (ACK) Output Parameter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Response == 0x31: (NACK) Error code</td>
</tr>
<tr>
<td>8</td>
<td>Response</td>
<td>WORD</td>
<td>0x30: Acknowledge (ACK).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x31: Non-acknowledge (NACK).</td>
</tr>
<tr>
<td>10</td>
<td>Check Sum</td>
<td>WORD</td>
<td>Check Sum (byte addition)</td>
</tr>
</tbody>
</table>

### Data Packet (Data)

<table>
<thead>
<tr>
<th>OFFSET</th>
<th>ITEM</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x5A</td>
<td>BYTE</td>
<td>Data start code 1</td>
</tr>
<tr>
<td>1</td>
<td>0xA5</td>
<td>BYTE</td>
<td>Data start code 2</td>
</tr>
<tr>
<td>2</td>
<td>Device ID</td>
<td>WORD</td>
<td>Device ID: default is 0x0001, always fixed</td>
</tr>
<tr>
<td>4</td>
<td>Data</td>
<td>N BYTES</td>
<td>N bytes Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The size is pre-defined per protocol stage</td>
</tr>
<tr>
<td>4+N</td>
<td>Check Sum</td>
<td>WORD</td>
<td>Check Sum (byte addition)</td>
</tr>
</tbody>
</table>

OFFSET[0]+...+OFFSET[4+N-1]=Check Sum
Once printing out fingerprints to the console was complete, saving the fingerprint templates as files was trivial. Simply put, a new file is made and the bytes of the template are written to it and stored in a directory. The issue was managing the templates. The enrollment process is as follows: the code iterates through the scanner, finding the first available space to place a finger; the user is prompted to place and remove their finger three times; the template is stored at that index. When a user orders a drink, he/she is put on the queue. The queue is sent to the Pi from the tablet as a list of user IDs. Every time the queue changes, it is because a user is no longer on the queue or a user is already on the queue. This means that the underlying structure has to be able to remove or put templates on the scanner. It was possible to delete all of the templates and refill the scanner every time a new queue was updated but that took too long. The solution was to have the templates added and deleted based on the changes to the queue and not the whole queue.

Phalange is the module that keeps track of the fingerprints on the scanner. Each fingerprint on the scanner is assigned an index once it is set that does not change until it’s deleted. This index is then associated with the user’s ID, in an object, phalange. All of the phalanges are put into a list, the phalangeList through the refill() function. The refill() function will append the phalanges on to the phalangeList and hold the items that are removed on a temporary list. It does this by populating a temporary phalangeList, appending the phalanges from the current phalangeList that are also on the queue as well as creating new phalanges for those that are on the queue and not on the current phalangeList. Once the temporary phalangeList is populated, it is compared to the current phalangeList. Entries that are on the current phalangeList but not on the temporary phalangeList are found by doing a set subtraction. The indices of these phalanges are then put into another list and returned to the calling function. The temporary phalangeList is set as the current phalangeList and the old one is deleted. The returned list of scanner indices is then deleted from the scanner. The appendPL() function will either append 0-199 if the template was successfully added to the scanner, or -1 if the template was not. For example, the phalangeList could look like:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8318675309</td>
<td>0</td>
</tr>
<tr>
<td>8185554201</td>
<td>12</td>
</tr>
<tr>
<td>4206666969</td>
<td>-1</td>
</tr>
<tr>
<td>8002255288</td>
<td>3</td>
</tr>
</tbody>
</table>

This means that in the scanner, there are 3 phone numbers: the 831, 420, and 800 numbers. Note they are not necessarily in order or sequential. This could be because:
- 13 users ordered drinks
- users at 1-11 picked up their drink,
- new users at 1 and 2 ordered and picked up their drink,
- 0, 12, and 3 are left

It is also important to note that the 420 number is on this list but does not have a fingerprint in the scanner. This means there is no fingerprint on the scanner for that ID, a result of the template for the user was not being found on the database. When this particular user goes to get their drink, they will find that their finger is not on the system and will be prompted to enroll. Once enrolled, their index is saved onto their phalange object.

Note that the phalange module knows nothing about the fingerprint templates themselves or anything about the scanner. It only keeps track of the indices in the scanner and associates them with the user. The phalange module is instead used by FingerprintGet_DB which handles everything the
Smartbar has to do with the templates. The module has several distinct components it uses to manage fingerprints: MySQL connector, phalanges, file system. FingerprintGet_DB connects directly to the MySQL database where it uploads and downloads templates, saving them on the Pi’s /tmp directory. At its most basic level, this module uploads fingerprints from the scanner to the database and downloads fingerprints from the database to the scanner.

When Control_IO calls $FPENROLL, FingerprintGet_DB then dictates program flow by prompting the user through messages sent to the tablet. First, the user is prompted to place their finger on the scanner. The scanner has a function IsPressFinger() which returns true or false depending on if the user has placed a finger on the scanner. When this returns true, the enrollment process begins. The scanner takes 3 different images by using the functions Enroll1(), Enroll2() and Enroll3(), prompting the user to remove their finger and replace it between each capture. Once the captures are complete, the scanner turns the 3 images into the fingerprint template and return a success message or a coded error message. If the enrollment fails, the scanner will return which capture the enrollment failed. If it succeeds, save_template() is called.

Save_template establishes connection to the MySQL server by using the MySQLdb.py library. It first connects to the database using the credentials saved in the function DB_connect. Save_template then sends the MySQL instruction as a string: “UPDATE users SET fingerprint = "%s" WHERE userPin = "%s" % (binascii.hexlify(varbin_value), str(fileID))”. Where varbin_value is the string of the fingerprint template.

When the tablet calls $FPQ,[list of user IDs], update_queue(queue) is called, which first downloads the templates at the userID into /tmp if and only if the template is not already on the scanner. It then populates the scanner with the appropriate templates using populate_scanner(). When this process finishes, /tmp is cleared and an ack is sent back.

Finally, when the tablet calls $FPIDEN,START, the module prompts the user to place their finger on the scanner. Once the scanner reads that a finger is on, the identification is started. The identification allows the scanner to find the finger in the pool of all templates. When the index of that finger is returned, it is found in the phalangeList and the user ID is returned.

8 User Interface

8.1 Approach

The Smart Bar requires a user interface such that the person ordering a drink can quickly verify their identity, BAC, and drink order. In order to make sure that this is done in a timely manner, a clear and intuitive interface is required. This interface must also communicate with low level hardware and a database simultaneously. The visual requirements of the UI such as screen resolution, brightness, and size were based on the specifications of current products in the market while also being constrained to an initial budget of $150.

The first design considerations were based around using a touch screen that could be added to a microcontroller with an onboard OS such as the BeagleBone or Raspberry Pi, but upon further research, most touch-screen displays made for microcontrollers were either 7” with high resolution such as 1280x800, or 10.1” with lower resolutions of 800x600 (see Raspberry Pi 10.1” Screen). These screens either were separated from most touch screens and would have required external controllers and touch screen interfaces or cost close to or above the UI operating budget. Furthermore, the options for programming a visual interface on the microcontroller alone would not only have taken most of the
memory and processing power of a microcontroller (such as Qt Creator), but also maintained an alien aesthetic that is unfamiliar to most users.

Research was then shifted to a tablet based UI where most of the screens met the visual requirements. Most tablets met either the budget requirement or visual requirements, but only a select few Android Tablets managed to meet both requirements. Since the mobile app was programmed on an Android device, the interaction with the database did not have to change drastically if the UI was also an Android application. The only requirement left to meet was for the UI to be able to interact with low level hardware.

Several means of controlling low level hardware were considered including using a Wi-Fi Direct or Bluetooth connection to a separate microcontroller, or a use microcontroller pre-flashed for Android Devices that can be connected serially. The research on serially connected microcontrollers brought us to investigate the Android Open Accessory protocol. The Android Open Accessory protocol is a method of connecting the Android device to a device with other specific parameters either over USB or via Bluetooth. This allowed us to choose a microcontroller better suited for our design as long as it had a USB port. Once all the design requirements were met for the UI, the RCA 10.1” Quad Core tablet was purchased.

RCA Pro II 10.1”

8.2 Process and Implementation
Communication from Android Tablet to Microcontroller

The first means of designing the full UI began with the Raspberry Pi sending simple messages to the tablet running the Android application, and, through a pseudo-console application, send back data. The Android Open Accessory gives both the table and the Pi bulk endpoints through which they can send
large amounts of data. On the Android Application end of the communication these are in the form of an Input Stream, Output Stream, and File Descriptor, which are maintained constant throughout the communication. Within the app all incoming data is read by a timer triggered interrupt service routine, which in the Android Development is simply done by periodically spawning a thread, reading the Input Stream for data, posting the data to the process buffer, and spawning a new thread after a period of time, killing the old thread.

```
Runnable mListenerTask = new Runnable() {
    @Override
    public void run() {
        try {
            if (isActive) {
                InMessage = PiComm.readString();
                String res = SystemCodeParser.DecodeAccessoryMessage(InMessage);
                String out = null;
                if (InMessage != null) {
                    mText.post(mUpdateUI2);
                }
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                //Restarts this thread.
            } catch (NullPointerException npe) {
                npe.printStackTrace();
            }
            if (PiComm.isDisconnected()) {
                finish();
            } else {
                new Thread(this).start();
            }
        }
    }
};
```

All information regarding the connection between the Tablet and the Raspberry Pi is stored in the CommStream class, which in the snippet of code above is referenced as a variable PiComm. The data is read in from PiComm.readString(), and decoded by the System Code Parser class, via DecodeAccessoryMessage().

