UNIVERSITY OF NOTRE DAME-ELECTRICAL ENGINEERING

Final Report

Team AutoBev

Elizabeth Clark, Lorena Garcia, Alex Macomber, Mark Pomerenke

5/9/2011

Contents

1 Introduction:

1.1 Problem Description:

One of the biggest challenges to both customers and owners of drinking establishments is that of lengthy wait times associated with the fulfillment of an order. In such an environment, profit for the owner, and happiness of the customer, is directly dependent upon efficiency of product deliveries. At many drinking establishments, customers waste many minutes each night waiting to be noticed by a bartender. Once noticed, the customer must then explain his or her order to the server. After making the order, the server then charges the customer. During this process, there is no guarantee that the server will hear the customer's order correctly, allowing for the possibility of inaccurate order fulfillment. Furthermore, if the customer chooses to pay with a credit card, the procedure is made even longer, due to the fact that the credit card verification process is very time-consuming.

With the technology available in our current society, it is apparent that waiting times can and should be reduced. The crucial minutes wasted while the customer explains their order to the server and while the server charges the customer, coupled with the possibility of an inaccurate order, can be eliminated through the utilization of current technology. By implementing a computer system that provides a means to order, pay for, and pour a beverage, without dependence on a restaurant employee, the time between request and delivery of a final product can be greatly reduced and streamlined. Additionally, through the elimination of verbal ordering, the accuracy of orders can be increased. Using a digital ordering process would also enable establishments to serve customers in the order in which they requested a drink, instead of in a random fashion.

It is reasonable to say that an establishment can serve more drinks by reducing wait times. This would inevitably lead to more revenue. Furthermore, by serving customers in order, instead of in a random fashion, customer satisfaction would also increase. Through the automation of the order and payment processes, the workload of employees can be decreased while increasing order fulfillment accuracy.

1.2 System Requirement:

Tables 1.1, 1.2, and 1.3 below show the requirements for the overall system, for the general subsystems, and for possible future enhancements as established at the time of the project proposal.

1.3 Overall System:

Overall System Requirements	
General	Must be capable of receiving and processing drink orders digitally Must use simple user commands for navigation (single click) Must have layer of protection between CPU and other sensitive electronics and users/ beverages
Size	Must fit onto a floor area no larger than 2' x 2' (not including any keg)
Power	Must be powered by wall plug in 120V AC
Compatibility	Must be able to connect to standard keg Must be expandable to include more extensive drink orders Must be in English

	Must be in compliance with current health standards

Table 1 Overall System Requirements

1.4 Subsystem and Interface Requirements:

Magnetic Card Reader	
General	Must be able to scan credit cards and send information to the PC
Size	Must not be bigger than 6”x6”
Power	Must be powered through USB interface
PC Software	Must be able to interface to a PC through USB Must be able to emulate a USB Human Interface Device (HID) keyboard Must turn on and off with PC
User Interface	
General	Must be implemented on a standard monitor
Power	Screen must be powered by 120 V AC
Compatibility	Must work on Windows PC Will Be Programmed Using Microsoft Visual Studio Will Program in C# Visual
Bartender Interface	
PC Software	Must be capable of running on a low performance windows machine Must list drink items in order in a FIFO queue Must assign item number to new drink items for claims purpose (will not need to issue ticket) Must be capable of adding orders to the queue from remote client (customer interface) Must be capable of deleting queue items from local user Must be able to handle bi-directional communication via Local Area Ethernet Must be compatible with UI program Bartender Server must be multi-threaded for expandability
Microcontroller	
General	Must handle at least 8 digital inputs * Must handle at least 4 analog inputs* Must supply at least 4 digital outputs * Must indicate the state of digital outputs with LED’s Must be programmable with C software Must have a universal asynchronous receiver transmitter (UART) Must be able to communicate serially with CPU over USB connection Must have non-volatile memory Must be capable of 5V operation. Must have On/Off
Power	Powered by 12 V supply from wall converter
Compatibility	Will receive signal from computer Will receive signal from flow sensor
Beverage Dispensing and Sensing	

General	Must have keg to pour drink from with appropriate keg components (spigot, keg adapter, pressure valve, tubing, and gas tank). Must have a solenoid valve to only allow beverage to pour through if already paid for. Must have a flow sensor to keep track of how much beverage has been poured. Must have Cup Senor to detect if cup is present Must have emergency stop
Power	Must be powered via plug into a 120 VAC outlet Must create intermediate voltage power supplies of 12 V and 5V regulated
Valve Permission Solenoids	Solenoids must receive 12 V across the coil to activate Must be able to control 12 V solenoid signal with a 5 V output Amplifier FET must be able to handle 0.53 A, and a max Vdd of at least 20V Must connect to 0.375" OD, .25" ID tubing
Flow Sensing	Requires specified resistive network for proper signal output Requires 5-24V power supply Must connect to 0.375" OD, .25" ID tubing
Cup Sensor	Must detect if cup is present underneath spigot Must toggle input to microcontroller from 5 to 0 V
Emergency Stop	Must allow user to stop beverage flow if anything goes wrong Should not be software actuated
Pressure System	Must create a regulated pressure of 12 psi Must attach to keg input by coupling to connector
Microcontroller Software	Must use less than 8K of program memory Must be able to communicate with solenoid(open/close it). Must be able to get input from flow sensor via interrupt Must be able to communicate with User Interface via USB Must operate with a baud rate of 9600
PC to Microcontroller USB Interface	
General	Must have USB interface User Interface PC must be able to communicate with microcontroller by sending bytes Microcontroller must be able to communicate with User Interface PC by sending bytes
Bartender/User PC UDP Interface	
General	Must connect using network UDP protocol Must operate on hard line and wireless connections to the network Bartender must be initialized before the User can access system User connections must receive updated bartender data upon request Must be send user data packets to the server with high reception certainty

Table 2 Subsystem and Interface Requirements

1.5 Future Enhancement Requirements

Future Enhancement Requirements	
Online Database	Must allow users to sign up to access online database that keeps track of past drink orders Must show information such as time, quantity, order, and expense
Charge Credit Cards	Must be able to complete charging process Upgrade from prototype that simply stores credit card information in a local database without charging the account
Serve Mixed Drinks	Must be able to properly make and serve "mixed drinks" in an automated manner
Cognitive Testing	Must be able to determine if customer is in suitable state for another drink
Interconnect Multiple Kiosks	Must allow for multiple dispensing units to be interacting with a centralized bartender interface within a single establishment

Table 3 Future Enhancement Requirements

Table 1 shows the overall system requirements necessary to solve the problem. The requirements in this table were all completed. Drink orders can currently be received and processed digitally through the use of a simple user interface. All sensitive electronics have been placed in encasings that provide protection from both users, and beverages. The unit is reasonably small enough to fit in a normal drinking establishment and can be powered through a normal wall outlet. It takes as an input, a standard keg, allows for the bartender to automatically update the drink list for the customers to view, and is implemented in English and in compliance with current health standards.

Table 2 shows the subsystem and interface requirements necessary to solve the problem.

All requirements for the magnetic card reader subsystem were completed. A USB connection between reader and computer allows for the device to be powered, and also for the credit card track information to be conveyed to the user interface.

The requirements for the user interface subsystem were also met. This interface was programmed using Microsoft Visual Studio. It is implemented in Visual C#, using a Windows Form application. All forms within the application are sized to be readable but a user on a standard PC monitor powered by a wall socket. One additional requirement that was added to the user interface was the ability to allow users to create a username and password that would be tied to their account such that they would be able to use the Android Smartphone app. For simplicity, it was decided that the username must be all letters, and the password must be a four digit pin. Thus, the user interface had to be able to verify these two traits. Another additional requirement that was encountered was the ability of the user interface to receive updated drink lists and prices from the bartender. This requirement was successfully completed.

The initial requirements for the bartender interface were completed. Additional requirements were added to this subsystem throughout the design completion process. During this process, it was discovered that the bartender interface must be able to store and extract data from a database. A local database was created using Miscrosoft SQL Server. At the end of every night, it is possible for the bartender to save the data information to a text file in html format. The bartender can also choose to

upload and display data that has been saved from previous nights. Another requirement that was added to the bartender interface was that of the ability to receive drink orders from a Android Smartphone, along with the ability to send a message back to the Android upon receiving an order, and the ability to send a text message to the phone when the drink is ready. This requirement was fulfilled using UDP communication. The bartender interface is able to extract the username and password from its local database. It then must be verified that the username does exist, and that the password is correct for that username. If not, the user must receive a respond from the bartender interface (via text messaging) noting that the either the username did not exist, or that the password was incorrect. If both are correct, then the bartender must issue a UDP response to the Android from which the order was placed telling the user what number order they are. The bartender also needed to have the ability to check to see if a person (when trying to set up a username at an AutoBev kiosk) already had a username. This required conducting a query of the local database. This requirement was completed.

The microcontroller subsystem requirements changed very slightly. Instead of being capable of 5 volt operation, the requirement was changed to having the microcontroller be capable of five volt operation.

The beverage dispensing and sensing subsystem requirements were also met. The unit will only allow for a beverage to be dispensed given that it has been paid for, and that there is a cup present. The amount of volume that has been poured is constantly polled and reported back to the user interface. The power requirement of our system changed to being a regulated power of five volts. This was implemented through the use of a voltage regulator. Thus, the solenoid valve is now controlled with a five volt signal from an output pin on the microcontroller. The cup sensor was also changed from a light sensor to a limit switch. This switch had to be sensitive enough to allow for trigger due to a plastic cup, but strong enough to not break easily due to a careless user.

The PC to microcontroller USB interface subsystem requirements did not change and were successfully implemented.

1.6 High Level Description:

The AutoBev System consists of a main user interface, a bartender interface, a beverage dispensing and sensing unit, a localized database, and a Android Smartphone application. The user and bartender interfaces were programmed using Microsoft Visual Studio.

The user interface enables a customer to begin a session by swiping a credit card. After this occurs, the user interface verifies that the track read by the magnetic stripe reader corresponds to a credit card. Once this is done, the user is able to chose if they would like to order a specialty drink (to be made by the bartender), if they would like to pour their own beverage, or if they would like to set up a username and password to be used with a Android Smartphone application.

If the user chooses the option to order a specialty drink, they are presented with a list of drinks and prices from which they may order. The user interface checks for an updated list every time a person begins a session. Once an order has been placed, it is sent to the bartender interface to be added to a drink list queue. The bartender interface sends a message back to the user indicating what drink order number they are, along with how many drink orders are in front of them. The bartender interface appropriately updates the total to be charged to that customer within the local database.

If the user chooses the option to pour their own beverage, they are then presented with the option to select a predetermined size (either twelve, sixteen, or sixty ounces) or to pay per ounce. Updates in price per ounce are checked for every new user. The user must then place their cup against a limit switch and hit a green start button on the user interface to begin pouring. When the user is done pouring, they must stop the flow and proceed to the checkout. At this time, the bartender interface will be informed of the amount of volume poured, and the cost of the purchase. This cost will be added to the user's total in the local database.

The beverage flow is controlled by a PIC18F4320 microprocessor and a solenoid valve. The microcontroller receives instructions from the user interface via serial communication. The user interface sends a single byte to the microcontroller indicating the volume limit of the beverage to be dispensed, and whether or not the user wants to begin pouring or stop pouring. Once a size has been set and the user indicates that they would like to begin pouring, the microprocessor polls an input to check if for the presence of a cup. Cup presence is determined using a limit switch. The switch is constantly polled while the unit is pouring to ensure that a cup is still present. If a cup is there, the solenoid valve opens and allows for liquid to be dispensed. Simultaneously, the flow sensor begins to deliver pulses to another input on the microcontroller. The pulses are used to keep track of the amount of volume that has been dispensed. Periodically, the microcontroller sends a status byte (indicating if it is still pouring and a cup is still present) and two bytes containing the volume dispensed, back to the user interface. At any time, if a cup is not present, the user is notified and the system stops dispensing. Otherwise, the system stops dispensing when either the volume limit is reached or the user indicates that he or she is finished pouring. At this time, the solenoid valve is closed.

If the user chooses to set up a username and password for a Smartphone, they are presented with a form that allows him or her to enter a username (consisting solely or letters) and a password (a four digit pin). Once these requirements are met, the user can choose to save the username and password. These are sent to the bartender interface which then queries the local database to see if a user already has a username and password. If they do not, the database is updated and the chosen username and password and stored to be associated with the credit card number that was initially read by the magnetic stripe reader.

After proceeding through one of these options, the user may chose to order again, or to checkout of the AutoBev system. If they want to order again, their total will be continuously updated in the database. If they chose to check out, the system will wait for a new user to swipe a credit card to begin a new session.

The AutoBev system has the capability to receive drink orders from a Android Smartphone application. As was described above, a user may set up a username and password at the main AutoBev kiosk. These are associated with a credit card track number and a nightly total. A customer must enter their username and password into the phone application and then pressed a button indicating the specialty drink they desire to order. Before the order is sent to the bartender, it is verified that the customer is willing to pay the cost of the drink. Once this is done, the bartender interface verifies the existence of the username, and that the user has entered the correct password. If the username or password is

incorrect, the phone will be sent a text message indicating a failure. Otherwise, the customer is sent a message with his or her drink order number, along with how many orders are in front of them. When the customer's drink is finished and cleared from the bartenders drink queue, the phone is sent a text message indicating that the order has been completed.

1.7 Expectations:

The design met the expectations of Team AutoBev. The user interface provides customers with an easy-to-use form that enables automated ordering of specialty drinks, as well as the capability to pour your own beverage. The volume that has been poured is kept track of using pulses from the flow sensor. The accuracy of this measurement is as was expected. It was known that 5600 pulses corresponded to a single liter of volume being dispensed. The number of pulses detected is reported to the user interface where pulses are converted to volume. Based on multiple trials, it has been seen that this method is extremely accurate. Despite the fact that there is the possibility of a pulse being missed, the accuracy is still very high due to the low amount of volume associated with each pulse.

The navigation through the user interface is simple and secure. It provides the user with up-to-date prices for specialty drinks, as well as for the self-dispensing beverage. The bartender is able to view a drink queue indicating the drinks to be made. The bartender interface allows the bartender to clear drinks that have been made, as well as the ability to save and load database information from a specified date. The bartender is also able to process a drink request from a Android Smartphone. In both of these cases, the user is informed of what order number they are as well as how many orders are in front of them. The Android application exceeded the expectations of Team AutoBev. This part of the AutoBev system has proven to be very accurate. The appropriate drink order is always displayed on the bartender interface and the cost of that drink is accurately added to the user's nightly total.

The only expectation that was not met was the expectation that the user would only need to have access to the mouse to use the user interface. Because the Android application was implemented, it was necessary to change the system to allow for the user to also have access to a keyboard (on which they can enter a username and password).

2 Detailed Project Description:

2.1 System Theory of operation:

A user is able to interact with the AutoBev system via a kiosk located within the establishment or through the use of a Android Smartphone. When a user orders a specialty drink, the order is converted to bytes and sent as a datagram packet to the bartender. In order to send the order, the IP address of the bartender must be known by the customer. The bartender (or server) accepts input from any IP address. The server sends messages back to the user interface or Android by also using UDP communication with datagram packets. The bartender can choose to save or load the database information from a given night in a text file. The text file is written using html format. The system is displayed on two PC monitors that are powered through a standard wall outlet.

A user is able to control the dispensing of a beverage through the user interface. A single byte indicating desired size and action (start or stop pouring) by the user is sent from the user interface to the microcontroller. A Universal Asynchronous Receiver/ Transmitter (UART) is used to send the byte. The RS-232 protocol is used. The byte is processed by microcontroller code which is implemented in C. The microcontroller then sets control signals to either open or close the solenoid valve. It also polls an input port to determine if a cup is present under the dispenser. When liquid begins to flow, the flow sensor begins to pulse and these pulses are processed as interrupts by the microcontroller. Each time an interrupt occurs, the total pulses are updated. A status byte, along with two bytes holding the number of pulses detected are periodically sent via the UART to the user interface so that the user can see how much volume he or she has dispensed.

Figure 1 below shows a block diagram of the entire AutoBev system.

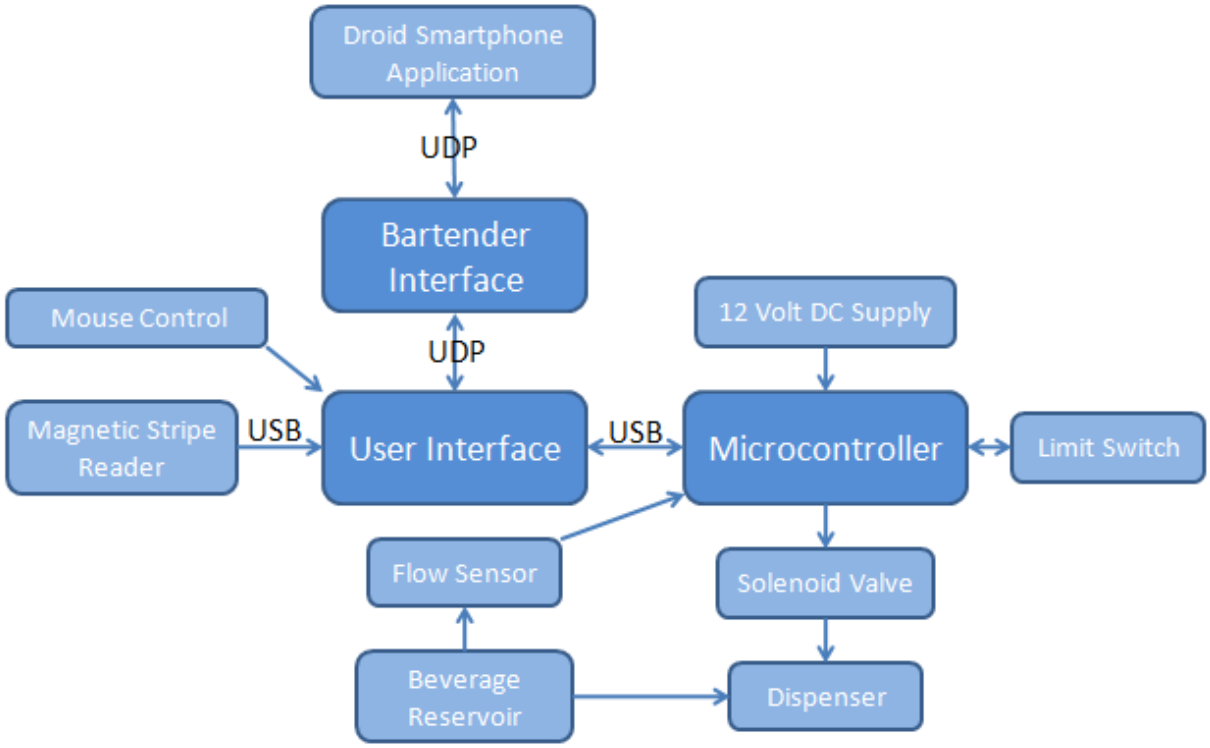


Figure 1 System Block Diagram

Power Supply:

A twelve-volt DC power brick power the board. It will be used to provide a five volt regulated source to the solenoid valve.

Microcontroller:

The microcontroller will receive data from the user interface via serial communication through the UART. It will report its status and the volume of liquid that has been dispensed back to the user interface. The microcontroller will control the solenoid valve to either allow or prohibit beverage

dispensing. Additionally, the microcontroller will receive input from a limit switch detecting the presence of a cup.

Beverage Dispensing:

While the user is pouring, the beverage will flow through a flow sensor that delivers pulses to the microcontroller. One liter of volume corresponds to 5600 pulses. The beverage dispensing unit will be controlled with a solenoid valve which is turned on and off by the microcontroller.

User Interface:

The user interface will take input from a mouse, a keyboard, and a magnetic stripe reader. The mouse and keyboard will be used to allow the customer to set up a username and password, and to allow the customer to navigate through the interface. The magnetic stripe reader will be used to obtain credit card information from the customer. The user interface will direct input to both the microcontroller and the bartender interface. It will be displayed on a standard PC monitor.

Bartender Interface:

The bartender interface will take input from the Android Smartphone app and the user interface. It will receive data via a UDP connection. It will respond to the user indicating their drink number, and the number of drinks in front of them. It will be displayed on a standard PC monitor

Interfaces:

User/Android to Bartender: UDP connection using sockets, send orders in bytes that can be decoded into strings

User to Microcontroller: asynchronous communication using Universal Asynchronous Receiver/ Transmitter (UART), following RS-232 protocol

2.2 Detailed operation of Subsystems

Card Scanner

The card scanner subsystem is used to extract credit card information from a user before they start the ordering process. At the beginning of a session, a customer must first swipe their credit card. The number on the credit card is sent over the serial connection to the user interface where it is verified to be a valid credit card number. In the AutoBev system, the first track of the card is read and stored as a means to identify the customer. For the whole night, every time a customer orders a drink, the total associated with the track read by the MSR is updated.

The card scanner subsystem consists of a magnetic stripe reader (MSR). The MSR is located alongside the customer interface so that the customer can scan his or her debit or credit card. The scanner is a “plug and play” device that simply connects to a USB port of any standard PC and emulates keyboard inputs without the need for additional software. The card scanner was programmed to output track one of the credit card according to the ISO/ IEC 7813 standard. This standard defines properties for the cards used in financial transactions such as debit or credit cards. There are two and sometimes three tracks on the cards that conform to this standard. The format of track one is shown below:

Track 1, Format B:

- **Start sentinel** — one character (generally '%')
- **Format code="B"** — one character (alpha only)
- **Primary account number** (PAN) — up to 19 characters. Usually, but not always, matches the credit card number printed on the front of the card.
- **Field Separator** — one character (generally '^')
- **Name** — two to 26 characters
- **Field Separator** — one character (generally '^')
- **Expiration date** — four characters in the form YYMM.
- **Service code** — three characters
- **Discretionary data** — may include Pin Verification Key Indicator (PVKI, 1 character), PIN Verification Value (PVV, 4 characters), Card Verification Value or Card Verification Code (CVV or CVK, 3 characters)
- **End sentinel** — one character (generally '?')
- **Longitudinal redundancy check** (LRC) — it is one character and a validity character calculated from other data on the track. Most reader devices do not return this value when the card is swiped to the presentation layer, and use it only to verify the input internally to the reader.

Figure 2 Track 1 of ISO Standard

http://en.wikipedia.org/wiki/Magnetic_stripe_card

After the user swipes his or card, the information contained in track one is sent over a serial connection to the user interface. Here, the track number is checked against the standard seen above in Figure 2 to ensure that it is a valid credit card number. If it is determined to be a valid number, an account is created for that track number. All future purchases associated with that track will be added to a total kept in a local database on the bartender interface.

The MSR chosen for this project was Unitech's MS240 Magnetic Stripe Reader. This product meets all stated requirements for reading credit cards and is relatively inexpensive. The MS240 interfaces with both a Mac and a PC and is able to emulate a keyboard. An additional benefit of the MS240 is that it is programmable. By using the 'Reader Configuration Manager,' it is possible to download settings to the MS240. These include, but are not limited to, turning a beeper on or off when a card is read, specifying which tracks to read, deciding on how information is displayed on the PC end, and choosing the type of 'end' character. The end character is what is sent to the PC to indicate that the reading has finished. In our case, the end character was set to be the enter key.

In order to test the functionality of this subsystem, the MSR was plugged into a serial port on a standard PC while a text file was opened. It was verified, using ten different credit cards, that the MSR was in fact reading track one of the credit cards. The MSR was then connected to the PC while the user interface was running. It was verified that when the reader read a valid credit card, the user interface proceeded to the second form in the drinking ordering process. This indicated that the MSR had read a track and then sent the appropriate end character, the enter key.

User Interface

Error! Reference source not found. below displays a flow chart representing the paths that the user may take through the interface.