The communication between the Android and the Raspberry Pi maintained a predefined command packet structure in order to run state machines on both devices or directly access low-level hardware. (Table Below). Most processes are started by a user interfacing with the tablet, upon which a command packet, usually with "START" parameters, is sent to start a process on the Raspberry Pi. Most processes send back information to the tablet, which in turn runs the corresponding state machine.
<table>
<thead>
<tr>
<th>Command</th>
<th>Subsystem</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SYSMAIN</td>
<td>Communication Link</td>
<td>START</td>
<td>Starts all communication with aux devices</td>
</tr>
<tr>
<td>$FPQ</td>
<td>Fingerprint</td>
<td>,##########,...</td>
<td>Sends lists of phone numbers whose fingerprints need to be loaded</td>
</tr>
<tr>
<td>$FPIDEN</td>
<td>Fingerprint</td>
<td>START TIMEOUT SUCC</td>
<td>Check the fingerprint against the templates on the scanner</td>
</tr>
<tr>
<td>$FPENROLL</td>
<td>Fingerprint</td>
<td>,############### EN* RM* TIME OUT ERR, * SUCC</td>
<td>Enrolls the user, returning messages to either place the finger, remove the finger or indicate success/failure</td>
</tr>
<tr>
<td>$BAC</td>
<td>Blood Alcohol Sensor</td>
<td>START, TIMEOUT, CHECKING, PASSED,### FAIL,###</td>
<td>Checks for alcohol in breath over sensor, and returns whether or not the user passed/failed</td>
</tr>
<tr>
<td>$CUP</td>
<td>Cup Detection Sensor</td>
<td>START, END, ON, OFF</td>
<td>Starts a cup detection loop which periodically returns whether there is an obstacle in the holder or not.</td>
</tr>
<tr>
<td>$DO,$DS,</td>
<td>Dispensing System</td>
<td>#ofL,#ofM@Spirit,Brand,vol...</td>
<td>A formatted string containing the recipe for the drink about to be poured</td>
</tr>
<tr>
<td>$DO,$PG</td>
<td>Dispensing System</td>
<td>START</td>
<td>Requests a purge from the dispensing system to clean the grate/nozzle</td>
</tr>
<tr>
<td>$IV</td>
<td>Inventory</td>
<td>$ofL,$ofM@container#,spirit,brand,volume...</td>
<td>A formatted string representing the current inventory of drinks</td>
</tr>
<tr>
<td>$LOG</td>
<td>Logging System</td>
<td>,*****</td>
<td>A logging system by which the tablet can send any useful information to be read on the terminal of the raspberry pi.</td>
</tr>
<tr>
<td>$WATCH</td>
<td>WatchDog Timer</td>
<td>WOOF, OKAY BARK</td>
<td>Watchdog Timer requests/response</td>
</tr>
</tbody>
</table>

The communication thread runs in the background of the main application which, when successfully connected, begins the Smartbar main User Interface. Any messages from the Pi that are relevant to the User, or require the screen to update its appearance are forwarded to the Main UI thread, which the Android OS is constantly running beginning when the application is opened.
Android Communication Overview

Communication from Android Tablet to Database

Throughout the application the tablet needs to access data relevant to both the users on the drink queue, and also information about what drinks are in the library based on the current inventory. The communication to the database is performed via a HTTP post on a separate thread from the main UI thread. Any information received back from the database is in the form of a JSON object, which is parsed and decoded for use in the current activity.

The asynchronous thread is called with format of new AttemptGetDesiredInformation().execute(), and each task makes the HTTP post with its own predefined URL by calling jsonParser.makeHttpRequest(ServerAccess.URL, “POST”, params). The application concurrently logs all information getting posted to the server, and upon the successful return of the JSON object, the data is sent through the same SystemCodeParser that the data from Pi is sent through.
Example of accessing the database via calls to the web server.

**Communication with User**

From the beginning the first input to the application is done via the user, whether to pick up their drink or order one from the kiosk. To keep the flow of the UI simple, most screens limited the number of choices that the user could input, and most of the time it gave instructions to interact, rather than freely allow interfacing with the system. This design choice limited the amount of user error that could be introduced into the system, while simultaneously preventing the application to bloat with unnecessary screens.
The flow chart above describes the essential and unique screens that the user will encounter throughout the flow of the system. Screenshots of each screen can be seen in Appendix G. Each screen is created by using a new Android Activity, which is the base class the Android SDK uses to add visuals. Each activity has a “lifetime” based on whether another screen is placed on top of another, it is minimized, or a pop-up comes to the forefront of the screen.

Each of the individual screens is ran by its state machine, which can take inputs from both the use and the Pi. Should any errors occur, the appropriate error dialogs/messages will appear, prompting the user to correct the issue if possible, or alerting them to wait until the issue is fixed.
Lifetime of Activities

Early on, a design choice was made to remove all screen history, meaning that once a screen was left to move onto another, all variables and memory related to the screen are destroyed. Only two activities kept history throughout the application, first the Main Activity which held the console where the communication between the Pi and the tablet, and second the Main Menu, which acts as the root of the kiosk for bar owners. Due to the looping nature of the system flow, and the resetting of the multiple state machines, using existing activities began to bloat the application, and variables began getting hard to keep track of. Destroying the activities required additional static classes to maintain information that could be used by newly allocated activities, which led to the creation of the Drink Order, Server Access, and Inventory classes. These static classes held information about the current user interacting with the Smartbar kiosk such as phone number, drink recipe, fingerprint data, bac data, email etc... The Drink Order class also held function calls to parse up the drink recipe into individual components. The Server Access class contained all the variables and urls needed to access the database, and simply retrieve information not available from the user input. Finally the inventory class contained information about the current stock of the Smartbar by which an administrator inputs into the kiosk.

Creating a Kiosk

Once the main Smartbar UI was running, we wanted to ensure that the user could not exit the application and/or retrieve access to low level hardware systems. This proved a challenge as the default
screens not only displayed the top menu bars that could be dragged down, but also the home menu was constantly displayed on the bottom of the screen.

To mitigate this potential problem, a series of flags needed to be set for the screen settings within the Application itself. At the time there was no name for setting all the flags, but after the release of Android 5.0, the mode is now considered Immersive mode. Immersive mode instantiates the screen as a Window object by which certain flags can be set to hide all the menu bars.
The only issue that occurs when using the “Immersive Mode”, is that if the user swipes from the top or bottom of the application, the menus reappear, making the application vulnerable for the user to exit the application. The solution to this was to periodically reset the Immersive mode within a TimerTask. A TimerTask in Java is simply a small thread that is attached to a Timer and runs until the task is cancelled. The TimerTask was initially set to trigger every 100ms, but after watching users forcibly try to exit the app, a shorter time of 10ms was set. With the new time, users struggled to exit the app, and quickly gave up.

Another solution that was considered in securing the application was to root the Android Tablet and configure several of the low-level hardware settings to remove the menu bars. This would also allow the tablet to forcibly only run one application. Doing so not only required external software to implement (which would require extra manufacturing to perform on each tablet), but also held a risk of ruining the tablet permanently if performed improperly. For the system proof of concept the periodic hiding of the screens proved successful.

```java
private void hideSystemUI() {
    // Set the IMMERSIVE FLAG
    // Set the content to appear under the system bars so that the content
    // doesn't resize when the system bars hide and show.
    View mDecorView = getWindow().getDecorView();
    mDecorView.setSystemUiVisibility(
            View.SYSTEM_UI_FLAG_LAYOUT_STABLE|
            View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION|
            View.SYSTEM_UI_FLAG_HIDE_NAVIGATION|
            View.SYSTEM_UI_FLAG_FULLSCREEN|
            View.SYSTEM_UI_FLAG_FULLSCREEN // hide status bar
            View.SYSTEM_UI_FLAG_IMMERSIVE_STICKY);
}

    TimerTask HideTask = new TimerTask() {
        @Override
        public void run() {
            IdleMenu.this.runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    hideSystemUI();
                }
            });
        }
    };
```

Full Screen Immersive Flags
9 Dispenser

9.1 Dispensing System Requirements

The dispensing system has several main requirements that are essential to dispense beverages:

- Dispense soda, juice, and water
- Make soda/juice by mixing syrup and water
- Make carbonated water
- Cool carbonated/noncarbonated water before mixing
- Supply water to system
- Supply pressurized gas to the system safely

These tasks can be executed by various dispensing components, each with different characteristics intended for a specific application.

9.2 Dispensing System Research

There are many different beverage-dispensing systems available with varying abilities based on the intended application. We will create our own custom system because our requirements are not fulfilled by many commercially available dispensing systems. Fortunately, dispensing components are very modular due to the need for replacement and repair in high workload environments such as restaurants. Each requirement was researched to discover the available options before considering specific components or systems.

Dispense soda, juice, and water

There are a variety of systems used in beverage dispensing applications, which fall into two main categories: post-mix and pre-mix. Pre-mix systems use kegs of pre-made soda or juice that must be refrigerated and pressurized with CO2 to retain carbonation. These systems are very large because the water has already been mixed into the soda and also expensive because each of the kegs must be refrigerated. These characteristics make pre-mix systems impractical for our application. Post mix systems combine cooled carbonated/noncarbonated water with concentrated syrups to make fresh soda or juice. The ratio of water to syrup is typically around 5:1, meaning that the majority of the liquid in the system is fresh water. This is ideal for our application because it allows us to have many different dispensing options available in a small space.

Make soda/juice by mixing syrup and water

Typically each flavor of soda/juice is made in an individual specialized valve as seen in soda fountain dispensing machines in restaurants. This is not ideal for our system because these valves are large and not designed to dispense liquid to another location through a tube. Alternatively, soda/juice may be dispensed in a more compact way by using a bar gun, which concentrates all the beverage lines into a single nozzle. Flavors are then selected by pushing buttons on the bar gun which opens up the proper
valves to be mixed by the specialized nozzle. This small scale dispensing apparatus is ideal for our application and influenced the final dispenser nozzle design.

**Cool carbonated/noncarbonated water before mixing**
Immediate cooling can be achieved in various ways, some are very expensive and some are very inexpensive. The most optimal method is to use a refrigeration system that is designed to instantly cool any liquid that passes through it. These refrigeration systems use a compressor to freeze a large block of ice that is submerged in a water bath containing the beverage lines. Another commonly used cooling system is the aluminum block cooler. These aluminum blocks contain narrow hollowed out channels that cause the liquid to circulate many times before exiting the block. The thermal properties of aluminum (Thermal conductivity of Aluminum (@25°C) = 205 W/(mK) vs Ice(@0°C) = 2.18 W/(mK)) result in the rapid transfer of heat from the liquid to the block causing the liquid to become roughly the same temperature as the block.

**Supply water to the system**
Most dispensing systems are connected directly to an external water line with pressure supplied from a separate system. Our system needs to be portable so we will use a pump to supply pressurized water from a reservoir. There are many affordable pumps on the market (for boats, RV’s, agriculture, etc) that are designed to mimic the water pressure supplied by a city’s water system. Our system will also be compatible with an external water supply by switching the pump out for a water pressure regulator connected to a pressurized water source.