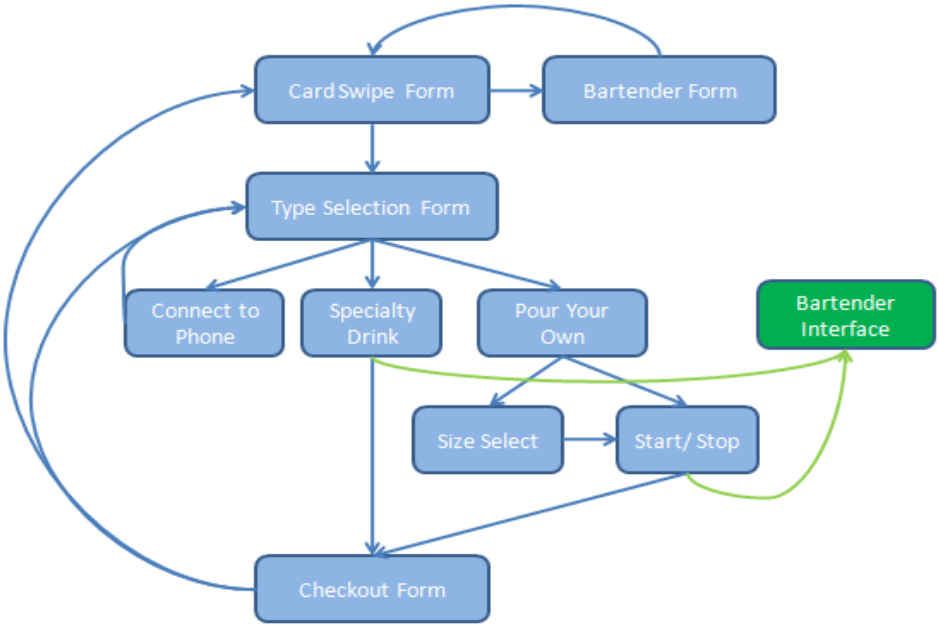


Figure 3 User Interface Flow Chart

Function of Subsystem

The user interface allows the user to interact with the AutoBev system. It is displayed on a standard PC monitor. The user interacts with the interface using a keyboard and a mouse. The interface consists of nine different forms.

Software Flow:

Welcome Form:

The first form with which the user is presented is the 'Welcome Form.' This form instructs the user to swipe their credit card. Track one of the credit card is read using the MSR and is input into a hidden textbox located on the form. When the specified 'end character' is detected as an input into the text box, the form checks the contents of the text box. Two types of input cause the form to respond. The first is if the bartender swipes a special access card. The code of this card is stored within the user interface. If this code is detected, then the user interface brings up a form that only the bartender can access which allows the bartender to dispense a beverage from the AutoBev system without being charged (sets the price of the dispensed beverage to zero dollars). This functionality is useful for when the reservoir needs to be changed. The other type of input that causes the form to respond is if a valid credit card has been swiped. When the end character is detected, the contents of the text box are also compared against the standards for a valid credit card. Upon validation, the user interface proceeds to the 'Type Selection Form.' The track that is read is stored as the 'cardNumber' property in a new instance of the class 'Person.' A credit card is identified when the 'cardNumber' is a semi-colon.

When a server is not detected upon opening the user interface, an error message appears on the welcome form which tells the user of the error. When this occurs, a button labeled 'Retry' also appears.

Pressing this button re-checks for a server connection. If a connection is detected, the Welcome Form appears as it does below in Figure 4.

Figure 4 Welcome Form

Bartender Form:

When the Welcome form identifies that a bartender has swiped a special access card, the Bartender Form is opened. The bartender may then choose to pour a beverage. When this option is selected, the bartender is taken to the Start and Stop pouring form. However, the cost associated per ounce with this transaction is set to zero. Additionally, the limit that the bartender can pour is set to the maximum amount. Through the use of this form, it is possible for the bartender to refill the beverage reservoir with ease. The bartender form can be seen below in **Error! Reference source not found.**.

Figure 5 Bartender Form

Type Selection Form:

As soon as the type selection form is loaded, a request is sent to the bartender interface for an updated list of drinks and prices. This list is received over a UDP client-server connection. A 'ReceiveData' function runs on a separate thread than the actual form to enable the reception of asynchronous data. The list that is received is then stored in a byte array which is part of the Person class. The user is presented with three options on the Type Selection Form. The user can choose to set up a username and password to be used with the Android application, to order a specialty drink, or to pour their own beverage. Clicking the button corresponding to any of the aforementioned options will bring the user to a new form.

Figure 6 Beverage Select Form

Username Form:

The Username form allows the customer to set up a username and password that can be used on an Android Smartphone. The username form requires the user to input a username and a password into a text box. This is done through the use of a virtual keyboard so that the need for an actual keyboard by the user was eliminated. When the user clicks the save button, the form verifies that the username consists of all letters and that the password is a four digit pin. The decision to create these constraints on the username and password came out of a desire to simplify the design. If the constraints are met, the track (stored in the current instance of the Person class), the username, and the password are sent to the bartender interface. Here, they are stored in a local database to be used as a means of identifying the user when they place an order from their Android. If a user forgets their password, they may swipe their card at the AutoBev kiosk and again choose the option to connect to their phone. When they do so and click 'save' the server sends back a message informing the user of the username associated with their account. The username form can be seen below in Figure 7.

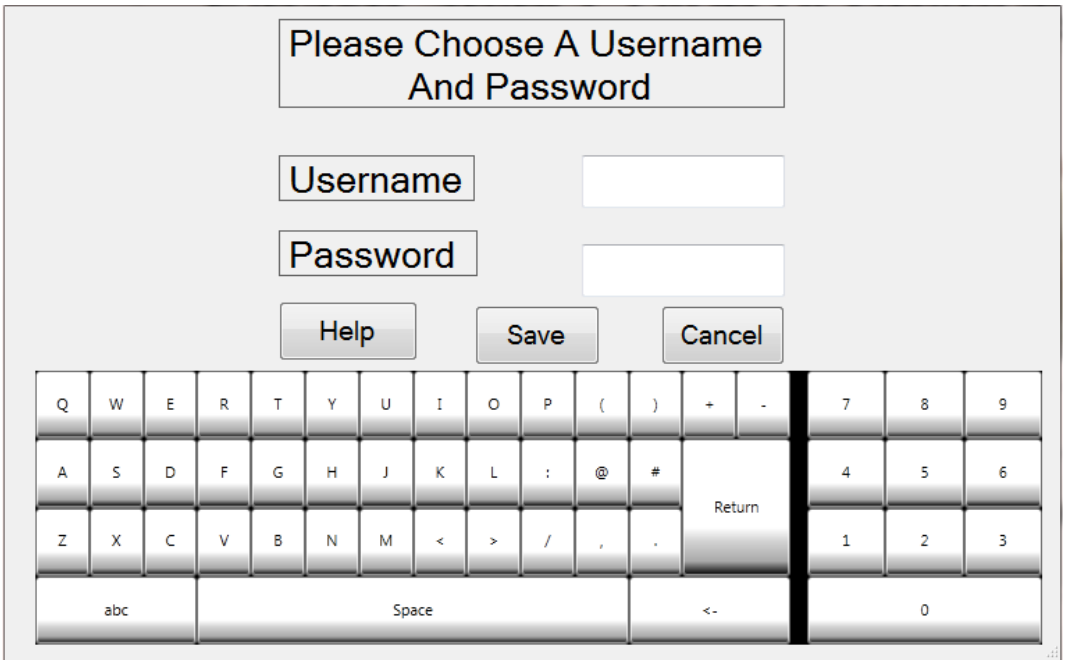


Figure 7 Username Form

Specialty Drink Form:

The Specialty Drink form allows the user to select a drink from a list of six. The price corresponding to the drink is displayed below the corresponding drink button. The list of drinks and prices is set from the 'Drink' property in the current instance of the 'Person' class. When a button is pressed, the drink order, along with the track is sent to the bartender interface. Here, the users total is updated. The order is sent over a UDP client-server connection. After sending the order, the specialty drink form waits for a response from the bartender. This is done by running a thread separate from that on which the form is running to detect when asynchronous data has been sent to the user interface. The information sent back contains the order number, the number of orders in front of the current order, and how much the credit card associated with the purchase has been charged. This information is displayed to the user on the Checkout form. The specialty drink form can be seen below in **Error! Reference source not found.**.

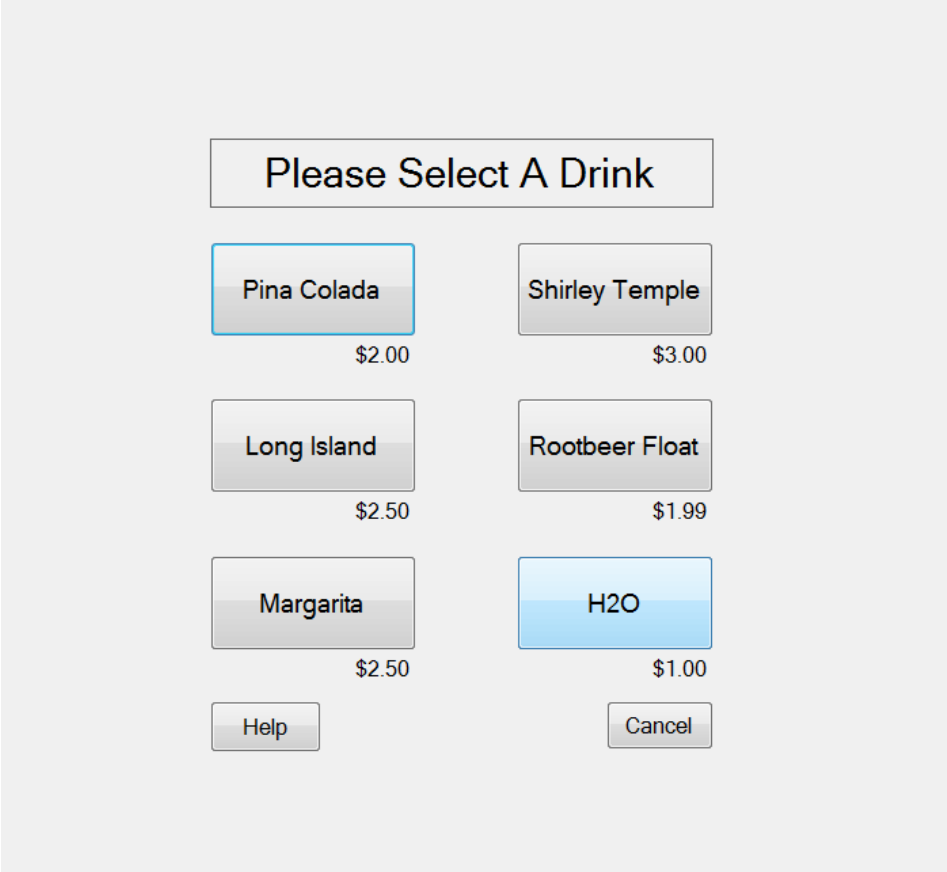


Figure 8 Mixed Drink Form

Beverage Dispensing Type Form:

When the user selects the option on the Type Selection form to 'Pour You Own,' the user interface attempts to establish a serial port connection with the microcontroller. If the connection is successful, then the user is presented with the Beverage Dispensing Type form. If a connection cannot be made, then the user is informed of the error with an error screen. The user may try to establish the connection again by pressing the 'Retry' button on the form. If a connection is made, the user is given the option of selecting a size of beverage to be poured, or of paying per ounce. When a user decides to pay by the ounce, the limit is set to three pitchers (or 180 ounces). This action is indicated to the microcontroller by setting the bit three of the status byte. If the user selects this option, the user interface attempts to establish a serial port connection with the microcontroller. If it can do so, then the user is directed to the start and stop pouring form. If it cannot, then the software proceeds to the start and stop form. If the user chooses to select a size, bit three of the status byte is cleared. The Beverage Dispensing Type form and the possible error screen can be seen below in **Error! Reference source not found.**.

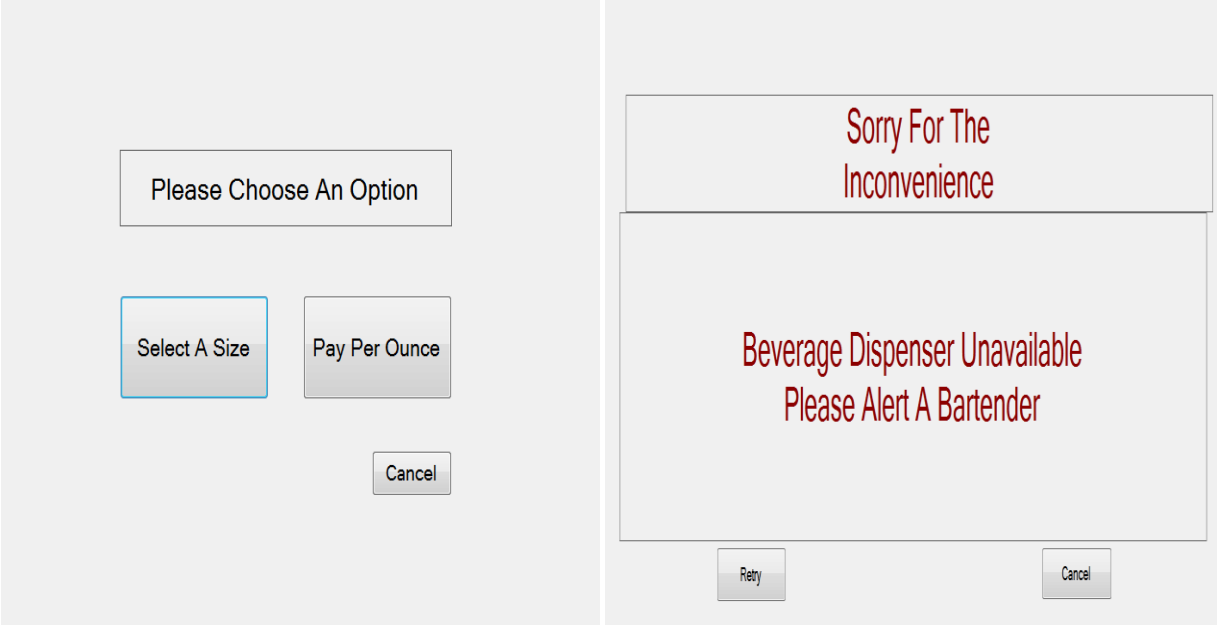


Figure 9 Beverage Pour Type Form/Error Message

Beverage Size Select Form:

The Beverage Size Select form presents the user with three size options. A user may choose to dispense twelve, sixteen, or sixty ounces. The prices of the three sizes are displayed below their respective buttons (and correspond to the price that was sent in the price list when the user began their session). The fourth, fifth, or sixth bit of the PC to microcontroller status byte is set when the user selects to pour twelve, sixteen, or sixty ounces, respectively. After a size is selected, the status byte is sent over the USART to the microcontroller. The user interface waits to bring up the Start and Stop Pouring form until a status byte indicating that the volume limit has been successfully set in the microcontroller code is sent from the microcontroller to the PC. If an error status byte is sent back, the user is asked to select the size again. The Size Selection form can be seen below in **Error! Reference source not found.**.

Figure 10 Beverage Size Select Form

Start and Stop Pour Form:

After making a successful serial port connection between the user interface and the microcontroller, the user is presented with the Start and Stop Pour form. This form gives the user three options. The first option is to start pouring. When selected, the first and second bits of the PC to microcontroller status byte are set and the byte is sent to the microcontroller. The microcontroller then opens the solenoid valve and allows a beverage to be poured. The second button on the form is the stop pouring button. When selected, the second bit of the PC to microcontroller status byte is cleared and the byte is sent to the microcontroller, which will then close the solenoid valve and stop the flow of beverage. This form has a text label which is constantly updated to display to the user how much liquid has been poured. This form is in constant communication with the microcontroller which periodically updates the user interface with its status and the volume dispensed. Each time data is received from the microcontroller, the form checks the status byte. This byte indicates if a cup is still detected by the microcontroller and if the volume limit has been reached. If it has been determined that a cup is not present, the form instructs the user to place a cup under the dispenser and begin pouring again. In a similar manner, if the volume limit has been reached, the customer is notified and instructed to proceed to checkout. The lsat option on the form is to 'finish pouring.' When the user selects this button, bit seven of the status byte is set and then sent from the PC to the microcontroller to indicate that the current session is over. This reinitializes total volume poured to zero ounces and the microcontroller then beings waiting for further instruction from the user interface. The Start and Stop Pour form can be seen below in **Error! Reference source not found.**.

Figure 11 Start and Stop Pouring Form

Checkout Form:

The Checkout form presents the user with two options. The first is to checkout. If the user chooses to checkout, the user interface proceeds to the Welcome Form and waits for a new user. The 'cardNumber' property of the Person class that was created when the person first scanned into the system is cleared in order to prepare for a new customer. If the user chooses to order again, the user is taken to the Type Selection form. The track that was read when the person first scanned into the system is kept as the 'cardNumber' property for the current instance of the Person class that had previously been created. This is again used as a means to identify the customer and update the associated nightly total in the local database. If the user has just ordered a specialty drink, the checkout form displays the order number, how many orders are in front of the customer, and the amount that has just been charged to the user. If the user has just poured their own beverage, the checkout form tells the user how much he or she has just added to their total. The total is retrieved from the database located in the bartender subsystem. An example of a checkout form (when the user ordered a specialty drink) can be seen below in **Error! Reference source not found.**.

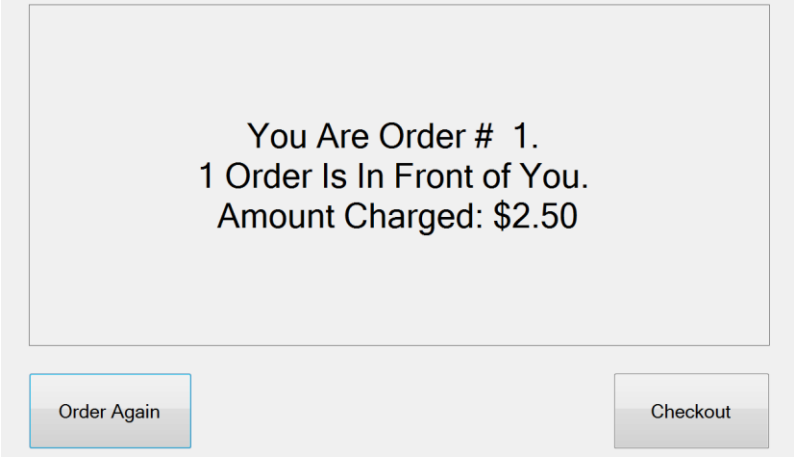


Figure 12Checkout Form (Specialty Drink)

New Class: Person.cs

The person class is a class that holds two important variables. A new instance of 'Person' is created when the program opens. This instance is continuously updated with information about the current user and is passed between forms to convey information. This class has four important properties. The first is a string called 'cardNumber.' When a user swipes their credit card on the first screen, track one of the credit card is stored in this variable. It is later sent to the bartender interface such that the appropriate user may be charged for the order. The second property of the person class is a string called 'username.' This is used to store the username of the current customer should they choose to connect their Android phone to the AutoBev system. Similarly, there is a property that is a string called 'password' which is used to store the four digit password that the customer would like to associate with his or her username. Lastly, the 'Person' class has a property called 'drink.' This is a string array with a length of fourteen. It is used to store the names of the six specialty drinks, their prices, the name of the 'pour your own beverage', and the prices per ounce of that beverage. The cardNumber property of the instance of Person is cleared every time a customer checks out. The username and password properties are also cleared. The drink array is updated every time a user chooses the option to order a specialty drink. At this time, the bartender is contacted via the UDP client-server connection and responds with a string containing a list of drinks and prices which is then displayed to the user on the Specialty Drink Selection form.

Interfaces to Other Subsystems:

On many of the forms, especially those with options, there is a 'help' button. When the user clicks this button, a pop-up box appears with information about the form. These help windows are unique to the form on which they are displayed.

Communication Protocols:

The user interface communicates with both the microcontroller and the bartender interface.

A Universal Asynchronous Receiver/ Transmitter (UART) is used to communicate between the user interface and the microcontroller using the RS-232 protocol. A single status byte is sent from the PC to

the microcontroller where the individual bits are checked to determined the action desired by the customer. The microcontroller responds with a single status byte and two bytes that are combined by the user interface to represent a 16 bit volume. The bits of the status byte sent from the microcontroller to the PC are individually checked to determine the state of operation of the microcontroller. The meaning of each bit in these bytes are further described in section X.X. liz

Subsystem Testing:

The user interface was tested by stepping through each path that a customer may take while placing an order. Through this process, it was possible to verify the functionality of each command. Additionally, message boxes (which act like a pop-up message in a web browser) were placed in strategic locations to verify that information was being transmitted and stored properly. Furthermore, verification statements were output to a text file at strategic points throughout the interface. This text file was then viewed and essentially displayed the progression of the customer through the interface.

It was essential to use the debugger that is available in Visual Studio. During the testing of each separate form, break points were inserted before every function (or event callback which was triggered by any action performed on the form). The debugger was then used to 'step through' the function. During this process the value of each variable was verified to be correct at every point within the code.

In order to test the integration of the user interface with the microcontroller/ flow sensing hardware, it was verified that the volume specified by the user on the Beverage Size Selection form was that which was poured by the beverage dispensing system. For example, on the user interface, a twelve ounce beverage was selected. Twelve ounces was then measured manually and then allowed to flow through the beverage dispensing system. As this occurred, it was verified that the amount poured was updated on the Start and Stop Pouring form, and that the solenoid valve closed and prohibited further dispensing after twelve ounces had been poured. This was repeated for the other two sizes.

In an effort to make the C# code as error-proof s possible, try-catch statements were used extensively. They were especially important to use on forms that interface with other subsystems. For example, every time the bartender is contacted, a UDP connection must be established. However, if a bartender is not available, it will not be possible to establish a connection. Therefore, a try-catch statement was used to test for a possible connection. The same logic was used when deciding to use a try-catch statement when attempting to interface with the microcontroller through a serial port.

Why Engineering Decisions Were Made:

The user interface was implemented as a Windows Form Application using the Microsoft Visual Studio 2010 IDE. After investigating and comparing several languages supported by this environment, it was decided that the best way to achieve a professional-looking and easy-to-use interface was to program using Visual C#. The combination of Microsoft Visual Studio and Visual C# enabled access to many useful predefined libraries. These existing libraries made it relatively simple to communicate with the microcontroller, the bartender interface, and the magnetic stripe reader.

The decision to stylize the user interface as Windows Forms was based on the fact that forms are easy for users to navigate. During the drink ordering process, every time the user needs to make a selection, he or she is presented with a self-explanatory form. The interface then proceeds to the next form. The ability to separate each step of the drink ordering process with the creating of multiple forms greatly simplified the development of the user interface.

Interface to Other Subsystems/ Communication Protocols:

The user interface communicates with both the microcontroller and the bartender interface.

A Universal Asynchronous Receiver/ Transmitter (UART) is used to communicate between the user interface and the microcontroller using the RS-232 protocol. A single status byte is sent from the PC to the microcontroller where the individual bits are checked to determined the action desired by the customer. The microcontroller responds with a single status byte and two bytes that are combined by the user interface to represent a 16 bit volume. The bits of the status byte sent from the microcontroller to the PC are individually checked to determine the state of operation of the microcontroller. The meaning of each bit in these bytes is further described in Section 2.2 Microcontroller/PC interface.

How this subsystem was tested

The user interface was tested by stepping through each path that a customer may take while placing an order. Through this process, it was possible to verify the functionality of each command. Additionally, message boxes (which act like a pop-up message in a web browser) were placed in strategic locations to verify that information was being transmitted and stored properly. Furthermore, verification statements were output to a text file at strategic points throughout the interface. This text file was then viewed and essentially displayed the progression of the customer through the interface.

It was essential to use the debugger that is available in Visual Studio. During the testing of each separate form, break points were inserted before every function (or event callback which was triggered by any action performed on the form). The debugger was then used to 'step through' the function. During this process the value of each variable was verified to be correct at every point within the code.

In order to test the integration of the user interface with the microcontroller/ flow sensing hardware, it was verified that the volume specified by the user on the Beverage Size Selection form was that which was poured by the beverage dispensing system. For example, on the user interface, a twelve ounce beverage was selected. Twelve ounces was then measured manually and then allowed to flow through the beverage dispensing system. As this occurred, it was verified that the amount poured was updated on the Start and Stop Pouring form, and that the solenoid valve closed and prohibited further dispensing after twelve ounces had been poured. This was repeated for the other two sizes.

In an effort to make the C# code as error-proof s possible, try-catch statements were used extensively. They were especially important to use on forms that interface with other subsystems. For example, every time the bartender is contacted, a UDP connection must be established. However, if a bartender

is not available, it will not be possible to establish a connection. Therefore, a try-catch statement was used to test for a possible connection. The same logic was used when deciding to use a try-catch statement when attempting to interface with the microcontroller through a serial port.

The user interface was implemented as a Windows Form Application using the Microsoft Visual Studio 2010 IDE. After investigating and comparing several languages supported by this environment, it was decided that the best way to achieve a professional-looking and easy-to-use interface was to program using Visual C#. The combination of Microsoft Visual Studio and Visual C# enabled access to many useful predefined libraries. These existing libraries made it relatively simple to communicate with the microcontroller, the bartender interface, and the magnetic stripe reader.

The decision to stylize the user interface as Windows Forms was based on the fact that forms are easy for users to navigate. During the drink ordering process, every time the user needs to make a selection, he or she is presented with a self-explanatory form. The interface then proceeds to the next form. The ability to separate each step of the drink ordering process with the creating of multiple forms greatly simplified the development of the user interface.

Bartender Interface

Figure 13 below displays a flow chart representing the paths that the bartender may take through the interface.

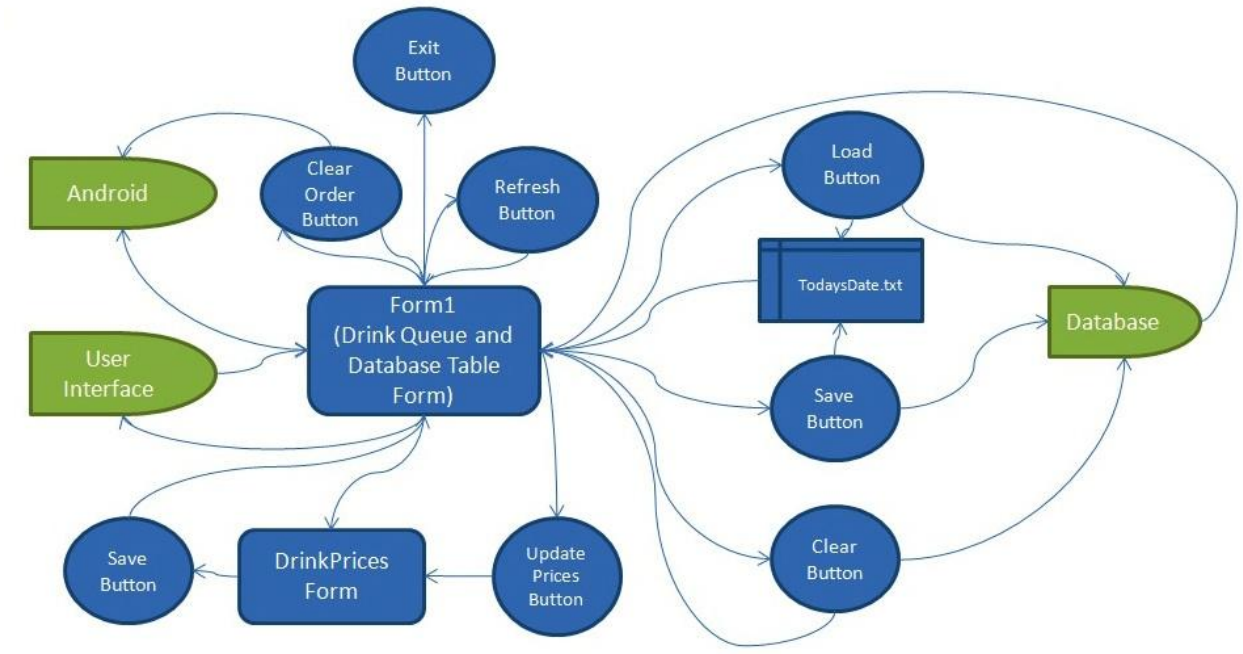


Figure 13-Bartender Interface Flow Chart

The fundamental purpose of the bartender interface is to serve as a database of recently ordered specialty drinks. The software allows the bartender to read drink entries made by the customer at the user hub, and then delete the orders after they have been prepared. The display for the bartender is a drink list that is updated with additions from the User hub. It provides options for

the bartender to scroll through the orders and delete selected drinks. It also allows the bartender to update a drink list and prices at the beginning and throughout the night.

The bartender needs to be able to interface with the customer. This takes place over two separate computers. A UDP Server/Client protocol was used for communication between the two computers, the User and Bartender hub, over a local area connection. A UDP protocol was chosen because two of the team members had experience with computer networks and because the amount of information being communicated is extremely small and only needs to be sent occasionally in packets, which is ideal for UDP. The protocol was written in C# so that it can seamlessly interface with the customer interface. For a more detailed description of the client/server interface refer to section 2.2.

Upon running the AutoBev Bartender Interface program the code calls on the function InitializePrices() and the class DrinkPrices is called. From within this class the function DrinkPrices() is called and the bartender is shown a DrinkPrices form with a drink list and prices. From this form, the bartender can udpate the drinks and drink prices at the beginning of the night. The form can be seen in Figure 14 below.

Figure 14 Bartender Interface Drink Prices Form

The drink names and prices that are initially shown on the list are read from a file DrinkPrice.txt that is saved on the PC that the bartender is using. The file DrinkPrice.txt is written throught the use of a delimiter character, ' / ', that separates the acutual drink names and prices from each other. When the bartender saves the new drink names and prices that they have typed out by clicking the save button, their input into each textbox is written back to the file DrinkPrices.txt, again through the use of the delimiter character , ' / '. The DrinkPrices from is then closed and the bartender is shown form1, which is the main Bartender Interface form. The main program is still in the InitializePrices() function. At this point, the program will read the new DrinkPrice.txt file and save the inputs into an array with every other value in the array being the delimiter character. This array is stored for the droid application as droidPriceList, which is a string with each of the inputs in the array being sent with a delimiter character, '#', separating each input. Once the program server has established a connection to the client on the user interface, the program sends a string with

the updated drink price list to the user interface , so that the drink options on the MixedDrinksForm on that interface can be updated appropriately. This is done through the function UpdateDrinks() in the Bartender Interface program. The main form of the Bartender Interface also has a button titled 'Update Prices' , which also allows the bartender to update the drink list throughout the night. This button has essentially the same function as the InitializePrices() function and will also bring up the DrinkPrices form, save the bartender's inputs, store, and send them to the user.

When the bartender is not updating the drinks list, form1, the main bartender interface form is shown. The function of this form is mainly for the bartender to be able to see the drink orders and order numbers for the night. The drink order and numbers are stored to different arrays within the program. Two variables, *counter* and *i* were initialized to 0 at the beginning of the program and were incremented by 1 each time a new order was processed. The integer value of the variable *counter* was placed in a *count* array and was the number sent to the user as their order number after an order was placed. The variable *i* reflects the number of the last index. Each time a new order is received by the bartender it is sent using a specific protocol that indicates the type of order being processed. If the order is a kiosk order or a an order from a phone, the characters ' # ' and ' &' are sent, respectively, as the start characters of a string message. Each string message contains the start character along with five other separate strings that are stored into a *words* array after being separated via the use of a delimiter. If the order was a kiosk order, the name of the drink order is stored in *words[1]* and the cost of the order is stored in *words[3]*. If the order was a phone order, the name of the drink order is stored in *words[5]* and the cost is stored in *words[6]*. The drink name is then saved into an array, *_drink,* with each drink order saved to the next available index. Each time a drink order is processed the drink is added to the bartender interface list using the function *DrinkQueue.Items.Add*, which is called using the function *updateLabelText* in the program. The input into the *updateLabelText* function is a string, "*count[i - 1]* + " " + *_drink[i - 1]*". The logic for updating with '*i-1'* rather than '*i'* is because the variable *i* is already incremented before the order is written to the bartender list and we want to write the order in the appropriate location on the list. To allow for the user to know how many orders are ahead of them in the list, whenever their order is processed the program counts all items on the list using the function *DrinkQueue.Items.Count,* stores this value in a variable *numItemsInList,* and sends the value in this variable to the user as a string. The complete message sent to the user is in the form of the following string:

Your Are Order # " *count[i-1]* ".
There are " *numItemsInList* " Orders Are Front of You.
Amount Charged: $ "*cost"*

The purpose of storing the variable in a *count* array was also to make it so that when a drink order was cleared the *count* array would shift all values in the *count* array after the *count* index that value is stored in up into the previous index. Once this was done, the variable *i* was decremented by one to keep track of how many values were actually being stored in the *count* array. This function occurrs any time the Clear Order button is pressed on the bartender interface. The purpose of this was so that the order numbers that were already served were no longer being stored and also for the correct order number to be written on the list next to the appropriate drink order. The value that is stored in *count[0]* is always the order number of the first drink on the list and so on. Every time the Clear Order button is selected the program makes sure that one of the items on the drink queue is actually being selected. If it is not, then the program issues a message

prompting the bartender to select an item. Once an item is selected and the Clear Order button is clicked, the selected index is stored to a variable *numVal*. The purpose of *numVal* is to appropriately shift the *count* array for all values stored in the array at and after the index that is equal to *num*Val. Furthermore, *numVal* is used to shift numbers stored in a string array *_number*. The string in the array *_number* is set to "0" if the order is a kiosk order and to the actual phone number when a phone order is placed. This is because the phone number is sent to the bartender and stored in *words[1]*, which is then stored to the *_number* array. When the clear button is clicked the program calls on the function *DrinkQueue.Items.RemoveAt* with the input *DrinkQueue.SelectedIndex* when clicked. This function removes information at the selected index from the list. Then, the program checks to see if the order was a drink order by checking to see if the string *_number[numVal]* is not equal to "0" or "null". Then it will send the appropriate phone user an SMS message to the number saved for that current index indicating that their order is ready. Our current bartender code has this section of the code commented out and is actually sending one of the group members an e-mail when the order is complete. This is because the group was testing in an area of poor phone reception. However, tests were done in which an actual text message was sent to the correct phone, indicating that our bartender interface has this capability. The function of the android application is further detailed in section 2.2.

The main form of the bartender interface also offers the bartender the options of saving the database that is storing the track numbers, total charges, user names, and passwords for the night. This is to ensure that the bartender can store the information to charge at the end of the night

Using the protocol for orders processed by the bartender a table, *customers,* was able to be updated on the bartender interface with the appropriate track number, total charges, user names, and passwords. The database is stored on the bartender's PC and is read from and written to the table using XML format. The track numbers of the credit cards swiped at the user interface are sent to the bartender and updated to the database. Furthermore, each track number has a charge associated with it that is updated throughout the night each time a new order is processed from that card. Using the mentioned protocol, the bartender interface also receives a pulse count that reflects the amount of drink poured via a 'pour your own' order at the kiosk and converts this count to reflect ounces and then a price for the amount pourn. This price is added to the total charge for a given track number and is sent back to the user.

Upon starting up the bartender interface program, the program calls on *DateTime.Now* to get the actual date. It then stores the day, month, and year collectively to a string entitled *TodaysDate*. When the save buton is selected the bartender interface will save the current database to a text file with the filename that is the string stored in *TodaysDate.* If a file is already stored to the bartender's PC using todays date, then the program will add a version number to the filename. This version number is incremented by 1 each time a file is saved for the current date. For example, if the current filename is *TodaysDate_2.txt*, then the next file saved has the name *TodaysDate_3.txt.*

The bartender interface also offers the bartender the option of loading a database that has already been saved. This is done by selecting the load button on the bartender interface. When the load button is selected the program first saves the current database in a way that is essentially the same as that done when the save button is selected. Then, a textbox appears and the user is prompted to enter a filename. The bartender enters a filename and presses enter on the keyboard. If the filename does not exist, then the program issues a message box informing the user that a file for the selected date has not yet been created. If the filename does exist, then the program loads the

database for that filename to the form and the fileName text box and save button are no longer made visible, while a Clear button appears on the screen. The Clear button, when selected, clears the database information that is being shown from the screen. There is also a Refresh button on the bartender interface, which simply refreshes the current form when selected. Finally, the bartender interface also has an Exit button, which when selected calls on the function *this.Close().* This completely closes the bartender interface program.

The form for the bartender interface can be seen in Figure 156 below and is called Form1 in the bartender program.

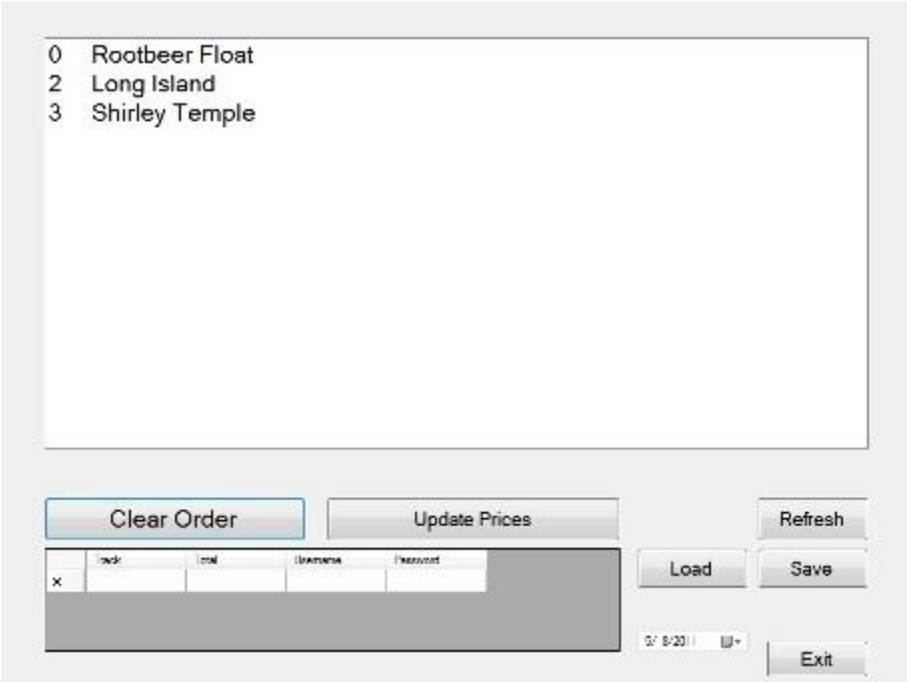


Figure 15- Bartender Interface Drink List Form

The bartender interface was tested by verifying that the drink selected on the user interface or android was properly displayed on the bartender interface along with its assigned drink order number. This process was repeated for each possible drink selection and for multiple orders to ensure that the server was updating the list correctly. We also needed to test that the clear order button worked correctly and did this by selecting orders from various places in the list and making sure that it updated correctly and sent the proper message back to the android. In order to debug the program when things were not appearing as expected on the list we had message boxes pop up with the current values of the different variables and arrays to make sure they were what we expected them to be. If they were not as expected we could have a better idea of what the program was doing based on the values given and reviewing what areas of the code would have caused them to have that value.

Microcontroller

Function

The microcontroller functions to control the state of a solenoid valve, to detect when a cup is present at a limit switch, and to keep track of the number of pulses put out by a flow sensor and communicate a status dependent on these factors to the computer. Therefore, the AutoBev kiosk requires a microcontroller capable of serial communication with a computer, at least 4 I/O pins, with at least 2 interrupts (16 and 8 for expandability), capability of handling C programmed routine of about 2KB, and an adjustable baud rate of 9600, and ability to hand solder. For these reasons, the PIC18F4320 by Microchip was chosen.

The PIC18F4320 has an internal flash memory of 8KB, serial transmit and receive pins, and 34 I/O pins. The PIC18F4320 is programmed in simple C and is compatible with Sourceboost compiler and the MeLabs programmer which was available on the learning center computers. The group also has some past experience using a similar microcontroller which made it a good choice. The last requirement that the chip be hand soldered was fulfilled by the .8 pitch spacing between pins in the TQFP- 44 pin layout.

The PIC18F4320 can be powered using a 3.3 or 5 V supply, the AutoBev controller is powered by a regulated voltage of 5 V which was stepped down from 12-volt supply that also powers the solenoid coil driver circuit.

A pin layout of the microcontroller can be seen below in Figure 17.

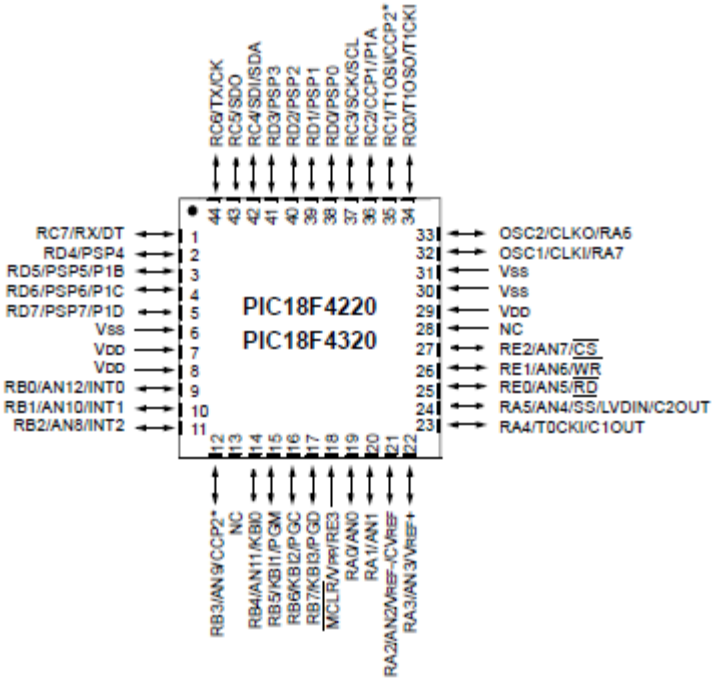


Figure 16 Pin Layout of the 44-Pin PIC18F4320

Interfaces to other subsystems

The microcontroller communicates with the user interface through UART serial communication. The program on the microcontroller is designed to wait until a status byte is sent from the user interface to

begin a new drink order session. This status byte designates instructions which include the volume to be poured, and additionally if the user is requesting to start or stop pouring. The microcontroller responds to the user interface with a status byte and two bytes which together form a six bit number holding the volume that has been poured.

How subsystem was tested

The subsystem was first tested by simulating both pulses sent by the flow sensor and the presence of a cup using I/O ports that were otherwise unimplemented. Port D six was toggled and connected to the input looking for flow sensor pulses. This was done to ensure that the code was able to recognize when an interrupt occurred (i.e. a pulse was sent from the flow sensor indicating an increase in volume) and react accordingly. The limit switch was simulated by applying a high voltage to port D 0. By connecting and disconnecting this voltage, it was possible to simulate the presence or absence of a cup.

After verifying that we were able to detect interrupts, an actual flow sensor was attached to the limit switch. A program was created in Visual Studio (in a Windows Form Application) that could send every possible status that the user interface may send the microcontroller in order to test if the microcontroller responded in a correct manner. This included sending a byte that was an error, setting the volume limit, indicating if the microcontroller should start or stop pouring, and detecting when a user wanted to check out. It was verified that the microcontroller responded in the appropriate manner by checking that the volume dispensed actually matched that which was requested by the PC.

For example, during one test, the PC sent a status indicating that the user desired to pour twelve ounces. We collected the liquid that was dispensed verified that the volume (updating on the PC monitor, due to the volume being sent back in two of the three status bytes from the microcontroller to PC) , matched that while was being poured. Additionally, while pouring, we pulled the input to the limit switch low and made sure that the microcontroller stopped pouring. We also made sure it stopped pouring when the status byte from the PC to the microcontroller indicated ‘stop pouring.’ Finally, when the program closed the solenoid valve, the amount of volume poured was carefully measured and verified to be twelve ounces. This test was repeated for other volume sizes and other start/stop scenarios.

Hardware

Figure 17 below shows a block diagram of the connections between the microcontroller and other hardware components in the AutoBev System.

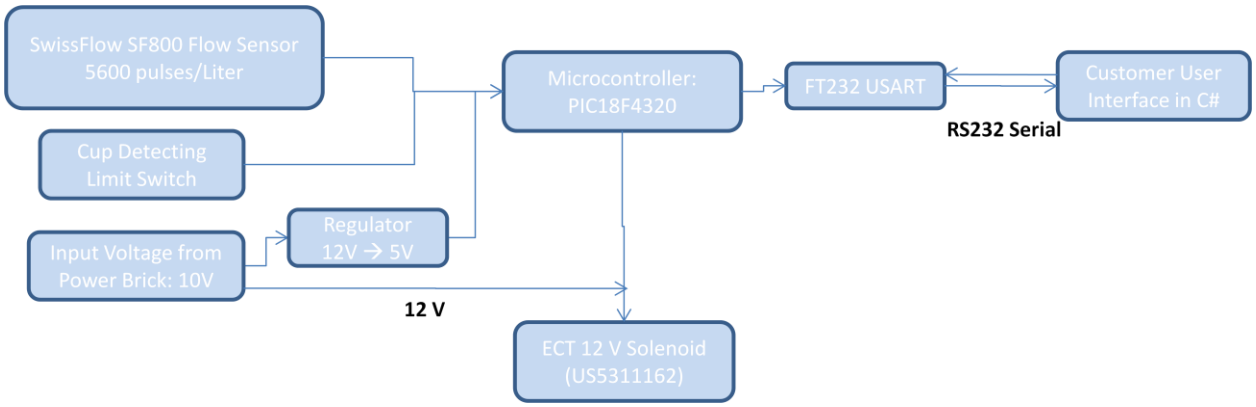


Figure 17 Microcontroller Hardware Connections

As can be seen, the microcontroller takes input from a voltage regulator that step down an input voltage of twelve volts from a power brick, to a five volt input to drive the microcontroller. Communication between the user interface and the microcontroller was achieved over the FT-232 USART. Additionally, the microcontroller took inputs from the SwissFlow SF800 Flow Sensor and a limit switch, while providing a five volt output to an ECT 12 volt solenoid.

Description of overall flow of the code

The software for this project is essentially interrupt driven. In the main function, the code begins by enabling global enabling global interrupts. This was necessary so that the code would react to every pulse sent by the flow sensor. The code flow sensor. The code then set all input and output ports to digital. This was done to allow for detection of a missing cup detection of a missing cup (using a limit switch) and to provide the ability to control the solenoid valve (using a five volt (using a five volt output). The clock source frequency was set to 8MHz. The Input/ Output pins were defined next. Two pins defined next. Two pins were associated with inputs from the flow sensor. These two pins, port B zero and port B one and port B one correspond to the INT0 and INT1 interrupt flags. Therefore, they could be used to detect external interrupts. external interrupts. Port d zero and port D one were set to input ports for the limit switch. When these port were low, it port were low, it was indicative of a missing cup. There were continuously polled in the main program to test for a high to test for a high voltage, and thus a cup. Port B two and port B 3 were initialized as output pins. They were used to control were used to control the solenoid valve and were set when the solenoid valve was to be opened. Table 4 Input/Output Ports

Table 4 below summarizes the input and output ports that are used by the program.

Pin	Type	Function
BO	Input	Flow Sensor
B1	Input	Flow Sensor
B2	Output	Solenoid Valve
B3	Output	Solenoid Valve
D0	Output	Limit Switch
D1	Output	Limit Switch

Table 4 Input/Output Ports

Following the configuration of I/O ports, general variable were initialized. A list of variables is below in **Error! Reference source not found.**.

Variable	Type	Initial Value	Meaning
inc	int	1	For increasing volume dispensed
limit	int	-	Pulse limit for selected size
count	int	0	For reporting status to PC
PIC_status	char	0	Status of dispensing system
upper	byte	-	Upper 8 bits of volume poured
lower	byte	-	Lower 8 bits of volume poured
volume	long	0	Total pulses counted (volume)
keepPouring	bool	false	Allows for opening of solenoid valve
receivedByte	unsigned char	-	Stores byte sent from PC
volumeLimit	short	-	Limit sent by PC

Table 5 General Variables, Type, and Initial Value

After the general variables were defined, the usart was initialized. The 'init_usart()' routine set the baud rate to 9.6k. It also enabled the serial port, enabled asynchronous reception, and set the bit for in high-speed baud rate mode.

After this, the program entered the main while loop that ran continuously during program execution. The program begins by waiting for a byte to be sent over the USART from the PC. This byte is the status byte that is sent from the user interface. The program waits until it receives data before proceeding. The 'recevieByte()' routine was used to wait for incoming data. This routine waits until the receive flag in the PIR1 register is set. When it is, the program grabs the data in the receive register. This is stored as the received byte.

After returning from this routine, the main program begins checking each bit in the received byte. This is accomplished using an if-statement, followed by two else-if statements. The if-statement checks to see if bit zero is cleared. If so, this is indicative of an error in the status byte that was sent from the PC to the microcontroller. In this case, the status of the microcontroller byte (PIC_status) is set to zero and sent back to the PC to indicate that it detected an error. It is essentially asking for the PC to resend the status. If bit zero is set, the value of bit one is checked. When clear, it means that a new order is being placed and the volume limit must be set. Bits three through six indicate what the volume limit should be. If none of these four bits are sent, the microcontroller status byte is set to indicate an error and is sent to the PC to request for retransmission of the last status. If bit one is set, it means that the user is requesting to either start or start pouring. Bit two set means to start pouring, and bit two cleared means to stop pouring. If the user wants to start pouring, the program checks port D zero to see if it is high. When it is high, a cup is present and the program opens the solenoid valve by setting port B two. It also sets the variable 'keepPouring' to true and 'volFlag' to false. Otherwise, the solenoid valve is kept closed. The status of the microcontroller as well as the total volume poured is sent back to the PC using the 'sendByte' routine. This routine waits until the transmit flag in the TXSTA register is set. When it is, the data is put in the transmit register and sent to the PC. Finally, if bit seven is set, it is indicative that the user wishes to end his or her session. In this case, all the variables are set to their initial values.