**Supply pressurized gas safely**
The carbonator and many of the potential pumps are powered by pressurized gas, either CO2 or compressed air. CO2 is the ideal gas to use in this system because a large amount can be stored in a small canister and the carbonator requires CO2 in order to carbonate the water. A 5lb CO2 tank will be used because it is the smallest canister available that will suit our application. We will use high quality adjustable CO2 regulators to ensure that the components will always receive the correct amount of pressure for operation. This will also allow us to adjust the system if it becomes necessary later on. Although the end product will use CO2, we will be using compressed air for most of our testing because it is safe to use inside. It is very possible that there will be gas leaks in our first prototypes so it is much better to use a gas we can breathe harmlessly.

### 9.3 Dispensing System Design

The general design of the dispensing system is displayed in figure (DispFigNum).1 to illustrate the concept. A more detailed version of the design is shown in the system diagram section (DispFigNum).4. The dispensing system includes several different component types, which are discussed in the following sections.

- **Water Source** – provides water for the carbonator and to make juice
- **Carbonator** – carbonates water to make soda
- **Pumps** – supply ingredients to be dispensed or mixed with water to make soda or juice
- Cooling system – rapidly cools all beverage lines to always dispense cold beverages
- Dispensing apparatus – mixes ingredients together and supports flow rate calibration
- Valves - open and close beverage lines as necessary
- CO2 Source – powers the carbonator and potentially the pumps
- Fittings/Tubing - food safe and pressure capable transport of gas or beverage

Figure (DispFigNum).1 : Dispensing system concept diagram

**Cooling System**

Specific Requirements:
- Must be able to cool 128 ounces of water per hour
- Water must be cooled to below 40 degrees Fahrenheit
- Must be small enough to fit within the structure

There are two cooling systems that are widely used in beverage dispensing: cold plate cooling blocks and remote coolers. Cold plate cooling blocks are large pieces of aluminum with channels hollowed out for beverage lines to run through. These blocks are placed in an ice bin, which needs to be drained and filled with ice continuously. The carbonated water is the most important thing to keep cold because it retains much less of the CO2 if it warms up.

The cooling system integrated into the SmartBar must be able to cool enough water to produce 20 drinks per hour. Standard cocktails can contain up to 8 ounces of soda, which requires 6.4 ounces of water to mix. The water exiting the cooling system must be below 40 degrees Fahrenheit in order for the CO2 (from the carbonator) to remain dissolved.
- Must be able to cool 128 ounces of water per hour
- Water must be cooled to below 40 degrees Fahrenheit
- Must be a space efficient shape able to fit within the dimensions of the SmartBar
- Must be small enough to fit within the SmartBar

A remote cooling system was out of our budget but we received one as a generous donation from International Carbonics Incorporated. The LFMS unit we were given easily fulfills these requirements also eliminates the need for an external carbonator

**Carbonator**
The most ideal carbonators available are built into remote cooling systems to take advantage of the rapid cooling. Carbonators contained within a cooling unit achieve better carbonation because CO2 is more soluble in water at lower temperatures. Though remote cooling systems are optimal carbonators and we were lucky enough to get one, external systems are a more affordable option that could be used if a less expensive version were to be built. McCann, Lancer are the two main brands that produce carbonators; McCann carbonators, which are approximately the same price as lancer carbonators, appear to be widely preferred and are much more available. There are two models of McCann carbonators, the 43 series and the “big mac” series; The “Big Mac series is slightly more expensive and slightly larger than the 43 series but can has the five times the carbonation tank capacity. I believe that the “Big Mac” series carbonator would be the best fit for our system because it has five times the carbonated water capacity and is only slightly larger and more expensive.

Water Pump and Supply
Specific Requirements:
- Pump water at sufficient pressure and flow rate to carbonator
- Automatic shut off when water line is closed
- Self-priming

The typical water pressure range for fountain dispensing systems is 35-50 psi, and the carbonator requires a minimum flow rate of 1.4 gallons per minute. The automatic shutoff is necessary to avoid damage to the pump from running when the line is sealed. The Flojet Ecotemp 03526-14A was chosen meets all of these requirements with 2.9 gallon flow rate at 50psi and fits our budget. Additionally, a Flojet water filter is attached to the pump input to ensure no particulates enter the pump. Standard 3 or 5 gallon water jugs are used to supply fresh water to the pump.

Syrup Pumps
An analysis of dispensing methods was done and pneumatic syrup pumps were determined to be the best choice. This analysis can be found in appendix section 6.7
Specific Requirements:
- Pump liquor and concentrate
- Self-priming
- Constant flow rate

The Flojet model n5000 is the most widely used and affordable syrup pump for beverage dispensing and is also the least expensive. It was chosen because it is reliable and can pump many different types of liquid at a constant rate.

CO2 Tank and Regulator
Specific Requirements:
- Safe
- Tank must fit onto structure
- 2 separate adjustable pressure outputs on regulator
Our pneumatic system uses two CO2 lines at different pressures thus a dual regulator is required. There aren’t very many options for CO2 Tanks and dual regulators, and only a handful are designed for pneumatic dispensing. Our application requires one line to be up to 100 psi, and the other up to 80 psi. A standard 5-pound aluminum CO2 tank and Cornelius dual gauge regulator, which is designed for dispensing systems, were chosen because they meet these requirements.

**Tubing and Connections**

Specific Requirements:
- Able to handle pressure of high pressure line
- Food Safe

The tubing needs to be made out of FDA approved food grade material and must be able to withstand at least 60psi. There are types of specialized tubing designed for beverage dispensing applications, which are both food safe and support high pressures. Standard quarter inch beverage line braided tubing and antimicrobial fittings were chosen because they meet these requirements and are designed for use in beverage dispensing applications.

**Dispensing Apparatus**

Many different ideas for a dispensing apparatus were considered and ultimately the good parts of some designs were combined to create the dispensing apparatus. These concepts can be found in appendix section 6.6

Specific Requirements:
- Connect all dispensing lines
- Mix ingredients together before dispensing

We chose to design our own mixing tube and nozzle, with a water and 3d printed it out of PLA. This design took several iterations but the finalized version is shown below in figure (DispFigNum).2. The water diffuser splits the water and carbonated water into many small streams that are injected into the manifold to mix more effectively. The diffused water stream also cleans the nozzle better than a single jet and mounts to the top of the manifold with threaded connection. The nozzle has cross hatching bars that mix the ingredients by forcing the mixture to change its path as it travels out into the cup.
9.3.1 Dispensing System Diagram
### 9.4 Dispensing Control Hardware Requirements

The dispensing system has several main requirements that are essential to dispense beverages:

- Control all dispensing components with microcontroller
- Store data on external registers to conserve GPIO
- Safely operate and protect dispensing components
- Scalable and modular design for easy maintenance and dispenser size compatibility

These requirements led to a basic general design concept where the raspberry pi controls registers that drive the switches to supply power to the valves, which are each protected by a fuse.

![General dispensing control hardware diagram](DispFigNum).4

### 9.5 Dispensing Control Hardware Component Selection

The components were selected with more specific requirements defined after the design concept was determined. Fuses are used to protect the valves and are discussed in the design section.

**Data Register**

Specific Requirements:
- 3.3V Logic
- 8-10 Control Bits

The most suitable data register for our application is the shift register because it is controlled using only three GPIO pins that can be used to control additional shift registers as well. Using shift registers allows the control hardware to be designed to be a chain of identical boards that can control as many valves as needed. Using SPI to control the registers was considered but requires two additional GPIO per additional control board, which limits the scalability of the system. SPI control would yield a faster response than chained shift registers when controlling a large number of boards but the speed is not a constraint on our system. A thousand chained 8-bit shift registers (which would control 8000 valves) operating at 1MHz would take .8 milliseconds to store data, which is a negligible amount of time for beverage dispensing. The SN74HC595N shift register was chosen because of its high data clock rate, reliability, and resource documentation available. It is also a safe choice because it is inexpensive and very common.
Power Switch
Specific Requirements:
- Supply 500mA
- Low Voltage Drop
- Reliable in applications with frequent switching

There were several options for the valve power switches that were researched: BJT s, MOSFETs, and relays. Relays are not ideal for applications where the power will be switched on and off frequently because they have a limited lifespan compared to transistors. BJT transistors were chosen over MOSFET because they can achieve a very low on resistance which results in a lower voltage drop than MOSFET with low impedance loads, such as solenoid valves. MOSFETs also require a high input voltage to turn on as opposed to BJTs, which can be driven with any voltage. The TIP 41C NPN Transistor was chosen because it is capable of achieving an extremely low voltage drop when sinking the valve current, and is very inexpensive. An additional PN2222A NPN transistor is used to supply the base current from the power supply to reduce the current draw from the shift register and raspberry pi. Although the SN74HC595N shift register is capable of supplying the current to drive this transistor, it is close to the rated absolute maximum. The current limitation of the raspberry pi wouldn’t be exceeded with the 3 shift registers we plan to use, but it would be somewhat close to the maximum, and would not support additional control valves if they were to be added later.

9.6 Dispensing Control Hardware Design

The dispensing control boards are designed to be modular pieces of the dispensing system, which allows the number of valves controlled to be scalable. This also makes the boards easily replaceable and supports pre-existing system upgrades. Each board contains one shift register and is able to control eight valves with a simple transistor configuration as shown in figure (DispFigNum).5:
The transistor pair to sink current resembles a Darlington transistor but achieves a much lower voltage drop across the larger transistor by powering the small transistor externally. The resistor values were calculated based required current to source 500 mA through the TIP41C. The TIP 41C has a gain of 50 when sinking 50, which requires a 10mA base current to turn on. This results in a collector emitter voltage drop of .08V and a base emitter voltage drop of .75V. With a 12V power supply the resistor value can be calculated and rounded down to the closest standard resistor value.

\[ R_{TIP41C} = \frac{V}{I} = \frac{12V - .08V}{.01A} = 1,192\Omega \approx 1,000\Omega \]

This resistor value is used to calculate the current through thee PN2222A:

\[ I_{PN2222A} = \frac{V}{R} = \frac{12V - .08V}{1000\Omega} = .01192A = 11.92mA \]

The PN2222A has a gain of 35 when sinking 11.92mA, which requires a .341 mA base current to turn on, and results in a base emitter voltage drop of .7V. The resistor value for the base of the PN2222A can be calculated using the 3.3V power supply and the base emitter voltage drop of each transistor. This value is also rounded down to the nearest standard resistor value.

\[ I_{PN2222A} = \frac{V}{I} = \frac{3.3V_S - .75V_{TBE} - .7V_{PBE}}{.000341A} = 5,425\Omega \approx 5,200\Omega \]

Each of the shift register inputs (data, clock, and store) is separated from the input by 1000 \( \Omega \) resistor to protect the pins of the raspberry pi. The fuse value is determined by multiplying the load current by a safety factor of 1.3 and then rounding up to the next standard fuse value:

\[ A_F = I \times SF = .5A \times 1.3 = .65A \approx 800mA \]

Our design uses three of these dispensing boards, which are all connected through an additional signal distribution board. The signal distribution board splits the clock and store outputs from the raspberry pi and connects the data output to the first dispensing board, which is passed to the following boards.