The program then enters a while loop if the volume limit has not been reached, if a cup is present, and if the user indicated that he or she wanted to start pouring. If all of these things are true, the program

again checks if a cup is present, and if so, the solenoid valve stays open, and the PIC status byte is set to indicate success. If not, the valve is closed and the microcontroller reports a status to the PC indicating that it has stopped pouring because no cup was detected.

The program then checks if the received flag is set in the PIR1 register indicating the presence of new data received over the USART. If data has been received, the byte is checked to see if the user wanted to start or stop pouring and the solenoid valve is opened or closed accordingly.

Next an if statement was used to test if an interrupt had been detected. When the microcontroller detects a pulse from the flow sensor, it enters the interrupt routine for the INT1F interrupt flag. In this routine, 'increaseVolume' is set to true. When it is true, the value of volume poured is incremented, and 'increaseVolume' is set back to false. The total volume is compared to the volume limit. If the volume poured has exceeded the volume limit, the microcontroller reports this in a status back to the PC and resets all variables to their initial value. Otherwise, the variable 'count' is increased. Count is continuously increased regardless of if a pulse has been detected or not. Every time count reaches the value of 1000, the status of the microcontroller and the total volume poured is reported to the PC to such that the user can see how much volume they have dispensed.

If at any time, a cup is no detected, or the user sends a status indicating that they want to stop pouring, the while loop that allows for pouring is broken and the program returns to the top of the main while loop where it stops on the receviedByte() routine and waits for the next instruction from the PC.

Note that the reason two inputs and two outputs were configured to perform the same function (i.e. detect the presence of a cup, control the solenoid value, and detect flow sensor pulses) was to allow for expandability to the AutoBev system. At this point in time, the software is configured to only allow control of a single solenoid, and detection of one limit switch and one flow sensor.

A flow chart that demonstrates the layout of the software can be seen below in Figure 18.

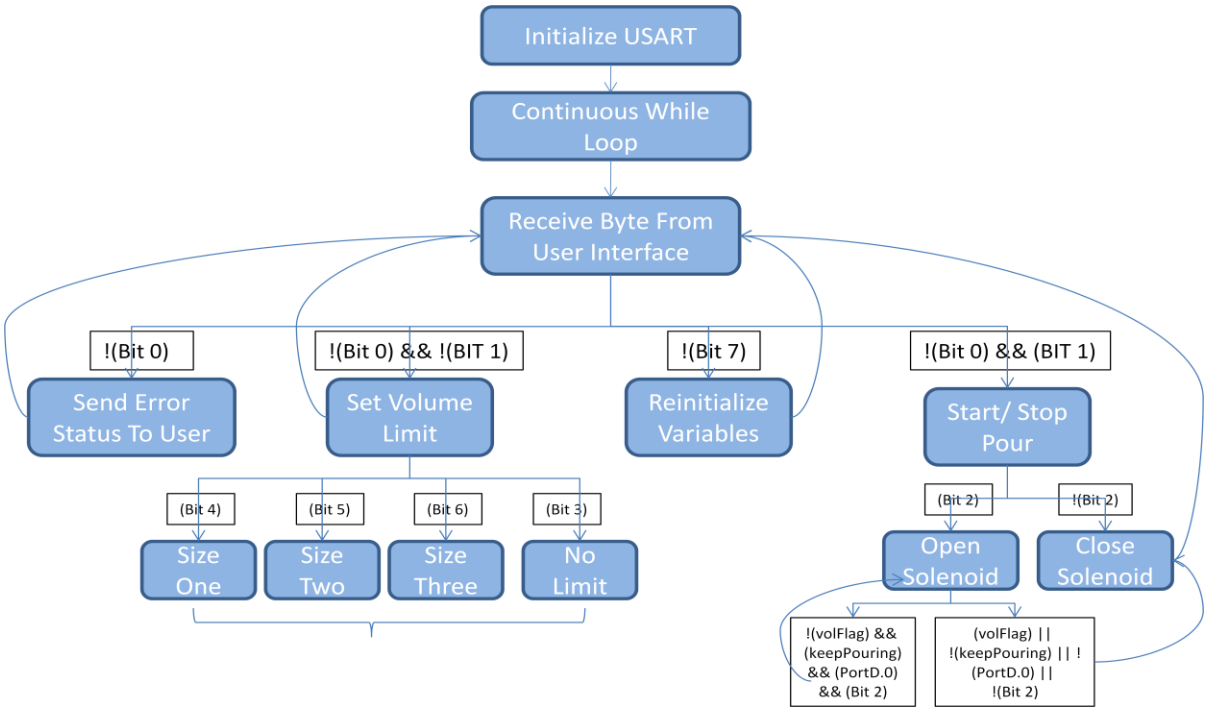


Figure 18 Flow Chart of Microcontroller Software Flowchart of Microcontroller Software

Communications protocols

The microcontroller communicates with the user interface serially using the FT-232 USART. The user interface send one status byte to the microcontroller where each bit has a different meaning. The microcontroller sends three bytes back to the user interface. The first byte is the status of the microcontroller and the second two bytes can be combined to form a sixteen bit volume (in pulses).

How subsystem was tested

In order to ensure the functionality of this subsystem, it was connected to serially to the user interface, and mechanically to the flow sensor, the limit switch, and the solenoid valve. Again, it was verified that the microcontroller responded to the status byte send by the user interface in the appropriate manner. Every possible status byte and path through the user interface that a customer could take was executed. It was possible to display the bytes send from the microcontroller to the PC using the debugging software in visual studio. Thus, each status could be verified as the appropriate response to the instruction sent from the user interface. It was also verified that the solenoid valve opened and closed when the user indicated that he or she wanted to start of stop pouring, and that it closed when the limit switch was not shut. On the user side, error messages were inserted into the start and stop form that appeared according to the status sent by the microcontroller. During testing, errors (such as no cup present) were caused and it was verified that the appropriate error messaged appeared.

Android application

Flow Chart for Software

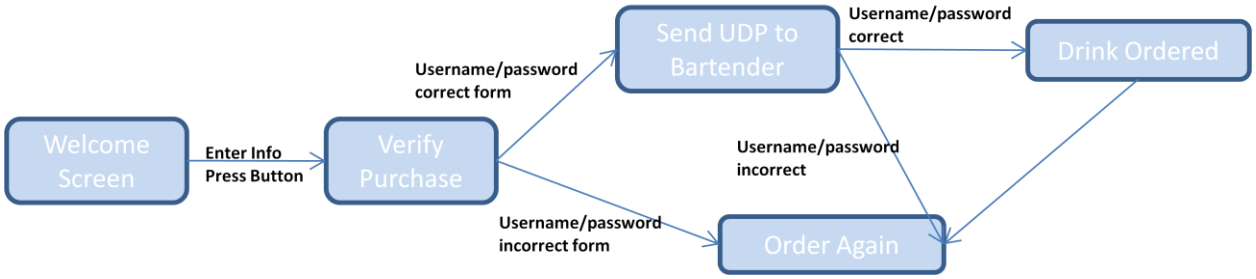


Figure 19 Android Application Flow Chart

Function

The Android application allows the user to interact with the AutoBev system with his or her Smartphone. The app has all the functionality of the AutoBev system contained at the kiosk except for the ability of the customer to pour his or her own beer.

Before using the AutoBev Android application, the user must sign in to an AutoBev kiosk at the bar. At the kiosk they will click on Connect to Phone after swiping his or her credit card. The user will then be prompted to enter a username and password which, along with the credit card track, will be sent to the bartender and saved to the database.

When the user opens the app he or she is presented with a welcome screen that contains two textboxes and six buttons shown in the figure below. The user enters his or her username and password into the respective textboxes and then clicks a button corresponding to the drink they wish to order. A message is sent over the wireless network using to the bartender's PC containing the username, password, phone number, IP address of the phone and the desired drink of the user.

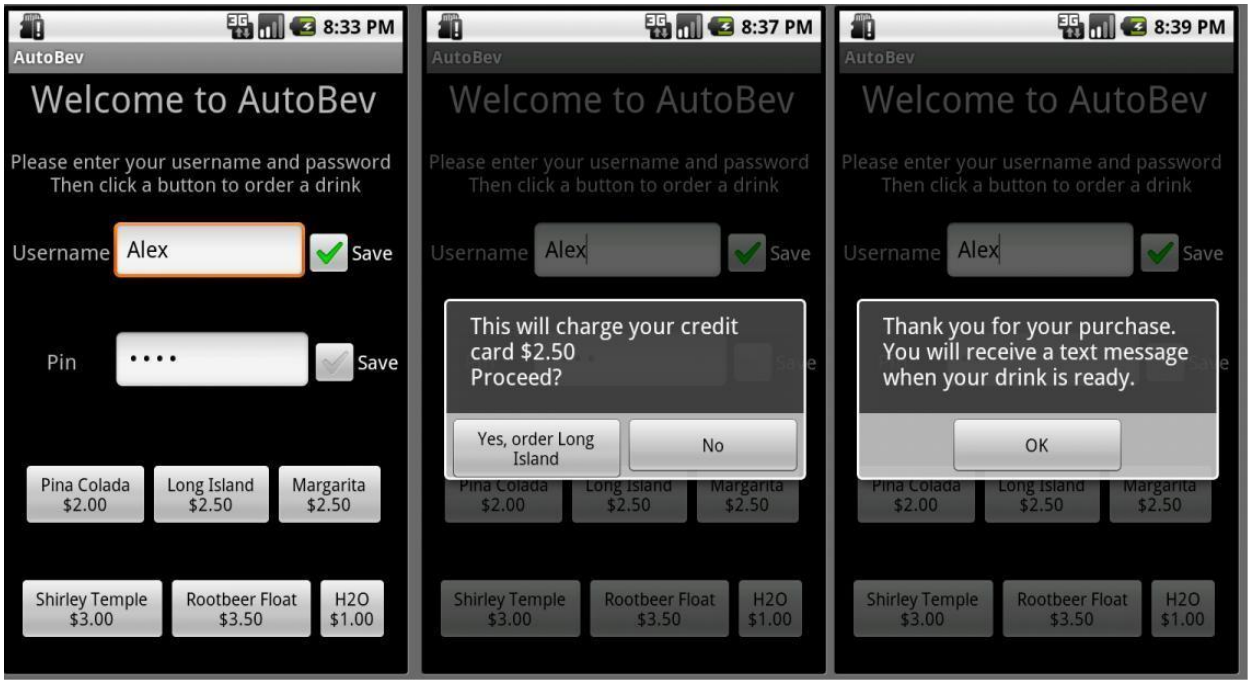


Figure 20 Android Application

The bartender's PC will verify that the username provided exists in the current database and that the password matches the username. If either of those conditions occur, the bartender will immediately send a message back to the phone telling the user to try again.

If the username and password are correct, the order is stored in the bartender's queue. A message is then sent to the app telling the user what drink number is his or her and the drink currently being served.

When the drink is filled by the bartender (i.e. the bartender clears it from his or her queue) the bartender's PC will send a text message and/or email to the Android user alerting him or her that the drink is ready.

Interfaces to other subsystems

The Android application will only interface with the bartender PC. The interface will involve a two way communication using the User Datagram Protocol (UDP) to send messages over the wireless network. Messages will also sent to the Android app via Short Message Service (SMS) text messages.

The Android application will have a separate receiving thread that is constantly looking for packets from the bartender. When the bartender sends a message to the application, the thread checks what the message's purpose is and acts accordingly.

When a user sends a drink order to the bartender, the first character of the message tells the bartender's program that the message is from the Android and the bartender's PC then extracts the information from the message to store a drink.

The SMS messaging works using the Simple Mail Transfer protocol (SMTP). The C# code uses an email account setup using Gmail. The code needs to know the phone number of the app, the service of the phone (i.e. Verizon, Sprint) and the email address and password of the sending account.

How subsystem was tested

The Android application was tested by stepping through the program so that every possible path is covered to ensure there are no errors. Debugging was achieved by printing to a terminal to know where code was hanging and to print variables.

The Android application was tested and performed flawlessly on both the Android 2.2 and Android 2.3 platforms. The different platform simply required choosing a different target in the development software Eclipse.

Why engineering decision were made

The decision was made to develop an application on an Android smartphone as opposed to a different operating system such as an iPhone. This choice was made for a variety of reasons. A member of the AutoBev team had experience with Java, the online support for Android development, the open source "Linux-like" community and finally the Notre Dame Computer Science Department develops on Androids and was willing to lend a phone to the group for development purposes.

The choice to use the User Datagram Protocol (UDP) for the interface between the bartender PC and the Android phone because that type of protocol was used for the bartender and user interface already and was easily adapted for the phone – PC communication.

Description of overall flow of the code

The Android application has two threads constantly running: the main UI thread and the UDP server thread.

The main thread takes input from the user via the phone's touch screen. The thread is constantly waiting for user action and does not act until the user has clicked one of the six drink buttons. Once one of the drink buttons is clicked, the code checks whether the username and password are of the right form. The username must contain only letters and the password must be four numbers. If the username and password are not correct then the program asks the user to try again.

If the username and password are of correct form, a popup form asks the user to verify that his or her credit card will be charged the cost of the drink. If he or she clicks yes then a message will be sent to the bartender containing the username, password, phone number, IP address of the phone, and the desired drink of the user.

The bartender checks to see if the username and password are correct and then sends a message back to the phone stating that the drink order has been placed and gives the drink serving number the user is, along with the current drink being served.

The second thread is the UDP server thread that is constantly waiting for a message from the bartender. There are three types of messages which are distinguishable by the first character of the received message corresponding to: wrong password, username does not exist, and order placed successfully.

A while loop runs in the main thread until a message is a received which sets a Boolean flag to true. If a message is not received in a limited time (approximately five seconds) a popup box says that the bartender did not respond and to try again. This is to prevent the code from hanging on the "receive" if there is a communication breakdown.

Communication protocols

The communication protocol is described in great detail in the Bartender to User Interface section.
The message sent from the Android phone to the bartender is of the form: "&" + Phone Number + "#" + IP Address + "#" + username + "#" + pin + "#" + drink order + "#".
The message received by the Android phone from the bartender is of the form: "X#" + count[i] + "#" + count[0] + "#", AndroidIP. Where X tells the phone what type of message is received, count[i] is the user's order number, count[0] is the order currently served and AndroidIP is the phone's IP Address.

Beverage Dispensing and Sensing

After a drink has been ordered, this feature allows for drinks that do not need to be made by a bartender to be instantly available. The computer that serves as the user interface communicates with the microcontroller via a USB interface, which controls all functions associated with the keg. Customers are able to place their cup under a tap and pour their selected beverage once they have completed all necessary steps on the user interface. The mechanical system to control flow is created to connect to a keg of liquid. In order to keep a constant pressure in the keg, a gas tank with a constant pressure valve is used. A spigot, keg adapter, and tubing are also included in the keg setup. The solenoid valve is attached to the part of the tube close to the tap and is actuated by a digital output from the microcontroller. The microcontroller signals the solenoid to open if a customer has placed their order and is pouring their own drink. A flow sensor is also placed in the flow path to send a signal to the microcontroller which specifies how much drink has been poured for pricing purposes and to monitor flow in case the customer opts to have a specific amount of drink poured rather than to pay by ounce. We also have a limit switch by the tap to indicate that a cup has been placed there in the chance that someone is trying to pour a drink without a cup. Additionally, our current system of flow sensor, limit switch, solenoid, and microcontroller board are attached to the output of a Danby 5.8 Cu. Ft. Capacity Keg Cooler (Model # DKC645BLS). The keg cooler has a manual spigot that we are using as an emergency stop that is independent of the software in case anything goes wrong and we want to stop beverage flow. Upon completion of serving a drink, the microcontroller sends information back to the user interface so that the tab can be updated.

Note that our system is designed to be expandable and the microcontroller has circuitry to handle up to 2 each of the solenoid, flow sensor, and limit switch and has available I/O ports for up to a total of 4.. The actual system demonstration only utilized one of the sets.

The SwissFlow SF800 flow sensor was selected as the flow sensor for our system. In reviewing different sensors online, this sensor was used by others in measuring beer poured from a keg and was said to have accurate results. It also is made so that communication with a microcontroller is

feasible via a pulsed output. Furthermore, the people of SwissFlow were willing to send the group free samples, which allowed for us to be able to have extra sensors in case anything went wrong during testing. The SwissFlow SF800 flow sensor is supplied by a 12V source and requires a resistive network as seen in Figure 21 below. The output of the sensor is a pulsed output that generates 5600 pulses/liter. The flow sensor allows the microcontroller to know how much liquid has been poured. By counting the pulses our program is able to calculate the volume of the liquid that is passing through the tube.

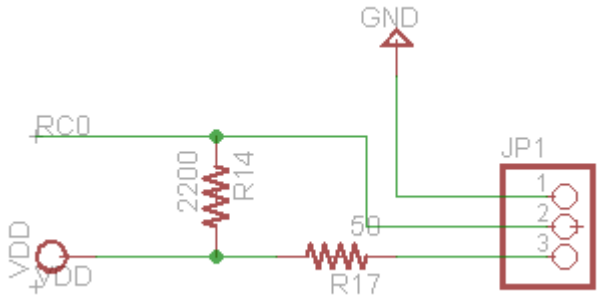


Figure 21 Flow Sensor Circuit

A limit switch acts as a safety for the system such that a beverage will not be dispensed unless there is a cup present beneath the spigot. A normally closed switch is connected to input pin RD1 and holds the pin at VDD until the switch is pushed where it sets the pin low to GND. The circuit can be seen below.

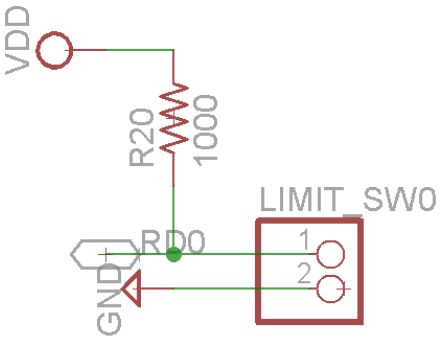


Figure 22 Limit Switch

Our solenoid of choice was the ECT 12 V solenoid (US5311162). The solenoid takes in digital input, which is necessary for communication with the microcontroller. Furthermore, the solenoid was provided to the group for free by Professor Mike Schafer of the University of Notre Dame. The solenoid serves as the main action unit as it will open or close to allow or stop the beverage from flowing. The ECT 12 V solenoid (US5311162) is a normally closed valve that will open when 12 V is applied to the coil. The internal resistance of the solenoid coil is about 23 ohms, therefore to switch open the solenoid requires about .53 amps. A power MOSFET, NDS355, was chosen appropriately for the coil driving circuit, because it can handle up to 1.7 amps and up to 30 V, and can be actuated by a 5V output pin connected to the ground through a 500 ohm resistor. This digital output has a corresponding indicator LED for troubleshooting so that the owner can check the status of the solenoid. The driving circuit can be seen in 24.

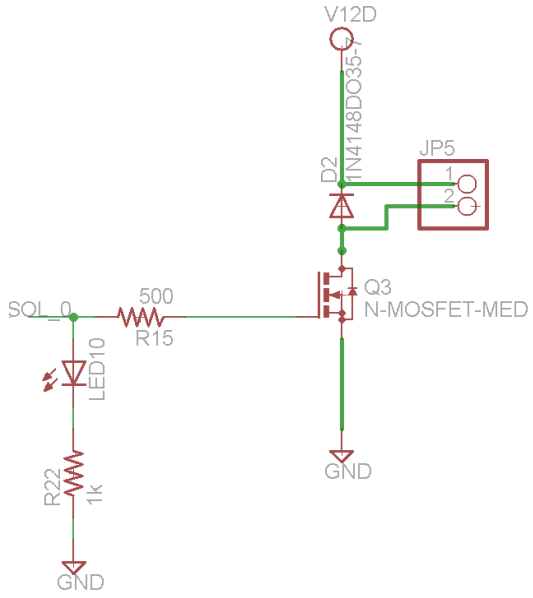


Figure 23 Solenoid Circuit

The control circuitry was tested using breadboards and similar components as the parts on the board. The solenoid MOSFET was connect as shown in the circuit above, and the input signal on the gate was ranged from 0-5 V to see when the transistor state would switch and allow current to flow. The actual required voltage was about 2.2 V so an output signal of 3.3 or 5V would be well enough to actuate the signal, and the resistor value was chosen to limit the current. The flow sensor circuit was tested by connecting the sensor to VDD through 250ohm resistor and the output was connected through a 2200 ohm in parallel to VDD and directly to an input on the logic analyzer. We then blew through the sensor and watched how the pulses would change based on flow. The final circuit was tested by measuring and noting that the amount of liquid poured was equal to the amount requested.

SQL Server Compact 3.5 Database:

Function

The SQL Server Compact Database is used to store transaction data. The data that needs to be saved includes the track number of the customer's credit card, the amount that customer will be charged, the username and the password for the Android application.

Interfaces to other subsystems

The SQL Server Database is exclusively accessed by the bartender's PC. Microsoft Visual provides an intuitive method in order to interface with databases using C#. After downloading the SQL Server 2005 Compact Edition by Microsoft, the database can be connected to the bartender's Windows Form Application using graphical tools.

The database and the Windows Form Application communicate using queries that conform to ANSI-SQL. Each query was pre-defined by the programmer and then later called in the Windows Form Application

similar to that of regular function all in C++ or Java. Queries come in various types including Select, Update, Delete and Insert. Each contributes to the overall function of the system.

How subsystem was tested

The SQL Server Database was tested by simply executing different queries and verifying that the output was as expected. This was accomplished by using the Query Builder within Visual Studio's database editor. The query builder allows the developer to graphically select which attributes and commands he or she needs to use.

A full integration testing was performed by running the bartender program and viewing the database throughout each stage of the program to ensure that each value was the correct one.

Why engineering decision were made

The SQL Server Compact edition was chosen because it is free to download and has outstanding support on the Microsoft Developer Network (MSDN). SQL is well known database computer language that has a short learning curve and has a lot of online support.

The database was placed on the bartender's PC rather than the customer's PC so that the bartender has easier access to the transaction data. The database was also put on the bartender side so that more than one kiosk and phone app could talk to the same bartender without each individual kiosk having its own credit card charging capabilities.

2.3 Interfaces and Sensors:

PC To Microcontroller Interface:

The PC to Microcontroller interface functions to connect the user interface to the microcontroller. A Universal Asynchronous Receiver/ Transmitter (UART) is used to transfer bytes between the two subsystems in accordance with the RS-232 standard. In order to convert the asynchronous serial messages sent by the microcontroller into standardized USB signals (recognized at the COM port of the PC), an intermediary is needed. The device used to do this is the FT-232-RI from FTDI. Professor Mike Schafer of the University of Notre Dame designed the circuit governing the operation of this device. The circuit was used with his permission and can be seen in the following figure.

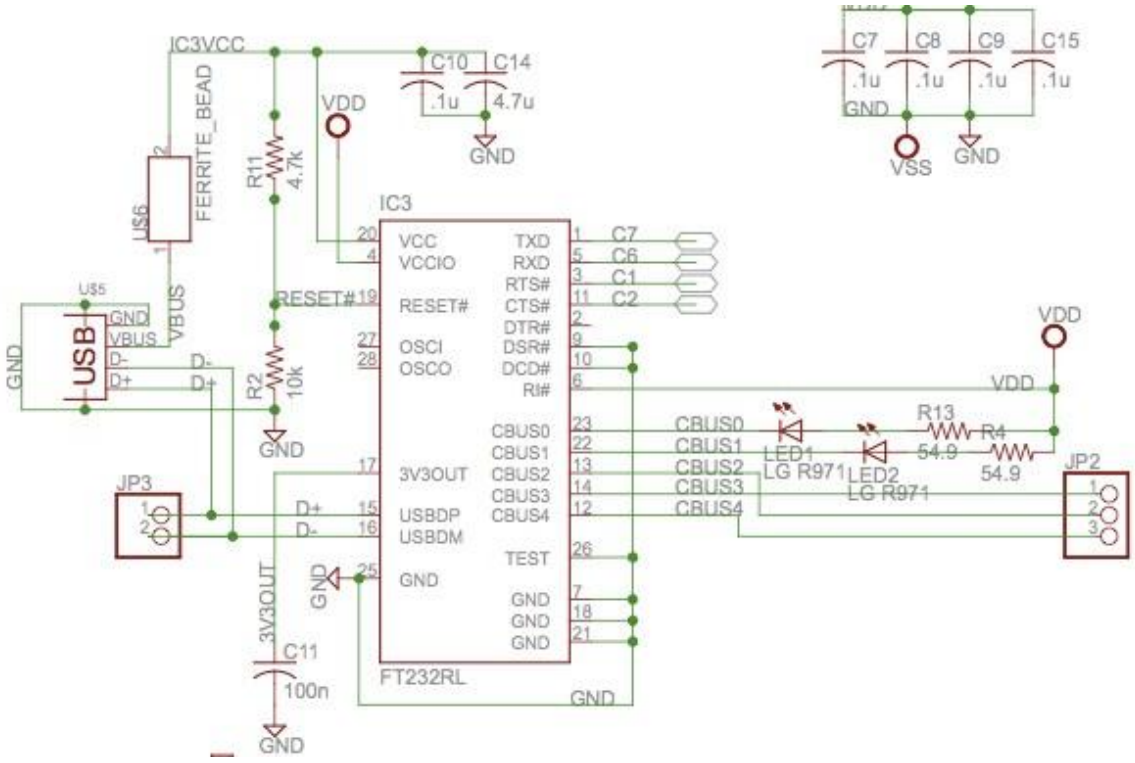


Figure 24 FT-232 RL Schematic

The FT232 is powered on pin 20 VCC by the USB connection, and the I/O pins are set to the voltage VDD of the voltage regulator output and connected to pin 4. Both of these values are 5V. The USB signal input/output is on pins 15 and 16 and the converted transmit and receive signals are on pins 1 and 5 respectively and go to the microcontroller pins RC7 and RC6. The state of these pins can be seen on CBUS0 and 1 and indicated on the board by a green (transmit) and yellow (receive) LED. Pins 3 and 11 are connected to the microcontroller pins RC1 and RC2 and held low to enable proper asynchronous communication.

The user interface communicates with the microcontroller using a single byte. The seven bits of this status byte indicate the status of the PC, the desired beverage size, if the user wants to start or stop pouring, and if the user has finished pouring.

The organization of the PC to microcontroller status byte can be seen in Table 6 below.

PC Microcontroller

Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
Success/Error	Message Type	Start or Stop	Order Type	Size 1	Size 2	Size 3	Filler
1 = success 0 = error (resend)	1 = start/stop 0 = new order	1 = start 0 = stop	1 = preset size 0 = pay per oz.	1 = 12 ounce 0 = no order	1 = 16 ounce 0 = no order	1 = 60 ounce 0 = no order	Reset Order

Table 6 PC to Microcontroller Status Byte

As can be seen in Table 6 above, the first bit of the status byte allows for indication of an error state by the PC. When this bit is zero, the microcontroller takes no action in response to the byte. It is assumed that an incorrect byte has been sent. Bit one of the status byte indicates the type of message that is being sent. If bit one is cleared, it is indicative that a new order is being placed. When this occurs, the microcontroller must check bits 3 through 6 to see to what the volume limit must be set. Only one of these bits is set in each status byte. If none of bits three through six are set when bit one is cleared, the microcontroller issues an error message to the PC and the user is asked to choose a size again. After the size has been set, the user interface sets bit one to indicate that the instruction is either to start or stop pouring a beverage. The microcontroller continues to listen to PC status bytes to detect when the user wants to stop pouring (bit 2 cleared). Bit seven is used to indicate when a user has finished pouring. When this bit is set, the variables in the microcontroller are reinitialized to their starting value and the microcontroller begins waiting for a new order from the PC.

The Microcontroller sends three bytes back to the user interface. The first byte contains the status of the beverage dispensing system, and the second two bytes hold a sixteen bit number that corresponds to the number of pulses that have been detected. The organization of these three bits can be seen in Table 7 through Table 9 below.

Microcontroller / PC

Byte 1 (Status):

Bit 0	Bit 1	Bit 2	Bit 3-7
Success/Error	Status	Cup Sensor	Filler
1 = success 0 = error (resend)	1 = still pouring 0 = done pouring	1 = cup present 0 = no cup	DC

Table 7 Status Byte from Microcontroller to PC

Byte 2 (Last 8 bits of 16-bit value of volume):

Bit 8-15
Bits 8-15 of Volume Poured Value (pulses)

Table 8 Lower Eight Bits of Volume (in terms of pulses)

Byte 3 (First 8 bits of 16-bit value of volume):

Bit 0-7
Bits 0-7 of Volume Poured Value (pulses)

Table 9 Upper Eight Bits of Volume (in terms of pulses)

Table 7 through Table 9 show the organization of the bits in the three bytes from the microcontroller to the PC. The first byte represents the status of the microcontroller. When bit zero is cleared, this indicates that the microcontroller has encountered an error. When bit zero is set, the microcontroller is successfully functioning. Bit one is set when the microcontroller is in a state of pouring. This occurs when a 'start pouring' instruction has been received from the user and a cup has been detected by the

limit switch. Only under these two conditions will bit one be set. Bit two is used to indicate when a cup is present. If the limit switch detects that there is no cup, both bit zero and bit two are cleared and the status is sent to the user interface which instructs the user to place their cup and press start again.

The second byte contains the upper eight bits of the amount of volume that has been dispensed. The third byte contains the lower eight bits of the amount of volume that has been dispensed. The volume is in units of flow sensor pulses. Every fifteen pulses (0.091 ounces) the microcontroller reports its status and the amount of volume dispensed back to the user interface. The microcontroller also sends these three bytes of data any time a missing cup is detected or when the volume limit has been reached.

How the interface tested

The interface was tested by once again using the debugger in Visual Studio. A breakpoint was placed on the function that executes when asynchronous data is received through the serial port. When the microcontroller sends the three status bytes, this function is called. It is then possible to open the buffer that holds the received data and check the individual bits of the status. This was done multiple times to ensure functionality. Additionally, pop-up message boxes were inserted into the user code that displayed the Pc to microcontroller status byte. These bytes were verified for every possible message that could be sent in either direction.

Design decisions

The number of status bytes used to send information from the user interface to microcontroller and from the microcontroller to the user interface was based on the desire to keep the protocol as simple as possible. In both cases, the minimal amount of bytes that could be used to communicate the necessary data was used.

Bartender To User Interface

The user to bartender interface functions to convey drink orders, 'pour your own' charges, and new usernames and passwords from the user interface to the bartender interface.

As detailed in Section 2.2 a UDP Server/Client protocol was chosen for communication between the two computers, the User and Bartender hub, over a local area connection. This enables two way communication between the bartender and user. This could also be done using a TCP protocol, the amount of information being communicated is extremely small and will only need to be sent occasionally in packets, which is ideal for UDP. UDP applications use datagram sockets to establish host-to-host communications. An application binds a socket to its endpoint of data transmission, which is a combination of an IP address and a service port.The basic flow behind the UDP Server/Client can be seen in Figure 25 below. The protocol was written in C# so that it could easily be integrated into the user and bartender programs.

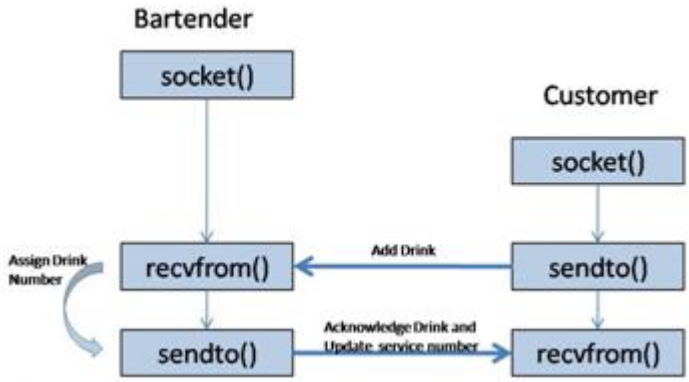


Figure 25 Flow of Client/Server

The client UDP protocol program is embedded within the UI program as a separate function on the MixedDrinksForm. Upon selecting a mixed drink order on the form, the client function is called with the selected drink as an input. The function sends a string message to the server that included the selected drink. The client UDP protocol program that was embedded into the Android application is similar and was discussed in Section 2.2.

The server UDP protocol program is embedded within the bartender program. It has the capability to open multiple communication threads in order to allow the owner to add multiple machines to the system. This was made possible by the use of multithreading in the program. Multithreading also made it feasible for the application to remain responsive to input, while still being able to add and clear orders from the drink list. In a single-threaded program, if the main executions thread blocks on a long running task, the entire application can appear to freeze. The initial program was experiencing this problem and, as a result, was not printing orders beyond the first one placed to the drink list. It was still processing the orders, but was not able to add them to the list and make them visual to the bartender. By moving such a long running task as the ability to continuously take in drink orders to a worker thread that runs concurrently with the main execution thread, it is possible for the application to remain responsive to user input while executing tasks in the background. Once multithreading was integrated into the server program the drink list which was called *DrinkQueue* in the program was able to add each new drink order processed to the list.

There are four possible types of messages that the user may send to the bartender over the UDP client/server. The first type of message is a specialty drink order. The second type of message is a 'pour your own beverage' order. The third type of message is the setting up of a username and password by a customer to be later used with the Android Smartphone application. The last type of message is a request by the user for an updated list of drinks and prices. The bartender can also receive a string message from an Android Smartphone through the UDP connection. In all five cases, the message that is sent from the user to the bartender is a string. This ASCII code for the string is encoded in bytes and sent via the UDP connection to the bartender where it is decoded. The bartender responds to the message based on the first character that is sent. Each order string is further subdivided into several other strings separated by a pound sign. Based on the first character in the string, the bartender is able to decode and use the information in the below shows the organization of these substrings.

Start Character	String 1	String 2	String 3	String 4	String 5	Order Type

#	Drink Name	Track Number	Cost	Null	Null	Kiosk Order
@	Track Number	Pulse Count	Null	Null	Null	Pour Your Own
$	Track Number	Username	Password	Null	Null	Android App Initialization
&	Phone Number	IP Address	Username	Pin	Drink	Phone Order
!	Null	Null	Null	Null	Null	Price Request

Table 10: String Format for Orders between Bartender and User

Table 10 above shows the format used for the strings sent from the user to the bartender. Each string is separated by a pound sign. Using the pound sign as a delimiter allows the bartender to extract the necessary information from each user request. Depending on the type of request, the bartender responds with a message in string format. It also updates the local database as needed.

The above protocol was decided upon to allow for multiple types of orders to be sent to, and processed by, the bartender interface. Without the use of a start character and a delimiter character, it would not have been possible to send five different types of messages to the bartender. In the above table, a null string simply means that no string is sent. Thus, for a Price Request order type, a single character is sent from the user interface to the bartender interface. This was meant to keep the data transmitted to a minimum.

The system was tested by sending each type of order from the user interface. The entirety of the message was first displayed to make sure that the whole string was sent successfully. Next, the individual substrings were displayed on a list in the bartender interface. Finally, pop-up message boxes were added to the user interface to show the string message that the bartender sends back in response to an order. These messages were unique to the type of order request. Thus, it was possible to detect if two way communication was successful.

3 System Integration Testing

3.1 How the integrated set of subsystems was tested

After the design and testing of the subsystems and their respective interfaces was complete a system integration test was required to ensure that the entire system would perform as expected with each subsystem connected to one another.

The overall system integration test involved completing similar testing that is detailed in section2.2: Detailed Operations of Subsystems while the system was fully connected.

In order to determine if the system was working properly, every possible path through the user interface was each tested ten times to ensure that every path through the software is bug free. To test system robustness, the bartender system was sent many orders by two different kiosks and the Android application simultaneously. If the bartender was able to communicate with the three clients for over 50 orders then the system was deemed robust enough to handle maximum demand.

3.2 How the testing demonstrates that the overall system meets the design requirements

The following table details the design requirements for the overall system and explains how the testing procedures ensured proper functionality.

System Requirement	How Requirement was Tested
Capable of receiving and processing drink orders digitally	Bartender interface received 100% of drink orders placed by customer interface and Android application.
Must use simple user commands for navigation (single click)	The simplicity of the customer user interface was tested by allowing an individual outside the design team to use the system.
Must have layer of protection between CPU and other sensitive electronics and users/ beverages	Throughout the testing the system was shown to be 100% leak proof.
Magnetic card reader must be able to scan credit cards and send information to the PC	The magnetic card reader was able to login the user by the swipe of a credit card. Non credit cards would not allow the user to enter into the system.
Customer and Bartender UI must run on Windows PC from executable	The customer and bartender UIs ran on several different PCs using a single executable file.
Board must be powered by 12V power brick which is regulated to 5V	12V was present at the pins of the solenoid and 5V was present at various test points throughout the board.
Microcontroller must be able to control 12V solenoid	Solenoid turned on and off when it was supposed to during testing, this allowing liquid to flow.
Flow sensing system must detect accurate flow rate and return number of ounces poured.	Several tests determined that the amount poured measured out to be the amount selected.
Limit switch must detect if cup is present underneath spigot.	Several tests were completed to ensure that the system would not pour and would stop pouring when the cup was not present.
UI on PC must be able to communicate with microcontroller by sending bytes over an RS232 protocol	Because tests were completed without error or improper execution, this implied that the correct messages were being sent and received over the serial interface.
UI on bartender must connect to customer UI an d Android using network UDP protocol	Each order was successfully sent from the Customer via the kiosk and the Android application.

Table 11 System Requirement Testing

4 Users Manual/Installation manual

Thank you for purchasing the AutoBev System. We promise that this system will dramatically improve service at your establishment. If you recently purchased an AutoBev System you have everything you need to install and integrate one AutoBev kiosk into your current bar operations. If business is booming and you need to expand, please order additional kiosks at http://seniordesign.ee.nd.edu/2011/Design%20Teams/AutoBev/AutoBev_Base.html.

The original AutoBev System includes:

- One Bartender system including:
    - Computer and monitor (with all necessary connectors)
    - Mouse and keyboard
- One AutoBev Kiosk including:
    - Computer and monitor (with all necessary connectors)
    - Mouse (if monitor is not a touch screen)
    - Credit Card Reader
    - Bartender Access Card
    - Refrigeration Unit
    - Tubing and Keg Connections
    - CO_2 tank
    - Control Hardware

With the AutoBev system you will reap many benefits including:

- Shorter and fairer wait times
- Increased efficiency
- Easy expansion
- Increased order accuracy
- Ability to received orders from Smartphones
- Decreased employee workload

4.1 How to install your product

Installing the AutoBev System is as easy as booting up two computers and connecting a keg. Follow these basic steps in order to have your system fully integrated in no time at all.

Bartender System

1. Find a place behind the bar for the Bartender System with access to a wall outlet. Space reserved should be large enough to fit a computer box, a monitor, a keyboard, and a mouse.
2. Plug in the computer labeled “Bartender” into the wall outlet.
3. Plug in the keyboard, mouse and monitor into the computer.
4. Turn on the computer and monitor.
5. On the desktop double click on the executable called “Bartender Interface”.
6. The first form encountered is a list of drinks and prices. These may be updated in real time as often at the bartender desires (set up initially here, and then can be changed by clicking the ‘Update Prices’ button on the main bartender form.

AutoBev Kiosk

1. Find a place in your bar near a wall outlet from which you would like your customers to order. It is recommended for this location to be in a low-traffic area of the bar that is easily accessible to customers.
2. Plug in the main power into the wall outlet.
3. Plug in the mouse (if not touch screen) and monitor into the computer box.
4. Turn on the computer and monitor.
5. On the desktop double click on the executable called “User Interface”. Customers may then begin a session by swiping their credit card.
6. Connect the CO2 tank and keg according to Dispensing hardware instructions.

Dispensing hardware (how to replace a keg)

1. Attach 5/8” hose to pressure regulator and secure with the red pinch clamp
2. Attach C02 tank to the pressure regulator via the hex nut connector, make sure the regulator lever is perpendicular to the hose connection.
3. Run hose through the hole on the back of the refrigerator unit, and place CO2 tank in the metal bracket.
4. Attach the 5/8” to the keg connector and clamp to seal.
5. Connect 3/8” hose from the spigot to the Solenoid / Flow Sensor unit with the arrow on the flow sensor pointing in the direction of flow.
6. Connect Solenoid/Flow Sensor to the keg connector and clamp to seal.
7. Push and turn keg connector to the keg output and push down lever to seal.
8. Open CO2 tank, and move regulator lever to the downward position(in line with hose) and adjust regulator to 12psi.

What you need to Provide:

1. Local wireless network that both the AutoBev Kiosk and Bartender Interface are connected to.
2. A keg for the kiosk
3. Lots of Thirsty Customers

4.2 How to use your product

Since the AutoBev System comes completely configured and ready to use, if you have followed the above installation steps you are ready to go!

Using the AutoBev system is very intuitive. Customers order drinks at the AutoBev Kiosks and with their Smartphones. When orders come in, the Bartender System will display the order in the order in the order in which they were requested. which orders should be made in what order.

As orders are filled, the bartender will see a list of drinks that need to be made. After the bartender makes a drink he or she will click on the respective drink and hit “clear”. The customer will be alerted either via text message or by a screen displaying the order number currently being served.

At the end of the night the bartender will press the “Save” button in order to send the transaction information to the respective financial institutions to charge the credit cards stored in the database. The bartender may also view transaction data from previous dates by clicking “Load” on the Bartender Interface.

The AutoBev System is an isolated system so that it does not interfere with your current data handling and transaction operations. If you desire to have these integrated you must call the AutoBev support team and they will assist you in fully integrating AutoBev with your current system.

4.3 How the user can tell if the product is working

The AutoBev System error checks itself so that when a piece of hardware or software malfunctions, an error popup box will tell the user exactly what went wrong. Many times a hard reset is all that is needed for the system to start working properly.

On the Welcome Form of the User Interface:

When the user interface is opened, the application tries to connect with a server application. If a server is detected, the user is allowed to swipe their card and begin the drink ordering process. However, if a server is not detected, and error message that reads “Server is currently down, please alert a bartender” appears. When a bartender sees this message, the first thing that should be done, is to press the ‘Retry Connection’ button in the lower right hand corner. This button runs the function that searches for an active server. If a server is found, the proper welcome form will be displayed. If not, the problem is most likely that the server application is not running on another computer within the bar. The bartender should then go to the location of the server computer and make sure it is running. It is probable that one of these fixes will solve the problem.

On the Type Selection Form:

When the Type Selection Form is opened, the user is presented with three options. One of these options is to ‘Pour Your Own” Beverage. In order to do so, the user interface must be connected to the device running the beverage dispensing and flow sensing software. Therefore, before the form allowing a user to choose a size of beverage to dispense is allowed to opened, the application checks for a valid connection. When this does not exist, the Type Selection Form changes appearance and the error message “Sorry for the Inconvenience. Beverage Dispensing Unavailable. Please Alert a Bartender” appears. When this happens, the user should immediately request for the help of a bartender. The bartender has a few options. The first is to press the “Retry” button located in the lower left hand corner. This will recheck for a steady connection between the beverage dispensing and flow sensing hardware. If one is found, the user will be allowed to proceed to a form where he or she is allowed to choose the type of dispensing (either pre-selected size, or pay per ounce) for the order. If a connection is not found, the error message will remain on the form. When this happens, the bartender should ensure that the serial port (connecting the dispensing hardware is connected). If it is, the bartender should remove the USB connection from the computer and then plug it back in. If neither of these fixes works, the bartender should check to ensure that the hardware is being provided with power, by making

sure the regulator is plugged into the power strip, and to the power port on the board. If there is no power, the device needs to be plugged in. It is more than likely that after these three things are checked, the problem will be solved.

On the Checkout Form:

On this form, the user will receive one of two messages. If the user has ordered a specialty drink, the user is informed of their order number, how many drinks are to be made before their own, and how much they have been charged. When these three things appear on the form, an order was placed successfully. The other type of message occurs when the user has poured their own beverage. In this case the user will be informed of the amount charged to their card for that purchase. When these messages occur, the user can be confident that their order was completed.

On the Android Application:

When a user places an order from a Android Smartphone, he or she is required to provide the username and password that were established at an AutoBev kiosk earlier in the night. When the order is placed, the username and password, along with the drink order, are sent to the bartender interface where it is first verified that the username exists. The password associated with that username is then checked against that which was sent from the phone. When both of these two strings are identical, the drink order is placed. However, if the username is not found within the local database, the message "Username Does Not Exist" is sent to the phone and appears in a pop-up message box on the Android Smartphone. If the password associated with the sent username does not match that which is stored in the database, the message, "Password incorrect for existing username" appears in a pop-up box on the Android Smartphone. If either of these two messages occur, the user should re-enter their information and try ordering again. If this still does not work, the user may swipe the card that was used to set up the username and password at an AutoBev kiosk and proceed to the "Username" form. Here, the user may try to set up a new username and password. When the user clicks the "Save" button, a request to set up a new username and password is sent to the bartender. If it is determined that the user's card number is already associated with a username, the user is notified of the username and password in the form of a pop-up box that appears on the username form. It is important to note, that each night the user will be required to set up a new username and password at the kiosk. This is done for security reasons. It also makes it possible for the user to associate various night's transactions with different credit cards.

Overall, it is fairly intuitive to know if the AutoBev System is working correctly. When a drink order is placed from either an AutoBev kiosk or from an Android application, the drink appears in a queue on the bartender interface and a confirmation is send back to the user informing them that a successful order has been placed. Otherwise, an error message is sent to the user. When the user is sent an error message, the type of error that has occurred is sent as the message so the user knows exactly what has gone wrong. The bartender can use the error message to provide the user with assistance.

4.4 How the user can troubleshoot the product

The user can troubleshoot the product using error messages that appear when a failure occurs. These error messages are often self explanatory such as “No Server Available,” “Beverage Dispensing Unavailable,” and “Username/Password” incorrect. These messages are associated with specific areas of the application and narrow down the possible sources of error. When an error message associated with a specialty drink request is encountered, the connection between the bartender and user should be checked and then re-established. When an error message associated with a “pour your own” beverage request is encountered, the hardware connections between the user interface and the beverage sensing and dispensing hardware should be checked to ensure that they are stable.

AutoBev technical support is available during normal business hours to field any questions and direct the user through the troubleshooting process.

5 To-Market Design Changes

The following list of changes would be made before the AutoBev System was taken to the commercial market. Some features are required before deployment while others are desirable but not necessary.

1. Change: Android application to be able to update the drink list and drink prices.
   a. Feasible: 100% feasible. Some time was spent trying to implement this feature with some success; however, the feature was not completely finished to be shown in the prototype.
   b. Importance: This feature is absolutely necessary for the customer to know which drinks are being offered at which price. It is also necessary for the bartender to know the desired order.
2. Change: Touch screen for AutoBev kiosk.
   a. Feasible: 100% feasible. The software will work with a touch screen because touch screens emulate clicks of a mouse.
   b. Importance: This feature is desired so there is no mouse at the kiosk; however, it is not vital for the overall functionality of the system.
3. Change: Integrate with the current transaction tracking system in the establishment.
   a. Feasible: This is feasible; however, the feature will require technical support.
   b. Importance: This feature is desired so the bar has all the transactions on one system; however, it is not vital for the overall functionality of the system because the AutoBev System may run in complete isolation.
4. Change: Charge credit cards.
   a. Feasible: 100% feasible. Will require technical support to interface the AutoBev System with the financial transactions.
   b. Importance: This feature is absolutely necessary for the system to function as it is intended.
5. Change: Cognitive testing. Must be able to determine if the customer is sober enough to have another drink.
   a. Feasible: Not very feasible. The only way to truly tell if someone has consumed enough alcohol is with human inspection. The AutoBev System must remain in view of the bartender so the bartender can use his or her own discretion.

   b. Importance: This feature would be more of entertainment than a necessary function. Not very important.
6. Change: AutoBev kiosks and Android application to be able to determine IP address of Bartender automatically.
   a. Feasible: Extremely feasible. Several networking schemes exist that can accomplish this.
   b. Importance: This feature is desired to reduce the complexity of installation; however, it is not vital for the overall functionality of the system.
7. Change: Added security and encryption to the transferring of sensitive financial data and usernames and passwords
   a. Feasible: 100% feasible, could use any known secure transfer protocols such as Kerberos or for communication between bartender/kiosk could use a internally generated encryption key only known to the system
   b. Importance: This would be critical so that people would know that their information is secure and eliminate thievery.
8. Change: Add custom encasements for monitor/microcontroller and power management.
   a. Feasible: 100% would require a simple design and custom construction, but not affordable unless cost amortized over many units, not possible with time/budget constraint
   b. Importance: adds a sleek protective case for the unit and looks more attractive

6 Conclusions

The AutoBev system is the first step in automating the bar experience. AutoBev looks to innovate bartending by offering a seemless order and queuing system paired with self-serve beverage dispensing. AutoBev accounts can be created and accessed at any AutoBev kiosk with the swipe of a credit card and will track ordering activity, keep a running tab, and process a single transaction per customer at bar's close. This will reduce transaction fees, save a bar both time and money, and eliminate the risk of thievery. Speeding up the bartending process will allow bars to serve more drinks, and will reduce wait time for patrons, thus enhancing the overall bar experience. AutoBev kiosks can be specialized to any establishment with simple processes to change drink lists and prices. AutoBev is simple to use and would be a great addition to any drinking establishment and the expandability of the system to include additional self-service drinks, specialized drink lists and prices, process real transactions, and securely store data give the product a useful real life application.