### 9.6.1 Dispensing Software Requirements

The dispensing system has several main requirements that are essential to dispense beverages:

- Receive commands from UI
- Calculate dispensing control data from commands
- Precise operation of control hardware for accurate pours
- Maintain an inventory with data about all stocked ingredients
- Error checking that will only allow valid commands go through

### 9.6.2 Dispensing Software Design

The dispensing software uses various different modules within a main program to execute commands and perform tasks. The main part of the software is the dispenser class, which stores all of the system
data and does dispensing system use the parent of all modules. The dispenser receives commands from
the UI using a protocol that we created to handle communication throughout the entire system, which is
discussed in the following section). These commands are decoded and stored as necessary to execute
the series of function calls necessary to perform the task. All functions are comprised of many small
helper functions for simplicity and code maintainability.

9.6.3 Software Flow

The dispensing system performs tasks when a command is received from the UI through the buffer of
the Raspberry Pi. The Raspberry Pi’s main control software reads commands from the buffer and
determines which system the message is addressed to. Dispensing commands are passed on to the
dispensing manager where they are decoded and executed. A return code containing the command
execution status is generated and returned to the UI following every command. Before the code is
returned, a system scan is performed and any warnings or errors present are appended to the return
code. Every command is executed in this fashion to alert operators of any potential problems as soon as
possible.

![Dispensing software flow diagram](image)

Drink dispensing is a multi-stage process that catches errors before any valves are opened. If an error
occurs at any point in the program, dispensing is terminated and the return message containing all error
data is sent back. Initially, the drink order command is received and the software attempts to decode
d and store the packet. If the packet is successfully validated and stored, the inventory is checked for all of
the ingredients of the current drink order. The system will abort the drink order unless all of the drink
order ingredients are in stock with sufficient volume to pour the drink. Next, the open duration of each
valve is calculated based on the volume specified in the recipe. Once the valve timing has been
calculated and stored, the system sends power to the dispensing control board and controls the valves
using the calculated times relative to the system clock. When the drink has been successfully dispensed, an acknowledge message is generated and returned to the UI.

![Software flow for a drink dispense command](DispFigNum).7

9.6.4 Communication and Functionality

We developed a simple protocol that allows messages to contain a large amount of data in a form that can be processed automatically. Messages sent from the UI to the Raspberry Pi contain 3 pieces of information: message destination, command code, and input parameters. Each system has a unique address that allows the Pi’s main control software to deliver the message. Each task is mapped to a command code and executed each time the code is received. The messages input parameter contains any additional data needed to execute a command, such as the drink order recipe for a dispense command. Each of the fields is separated by an ‘@’ and the data within the field is separated by commas. The message destination and command code fields are prefixed with a ‘$’ to allow those pieces of data to be split off. The dispenser is controlled by the command messages, which are listed in figure (DispFigNum).8

**Command Format**: $(Destination – Dispensing System : DS),$(Command Code),$(Input Parameters)

<table>
<thead>
<tr>
<th>Command Name</th>
<th>Command Code</th>
<th>Input Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Dispenser</td>
<td>$SD</td>
<td>None</td>
</tr>
<tr>
<td>Quit Dispenser</td>
<td>$QD</td>
<td>None</td>
</tr>
<tr>
<td>Drink Order</td>
<td>$DO</td>
<td>Number of Alcohols, Number of Mixers, Recipe</td>
</tr>
<tr>
<td>Enter Cooling Mode</td>
<td>$CM</td>
<td>None</td>
</tr>
<tr>
<td>Change Inventory</td>
<td>$IV</td>
<td>Inventory String</td>
</tr>
<tr>
<td>Calibrate Valve</td>
<td>$CV</td>
<td>Valve Number, Calibration Volume to dispense</td>
</tr>
<tr>
<td>Water Purge Clean</td>
<td>$PG</td>
<td>None</td>
</tr>
<tr>
<td>Open Valve Timed</td>
<td>$OV</td>
<td>Valve Number, Valve open time</td>
</tr>
</tbody>
</table>

(DispFigNum).8: Table of dispensing commands
Return messages use all of the same field separation characters as the incoming commands but are structured differently. Return codes contain five information fields: System, Code type, initial command, return code, and return parameters. The system field is the same address used to receive messages and allows the UI to determine where the message came from. The initial command field contains the command code of the task that the response is addressing. There are three code types, Acknowledge, Not Acknowledge, and Error. Acknowledge codes are sent when a command has been successfully executed, if there are any non-fatal system warnings they are stored in the return parameters field. Not Acknowledge and Error codes contain information about why the command failed to inform the user that the system is not working properly. Some error codes are considered fatal and cause the UI to enter a locked state displaying a message to contact a system administrator. The dispensing system return code documentation can be found in the table below:

Return Code Format: $DS,(Code Type),(Initial Command),@(Return Code)@(Return Parameters)
Return Codes: AK = Acknowledge, ER = Error, NA = Not Acknowledge

<table>
<thead>
<tr>
<th>Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NF</td>
<td>Component of drink order not found in inventory</td>
</tr>
<tr>
<td>IO</td>
<td>Invalid Drink Order</td>
</tr>
<tr>
<td>LV</td>
<td>Volume of component in drink order is too low</td>
</tr>
<tr>
<td>II</td>
<td>Invalid inventory received from server</td>
</tr>
<tr>
<td>CR</td>
<td>Cant receive inventory because there is no connection to server</td>
</tr>
<tr>
<td>CS</td>
<td>Cant store inventory because there is no connection to server</td>
</tr>
<tr>
<td>ICV</td>
<td>Invalid calibration command</td>
</tr>
<tr>
<td>IOV</td>
<td>Invalid open valve timed command</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Not Acknowledge Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA</td>
<td>Command not performed</td>
</tr>
<tr>
<td>IC</td>
<td>Invalid Command</td>
</tr>
<tr>
<td>AA</td>
<td>Dispensing system already active</td>
</tr>
<tr>
<td>AO</td>
<td>Dispensing system already off</td>
</tr>
<tr>
<td>NS</td>
<td>Dispensing system not started yet</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Warnings</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WV</td>
<td>Container(s) are running out warning</td>
</tr>
</tbody>
</table>

(DispFigNum).9: Table of dispensing return messages

9.6.5 Modules
Modules contain data types and functions necessary for their dedicated purposes, which are listed in figure (DispFigNum).10.

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispensing Manager</td>
<td>Receive, decode, and execute commands; Initialize system and modules; Store operational system data</td>
</tr>
<tr>
<td>System Manager</td>
<td>Control and supply power to system components; Protects</td>
</tr>
</tbody>
</table>
components during start up and shutdown; Ensures components only powered when in use;

**Inventory Manager**
Stores data of all ingredients in stock; Monitors liquid levels of each ingredient; Maintains inventory on web server; Verifies drink orders contain stocked ingredients

**Valve Manager**
Calculates valve times to dispense any amount; Controls shift registers to operate valves; Monitors status of all valves during dispense to precisely open or close at calculated times

**Drink Holder**
Temporary data type to store drink order data; Stores similar data as inventory and is used to identify ingredients

**Print Filter**
Handles all print statements and system logs; Uses different print styles to easily differentiate messages; Able to record data to web server for remote debugging if needed

**Warning Manager**
Check system for warnings and errors such as low container volume or dropped internet connection; Scans system after each command and appends results to return code

10 Power
This section discusses the systems power supply, an analysis for total power consumption with relevance to bar and restaurant electronics is included in appendix section 6.5.

10.1 Requirements
The power supply required in this project will be used to power the sensor systems, dispensing system, raspberry pi, and the tablet user interface. These needs can be fulfilled by a variety of pre-existing power supplies on the market, or by building out own power supply. In order to determine the best solution for the SmartBar, the systems that require DC power must be defined:

<table>
<thead>
<tr>
<th>System</th>
<th>Voltage V</th>
<th>Current Draw</th>
<th>Consumption (Watts)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cup Detection</td>
<td>3.3</td>
<td>.25</td>
<td>.825</td>
</tr>
<tr>
<td>BAC Sensor</td>
<td>5</td>
<td>.25</td>
<td>1.25</td>
</tr>
<tr>
<td>Raspberry Pi</td>
<td>5</td>
<td>1.2 (Max)</td>
<td>6</td>
</tr>
<tr>
<td>RCA 10.1” Tablet</td>
<td>5V</td>
<td>1 (Max)</td>
<td>5</td>
</tr>
<tr>
<td>Dispensing (Valves)</td>
<td>12</td>
<td>10 (20 Valves @ .5A)</td>
<td>120</td>
</tr>
<tr>
<td>Dispensing (Water Pump)</td>
<td>12</td>
<td>5.4</td>
<td>64.8</td>
</tr>
</tbody>
</table>

Total Power Consumption: 197.875 Watts

(PowFigNum).1: Table of smartbar component power consumption
10.2 Research

10.2.1 Building a Power Supply

Before a pre-existing power supply can be compared to a custom power supply, one must understand how to build a power supply. Building a power supply is relatively simple in theory: First, the alternating current (120VAC@60Hz) from the wall needs to be dropped down to an appropriate voltage for the power supply. Research indicates that a good transformer voltage output is around 24V because it is high enough to fully supply the regulators discussed further along in this section. Next, the lowered voltage must be converted from AC to DC, which is done through rectification and regulation. A full bridge rectifier causes the voltage to oscillate with only a positive voltage, instead of swinging back and forth from positive to negative.

Once the voltage has been rectified, it must be leveled to a usable voltage that is closer to DC. This filtering is done using a large capacitor, whose value is chosen dependent on the requirements of the regulator that will be used to produce a constant DC voltage output.