7 Appendices

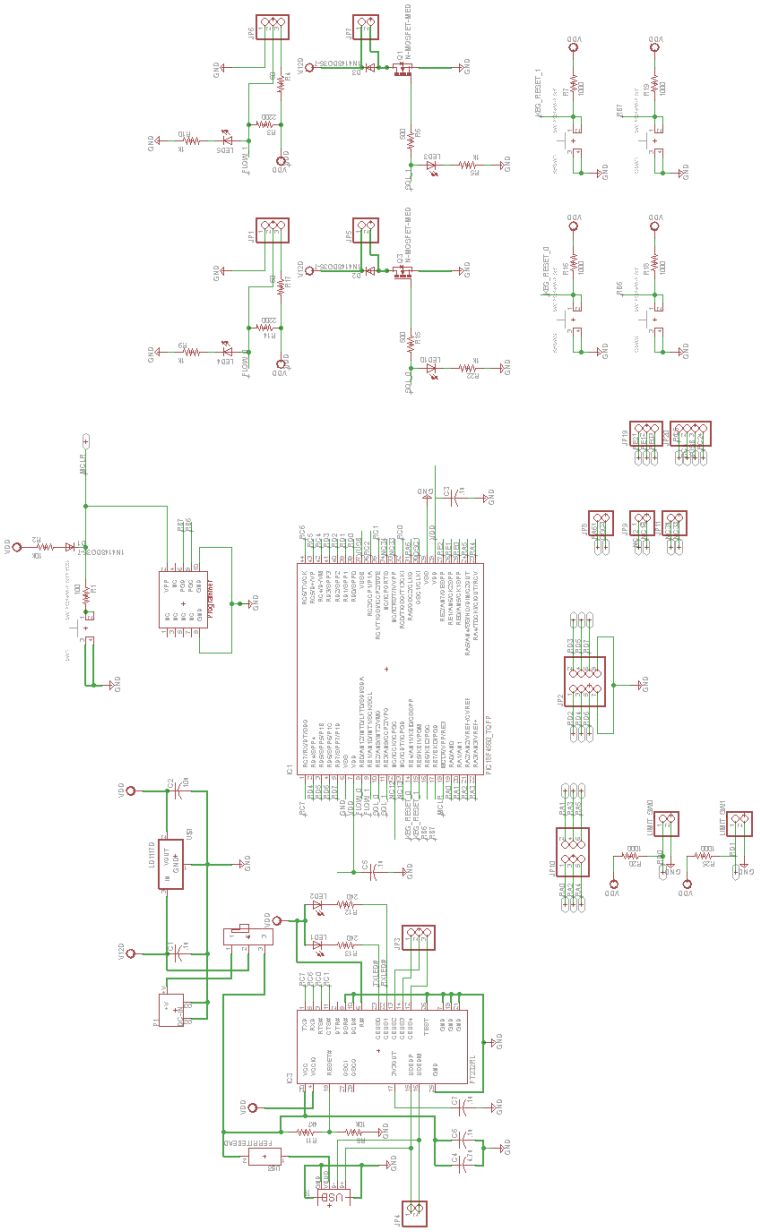
7.1 Bill of Materials

Subsystem	Description	Distributor	Part #	QTY	unit$	Total Cost
FS&CC	SwissFlow 3000	Swissflow		2	0.00	$0.00
FS&CC	ETI 12V Solenoid (US5311162)	stock		2	0.00	$0.00
FS&CC	MOSFET N-CH 30V 1.7A SSOT3	Digi-key	NDS355ANCT-ND	2	0.51	$1.02
FS&CC	50 ohm Resistor	stock		2	0.00	$0.00
FS&CC	2200 ohm Resistor	Digi-key	RHM10.0KCRCT	2	0.00	$0.00
FS&CC	500 ohm Resistor	stock		2	0.00	$0.00
FS&CC	Diode 1N4001	Digi-key	1N4001FSCT-ND	2	0.31	$0.62
FS&CC	toggle switch	stock		5	0.00	$0.00
FS&CC	Ander-Lign Compression Connector (Brass)	Home Depot		2	2.19	$4.38
FS&CC	Clear Vinyl Tubing	Home Depot		1	2.98	$2.98
FS&CC	1000 ohm Resistor	stock		11	0.00	$0.00
FS&CC	Danby 5.8 Cu. Ft. Capacity Keg Cooler	loaned		1	0.00	$0.00
FS&CC	12v voltage Power Brick	provided in kit		1	0.00	$0.00
Card Scan	Unitech America MS240 Mag Stripe Reader, MSR Track I, II&III, USB	Unitech America (via provantage.com)	Manufacturer Part# MS240-1T2	1	37.18	$37.18
Micro	slide DPDT switch	Digi-key	SW116-ND	1	0.81	$0.81
Micro	300 ohm SMD resistor	Digi-key	P300DACT-ND	10	0.20	$2.04
Micro	10k SMD resistor	Digi-key	P10KDACT-ND	2	0.00	$0.00
Micro	4.7k smd resistor	Digi-key	P4.7KDACT-ND	1	0.00	$0.00
Micro	.1uF smd capacitor	Digi-key	ECJ-2VB1E104K	7	0.00	$0.00
Micro	10uF capacitor	Digi-key	PCC2225CT-ND	1	0.00	$0.00
Micro	4.7uF capacitor	Digi-key	PCC1842CT-ND	1	0.00	$0.00
Micro	red led	Digi-key	160-1422-1-ND	1	0.00	$0.00
Micro	green led	Digi-key	160-1423-1-ND	9	0.00	$0.00
Micro	diode	Digi-key	1N4148WTPMSCT-ND	1	0.00	$0.00
Micro	ferrite beed	Digi-key	445-2201-1-ND	1	0.00	$0.00
Micro	5v voltage reg.	Digi-key	497-1235-1-ND	1	0.00	$0.00
Micro	20MHz ceramic resonator	Digi-key	490-4717-1-ND	1	0.00	$0.00
Micro	USB connector	Digi-key	609-3656-ND	1	0.00	$0.00
Micro	FT232RL-REEL	Digi-key	768-1007-1-ND	1	0.00	$0.00
Micro	PIC18L4320-I/P Microcontroller	Digi-key		1	0.00	$0.00
Micro	Power Connector	Digi-key	CP-002A	1	0.00	$0.00
Micro	Board Programmer	Provided in kit		1	0.00	$0.00
Bartender	LC Computer	Provided		1	0.00	$0.00
User Interface	LC Computer	Provided		1	0.00	$0.00
P & Flow	CO2 Tank refill	Action Cycle	NA	1	$20.530	$20.53
P & Flow	Rootbeer Keg and attachements	Simonton Lake Drive in	NA	1	$24.530	$24.53
P & Flow	5/8" 4'x8' composite board	Home Depot	77391400001	1	$10.670	$10.67
P & Flow	1/8 Hexnipple adapter	Home Depot	48643067561	2	$1.570	$3.14
P & Flow	Stretch and Seal Tape BLK	Home Depot	742366006295	1	$6.870	$6.87
P & Flow	10' 3/8" vinyl tubing	Home Depot	48643025523	1	$3.110	$3.11
P & Flow	teflon tape	Home Depot	78864177206	1	$0.780	$0.78
P & Flow	3/8" tubing barb adapter	Home Depot	48643071025	2	$2.290	$4.58
P & Flow	SS Clamp	Home Depot	78575126029	3	$0.850	$2.55
P & Flow	1/8- 1/4 reducer adapter	Home Depot	48643072169	2	$2.550	$5.10
					TOTAL:	$130.89

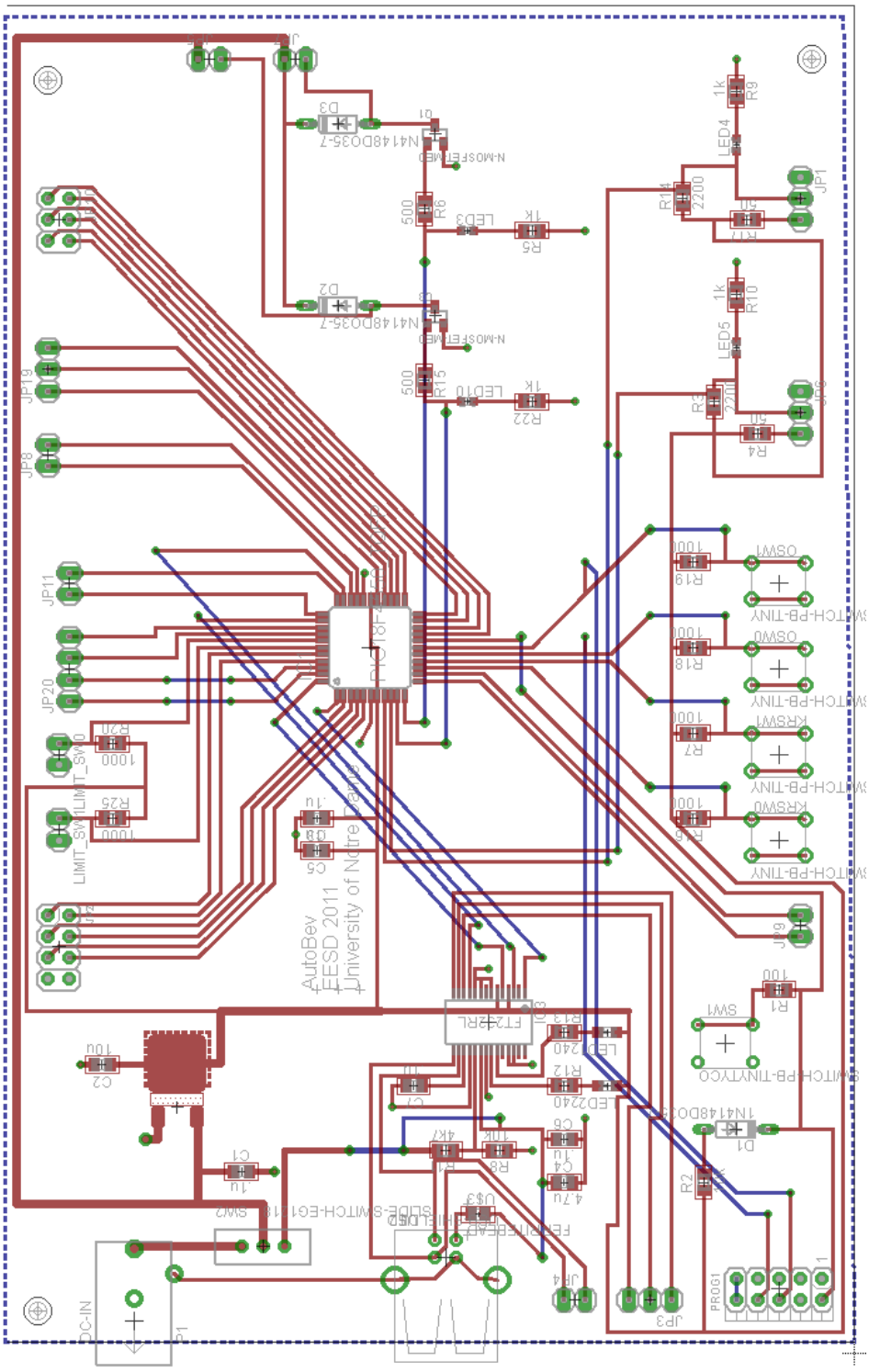
Datasheets:

- PIC18F4320- http://ww1.microchip.com/downloads/en/devicedoc/39599c.pdf
- FD233R- http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232R.pdf
- NDS355- http://pdf1.alldatasheet.com/datasheet-pdf/view/54177/FAIRCHILD/NDS355.html
- SwissFlow800- http://www.swissflow.com/sf800.html
- Solenoid- http://ip.com/patent/US5311162

7.2 Complete hardware schematics



7.3 Complete Board Layout



7.4 Microcontroller Code

```
#include <system.h>
#include "AutoBevLib.h"

#pragma DATA  _CONFIG1H, 0b11001001
#pragma DATA  _CONFIG2H, _WDT_OFF_2H
#pragma DATA  _CONFIG3H, _MCLRE_ON_3H
#pragma DATA  _CONFIG4L, _LVP_OFF_4L

////////////////////////////////////////////
//    AutoBev Senior Design Project       //
//                                        //
//    AutoBev.c                           //
//  Main code for the project             //
////////////////////////////////////////////

//--// Voltatile bit Declarations


      volatile bit toint@INTCON.2;          // Timer 0 Interrupt Flag
      volatile bit intedge1@INTCON2.5;  // Interrupt on rising edge
      volatile bit int1IF@INTCON3.0;            // Int 1 interrupt flag to Detect Flow Sensor Pulses
      volatile bit cren@RCSTA.4;            // For enabling reception of the USART interrupt
      volatile bit rcif@PIR1.5;             // Flag to detect received data

//--// Global Interrupt Semiphores

      bool increaseVolume = false;          // Flow Sensor Pulse Interrupt

//--// Other Semiphores
      bool volFlag=false;                   // Flag For Reaching Volume
      bool cupPresent = false;              // Cup Present for Dispensing


////////////////////////////////////////////
//    Function: interrupt
//    Purpose:  The interrupt service routine
//    Input:      Void
//  Output:   Void
////////////////////////////////////////////
void interrupt(void){
      if(int1IF){
            increaseVolume=true;                // Indicate that there is something to do
            int1IF=0;                           // reset interrupt flag
      }
}

//--// MAIN FUNCTION
void main(void)
{
      //--// Setup I/O Pins

            intcon=0b11000000;                  // enable global interrupts
            intcon2=0b00110000;                 // enable intedge interrupt
            intcon3=0b01001000;                 // enable interrupt flag
            adcon1=0b00011111;                  // make ports digital inputs
```

```
    cmcon=0b00000111;           // configure the comparators
    osccon = 0b01110000;        // set up oscillator control

    trisb.0 = 1;
    trisb.1 = 1;                // Pulse input from flow sensor
    trisb.2 = 0;
    trisb.3 = 0;                // Output to solenoid valve
    trisb.4 = 1;                // Keg override button
    trisb.5 = 1;                // Keg replacement override
    trisc.0=0;
    trisc.1=0;
    trisd.0 = 1;
    trisd.1 = 1;                // Input from limit switch
    trisd.6 = 0;
    trisd.7 = 0;

    latb.2 = 0;
    latb.3 = 0;                 // Close solenoid to begin
    latc.0 = 0;
    latc.1 = 0;
    latd.6=0;                   // Output for simulate pulses
    latd.7=0;                   // Output for testing


//-// General Variables
    int inc = 1;                // set increment value
    int limit;                  // set volume limit (# of pulses before increment)
    int count_2= 0;             // for reporting status back to PC
    char PIC_status;            // byte for status of microcontroller
    char upper;                 // for upper 8 bits of volume poured
    char lower;                 // for lower 8 bits of volume pourted
    long volume = 0;            // Volume Dispensed
    bool keepPouring = false;   // indicates it is ok to keep solenoid valve open
    unsigned char receivedByte; // status byte from PC
    unsigned short volumeLimit; // Volume Limit in Pulses

//--// Reset Status Byte
    PIC_status=0;

//--// USART Initialization
    init_usart();               // Initializes USART @ 9.6k baud rate


//--// USART Initialization//
while(1){
    //--// Instruction Checking Routine
    char receivedByte = receiveByte();          // Received byte from PC

    if (!(((receivedByte>>0))&0x01)) {          // Bit 0 cleared indicates error
        PIC_status.0=0;
        volume = 0;
        sendByte(PIC_status);                   // Send error Status to PC
    }
    //--// Size Check Routine
    else if (!(((receivedByte>>1))&0x01)) {     // Bit 1 cleared indicates new order
        if (!((receivedByte>>3))&0x01) {
            volumeLimit = 29810;                // Pulses in 3 pitchers (180 oz)
```

```
            volume = 0;                     // Initialize Volume
            volFlag = false;
            PIC_status.0=1;
            PIC_status.1=1;
            PIC_status.2=1;
            sendByte(PIC_status);
        }

        else if (((receivedByte>>4))&0x01) {
            volumeLimit = 331;//1988;       // Pulses in 2//12 oz
            volume = 0;
            volFlag = false;
            PIC_status.0=1;
            PIC_status.1=1;
            PIC_status.2=1;
            sendByte(PIC_status);
        }
        else if (((receivedByte>>5))&0x01) {
            volumeLimit = 663;//2650;       // Pulses in 4//16 oz
            volume= 0;
            volFlag = false;
            PIC_status.0=1;
            PIC_status.1=1;
            PIC_status.2=1;
            sendByte(PIC_status);
        }
        else if (((receivedByte>>6))&0x01) {
            volumeLimit = 994;//9937;       // Pulses in 6//60 oz
            volume = 0;
            volFlag = false;
            PIC_status.0=1;
            PIC_status.1=1;
            PIC_status.2=1;
            sendByte(PIC_status);
        }
        else {                              // No size selected--an error in the byte sent
            PIC_status.0=0;
            PIC_status.1=0;
            PIC_status.2=0;
            sendByte(PIC_status);
        }

    }
    //--// End Size Setting Routine

    //--// Start/ Stop Routine
    else if (((receivedByte>>1))&0x01){     // Bit 1 set indicates start or stop pouring

        if (((receivedByte>>2))&0x01){      // Bit 2 set indicates starting to pour
            PIC_status.0=1;
            PIC_status.1=1;

            if ((portd.0==true)){           // Check for cup present
                latb.2=1;                   // Open solenoid valve
                keepPouring = true;         // Allow for pouring
                PIC_status.2=1;             // Cup Present
                PIC_status.0=1;             // No error in pouring
```

```
                }
                else {
                    latb.2=0;                   // Keep valve closed
                    PIC_status.2=0;
                    PIC_status.0=0;             // no cup present, error in pouring
                }

                volFlag = false;
                upper = volume >> 8;
                lower = volume & 0xff;

            }
            else {                              // Bit 2 cleared indicates stop pouring
                keepPouring = false;
                volFlag = false;
                latb.2 = 0;
            }
            sendByte(PIC_status);
            sendByte(upper);
            sendByte(lower);

}
//--// End Start/ Stop Routine

//--// Start/ Stop Routine
if (((receivedByte>>7))&0x01){              // Bit 7 set indicates person has checked out
    keepPouring = false;
    volFlag = false;
    latb.2=0;
    volumeLimit = 0;
    volume = 0;
    PIC_status.0=1;
    PIC_status.1=0;
    PIC_status.2=0;
    sendByte(PIC_status);
}

//--// While Loop to Pour until volume limit is reached
while ((volFlag==false) && (keepPouring == true)) {

    if (((portd.0))==true){// || (portb.4))){        // Check if cup is present
        keepPouring = true;
        latb.2=1;                               // open the solenoid valve
        PIC_status.0=1;
        PIC_status.1=1;
        PIC_status.2=1;
    }
    else {                                      // Cup is not present
        keepPouring = false;
        latb.2=0;                               // Close solenoid valve
        // update status here
        PIC_status.0=0;
        PIC_status.1=1;
        PIC_status.2=0;
        upper=volume >> 8;
        lower = volume & 0xff;
        sendByte(PIC_status);
```

```
        sendByte(upper);
        sendByte(lower);
    }

    //--// Checking Async PC instruction
    if(rcif){                                   // Check for PC instruction
        receivedByte = rcreg;
        if (!(((receivedByte>>2))&0x01)){       // Bit 1 cleared means stop pouring
            latb.2=0;                           // Close solenoid valve
            keepPouring=false;
            PIC_status.0=1;
            PIC_status.1=1;
            PIC_status.2=1;
        }
        else {                                  // Bit 1 set means start pouring
            keepPouring=true;
            latb.2=1;                           // Keep valvue open
            PIC_status.0=1;
            PIC_status.1=1;
            PIC_status.2=0;
        }
    }

    //---// Interrupt handling routine
    if (increaseVolume) {
        volume = volume + inc;                  // Increase pulses received
        increaseVolume=false;                   // Reset interrupt flag
        //delay_ms(10);                         // delay used for testing

        if (volume > volumeLimit) {             // If volume has been reached
            latb.2=0;                           // Close solenoid valve
            volFlag=true;                       // Break out of while loop
            upper = volume >> 8;
            lower = volume & 0xff;
            PIC_status.0=1;                     // Set status
            PIC_status.1=0;
            PIC_status.2=1;
            sendByte(PIC_status);               // Send status and pulses to PC
            sendByte(upper);
            sendByte(lower);
            volume = 0;                         // Reinitialize variables
            volumeLimit = 0;
            keepPouring = false;
            PIC_status.0=1;
            PIC_status.1=1;
            PIC_status.2=1;

        } // End volume reached routine

    } // End interrupt handling routine

    count_2=count_2+1;                          //

    //--// Micro to PC status Report Routine
    if (count_2>1000) {
        upper=volume >> 8;
        lower = volume & 0xff;
```

```
                    sendByte(PIC_status);           // Send status and pulses to PC
                    sendByte(upper);
                    sendByte(lower);
                    count_2=0;
                    latd.6= ~latd.6;                // Toggle d.6 to simulate pulses
                }

            } // close volume pouring while loop

        }// End Main Loop
} //end main
```

```
#include <system.h>


void init_usart(void);
void sendByte(char value);
char receiveByte(void);
```

```
#include <system.h>

//////////////////////////////////////////
//    AutoBev                           //
//                                      //
//    AutoBevLib.c                  //
//  Serial functions for usart      //
//////////////////////////////////////////

// Function Prototypes are in AutoBev.h

// Global Volatile Bits

//////////////////////////////////////////////////////
//    Function: init_usart
//    Purpose:  initializes usart
//    Input:    void-set Baud Rate to 9600
//  Output:   Void
//////////////////////////////////////////////////////

void init_usart(void)
{
     spbrg = 51;                    // Set correct spbrg for 9.6k rate
     /**///rcsta default is 00000000b
               //rcsta = 10000000b;
               //bit 7: serial port enable - 1
               //bit 6: 8 bit reception    - 0
               //bit 5: D/C              - x
               //bit 4: Asynchronous Mode  - 1
               //bit 3: D/C              - x
               //bit 2: No framing error.  - x
               //bit 1: No overrun error.  - x
               //bit 0: D/C              - x

  rcsta.4 = 1;             //enables async reception
  rcsta.7 = 1;             //Serial port enabled


     /**///trxsta default is 00000010b
               //txsta = 00100100b;
               //bit 7: x                    -x
               //bit 6: 8-bit transmit       -x
               //bit 5: transmit enable      -1
               //bit 4: asynch mode          -0
               //bit 3: D/C                  -x
               //bit 2: high speed               -1
               //bit 1: trmt                 -x
               //bit 0: Parity bit           -x

  txsta.4=0;                //Asynchronous mode
  txsta.5=1;                //Transmit enabled
  txsta.2=1;                //High speed baud rate mode
}

//////////////////////////////////////////////////
//    Function: sendByte
//    Purpose:  puts a byte to terminal via usart
```

```
// Input:    value -> character to be typed (ascii)
// Output:   Void
////////////////////////////////////////////////////
/* send ascii character to terminal*/
void sendByte(char value)
{
    volatile bit txen@TXSTA.5;      // define transmit enable bit
    volatile bit trmt@TXSTA.1;      // define tranmist SR status bit
    txen = 1;                       // enable transmission
    while(true)
    {
        if(trmt)
        {
            txreg = value;              // put data in transmission reg
            return;
        }
    }
}
////////////////////////////////////////////////////////
//    Function: receiveByte
//    Purpose:  gets a characeter from terminal via usart
//    Input:      Void
//  Output:   The character present at the terminal
////////////////////////////////////////////////////////


char receiveByte(void)
{
    volatile bit cren@RCSTA.4;          // define enable reception bit
    volatile bit rcif@PIR1.5;       // define received flag
    char value;                         // allocate memory for byte of data

    cren = 1;                       // enable reception

    while(true)
    {
        if(rcif)                    // wait till recieve is clear
        {
            value = rcreg;              // get value
            //putc(value);              // print gathered value to terminal
            return value;
        }
    }
}
```