The next step is to choose a linear regulator, which takes this fluctuating input and creates a smooth continuous DC output. It is important to choose a capacitor value that will produce a fluctuating voltage that will never drop below the minimum voltage input requirements of the regulator chosen. The layout of the circuit will vary depending on each application, but can be summarized effectively by figure (PowFigNum).2.

![Components of a typical linear power supply](image)

(PowFigNum).2 : Power supply design information

10.3 Buy - Pros and Cons

The cost of each option was researched and is included in appendix section

- **Pros:**
  - Inexpensive
  - Well-Contained
  - (Relatively) Safe
  - Many self-cooling (contain fan) models available
  - Can be turned on/off by the raspberry pi
  - Many have dedicated always-on 5V rail for UI and raspberry pi
• Cons:
  • Designed for computers not SmartBars
  • Will fail eventually and need to be replaced
  • Difficult/Impossible to repair
  • Requires modification to use with SmartBar

10.4 Build – Pros and Cons

• Pros:
  • Good learning experience
  • Easily repaired and maintained
  • Will likely last longer than a purchased power supply
  • Designed exactly for needs

• Cons:
  • Expensive
  • Requires PCB/Perfboarding
  • Needs to be cooled
  • Somewhat dangerous to build (for inexperienced engineers)
  • Requires time to design, build, and debug

10.5 Component Selection

After all of the research, the optimal power supply choice for the SmartBar is the ATX at this time. In our situation, the ATX is a better choice than custom building a power supply because it is less expensive, less dangerous, and requires significantly less time to set up. Although it would be a great learning experience, it seems impractical to build our own power supply since it is more expensive and nobody on the team has built a power supply before. It may be practical to build our own power supply in the future if the SmartBar eventually becomes a product and we have more resources at our disposal. In that case we would be able to get components at wholesale cost, and have boards printed at a much lower cost.

10.6 Circuit Design

The power distribution board has a standard 24-pin Molex connector for the ATX motherboard supply lines, and an 8 pin Molex connector for the ATX EPS power lines. The motherboard connector has a 3.3 and 5V rail that are used to power the sensor board, raspberry pi, and the tablet. The tablet and raspberry pi are connected through a dual USB port, which allows data transmission while the ATX powers the tablet. Each of the EPS power circuits is an isolated power supply for the three dispensing control boards and the water pump. The power rails to the dispensing boards are controlled by relays so power is only supplied when the dispenser is operating. The water pump is also powered through a relay, and an external AC relay is used to power the cooling system.
10.7 Power Consumption Analysis

Power consumption of electronics is commonly measured in Watts (W) which is the rate at which work is done when current flows through an electrical potential difference (voltage). It is calculated as following: Watts (W) = Amperes (A) x Volts (V). The power consumption of devices over time is typically measured in Kilowatt-hours, which is the usage of 1000 watts for one hour.

10.7.1 Dispensing Machine Power consumption

Different kinds of dispensing machines vary in power consumption; These calculations are based on a 12 hour operational day (Smart bar power justification in Power Requirement and Analysis section)

<table>
<thead>
<tr>
<th>Machine Type</th>
<th>Power Consumption</th>
<th>Power Consumption Per Day</th>
<th>Cost Per Day (Based on Santa Cruz average)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Himalay Post-Mix 8 Flavor Fountain Dispensing Machine – No Ice maker</td>
<td>1275 Watts</td>
<td>15.4 kWh</td>
<td>$3.41</td>
</tr>
<tr>
<td>Servend CEV-40 Post-mix 8 Flavor Fountain dispenser – No ice maker</td>
<td>984 Watts</td>
<td>11.808 kWh</td>
<td>$2.62</td>
</tr>
<tr>
<td>Cornelius Vantage Post-mix 8 Flavor Fountain Dispenser – No ice maker</td>
<td>814 Watts</td>
<td>9.785 kWh</td>
<td>$2.17</td>
</tr>
<tr>
<td>SmartBar</td>
<td>1000 Watts</td>
<td>12 kWh</td>
<td>$2.66</td>
</tr>
</tbody>
</table>

10.7.2 Bar Appliance Power Consumption

There are a variety of electronics that can be found in bars, the power consumption of these electronics is calculated according to ANSI standards but many do not include this information in the specification sheets. These power consumption ratings are based on averages and sets of conditions for each different type of electronic device; For example, keg refrigeration units power consumption is calculated based on the average amount of hours of refrigeration per day, average number of times the keg is changed per day, average ambient temperature keg refrigerators are exposed to, etc. It would be difficult to calculate this power rating and since most specification sheets do not include this information I have come up with a simple way to calculate power consumption assuming 12 hours of operation per day (Same calculation used above). This compares all devices to the same standard, although devices will vary in power consumption because they will not be on all of the time. The smart bar will not be drawing the maximum amount of current at all times, so this comparison is fair to make. The power consumptions are displayed as a range from the smallest size appliance to the largest.
### Smart Bar – Final Report

<table>
<thead>
<tr>
<th>Appliance Type</th>
<th>Average Daily Power Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmartBar</td>
<td>12 kWh</td>
</tr>
<tr>
<td>Refrigeration</td>
<td>70 kWh</td>
</tr>
<tr>
<td>Oven</td>
<td>35 kWh</td>
</tr>
<tr>
<td>Fryers</td>
<td>11 kWh</td>
</tr>
<tr>
<td>Grills</td>
<td>37 kWh</td>
</tr>
<tr>
<td>Bain-maries</td>
<td>27 kWh</td>
</tr>
</tbody>
</table>

### 10.7.3 Restaurant Appliance Power Consumption

It may be important to compare the SmartBar’s power consumption to industrial kitchen appliances’ power consumption. I found a study done on restaurants equipment to find the average amount of power consumed by type of kitchen appliance. These can be compared to the SmartBar’s power consumption (based on 12 hour day).

<table>
<thead>
<tr>
<th>Appliance Type</th>
<th>Average Daily Power Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmartBar</td>
<td>12 kWh</td>
</tr>
<tr>
<td>Refrigeration</td>
<td>70 kWh</td>
</tr>
<tr>
<td>Oven</td>
<td>35 kWh</td>
</tr>
<tr>
<td>Fryers</td>
<td>11 kWh</td>
</tr>
<tr>
<td>Grills</td>
<td>37 kWh</td>
</tr>
<tr>
<td>Bain-maries</td>
<td>27 kWh</td>
</tr>
</tbody>
</table>

### 10.7.4 Bar Power Consumption Estimates

I found a commercial energy calculator that approximates the power consumption of different types of commercial businesses, including bars. I did some calculations to get an idea of how much bars spend on electricity per month. To make these situations as energy efficient as possible, I assumed a few things that are unlikely to all be true: New building – better insulation, No air conditioning, No electric water heater, No electric cooking equipment, No windows, No electric heater, no outdoor lighting. This calculator uses local energy costs and weather averages, though the heating/cooling is not factored in for my calculations. The business hours used in these calculations are: 12 hours/day, 6 days/week, 52
weeks/year, totaling 3744 total business hours. To get the best price range, the most energy efficient lights and the least energy efficient lights are used in each calculation.

<table>
<thead>
<tr>
<th>Bar Size (square feet)</th>
<th>Lighting Type</th>
<th>Annual Electricity Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>Incandescent</td>
<td>$5,260</td>
</tr>
<tr>
<td>2000</td>
<td>LED</td>
<td>$2,530</td>
</tr>
<tr>
<td>3000</td>
<td>Incandescent</td>
<td>$7,890</td>
</tr>
<tr>
<td>3000</td>
<td>LED</td>
<td>$3,795</td>
</tr>
<tr>
<td>5000</td>
<td>Incandescent</td>
<td>$13,151</td>
</tr>
<tr>
<td>5000</td>
<td>LED</td>
<td>$6,325</td>
</tr>
<tr>
<td>10000</td>
<td>Incandescent</td>
<td>$26,301</td>
</tr>
<tr>
<td>10000</td>
<td>LED</td>
<td>$12,650</td>
</tr>
</tbody>
</table>

10.7.5 Power Requirement and Analysis
Fountain dispensing machines without ice makers consume roughly between 800-1300 Watts when operational. A maximum power consumption of 1000 Watts would be appropriate for our machine because it is around the same power consumption as other dispensing machines on the market (which do not have a gravity feed component). The maximum power would be reached if every electronic device were to be turned on (All valves open, all solenoids actuated to dispense) at all times, which would never happen. This power budget can be compared with the results above (using the same business hours an energy costs above) to get an idea of the SmartBar’s relative energy consumption:

SmartBar cost to operate all year long at absolute maximum power consumption:

1 kWh * 3744 business hours per year * .2217 Average kWh commercial cost ≈ $830

If the average drink costs $10, the SmartBar would need to dispense at least 100 drinks per year to cover the electricity cost (probably closer to 125 drinks/year with the cost of syrup and liquor); Successful bars will likely sell more than 100 drinks on a weekend night. The cost increase of the SmartBar is relatively small considering that the SmartBar doesn’t need to be tipped or paid a salary. These cost calculations also account for refrigeration, which would not be needed as much with a SmartBar.