7.5 User Interface Code

```
//Program.cs
using System.Windows.Forms;
using Customer;
using System.Collections;

namespace WindowsFormsApplication1
{
   static class Program
   {

      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      [STAThread]
      static void Main()
      {

         Person person = new Person();
         BitArray status = new BitArray(8, false);
         person.CardNumber = "";
         person.SessionTotal = 0;
         Application.EnableVisualStyles();
         Application.SetCompatibleTextRenderingDefault(false);
         Application.Run(new FirstForm(person, status));
      }
   }
}
```

```
//FirstForm.cs
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO.Ports;
using Customer;
using System.Net.Sockets;
using System.Net;

namespace WindowsFormsApplication1
{
   public partial class FirstForm : Form
   {
      BitArray status;
      Person person;
      string[] drinkPrice;

     // variables for updating drink prices
      string ipAddress = "10.40.136.93";//"10.40.163.142";//"10.40.167.6";//"10.40.137.203";//"10.254.254.253";//"10.40.163.87";
      int portNumber = 9050;//50005;

      // variables to send data to bartender
      string stringData;
      byte[] data;
      IPEndPoint ipep;
      Socket server;

      // variables to receive data from bartender
      IPEndPoint sender;
      EndPoint Remote;
      int recv;

      bool bartenderAvailable= false;

     public FirstForm(Person Person, BitArray Status)
     {
        person = Person;
        status = Status;
        InitializeComponent();
        cardNumber.Text = "";
     }

     private void FirstForm_Load(object sender, EventArgs e)
     {
        cardNumber.Text = person.CardNumber;

        string updatedDrinks = getUpdatedDrinks();
        char[] delimiterChars = { '/' };
        drinkPrice = updatedDrinks.Split(delimiterChars);
        person.Drink = drinkPrice;
     }
```

```
private void cardNumber_KeyDown(object sender, KeyEventArgs e)
{
   if (e.KeyCode == Keys.Enter)
   {
      person = new Person();
      person.CardNumber = cardNumber.Text;
      person.Drink = drinkPrice;
      if (person.CardNumber.Length > 0)
      {
         // if the bartender swipes card, bring up the bartender screen
         if ((person.CardNumber == ";014240310265564?") |
            (person.CardNumber == ";014015890365564?") |
            (person.CardNumber == ";016107080365564?") |
            (person.CardNumber == ";013546550565564?"))
         {

            cardNumber.Clear();
            string updatedDrinks = getUpdatedDrinks();
            char[] delimiterChars = { '/' };
            drinkPrice = updatedDrinks.Split(delimiterChars);
            person.Drink = drinkPrice;
            Bartender_1 bartenderForm = new Bartender_1(person, status);
            cardNumber.Clear();
            bartenderForm.ShowDialog();
         }
         // otherwise bring up form to let user decide what type of drink they are ordering
         else if (person.CardNumber[0].ToString() == ";")
         {

            TypeSelection ts = new TypeSelection(person, status);
            cardNumber.Clear();
            string updatedDrinks = getUpdatedDrinks();
            char[] delimiterChars = { '/' };
            drinkPrice = updatedDrinks.Split(delimiterChars);
            person.Drink = drinkPrice;
            if (bartenderAvailable)
            {
               ts.ShowDialog();
            }
         }
         else
         {
            cardNumber.Clear();
            MessageBox.Show("Please Swipe A Valid Credit Card");
         }
      }
   }
}


public string getUpdatedDrinks()
{
   data = new byte[1024];
   ipep = new IPEndPoint(IPAddress.Parse(ipAddress), portNumber);
   server = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);

   data = Encoding.ASCII.GetBytes("!");
```

```
    // try sending the order to the bartender
    try
    {
       server.SendTo(data, data.Length, SocketFlags.None, ipep);
       bartenderAvailable = true;
    }

    catch (SocketException ex)
    {
       string message = ex.Message;
       bartenderAvailable = false;
       System.Windows.Forms.MessageBox.Show("Bartender Not Available, Please Request Help");

    }

    sender = new IPEndPoint(IPAddress.Any, 0);
    Remote = (EndPoint)sender;
    data = new byte[1024];

    // try to receive the data send from the server
    try
    {
       recv = server.ReceiveFrom(data, ref Remote);
       stringData = Encoding.ASCII.GetString(data, 0, recv);
       bartenderAvailable = true;
       errorLabel.Visible = false;
       connectionButton.Visible = false;
       helpButton.Visible = true;
       return stringData;
    }
    catch (SocketException ex)
    {
       string message = ex.Message;
       //System.Windows.Forms.MessageBox.Show("No Bartender Available, Please Ask For Assistance");
       stringData = "No Host Detected";
       errorLabel.Visible = true;
       connectionButton.Visible = true;
       bartenderAvailable = false;
       helpButton.Visible = false;
       return stringData;
    }
  }

  private void helpButton_Click(object sender, EventArgs e)
  {
    string helpMessage = "To begin, you must swipe your credit card, this will store " +
       "your payment information into a local database, all drinks you order tonight will " +
       " be charged using your card information";
    HelpForm helpForm = new HelpForm(helpMessage);
    helpForm.ShowDialog();
    cardNumber.Focus();
  }

  private void connectionButton_Click(object sender, EventArgs e)
  {
    cardNumber.Text = person.CardNumber;
    string updatedDrinks = getUpdatedDrinks();
```

```
            char[] delimiterChars = { '/' };
            drinkPrice = updatedDrinks.Split(delimiterChars);
            person.Drink = drinkPrice;
        }

        private void button1_Click(object sender, EventArgs e)
        {
            this.Close();
        }

    }
}
```

```
// UsernamePassword.cs
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using Customer;
using System.Collections;
using System.Net.Sockets;
using System.Net;


namespace WindowsFormsApplication1
{
   public partial class UsernamePassword : Form
   {
      Person person;
      BitArray status;
      string cardnumber;
      string username;
      string password;
      string ipAddress = "10.40.136.93";//"10.40.163.142";//"10.40.167.6";//"10.40.137.203";//"10.254.254.253";
      int portNumber = 9050;
      bool allLetters = false;
      bool allNumbers = false;
      bool keepChecking = true;
      bool keepChecking1 = true;
      bool correctLength = false;

      // for checking username and password
      string tempUsername;
      string tempPassword;

      byte[] data = new byte[1024];
      IPEndPoint ipep;
      Socket server;

      string drinkOrderRevised;
      byte[] data2 = new byte[1024];
      IPEndPoint Sender;
      EndPoint Remote;
      int recv;
      string message;
      string stringData; //drink is going to be the variable from the user interface

      public UsernamePassword(Person Person, BitArray Status)
      {
         person = Person;
         status = Status;
         InitializeComponent();
         this.elementHost1.Child = new SingerOSK.SingerOnScreenKeyboard(this.Handle);

      }
```

```
// send the data to the bartender interface to save the new username and password
private void saveButton_Click(object sender, EventArgs e)
{
   tempUsername = usernameTextBox.Text;
   tempPassword = passwordTextbox.Text;

   // check to make sure every character in password is letter
   if (tempUsername.Length > 0)
   {
      if ((!isLetters(tempUsername)))
      {
         allLetters = false;
      }
      else
      {
         allLetters = true;
      }

      if ((!isInteger(tempPassword)) | (tempPassword.Length != 4))
      {
         allNumbers = false;
      }
      else {
         allNumbers = true;
      }

      if (allLetters & allNumbers)
      {
         person.Password = passwordTextbox.Text;
         person.Username = usernameTextBox.Text;

         // get the track, username, and password to send to the bartender
         cardnumber = person.CardNumber;
         username = person.Username;
         password = person.Password;

         //set up a new connection
         ipep = new IPEndPoint(IPAddress.Parse(ipAddress), portNumber);
         server = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);
         drinkOrderRevised = "$" + cardnumber + "#" + username + "#" + password + "#";
         data = Encoding.ASCII.GetBytes(drinkOrderRevised);

         // try to send data over the new connection
         try
         {
            server.SendTo(data, data.Length, SocketFlags.None, ipep);
         }
         catch (SocketException ex)
         {
            message = ex.Message;
            System.Windows.Forms.MessageBox.Show(message);
         }
         // set up a connection to receive data
         Sender = new IPEndPoint(IPAddress.Any, 0);
         Remote = (EndPoint)Sender;
         // try to receive the data
         try
```

```
            {
                recv = server.ReceiveFrom(data2, ref Remote);
                stringData = Encoding.ASCII.GetString(data2, 0, recv);
                System.Windows.Forms.MessageBox.Show(stringData);
            }
            catch (SocketException ex)
            {
                message = ex.Message;
                System.Windows.Forms.MessageBox.Show(message + " \nYou will not be charged for this order");
                stringData = "No Host Detected";
                MessageBox.Show(stringData);
            }

            this.Close();
        }
        else
        {
            if (!allLetters)
            {
                System.Windows.Forms.MessageBox.Show("Username Must Be All Letters, Try Again");
                allLetters = true;
            }
            else if (!allNumbers)
            {
                System.Windows.Forms.MessageBox.Show("Password must be 4 numbers");
                allNumbers = true;
            }
        }
    }
    else
    {
        MessageBox.Show("Please Enter A Username");
    }
}

private void helpButton_Click(object sender, EventArgs e)
{
    string helpMessage = "Please select a username that consists of only characters " +
      "and a password that consists of four numbers.  After setting up a username and " +
      " password, you may download the AutoBev phone application for your Droid and order " +
      "drinks directly from your phone.  Your credit card information will be stored in " +
      " our database for charging purposes.";
    HelpForm helpForm = new HelpForm(helpMessage);
    helpForm.ShowDialog();
}

private void button1_Click(object sender, EventArgs e)
{
    this.Close();
}


public bool isLetters(String str)
{
    for (int i = 0; i < str.Length; i++)
    {
        int Ascii = (int)str[i];
```

```
            if (Ascii < 65 || Ascii > 122)
            { // check if it not a letter
               return false;
            }
         }
         return true;  // if all letters
      }

      public bool isInteger(String i)
      {
         try
         {
            Convert.ToInt16(i);
            return true;
         }
         catch (Exception e)
         {
            return false;
         }
      }

      private void button1_Click_1(object sender, EventArgs e)
      {
         this.Close();
      }

   }
}
```

```
// Bartender_1.cs
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using Customer;
using System.Collections;
using System.IO;

namespace WindowsFormsApplication1
{
   public partial class Bartender_1 : Form
   {
      string cardNum;
      Person person;
      BitArray status;
      string cardNumber;
      string[] drink = new string[14];

      BeverageSizeSelect SizeSelect;

      public Bartender_1(Person Person, BitArray Status )
      {
         InitializeComponent();
         person = Person;
         status = Status;
         drink = Person.Drink;
         SizeSelect = new BeverageSizeSelect(Person, Status);
         cardNumber = person.CardNumber;
      }

      private void PourButton_Click(object sender, EventArgs e)
      {
         person.Drink[13] = "0.00";
         StartAndStopPour startAndStop = new StartAndStopPour(person, status, SizeSelect);
         this.Close();
         startAndStop.ShowDialog();
      }

      private void ExitButton_Click(object sender, EventArgs e)
      {
         this.Close();
      }


   }
}
```

```
//BeveragePourType.cs
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using Customer;


namespace WindowsFormsApplication1
{
   public partial class BeveragePourType : Form
   {
      string cardNum;
      Person person;
      BitArray status;
      BeverageSizeSelect SizeSelect;
      public BeveragePourType(Person person1, BitArray Status)
      {
         person = person1;
         cardNum = person.CardNumber;
         status = Status;
         InitializeComponent();
      }

      // Clicked Select Size
      private void selectSize_Click(object sender, EventArgs e)
      {
         // set bit three to indicate order is predetermined size
         status[0] = true;
         status[1] = false;
         status[3] = true;
         BeverageSizeSelect beerSizeSelect = new BeverageSizeSelect(person, status);
         this.Close();
         beerSizeSelect.ShowDialog();
      }

      private void payPerOunce_Click(object sender, EventArgs e)
      {
         // clear bit three to indicate that this is a pay per oz order
         status[0] = true;
         status[1] = false;
         status[2] = false;
         status[3] = false;
         status[4] = false;
         status[5] = false;
         status[6] = false;
         status[7] = false;
         StartAndStopPour startAndStop = new StartAndStopPour(person, status, SizeSelect);
         this.Close();
         startAndStop.ShowDialog();
      }

      private void cancelButton_Click(object sender, EventArgs e)
```

```
      {
         this.Close();
      }
   }
}
```

```
//BeverageSizeSelect.cs
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using Customer;
using System.Diagnostics;
using System.Threading;
using System.IO.Ports;

namespace WindowsFormsApplication1
{
   public partial class BeverageSizeSelect : Form
   {
      string path = "C:\\Users\\eclark4\\Desktop\\WriteLines1.txt";
      string cardNum;
      Person person;
      BitArray status;
      string[] drink = new string[14];
      string size;
      int length;
      byte[] PICByte_Buffer;
      byte tempByte;
      string message;
      BitArray PICStatus;
      int bitLength;
      int limit;

      BitArray picByte = new BitArray(8);
      byte[] status_byte = new byte[1];
      public BeverageSizeSelect(Person Person, BitArray Status)
      {
         person = Person;
         cardNum = person.CardNumber;
         status = Status;
         drink = Person.Drink;
         InitializeComponent();

         //configuring the serial port
         serialPort1.PortName = "COM6";
         serialPort1.BaudRate = 9600;
         serialPort1.DataBits = 8;
         serialPort1.Parity = Parity.None;
         serialPort1.StopBits = StopBits.One;
      }

      public void sizeClick(string Size)
      {
         size = Size;
         status[0] = true;
         status[1] = false;
         status[7] = false;
         status.CopyTo(status_byte, 0);
```

```
    serialPort1.Write(status_byte, 0, 1);
}

private void size1_Click(object sender, EventArgs e)
{
    // set bit 4 indicating 12oz has been chosen
    status.Set(4, true);
    status.Set(5, false);
    status.Set(6, false);
    sizeClick(Size1.Text);
}

private void Size2_Click(object sender, EventArgs e)
{
    //set bit 5 indicating 16oz has been chosen
    status.Set(4, false);
    status.Set(5, true);
    status.Set(6, false);
    sizeClick(Size2.Text);
 }

private void size3_Click(object sender, EventArgs e)
{
    // set bit 6 indicating 60oz has been chosen
    status.Set(4, false);
    status.Set(5, false);
    status.Set(6, true);
    sizeClick(Size3.Text);
}

private void size4_Click(object sender, EventArgs e)
{
    status.Set(4, false);
    status.Set(5, false);
    status.Set(6, false);
    sizeClick(Size4.Text);
}

private void cancel_Click(object sender, EventArgs e)
{
    this.Close();
}

private void serialPort1_DataReceived(object sender, System.IO.Ports.SerialDataReceivedEventArgs e)
{
    length = serialPort1.BytesToRead;
    PICByte_Buffer = new byte[length];
    serialPort1.Read(PICByte_Buffer, 0, PICByte_Buffer.Length);
    PICStatus = new BitArray(PICByte_Buffer);
    bitLength = PICStatus.Length;
    limit = serialPort1.BytesToRead;

    char[] characters = System.Text.Encoding.ASCII.GetChars(PICByte_Buffer);

    tempByte = PICByte_Buffer[PICByte_Buffer.Length - 1];
```

```
        // check if the zero bit of the status byte is not a 1, this means there was an error
        if ((tempByte & 1) != 1) {
            // tell the user to chose the size again
            System.Windows.Forms.MessageBox.Show("Please Select Size Again");
        }
        else {
            serialPort1.Close();
            this.Invoke(new EventHandler(CloseForm));
            StartAndStopPour startAndStop = new StartAndStopPour(person, status, this);
            startAndStop.ShowDialog();
        }
    }
    private void CloseForm(object sender, EventArgs e)
    {
        this.Close();
    }

    private void BeverageSizeSelect_Load(object sender, EventArgs e)
    {
        try
        {
            if (serialPort1.IsOpen)
            {
                serialPort1.Close();
            }
            serialPort1.Open();
            price1.Text = "$" + Convert.ToString(Math.Round((Convert.ToDecimal(drink[13]) * 2)));
            price2.Text = "$" + Convert.ToString(Math.Round((Convert.ToDecimal(drink[13]) * 4)));
            price3.Text = "$" + Convert.ToString(Math.Round((Convert.ToDecimal(drink[13]) * 6)));

        }
        catch (Exception ex)
        {
            message = ex.Message;
            //System.Windows.Forms.MessageBox.Show(message);
            MessageBox.Show("Error Sending Request to Beverage Dispeser, \n Please See Bartender For Assistance");
            this.Close();
        }
    }

    private void helpButton_Click(object sender, EventArgs e)
    {
        string helpMessage = "Please select the size of drink that you would like to pour. " +
            "On the next screen you will be able to dispense your beverage.";
        HelpForm helpForm = new HelpForm(helpMessage);
        helpForm.ShowDialog();
    }
  }
}
```

```
//TypeSelection.cs
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using Customer;
using System.IO.Ports;
using System.Net;
using System.Net.Sockets;


namespace WindowsFormsApplication1
{
   public partial class TypeSelection : Form
   {
      private string cardnumber;
      private Person person;
      BitArray status;
      UsernamePassword userName;
      bool dispenseAllowed;


      public TypeSelection(Person Person, BitArray Status)
      {
         person = Person;
         status = Status;
         cardnumber = person.CardNumber;
         dispenseAllowed = false;

         InitializeComponent();
         retryButton.Visible = false;
         //configuring the serial port
         serialPort1.PortName = "COM6";
         serialPort1.BaudRate = 9600;
         serialPort1.DataBits = 8;
         serialPort1.Parity = Parity.None;
         serialPort1.StopBits = StopBits.One;
      }

      private void mixedDrinkButton_Click(object sender, EventArgs e)
      {
         // show the mixed drink form
         this.Close();
         MixedDrinkForm mixedDrinkForm = new MixedDrinkForm(person, status);
         mixedDrinkForm.ShowDialog();
      }

      private void pourYourOwnButton_Click(object sender, EventArgs e)
      {
         tryConnection();

         if (dispenseAllowed)
         {
```

```
        // set bits 1 and 2 of the status to say its a new order and start pouring
        status.Set(1, true);
        status.Set(2, true);
        this.Close();
        BeveragePourType beerPourType = new BeveragePourType(person, status);
        serialPort1.Close();
        beerPourType.ShowDialog();
      }
      else
      {
        //MessageBox.Show("error");
      }

    }

    private void cancelButton_Click(object sender, EventArgs e)
    {
      this.Close();
    }

    private void phoneAppButton_Click(object sender, EventArgs e)
    {
      // open phone app form
      userName = new UsernamePassword(person, status);
      this.Close();
      userName.ShowDialog();
    }
    public void tryConnection()
    {
      try
      {
        if (serialPort1.IsOpen)
        {
           serialPort1.Close();
        }
        serialPort1.Open();
        dispenseAllowed = true;
        dispenseLabel.Visible = false;
        retryButton.Visible = false;
        sorryLabel.Visible = false;

      }
      catch (Exception ex)
      {
        dispenseAllowed = false;
        dispenseLabel.Visible = true;
        retryButton.Visible = true;
        sorryLabel.Visible = true;
      }
    }

    private void helpButton_Click(object sender, EventArgs e)
    {
      string helpMessage = "On this form you must choose to either order a " +
        "mixed drink which will then be sent to the bartender to be made " +
        "or to pour your own beer.";
```

```
        HelpForm helpForm = new HelpForm(helpMessage);
        helpForm.ShowDialog();

      }

      private void retryButton_Click(object sender, EventArgs e)
      {
        tryConnection();
      }
    }
}
```

```
//HelpForm.cs
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{

   public partial class HelpForm : Form
   {
      string HelpMessage;
      public HelpForm(string helpMessage)
      {
         HelpMessage = helpMessage;
         InitializeComponent();
      }

      private void HelpForm_Load(object sender, EventArgs e)
      {
         helpMessageText.Text = HelpMessage;
      }

      private void exitButton_Click(object sender, EventArgs e)
      {
         this.Close();
      }
   }
}
```

```
// MixedDrinkForm.cs
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using Customer;
using System.Net;
using System.Net.Sockets;


namespace WindowsFormsApplication1
{
   public partial class MixedDrinkForm : Form
   {
      string cardNum;
      Person person;
      BitArray status;
      string[] drink = new string[14];
      string cost;
      bool ServerDetected = false;
      string userName;
      string password;
      string ipAddress = "10.40.136.93";//"10.40.163.142";//"10.40.163.142";//"10.40.167.6";//"10.40.137.203";//"10.254.254.253";
      int portNumber = 9050;

      // variables to send data to bartender
      string stringData;
      byte[] data;
      IPEndPoint ipep;
      Socket server;
      int count;
      string amount;
      string drinkOrderRevised;
      string drinkName;

      // variables to receive data from bartender
      IPEndPoint sender;
      EndPoint Remote;
      int recv;

      public MixedDrinkForm(Person Person, BitArray Status)
      {
         person = Person;
         status = Status;
         drink = Person.Drink;
         cardNum = person.CardNumber;
         userName = person.Username;
         password = person.Password;
         InitializeComponent();
      }

      public void drinkOrdered(string Drink, string Cost)
      {
```

```
    stringData = SimpleUdpClient(Drink);
    if (ServerDetected) {
        person.SessionTotal = person.SessionTotal + Convert.ToDecimal(cost);
        ThankYouForm thankYouForm = new ThankYouForm(person, stringData, status);
        this.Close();
        thankYouForm.ShowDialog();
    }
    else {
        System.Windows.Forms.MessageBox.Show("No Server Detected");
        this.Close();
    }
}

private void Drink1_Click(object sender, EventArgs e)
{
    drinkName = Drink1.Text;
    cost = drink[1];
    drinkOrdered(drinkName, cost);
}

private void Drink2_Click(object sender, EventArgs e)
{
    drinkName = Drink2.Text;
    cost = drink[3];
    drinkOrdered(drinkName, cost);
}

private void Drink3_Click(object sender, EventArgs e)
{
    drinkName = Drink3.Text;
    cost = drink[5];
    drinkOrdered(drinkName, cost);
}

private void Drink4_Click(object sender, EventArgs e)
{
    drinkName = Drink4.Text;
    cost = drink[7];
    drinkOrdered(drinkName, cost);
}

private void Drink5_Click(object sender, EventArgs e)
{
    drinkName = Drink5.Text;
    cost = drink[9];
    drinkOrdered(drinkName, cost);
}

private void Drink6_Click(object sender, EventArgs e)
{
    drinkName= Drink6.Text;
    cost = drink[11];
    drinkOrdered(drinkName, cost);
}

private void cancel_Click(object sender, EventArgs e)
{
```

```
    this.Close();
}

public string SimpleUdpClient(string drinkOrder)
{
    data = new byte[2048];
    ipep = new IPEndPoint(IPAddress.Parse(ipAddress), portNumber);
    server = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);
    count = 1000;

    // convert amount to string to send to server
    amount = count.ToString();
    drinkOrderRevised = "#" + drinkOrder + "#" + person.CardNumber + "#" + cost + "#";
    data = Encoding.ASCII.GetBytes(drinkOrderRevised);
    // try sending the order to the bartender
    try {
        server.SendTo(data, data.Length, SocketFlags.None, ipep);
    }
    catch (SocketException ex) {
        string message = ex.Message;
        System.Windows.Forms.MessageBox.Show(message);
    }

    sender = new IPEndPoint(IPAddress.Any, 0);
    Remote = (EndPoint)sender;
    data = new byte[2048];

    // try to receive the data send from the server
    try {
        recv = server.ReceiveFrom(data, ref Remote);
        stringData = Encoding.ASCII.GetString(data, 0, recv);
        ServerDetected = true;
        return stringData;
    }
    catch (SocketException ex) {
        string message = ex.Message;
        System.Windows.Forms.MessageBox.Show(message + " \nYou will not be charged for this order");
        stringData = "No Host Detected";
        ServerDetected = false;
        return stringData;
    }
}

private void MixedDrinkForm_Load(object sender, EventArgs e)
{

    Drink1.Text = drink[0];
    price1.Text = "$" + drink[1];
    Drink2.Text = drink[2];
    price2.Text = "$" + drink[3];
    Drink3.Text = drink[4];
    price3.Text = "$" + drink[5];
    Drink4.Text = drink[6];
    price4.Text = "$" + drink[7];
    Drink5.Text = drink[8];
    price5.Text = "$" + drink[9];
    Drink6.Text = drink[10];
```

```
            price6.Text = "$" + drink[11];

        }

        private void helpButton_Click(object sender, EventArgs e)
        {
            string helpMessage = "Please choose one of the mixed drinks from the order form. " +
                "Your order will be sent to the bartender and your card will be charged with " +
                "the price shown underneath the drink";
            HelpForm helpForm = new HelpForm(helpMessage);
            helpForm.ShowDialog();
        }
    }
}
```

```
// StartAndStopPour.cs
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Diagnostics;
using System.Threading;
using Customer;
using System.IO.Ports;
using System.Net;
using System.Net.Sockets;

namespace WindowsFormsApplication1
{
   public partial class StartAndStopPour : Form
   {
      Stopwatch stopWatch = new Stopwatch();
      int combined;
      string amount;
      double poured;
      bool allowPour = true;
      bool allowFinish = true;

      string cardNum;
      Person person;
      BitArray status;
      string[] drink = new string[14];
      byte tempByte;
      BeverageSizeSelect SizeSelectForm;
      byte[] status_byte = new byte[1];

      // variables for sending order to bartender
      byte[] data = new byte[1024];
      string stringData;
      string ipAddress = "10.40.136.93";//"10.40.163.142";//"10.40.167.6";//"10.254.254.253";//"10.40.137.203";
      int portNumber = 9050;
      IPEndPoint ipep;
      Socket server;
      string drinkOrderRevised;
      string message;

      // variables for receiving data from bartender
      IPEndPoint Sender;
      EndPoint Remote;
      int recv;

      // variables for receiving data from the microcontroller
      int length;
      byte[] PICByte_Buffer;
      BitArray PICStatus;
      int bitLength;
      int limit;
```

```
TimeSpan ts;

//need to pass in the size that the person chose, or infinity if it is a PPO
public StartAndStopPour(Person Person, BitArray Status, BeverageSizeSelect sizeSelectForm)
{
   person = Person;
   cardNum = person.CardNumber;
   drink = Person.Drink;
   SizeSelectForm = sizeSelectForm;
   if (cardNum == "b") {
      //price = 0.0m;
   }
   else {
      //price = .25m;
   }

   status = Status;
   InitializeComponent();
   beginButton.Enabled = true;
   stopButton.Enabled = false;
   //configuring the serial port
   serialPort1.PortName = "COM6";
   serialPort1.BaudRate = 9600;
   serialPort1.DataBits = 8;
   serialPort1.Parity = Parity.None;
   serialPort1.StopBits = StopBits.One;

}

private void beginButton_Click(object sender, EventArgs e)
{
   if (allowPour)
   {
      allowFinish = false;
      // set bits 1 and 2 of status indicating that the microcontroller should start pouring
      status[0] = true; status[1] = true; status[2] = true; status[7] = false;
      status.CopyTo(status_byte, 0);
      serialPort1.Write(status_byte, 0, 1);
      stopButton.Enabled = true;
      beginButton.Enabled = false;
      exitButton.Visible = false;
   }
}

private void stopButton_Click(object sender, EventArgs e)
{
   if (allowPour)
   {
      allowFinish = true;
      // clear bit 2 of status indicating that the microcontroller should stop pouring
      status[0] = true; status[1] = true; status[2] = false; status[7] = false;
      status.CopyTo(status_byte, 0);
      // send status to microcontroller
      serialPort1.Write(status_byte, 0, 1);
      stopButton.Enabled = false;
      beginButton.Enabled = true;
      exitButton.Visible = true;
```

```
    }
  }

  private void exitButton_Click(object sender, EventArgs e)
  {
    if (allowFinish)
    {
      status[7] = true;
      status.CopyTo(status_byte, 0);
      serialPort1.Write(status_byte, 0, 1);

      // clear all the status bits
      status[0] = false; status[1] = false; status[2] = false;
      status[4] = false; status[5] = false; status[6] = false;
      status[7] = false;

      status.CopyTo(status_byte, 0);
      serialPort1.Close();

      // send order to bartender
      ipep = new IPEndPoint(IPAddress.Parse(ipAddress), portNumber);
      server = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);
      drinkOrderRevised = "@" + cardNum + "#" + amount + "#" + "field3" + "#";

      data = Encoding.ASCII.GetBytes(drinkOrderRevised);
      // try sending the data to the server
      try
      {
        server.SendTo(data, data.Length, SocketFlags.None, ipep);
      }
      catch (SocketException ex)
      {
        message = ex.Message;
        System.Windows.Forms.MessageBox.Show(message);
      }

      // set up new connection
      Sender = new IPEndPoint(IPAddress.Any, 0);
      Remote = (EndPoint)Sender;
      data = new byte[2048];
      try
      {
        recv = server.ReceiveFrom(data, ref Remote);
        stringData = Encoding.ASCII.GetString(data, 0, recv);
      }
      catch (SocketException ex)
      {
        message = ex.Message;
        System.Windows.Forms.MessageBox.Show(message + " \nYou will not be charged for this order");
        stringData = "No Host Detected";
      }
      status[7] = false;
      allowFinish = false;
      ThankYouForm thankYouForm = new ThankYouForm(person, stringData, status);
      this.Close();
      thankYouForm.ShowDialog();
    }
```

```
    else
    {
      MessageBox.Show("You Must Stop Pouring Before You Can Finish");
    }
  }

  private void serialPort1_DataReceived(object sender, SerialDataReceivedEventArgs e)
  {
    // receive data from the microcontroller
    length = serialPort1.BytesToRead;
    PICByte_Buffer = new byte[length];
    // read the appropriate amount of bytes
    serialPort1.Read(PICByte_Buffer, 0, PICByte_Buffer.Length);
    PICStatus = new BitArray(PICByte_Buffer);
    bitLength = PICStatus.Length;
    limit = serialPort1.BytesToRead;

    // if the microcontroller has sent the status byte and the two volume bytes
    if (PICByte_Buffer.Length >= 3) {
      combined = PICByte_Buffer[PICByte_Buffer.Length - 2] << 8 | PICByte_Buffer[PICByte_Buffer.Length - 1];
      amount = combined.ToString();
      poured = combined / 165.61176;
      updateLabelText((Math.Round(poured,1)).ToString());
      // get the status byte from the byte array
      tempByte = PICByte_Buffer[PICByte_Buffer.Length - 3];

      // check the status
      // if bit 0 was not zero, then this indicates an error, check what kind of error
      if ((tempByte & 1) != 1) {
        //MessageBox.Show("Error");
        allowFinish = true;
        // if the second bit is zero, there was not a cup present
        if ((tempByte & 4) != 4) {
          this.Invoke(new EventHandler(CupErrorMessage));
          //System.Windows.Forms.MessageBox.Show("Place Cup And Press Start Again");
          //MessageBox.Show(Convert.ToString(tempByte));
          this.Invoke(new EventHandler(UpdateButtons));

        }
      }
      else if ((tempByte & 2) != 2) {
        this.Invoke(new EventHandler(VolumeReachedMessage));
        //updateLabelText("Volume Limit Reached Please Proceed To Checkout");
        allowPour = false;
        allowFinish = true;
      }
    }
  }

  // functions to update the amount poured label with
  delegate void updateLabelTextDelegate(string newText);
  private void updateLabelText(string newText)
  {
    if (volumePoured.InvokeRequired) {
      //this is the worker thread
      updateLabelTextDelegate del = new updateLabelTextDelegate(updateLabelText);
      volumePoured.Invoke(del, new object[] { "You have poured " + newText + " ounces" });
```

```
        }
        else{
            //this is the UI thread
            volumePoured.Text = newText;
        }
    }

    private void CupErrorMessage(object sender, EventArgs e)
    {
        volumePoured.Text = "Please Hold Down Lever With Cup And Begin Pouring Again";
        exitButton.Visible = true;

    }

    private void VolumeReachedMessage(object sender, EventArgs e)
    {
        volumePoured.Text = "Desired Volume Has Been Reached, Please Proceed To Checkout";
        exitButton.Visible = true;
    }


    private void UpdateButtons(object sender, EventArgs e)
    {
        beginButton.Enabled = true;
        stopButton.Enabled = false;
    }


    private void StartAndStopPour_Load(object sender, EventArgs e)
    {
        // try opening the serial port ocnnection
        exitButton.Visible = false;
        try
        {
            if (serialPort1.IsOpen)
            {
                serialPort1.Close();
            }
            serialPort1.Open();
            beginButton.Enabled = true;
            // if it is successful, set bit 0, and clear bit 1 indicating a new order
            status[0] = true;
            status[2] = false;
            status[7] = false;

            status.CopyTo(status_byte, 0);
            serialPort1.Write(status_byte, 0, 1);
            serialPort1.DiscardInBuffer();
            priceLabel.Text = "The Price Per Ounce is: $" + drink[13];
            status[1] = true;
        }
        catch (Exception ex)
        {
            string message = ex.Message;
            //System.Windows.Forms.MessageBox.Show(message);
            MessageBox.Show("Error Pouring Beer, Please Ask For Assistance");
            this.Close();
```

```
        }
      }

      private void helpButton_Click(object sender, EventArgs e)
      {
        string helpMessage = "You may control when you drink is poured by hitting the begin and " +
          "stop pouring buttons.  If you selected a predetermined size, when this limit is detected, " +
          "the flow will be stopped.  If you are paying per ounce, you may pour up to 180 oz during " +
          "a single purchase.";
        HelpForm helpForm = new HelpForm(helpMessage);
        helpForm.ShowDialog();
      }
    }
}
```

```
//ThankYouForm.cs
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using Customer;
using Microsoft.CSharp;

namespace WindowsFormsApplication1
{
   public partial class ThankYouForm : Form
   {
      string cardNum;
      Person person;
      BitArray status;

      public ThankYouForm(Person Person, string Order, BitArray Status)
      {
         person = Person;
         cardNum = person.CardNumber;
         status = Status;
         InitializeComponent();
         thankYouText.Text = Order;
      }

      public void orderAgain_Click(object sender, EventArgs e)
      {
         TypeSelection typeSelection = new TypeSelection(person, status);
         typeSelection.ShowDialog();
         this.Close();
      }

      public void checkoutButton_Click(object sender, EventArgs e)
      {
         // Get name from textbox
         string track = person.CardNumber;
         // Check if customer has a record, if not create one

         // Arbitrary charge amount
         decimal charge = person.SessionTotal;
         // Add charge to total

         person.CardNumber = " ";
         person.SessionTotal = 0;

         this.Close();
      }
   }
}

// Person.cs
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;

namespace Customer
{
   public class Person
   {

      private decimal sessionTotal;
      private string cardNumber;
      private string username;
      private string password;
      public string[] drink = new string[14];

      public decimal SessionTotal
      {
         get {
            return sessionTotal;
         }
         set {
            sessionTotal = value;
         }
      }

      public string CardNumber
      {
         get { return cardNumber; }
         set { cardNumber = value; }
      }

      public string Username
      {
         get { return username; }
         set { username = value; }
      }

      public string Password
      {
         get { return password; }
         set { password = value; }
      }

      public string [] Drink
      {
         get {
            return drink;
         }
         set {
            /*for (int i = 0; i < 14; i++)
            {
               drink[i] = value[i];
            }*/
            drink = value;
        }
      }
   }
}
```