<table>
<thead>
<tr>
<th>Bar Size</th>
<th>Lighting Type</th>
<th>Cost w/o SmartBar</th>
<th>Cost w/ SmartBar</th>
<th>Cost Increase (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>Incandescent</td>
<td>$5,260</td>
<td>$6,090</td>
<td>15.8</td>
</tr>
<tr>
<td>2000</td>
<td>LED</td>
<td>$2,530</td>
<td>$3,360</td>
<td>32.8</td>
</tr>
<tr>
<td>3000</td>
<td>Incandescent</td>
<td>$7,890</td>
<td>$8,720</td>
<td>10.5</td>
</tr>
<tr>
<td>3000</td>
<td>LED</td>
<td>$3,795</td>
<td>$4,625</td>
<td>21.8</td>
</tr>
</tbody>
</table>
### Smartbar Component Power Consumption

The components that have been selected for use in the smart bar can be totaled up:

<table>
<thead>
<tr>
<th>Component</th>
<th>Power Consumption</th>
<th>Quantity</th>
<th>Total Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispensing Solenoid Valve</td>
<td>6 Watts</td>
<td>20</td>
<td>120 Watts</td>
</tr>
<tr>
<td>Water Pump</td>
<td>62 Watts</td>
<td>1</td>
<td>62 Watts</td>
</tr>
<tr>
<td>LFMS Cooling System</td>
<td>165 Watts</td>
<td>1</td>
<td>165 Watts</td>
</tr>
<tr>
<td>Cup Detection</td>
<td>.825 Watts</td>
<td>1</td>
<td>.825 Watts</td>
</tr>
<tr>
<td>BAC Sensor</td>
<td>1.25 Watts</td>
<td>1</td>
<td>1.25 Watts</td>
</tr>
<tr>
<td>Raspberry Pi</td>
<td>6 Watts</td>
<td>1</td>
<td>6 Watts</td>
</tr>
<tr>
<td>RCA 10.1” Tablet</td>
<td>5 Watts</td>
<td>1</td>
<td>5 Watts</td>
</tr>
<tr>
<td><strong>Total:</strong> 300.075 Watts</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 10.8 Dispensing Aparatus Concepts

- **Goal:** mix syrup and water to make different flavors of fresh soda / juice

After researching the existing beverage dispensing methods, I wasn't able to find anything that is designed for our particular application. These existing systems could be modified to suit our application but even then they are not the optimal mixing solution. The ideal apparatus would be able to make fresh soda/juice as well as mix in liquor simultaneously so I have brainstormed a few designs that may be able to accomplish this. Each of these designs requires a way to adjust the flow rate in the lines before the mixing apparatus', which can be done using flow regulators or a bar gun manifold (this is not displayed in the diagrams). I have also found a potential solution to the contamination issues that are present in almost all designs; LiquiGlide is a "permanently wet, liquid-impregnated surface" that is food safe and can be coated on any material to almost fully repel any viscous liquid. I am not sure how available this material is to purchase since the company must be contacted directly for any inquires, but it is still worth mentioning (in this section instead of repeatedly in the sections bellow).
10.8.1 Static Mixing Tube Injector
Design a system that injects syrup and alcohol into a (carbonated) water stream and pass it through a static mixing tube instead of having the mixing done by a nozzle.

There are many different possibilities for this system but I have gone into detail on the two configurations that are best suited for the SmartBar. These designs are more ideal than the others (see sections below) because they are self-cleaning and do not require any custom made components to perform mixing.
10.8.1.1 Static Mixing Injector: Single Tube

This design operates by opening the appropriate syrup and alcohol dispense valve(s) at the “same” (syrup valve accounts for delay of water to reach syrup valve, alcohol doesn’t matter) time as the carbonated water line valve or regular water line valve and forcing all of the liquids through the mixing tube. The alcohol valves are before the syrup valves so that water is the only possible liquid that could splash back up into the liquor tubing to avoid contamination. This shouldn’t be much of a problem because theoretically liquid won’t enter any of the sealed lines (provided the sealed line is physically above the line it is feeding into) because the air in the line is trapped and would need to be displaced by the liquid (which won’t go against gravity if there is an alternate path). The valves would need to open at different times according to their position in the line (when dispensing from the valve on the very end of the line - syrup valve would open late and water valve would close early to maintain ratio). There are two potential ways of removing contamination:

- The syrup valve could be opened slightly before the water valve so that the water valve could stay open slightly after the syrup valve is closed in order to flush the system. This approach would not waste any water
- The entire mixing line could be flushed with water to clean any syrup or alcohol remnants that may be stuck to the tubing between mixing line and the valves.
10.8.1.2 Static Mixing Injector: Multiple Tubes

One mixing tube per flavor of soda/juice, this design operates similarly to the single tube arrangement except that each flavor has a three-way valve to distribute the water/carbonated water. One of the three way solenoid valves will be powered at a time switching it to the position that feeds water towards the static mixing tube. The rest of the three way valves allow water to flow along the line when unpowered. This would avoid contamination even more although it requires significantly more valves.

10.8.2 Vortex Funnel
Create a closed funnel-like object with inlets that inject carbonated water, syrup, and liquor to be mixed.
- (Carbonated) water inlets are perpendicular to the bottom of the funnel so that the (carbonated) water enters the nozzle horizontally.

- Above the (carbonated) water inlets are inlets that inject syrup slightly downwards at the side of the funnel, directly into the spiraling water to be mixed.
• Inlets arranged in a circular pattern on top of the nozzle inject liquor directly into the spiraling water to be mixed

Problems:
• Any leftover syrup in the injection lines could drip at any point and contaminate a drink being poured
• Injection angle could potentially splatter soda/juice onto top of funnel assembly, which could drip down

10.8.3 Channeled Block

3D print a block that has input ports for the syrup and water lines that feed into channels, which get ejected into some kind of nozzle to be mixed.

• (Carbonated) water channels could be diffused into multiple exits into the nozzle
• Syrup Channels could exit the block at angles into the nozzle to be mixed with the (carbonated) water
• Syrup channels could also be diffused to exit the block at an angle from multiple locations to potentially improve mixing
• Could include channels for liquor to get mixed in as well

Problems
• Any leftover syrup in the channels could drip at any time causing contamination
• 3d printed material may not be able to withstand the pressure in desired size, block might need to be very large
10.8.4 Pre-Dispense Drink Mixer

Concept: Syrup, (carbonated) water, and alcohol lines all dispensed into mixing container with magnetic stir bar at bottom rotated by magnetic stir plate under container and drain connected to solenoid valve. The bottom of the container would be slightly angled to direct drink towards drain but not enough to cause stirring bar to roll. An external stirring mechanism could be used instead so that bottom of container could be funnel shaped.

- All liquid is dispensed into mixing container simultaneously until all necessary amounts have been poured.
- All contents remain in container while being stirred for appropriate amount of time
- When container contents have been mixed, drain valve opens and drink is dispensed

Problems
- Container would need to be cleaned after every drink is dispensed
- Any debris that ends up in the container will clog up the solenoid valve
10.8.5 Master Nozzle/ Mixer

Concept: All the lines coming from both the liquor gravity feed system, and soda system, which includes a total of 20 liquid lines would route to a single apparatus. The nozzles would point toward the center of the apparatus so that the liquids would flow into a vortex centerpiece, dispersing the liquids throughout the apparatus evenly. The Water lines (both carbonated and non-carbonated) would be attached toward the top so that the nozzle can be flushed with water from the top.

Pros:

- Simple enough nozzle could be 3D printed.
- 3D Printing material is only cost.
- Allows for cleaning/flushing of master nozzle.
- Allows for mixing of both liquor and soda in same area as opposed to bar gun only mixing soda/syrup.

Cons:

- Routing Lines together may be difficult.
- Mixing ratios for Soda/syrup will need to be calibrated by trial and error
10.8.6 Venturi Manifold

Concept: The syrup lines feed into the manifold via barbed fittings and then get injected into passing carbonated or non-carbonated water. The two then go into a nozzle that swirls the mixture to further incorporate them. A secondary nozzle allows the liquor to pass through unaffected by the soda.

Pros:

- Simple enough nozzle/manifold could be 3D printed.
- 3D Printing material is only cost.
- Cleaning automatically done by passing liquid
- Would mix soda thoroughly

Cons:
- May be difficult to model/print

10.8.7 Fountain Dispense Valve

Normal Operation Overview:
- Two valves make up system, one for syrup one for water / carbonated water
  - Each valve has a regulator to adjust flow rate
  - Flow rates are adjusted for correct water to syrup ratio (usually 5:1)
  - Incoming water and syrup lines are always under pressure
- Push lever or button to open valves and dispense
  - Both valves open simultaneously, diffusing the syrup which is then forced through the baffle where it gets mixed with the water before the mixture is dispensed

Pros
- Controlled by an electrical signal (mechanical actuation required though)
- Each flavor has its own nozzle - no cross contamination
- Able to adjust input flow rates to achieve correct ratio
Portion control valves dispense select volumes when button is pressed
  - Con: Would require more complex modifications to be controlled electronically

**Cons**

- Any liquid in the line between the cooling system and valve will become warm and lose carbonation
- Need to be modified to be electronically controlled
  - Requires circuitry changes (normally controlled by mechanical switch)
- Need to add a tube to transfer dispensed soda to cup
  - Not designed to be attached to a tube
    - nozzle end would need to be modified to attach a tube
      - nozzle is required for mixing - modifications may not be possible
      - nozzle end is very large - would not fit small tubing, adapters would be required
  - Forcing the soda through tubing and size reductions would agitate the mixture and likely lose some carbonation
  - Any soda stuck to the sides of these tubs will drip randomly
    - Flushing the line with water is not possible
- Would need 8 individual valves
  - Limited to carbonated or non-carbonated for each flavor, not both
- Large - 3” width x 4.5” height x 3.5” depth
- Designed to be located above cup
  - Gravity dispensing system located above cup, no room for large valves
  - Operation behavior unknown if used to pump upwards as is desired in our application
- Any liquid in the line between the cooling system and valve will become warm and lose carbonation (as with any dispensing system currently available)
- Expensive - see available options below

**Available Options**

- Cornelius
  - UF-1 Lever Valve - $143.00
  - UF-1 Push Button Valve - $153.00
• UF-1 Portion Control Valve - $207.00

• Lancer
  • 100 Sanitary Lever Valve - $135.00
  • 100 Lever Valve - $130.00 (not sanitary?)
  • 100 Push Button Valve - $140.00

• Flomatic
  • 424 Lever Valve - $98.00
  • 424 Push Button Valve - $89.00
  • 424 Portion Control Valve - $124.00

10.8.8  Bar Gun

Normal Operation Overview:

• All lines (syrup, water, and carbonated water - always under pressure) are connected to manifold, which connects to bar gun
  • Manifold allows for flow rates to be adjusted for desired ratio

• Buttons are pressed on the bar gun to physically open correct valves
  • Metal plates below the buttons determine if water or carbonated water valve gets opened (Wunderbar & Schroeder have same plate style, Cornelius has different layout but same concept)
    • Two water valves and carbonated water valves on bar gun, one per side
    • When button is pressed, corresponding syrup valve is opened and metal plate opens either carbonated water or regular water valve depending on plate configuration
    • Metal plate has small holes so that only the syrup line corresponding to the button press gets opened
• The (carbonated) water is diffused and forced out of the tiny holes around the edge of the nozzle
• Syrup is injected into nozzle at an angle to create a vortex that mixes the two

Pros
• Compact design - all lines compressed to one small location
• Designed to dispense all beverages out of one nozzle
• Nozzle can be flushed with water between dispenses if cleaning is necessary
• Able to tune flow rates of incoming beverage lines for perfect mixing ratio
• Has flow regulators which keep output line pressure constant even if input pressure fluctuates
  • Operate by a piston connected to a spring which expands the port if the inlet pressure drops and contracts if pressure increases
• Can be controlled electronically with or without being modified - see Control with/without Modification
  • Modification allows syrup to be mixed with either carbonated or non-carbonated water (or a mixture)
  • Modification allows for new flavors, multiple flavors can be injected at once (would need to be pulsed to maintain ratio)

Cons
• Patented
  • Must buy from a manufacturer
• Not specifically designed for automatic dispensing
  • Designed for human physical force on button to open valve
• Additional components and/or modifications required to dispense automatically
• Paying for features that aren't being used
• Dispensing out of single nozzle makes contamination possible (although gun is designed to dispense this way)
  • Associated Pro: Small amount of water can be dispensed between drinks to clean nozzle
• Any liquid in the line between the cooling system and valve will become warm and lose carbonation (as with any dispensing system currently available)

Available Options
• Wunderbar Post-Mix 10-Button M4 Bar gun - $260
• Wunderbar Post-Mix 10 Button Standard Bar Gun - $230
• Schroder Post-Mix 10 Button Bar Gun - $330
• Cornelius Post-Mix 10 Button Bar Gun ~ $75 (No online dealer found, available on ebay)

10.9 Dispensing Method Analysis

There are several options that were researched in detail because each meets the requirements and was a suitable option. Additional information was defined before any decisions were made to help ensure the best components were selected.

• Goal: Dispense all different kinds of liquor and ingredients very accurately

• Universal Challenges
  • Different alcohol types and concentrates have different viscosities
  • Different alcohols and concentrates have slightly different densities

• Options
  • Pump system - Use pumps to dispense liquor
  • Gravity feed system - Dispense liquor from elevated containers (using some kind of electronic valve system)
  • Pressurized system - Use containers under constant pressure to dispense liquor
10.9.1 Pump Overview
There are two types of pumps, peristaltic and diaphragm pumps, which are the most suitable for dispensing liquor. The pros and cons of each type of pump are discussed in later sections.

- **General Pump System Pros**
  - Containers can be located anywhere (within the pumps priming range)
  - Flow rate not dependent on amount of liquid in containers
  - Dispensing location can be located anywhere (within pump range)
  - Able to dispense liquid straight from original container (would need some kind of lid that can vent though)
  - No algorithms required to determine dispensing time

- **General Pump System Cons**
  - Flow rate consistency depends on type of pump
  - Pumps require maintenance and contain many moving parts which can fail
  - Difficult to achieve constant flow rate – pumps generally not designed to accurately dispense small amounts of liquid

10.9.2 Diaphragm Pump
- Operation (General concept - there are a few different types of diaphragm pump)
  - Chamber with entry and exit ports with a nonpermeable membrane as part of the chamber wall
  - Phase 1: Entry port is opened, exit port is closed, and membrane is flexed outward causing low pressure (lower than external pressure) inside chamber to draw liquid in
  - Phase 2: Entry port is closed, exit port is opened, and membrane is flexed inward causing high pressure (higher than external pressure) inside chamber to force liquid out
10.9.2.1 Diaphragm Pump: Pros

- Self-priming
- Able to handle viscous liquids
- Can be run dry (no liquid being pumped)
- Some pumps are designed for beverage dispensing (although not for spirits)
- Gas powered pumps available - system already has CO2 supply
  - Electric diaphragm pumps available also
- The syrup pump options for the pneumatic system are also options here
  - We will be able to test out liquor dispensing with whatever pump we choose
- Some electric pumps are submergible - could be built into liquor containers

10.9.2.2 Diaphragm Pump: Cons

- Must be food grade material - always in contact with liquid
  - Eliminates many options
- Existing beverage dispensing pumps not designed to dispense spirits
• high flow rates (see options section – these ) make accurate dispensing more difficult – variations in flow rate have significant effect
• designed to dispense beer, wine, or concentrated syrups
• Everything constantly under pressure
  • Requires high pressure tubing for dispense lines and gas lines
  • Requires a valve (that can support higher pressure) at end of line to control when liquid is dispensed
  • Possibility of leaks at any time, CO2 leaks dangerous
• Most pumps designed for high flow rate applications
• Beverage dispensing pumps are somewhat Large ~ 6" Width x 6" Height x 3" Depth
• Gas powered pumps all need to be located in same general area

10.9.2.3 Diaphragm Pump: Potential Options (Food safe with low enough flow rate)

• Seaflo SFDP1 - $19.99 – Flow Rate 71.7 ml/sec (Electric)
• Flojet T5000 - $42.00 - Flow Rate 118.3 ml/sec (Gas)
• Flojet N5000 - $34.75 - Flow Rate 118.3 ml/sec (Gas)
• Flojet G55 - $52.00 - Flow Rate: 207 ml/sec (Gas)
• Shurflo Beverage Pump - $42.00 - 100.6 ml/sec (Gas)

Note: A standard 1.5 oz shot is equivalent to 44.36ml

10.9.3 Peristaltic Pump
Operation - "pushes" liquid by rotating rollers which moves a compressed section of the line
10.9.3.1 Peristaltic Pump: Pros

- Designed to dispense specific amounts of liquid
- Pump does not come in contact with liquid
  - Associated Con: many inexpensive pumps have tubing built into the head and are not food grade, designed for pools, aquariums, chemical mixing, etc
- Self-priming
- High suction force - able to pump large vertical distance
- Able to handle high viscosity fluids
- Prevents backflow
- Low maintenance required
  - Design has no valves or seals
  - No cleaning necessary - no contact with liquid
- Can be run dry (no liquid being pumped)
- Used in many bartending robotics projects

10.9.3.2 Peristaltic Pump: Cons

- Not designed for beverage dispensing (Except for Bartendro pump - see options)
- Expensive for pumps with sufficient flow rate (see options)
- Designed for low flow rate - designed to dispense given amount of liquid over long period of time (medicine dosing)
- Tubing must be replaced periodically - gets worn out by pump mechanism
- Flow is pulsed, not good for applications where constant flow is required (our application)

10.9.3.3 Peristaltic Pump: Potential Options

- Bartendro Peristaltic Pump & Controller - $119.95 - 700ml/min
  - Designed to dispense liquor
- Simply Pumps PM200F - $115.88 - Max Flow: 320ml/min
• Simply Pumps PM300F - $144.61 - Max Flow: 946ml/min
• Welco WP1000 - $62.00 - Max Flow: 350ml/min
• Welco WP1100 - Max Flow: 670ml/min

10.9.4 Gravity Feed

Liquid can be dispensed from elevated containers electronically using solenoid valves. The time to dispense a given quantity can be calculated:

\[ t = \frac{4\sqrt{h_0}}{\sqrt{16h_0 - \frac{16C_d A_a 2\sqrt{2g}}{wl} \Delta V}} - \frac{2C_d A_a \sqrt{2g}}{wl} \]

- \( w \): Container base width
- \( l \): Container base width
- \( g \): Acceleration due to gravity (9.81m/s\(^2\))
- \( A_a \): Aperture area (diameter of connection before solenoid valve)
- \( C_d \): Coefficient of discharge (solenoid valve constriction coefficient)
- \( \Delta V \): Volume to be dispensed (in mL)
- \( h_0 \): Liquid height above valve (container liquid height + dispense line height)

10.9.4.1 Gravity Feed Dispensing: Pros

• Flow rate can be as slow as needed
  • The slower the flow rate, the more negligible random flow disturbances become
  • Solenoid valve orifice diameter dictates
• Flow rate is stable and predictable
  • No fluctuations in dispensing pressure
• Gravity is free - won't fail, never needs to be replaced
• Gravity is very consistent - won't ever change (unless you change elevation - which can be accounted for)
- Electronically controlled without modification - solenoid valves
- Low maintenance - only moving part is solenoid valve
  - Can be easily removed for cleaning or replacement
- System is self-priming (dispense line before valve fills automatically)

### 10.9.4.2 Gravity Feed Dispensing: Cons

- Containers must be placed above dispensing point
  - Associated Pro: There is a lot of free space up higher on the SmartBar
- Volume of liquid in container must be known to predict flow rate
- Valves must be below containers
- Most solenoid valves are not designed for beverage dispensing applications
- Flow rate is not constant
  - Flow rate is non-linearly related to height of liquid above dispense point
    - Associated pro: Can be predicted accurately by an algorithm
- Containers need a vent for airflow for proper dispensing
  - pressure inside the container must be equal to pressure outside of the container
  - need a way to open and close vent so vent is only open when dispensing to keep contents as fresh as possible

### 10.9.4.3 Gravity Feed Dispensing: Potential Valves

- Deltrol Controls DSVP11 - $12.50 - Orifice Diameter 8mm
- EchoTech DDT-CD-12VDC - $13.11 - Orifice Diameter 8.5mm
- YongChuang YCWS10-02 - $2.00 - Orifice Diameter 10mm
- YongChuang YCWS1 - $1.90 - Orifice Diameter 2.5mm

### 10.9.5 Pressurized System

Liquid can be dispensed from a container under pressure:
10.9.5.1 **Pressurized System: Pros**

- No pumps required
- Constant flow rate out of container if constant pressure is maintained
- One single pressure source can supply pressure to many containers
- Space efficient containers could be designed to maximize amount of liquid in given volume
- Containers can be located anywhere (provided great enough pressure to move liquid to dispensing location)
- Silent
10.9.5.2 Pressurized System: Cons

- Not enough resources to design safe containers for this system
- Must cut off pressure lines and depressurize container to refill/replace
- Relatively unsafe (compared to pumps and gravity feed)
  - Forgetting to cut off pressure supply before opening container could turn container cap into a projectile
  - High system pressure - any leak would cause gas or liquid to be ejected rapidly, could be fatal if using CO2 in room that is not well ventilated
- Specialty containers must be used for safety
  - Most pressurized containers are not designed for beverage dispensing (beer kegs are the exception)
  - Pressurized containers are round for safety reasons - not space efficient at all
  - Side Note: 2 Liter soda bottles are designed to withstand pressure safely
    - Not designed to be re-used nor to be used for anything other than containing soda
    - Not rated for any particular pressure
- Pressure needs to remain constant to maintain constant flow rate
- Everything must be safe for contact with food
  - All gas line components must be selected carefully, for example CO2 in contact with copper (or any alloy containing copper) produces toxic compounds.
11 Final Product

The Smartbar V1.0 Kiosk
12 References


13 Appendix A Circuits

13.1 Dispensing Breakout
13.2 Communication Breakout

13.3 Pi Breakout Board
13.4 Power Distribution Board
13.5 Sensor Breakout Board
14 Appendix B PCBs

14.1 Sensor Breakout

14.2 Raspberry Pi Breakout Board
14.3 Power Distribution

14.4 Dispensing Daughter Board
14.5 Signal Distribution Board
## 15 Appendix C

<table>
<thead>
<tr>
<th>Raspberry Pi Breakout</th>
<th>Part Reference</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>1N5818</td>
<td></td>
</tr>
<tr>
<td>J1</td>
<td>HEADER 20x2/M</td>
<td></td>
</tr>
<tr>
<td>J2</td>
<td>HEADER 4</td>
<td></td>
</tr>
<tr>
<td>J3</td>
<td>Header 2x10M</td>
<td></td>
</tr>
<tr>
<td>J4</td>
<td>CONN SOCKET 10</td>
<td></td>
</tr>
<tr>
<td>LED1</td>
<td>Tricolor LED</td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td>330</td>
<td></td>
</tr>
<tr>
<td>U2</td>
<td>RJ45</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sensor Breakout Board</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>0.1uF</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>0.1uF</td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td>1N5818</td>
<td></td>
</tr>
<tr>
<td>J1</td>
<td>RJ45</td>
<td></td>
</tr>
<tr>
<td>J2</td>
<td>12 HEADER</td>
<td></td>
</tr>
<tr>
<td>J4</td>
<td>JUMPERx3</td>
<td></td>
</tr>
<tr>
<td>J5</td>
<td>Molex Two Pin</td>
<td></td>
</tr>
<tr>
<td>J6</td>
<td>JUMPERx2</td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>R5</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>R7</td>
<td>47000</td>
<td></td>
</tr>
<tr>
<td>R8</td>
<td>22000</td>
<td></td>
</tr>
<tr>
<td>R9</td>
<td>330</td>
<td></td>
</tr>
<tr>
<td>U1</td>
<td>SPI ADC</td>
<td></td>
</tr>
<tr>
<td>U3</td>
<td>MCP6004</td>
<td></td>
</tr>
<tr>
<td>U4</td>
<td>MCP6002</td>
<td></td>
</tr>
<tr>
<td>Power Distribution Board</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>10A</td>
<td></td>
</tr>
<tr>
<td>J11</td>
<td>JSM1-12V-5</td>
<td></td>
</tr>
<tr>
<td>J12</td>
<td>ATX 24 Pin Connector</td>
<td></td>
</tr>
<tr>
<td>J13</td>
<td>JSM1-12V-5</td>
<td></td>
</tr>
<tr>
<td>J15</td>
<td>Dual_USB</td>
<td></td>
</tr>
<tr>
<td>J16</td>
<td>Dual_USB</td>
<td></td>
</tr>
<tr>
<td>J17</td>
<td>ATX EPS12V Connector</td>
<td></td>
</tr>
<tr>
<td>J18</td>
<td>JSM1-12V-5</td>
<td></td>
</tr>
<tr>
<td>J19</td>
<td>JSM1-12V-5</td>
<td></td>
</tr>
<tr>
<td>PC2</td>
<td>2Pin</td>
<td></td>
</tr>
<tr>
<td>Q1</td>
<td>2N2222A</td>
<td></td>
</tr>
<tr>
<td>Q2</td>
<td>2N2222A</td>
<td></td>
</tr>
<tr>
<td>Q3</td>
<td>2N2222A</td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>470</td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td>470</td>
<td></td>
</tr>
<tr>
<td>R5</td>
<td>33k</td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td>22k</td>
<td></td>
</tr>
<tr>
<td>R7</td>
<td>33k</td>
<td></td>
</tr>
<tr>
<td>Signal</td>
<td>RJ45</td>
<td></td>
</tr>
<tr>
<td>U1</td>
<td>MC3458/50</td>
<td></td>
</tr>
<tr>
<td>U1</td>
<td>MC3458/50</td>
<td></td>
</tr>
<tr>
<td>Dispenser1_Out</td>
<td>WM1861</td>
<td></td>
</tr>
<tr>
<td>Dispenser2_Out</td>
<td>WM1862</td>
<td></td>
</tr>
<tr>
<td>Dispenser3_Out</td>
<td>WM1863</td>
<td></td>
</tr>
<tr>
<td>Sensor Power Connector</td>
<td>WM1864</td>
<td></td>
</tr>
<tr>
<td>WP_Connector</td>
<td>WM3901</td>
<td></td>
</tr>
<tr>
<td>Dispensing System</td>
<td>Reference</td>
<td>Value</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------</td>
<td>-------</td>
</tr>
<tr>
<td>C1</td>
<td>1uF</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>100uF</td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td>800mA</td>
<td></td>
</tr>
<tr>
<td>F10</td>
<td>800mA</td>
<td></td>
</tr>
<tr>
<td>F11</td>
<td>800mA</td>
<td></td>
</tr>
<tr>
<td>F12</td>
<td>800mA</td>
<td></td>
</tr>
<tr>
<td>F13</td>
<td>800mA</td>
<td></td>
</tr>
<tr>
<td>F14</td>
<td>800mA</td>
<td></td>
</tr>
<tr>
<td>F15</td>
<td>800mA</td>
<td></td>
</tr>
<tr>
<td>F16</td>
<td>800mA</td>
<td></td>
</tr>
<tr>
<td>J1</td>
<td>3Pin Header</td>
<td></td>
</tr>
<tr>
<td>J2</td>
<td>2Pin Header</td>
<td></td>
</tr>
<tr>
<td>J3</td>
<td>2Pin Header</td>
<td></td>
</tr>
<tr>
<td>J4</td>
<td>2Pin Header</td>
<td></td>
</tr>
<tr>
<td>J5</td>
<td>2Pin Header</td>
<td></td>
</tr>
<tr>
<td>J6</td>
<td>2Pin Header</td>
<td></td>
</tr>
<tr>
<td>J7</td>
<td>2Pin Header</td>
<td></td>
</tr>
<tr>
<td>J8</td>
<td>2Pin Header</td>
<td></td>
</tr>
<tr>
<td>J9</td>
<td>2Pin Header</td>
<td></td>
</tr>
<tr>
<td>Q1</td>
<td>PN2222A</td>
<td></td>
</tr>
<tr>
<td>Q2</td>
<td>TIP14C</td>
<td></td>
</tr>
<tr>
<td>Q3</td>
<td>PN2222A</td>
<td></td>
</tr>
<tr>
<td>Q4</td>
<td>TIP14C</td>
<td></td>
</tr>
<tr>
<td>Q5</td>
<td>PN2222A</td>
<td></td>
</tr>
<tr>
<td>Q6</td>
<td>TIP14C</td>
<td></td>
</tr>
<tr>
<td>Q7</td>
<td>PN2222A</td>
<td></td>
</tr>
<tr>
<td>Q8</td>
<td>TIP14C</td>
<td></td>
</tr>
<tr>
<td>Q9</td>
<td>PN2222A</td>
<td></td>
</tr>
<tr>
<td>Q10</td>
<td>TIP14C</td>
<td></td>
</tr>
<tr>
<td>Q11</td>
<td>PN2222A</td>
<td></td>
</tr>
<tr>
<td>Q12</td>
<td>TIP14C</td>
<td></td>
</tr>
<tr>
<td>Q13</td>
<td>PN2222A</td>
<td></td>
</tr>
<tr>
<td>Q14</td>
<td>TIP14C</td>
<td></td>
</tr>
<tr>
<td>Q15</td>
<td>PN2222A</td>
<td></td>
</tr>
<tr>
<td>Q16</td>
<td>TIP14C</td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td>1k</td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>1k</td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td>1k</td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td>1k</td>
<td></td>
</tr>
<tr>
<td>R5</td>
<td>1k</td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td>1k</td>
<td></td>
</tr>
<tr>
<td>R7</td>
<td>1k</td>
<td></td>
</tr>
<tr>
<td>R8</td>
<td>2.7k</td>
<td></td>
</tr>
<tr>
<td>R9</td>
<td>2.7k</td>
<td></td>
</tr>
<tr>
<td>R10</td>
<td>2.7k</td>
<td></td>
</tr>
<tr>
<td>R11</td>
<td>2.7k</td>
<td></td>
</tr>
<tr>
<td>R12</td>
<td>1k</td>
<td></td>
</tr>
<tr>
<td>R13</td>
<td>1k</td>
<td></td>
</tr>
<tr>
<td>R14</td>
<td>1k</td>
<td></td>
</tr>
<tr>
<td>R15</td>
<td>1k</td>
<td></td>
</tr>
<tr>
<td>R16</td>
<td>2.7k</td>
<td></td>
</tr>
<tr>
<td>R17</td>
<td>2.7k</td>
<td></td>
</tr>
<tr>
<td>R18</td>
<td>2.7k</td>
<td></td>
</tr>
<tr>
<td>R19</td>
<td>2.7k</td>
<td></td>
</tr>
<tr>
<td>U1</td>
<td>RJ-45</td>
<td></td>
</tr>
<tr>
<td>U2</td>
<td>SN74HC555N</td>
<td></td>
</tr>
</tbody>
</table>
16 Appendix D

16.1 Main Activity/ Console

![Console Log]

16.2 Container/Inventory Screen

![Inventory Screen]
16.3 Idle Menu

16.4 Recognize Fingerprint

Please hold your finger on the scanner.
16.5 New User

Please Enter Phone Number to Register Fingerprint

1-(555)-867-5309

16.6 Fingerprint Enrollment

Please hold your finger on the scanner.
16.7 Check BAC

**PLEASE BREATHE INTO THE SENSOR ON YOUR RIGHT**

Disclaimer: Recently drinking may result in a higher BAC reading.

16.8 Confirm Drink

**Your Order**

Pour Drink

- Vesper
- Drink Not Found
- Spirit:
- Vodka: 0.5oz
- Gin: 2.0oz
- Kina Lillet: .25oz

**$2.00**
16.9 User Sign In

![User Sign In Screen]

Welcome Back!

User Login

tylerjriver@gmail.com

Password

......

Done

(!)