7.6 Bartender Interface Code:

Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
  static class Program
  {
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
      Application.EnableVisualStyles();
      Application.SetCompatibleTextRenderingDefault(false);
      Application.Run(new Form1());
    }
  }
}
```

DrinkPrices.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO;

namespace WindowsFormsApplication1
{
  public partial class DrinkPrices : Form
  {
    string path;
    public DrinkPrices()
    {
      InitializeComponent();
      path = "C:\\Users\\lgarcia\\Desktop\\";
    }

    private void saveButton_Click(object sender, EventArgs e)
    {
      TextWriter tw = new StreamWriter(path + "DrinkPrice.txt");
      tw.WriteLine(textBox1.Text + '/' + textBox2.Text + '/' + textBox3.Text + '/' + textBox4.Text + '/'
        + textBox5.Text + '/' + textBox6.Text + '/' + textBox7.Text + '/' + textBox8.Text + '/' +
        textBox9.Text + '/' + textBox10.Text + '/' + textBox11.Text + '/' + textBox12.Text +
          '/' + textBox13.Text + '/' + textBox14.Text);
      tw.Close();
      this.Close();
    }
  }
}
```

Form1.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.IO;
using System.Net.Mail;

namespace WindowsFormsApplication1
{
   public partial class Form1 : Form
   {
      string[] count = new string[99];
      string[] _number = new string[99];
      int i = 0;
      int counter = 0;
      string message;
      int currentVersion = 1;

      // variables for setting prices
      string drinkPriceList;
      string[] drinkPriceArray;
      string droidPriceList;

      // variables for receiving data

      string order;

      // Initialise the EndPoint for the clients
      IPEndPoint ipep;
      Socket newsock;
      IPEndPoint sender;
      EndPoint Remote;
      int[] indexArray;
      int[] offsetArray;
      int index;

      string[] _drink = new string[99];
      string track;
      string stringCost;
      decimal charge;
      string username;
      string password;
      int numItemsInList;

      // for beer orders
      string pulseCount;
      uint amount;
      decimal volume;

      // from the phone
      string number;
      string pin;
      string DroidIP;

      byte[] data = new byte[2048];
      byte[] data2 = new byte[2048];
      decimal total = 0;
```

```
// for loading and saving database
string path = "C:\\Users\\lgarcia\\Desktop\\";
string drinkPricePath = "C:\\Users\\lgarcia\\Desktop\\DrinkPrice.txt";
// for the current date

DateTime Date;
int CurrentDay;
int CurrentMonth;
int CurrentYear;
string TodaysDate;

// for the date in the calendar on the form
int day;
int month;
int year;
string FullDate;

public Form1()
{
   InitializeComponent();
   fileName.Visible = false;
   clearButton.Visible = false;
   ipep = new IPEndPoint(IPAddress.Any, 9050);
   newsock = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);
   newsock.Bind(ipep);
   sender = new IPEndPoint(IPAddress.Any, 0);
   Remote = (EndPoint)(sender);

   Date = DateTime.Now;
   CurrentDay = Date.Day;
   CurrentMonth = Date.Month;
   CurrentYear = Date.Year;
   TodaysDate = CurrentMonth.ToString() + "_" + CurrentDay.ToString() + "_" + CurrentYear.ToString();

   day = CurrentDate.Value.Day;
   month = CurrentDate.Value.Month;
   year = CurrentDate.Value.Year;
   FullDate = month.ToString() + "_" + day.ToString() + "_" + year.ToString();

   // bring up form for bartneder to set prices
   initializePrices();

   // begin receiving async data
   newsock.BeginReceiveFrom(data, 0, data.Length, SocketFlags.None, ref Remote, new AsyncCallback(ReceiveData), Remote);
}

public void clearOrderButton_Click(object sender, EventArgs e)
{
   //String DrinkSelected = DrinkQueue.Text;
   string drinkSelected = DrinkQueue.Text;


   if (DrinkQueue.SelectedIndex == -1)
   {
      MessageBox.Show("Please select an item first.", "No item selected", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
   }

   else
   {

      int numVal = DrinkQueue.SelectedIndex;
      if (drinkSelected[1] == ' ')
      {

         drinkSelected = drinkSelected.Substring(6, (drinkSelected.Length - 6));
```

```
        }
        else
        {

            drinkSelected = drinkSelected.Substring(7, (drinkSelected.Length - 7));
        }


        DrinkQueue.Items.RemoveAt(DrinkQueue.SelectedIndex);
        if ((_number[numVal] != "0") & (_number[numVal] != null))
        {

            sendSMS("Your " + drinkSelected// _drink[Convert.ToInt32(count[numVal])]
                + " is ready. Please retrieve it at the bar. Make sure to tip your bartender!",
                "macomber.alex@gmail.com");
        }
        for (int k = numVal; k < counter; k++)
        {
            count[k] = count[k + 1];
            _number[k] = _number[k + 1];

        }
        i = i - 1;
    }
}

public void ReceiveData(IAsyncResult e)
{
    this.customersTableAdapter.Fill(this.bartenderDBDataSet.Customers);
    ipep = new IPEndPoint(IPAddress.Any, 0);
    // Initialise the EndPoint for the clients
    Remote = (EndPoint)ipep;
    // Receive all data
    newsock.EndReceiveFrom(e, ref Remote);
    if (Encoding.ASCII.GetString(data, 0, 1) == "!")
    {
        updateDrinks();
        data = new byte[1024];
        newsock.BeginReceiveFrom(data, 0, data.Length, SocketFlags.None, ref Remote, new AsyncCallback(ReceiveData), Remote);
    }
    else
    {
        count[i] = Convert.ToString(counter);
        indexArray = new int[4];
        offsetArray = new int[4];
        //To Write the Order
        index = 0;
        while (data[index] != 0)
        {
            index = index + 1;
        }
        char[] delimiterChars = { '#', '$', '@', '&' };
        order = Encoding.ASCII.GetString(data, 0, index);
        string[] words = { " " };
        words = order.Split(delimiterChars);

        // this is a drink order from the kiosk
        if (Encoding.ASCII.GetString(data, 0, 1) == "#")
        {
            // Convert string to int
            int orderID;
            int.TryParse(count[i], out orderID);

            // Add order to datatable Orders (track, drink, number, orderID)
            //MessageBox.Show(orderID.ToString());
            //ordersTableAdapter.AddOrder("kiosk", words[1], "fromKiosk", orderID);
```

```
_drink[i] = words[1];
_number[i] = "0";
track = words[2];
stringCost = words[3];

// covert the cost of the mixed drink to a decimal to be added to the total
charge = Convert.ToDecimal(stringCost);

// send message back to kiosk indicating drink number and current order
numItemsInList = DrinkQueue.Items.Count;

// Check if customer has a record, if not create one
try
{
   if (customersTableAdapter.GetTotal(track) == null)
   {
      customersTableAdapter.NewCustomer(track, null, null);
      //message = "Customer Created in drink order sent from kiosk";
   }
   //update index for the counter and drink arrays & update list on bartender
   i++; counter++; index = 0;
   updateLabelText(count[i - 1] + "     " + _drink[i - 1]);
   //updateLabelText(track);
}

catch (Exception e2)
{
   message = "Customer Not Created In Drink Order Sent From Kiosk";
   System.Windows.Forms.MessageBox.Show(e2.ToString());
   MessageBox.Show(message);
}

// Add charge to total
customersTableAdapter.AddToTotal(charge, track);
// Get new total
try
{
   total = (decimal)customersTableAdapter.GetTotal(track);
   message = "Successfully obtained total";
}
catch (Exception e1)
{
   message = "Total Could Not Be Obtained From Database";
   System.Windows.Forms.MessageBox.Show(e1.ToString());
   MessageBox.Show(message);
}

// Update DataGridView on form
customersTableAdapter.Fill(bartenderDBDataSet.Customers);
this.tableAdapterManager.UpdateAll(this.bartenderDBDataSet);
//send message to customer

if (numItemsInList.ToString() == "1")
{
   message = "You Are Order #  " + count[i - 1] + ". \n" + numItemsInList.ToString() + " Order Is In Front of You. " +
  "\nAmount Charged: $" + stringCost;
}
else
{
   message = "You Are Order #  " + count[i - 1] + ". \n" + numItemsInList.ToString() + " Orders Are In Front of You. " +
      "\nAmount Charged: $" + stringCost;
}

data2 = Encoding.ASCII.GetBytes(message);
newsock.SendTo(data2, data2.Length, SocketFlags.None, Remote);
```

```
    // Begin waiting for async data to be sent from client
    data = new byte[1024];
    newsock.BeginReceiveFrom(data, 0, data.Length, SocketFlags.None, ref Remote, new AsyncCallback(ReceiveData), Remote);
  }
  // this is a drink order from an Android phone & the database needs to be updated
  else if (Encoding.ASCII.GetString(data, 0, 1) == "&")
  {
    // number IP username pin drink
    if (words[1] == "null")
    {
      words[1] = "macomber.alex@gmail.com";
    }
    number = words[1];
    _number[i] = words[1];

    // Convert string to int

    // Add order to datatable Orders (track, drink, number, orderID)
    //ordersTableAdapter.AddOrder("phone", words[5],words[1],orderID);

    DroidIP = words[2];
    username = words[3];
    pin = words[4];
    _drink[i] = words[5];
    stringCost = words[6];
    // covert the cost of the mixed drink to a decimal to be added to the total
    charge = Convert.ToDecimal(stringCost);
    track = customersTableAdapter.FindTrack(username);

    // query the database for the username
    if (customersTableAdapter.CheckUsername(username) != null)
    {
      // if the username exists make sure the password is right
      if (customersTableAdapter.GetPassword(username) == pin)
      {
        //check if password is right
        message = "Password correct";
        // send message back to droid indicating drink number and current order
        // the number of drinks in the list in counter
        sendUDP("1#" + count[i] + "#" + DrinkQueue.Items.Count.ToString() + "#", DroidIP);
        // only update index values 1if password is correct and order is placed
        i++; counter++;
        // charge the username and password
        updateLabelText(count[i - 1] + "     " + _drink[i - 1]);
        // Add charge to total
        customersTableAdapter.AddToTotal(charge, track);
      }
      // otherwise the password was not correct
      else
      {
        sendUDP("3#", DroidIP);
        message = "Password incorrect for existing username";
      }
    }

    // otherwise the username did not exist
    else
    {
      sendUDP("2#", DroidIP);
      message = "Username does not exist";
    }
    //System.Windows.Forms.MessageBox.Show(message);
    // start to receive async data again
    data = new byte[1024];
    newsock.BeginReceiveFrom(data, 0, data.Length, SocketFlags.None, ref Remote, new AsyncCallback(ReceiveData), Remote);
  }
```

```
// this is a setting up a username with a track number and password
else if (Encoding.ASCII.GetString(data, 0, 1) == "$")
{
   track = words[1];
   username = words[2];
   password = words[3];

   // Check if customer has a record, if not create one
   try
   {
      // check if the user already has an account
      if (customersTableAdapter.GetTrack(track) != null)
      {
         // check if the customer does not have a username yet
         if (customersTableAdapter.GetUsername(track) == null)
         {
            // if they don't have a username, check to make sure username is unique
            if (customersTableAdapter.CheckUsername(username) == null)
            {
               customersTableAdapter.SetUsername(username, password, track);
               message = ("UserName Updated");
            }
            // if it's not unique don't create username
            else
            {
               message = "UserName already taken for account already in existence";
               //System.Windows.Forms.MessageBox.Show("UserName Updated");
            }
         }
         // otherwise they already have a username, do not change it
         else
         {
            message = "You already have a username: It is: " + customersTableAdapter.GetUsername(track);
            //System.Windows.Forms.MessageBox.Show(message2);
         }
      }
      // otherwise it is a first time user, an account needs to be created
      else if (customersTableAdapter.GetTrack(track) == null)
      {
         // check if the username is unique
         if (customersTableAdapter.CheckUsername(username) == null)
         {
            //allow for creation of username
            customersTableAdapter.NewCustomer(track, username, password);
            message = "Username Created";
         }
         // otherwise the username is taken, do not create the username
         else
         {
            message = "Username already in Use, Please Select Another";
         }
      }
   }
   catch
   {
      message = "Unable to complete try statement for creating username and password";
   }
   // System.Windows.Forms.MessageBox.Show(message);

   data2 = Encoding.ASCII.GetBytes(message);
   newsock.SendTo(data2, data2.Length, SocketFlags.None, Remote);

   // Update Datagrid view on form
   customersTableAdapter.Fill(bartenderDBDataSet.Customers);
   this.tableAdapterManager.UpdateAll(this.bartenderDBDataSet);
```

```
            // start receiving async data again
            data = new byte[1024];
            newsock.BeginReceiveFrom(data, 0, data.Length, SocketFlags.None, ref Remote, new AsyncCallback(ReceiveData), Remote);
        }
        // this is a beer order
        else if (Encoding.ASCII.GetString(data, 0, 1) == "@")
        {
            track = words[1];
            pulseCount = words[2];
            if (pulseCount == "")
            {
                pulseCount = "0";
            }
            // convert the pulse count from a string to an unsigned int, then convert to volume in ml, get that as a decimal
            amount = Convert.ToUInt32(pulseCount);
            volume = Convert.ToDecimal(amount / 165.61176);
            // determine the price of the volume poured
            if (track[1].ToString() == "0")
            {
                charge = Convert.ToDecimal(0);
            }
            else
            {
                charge = volume * Convert.ToDecimal(drinkPriceArray[13]);
            }
            string stringCost = charge.ToString();
            index = 0;
            // Check if customer has a record, if not create one
            try
            {
                if (customersTableAdapter.GetTotal(track) == null)
                {
                    customersTableAdapter.NewCustomer(track, null, null);
                    message = "Customer created for beer order";
                }
            }
            catch
            {
                message = "Could not create customer for beer order";
            }
            //  System.Windows.Forms.MessageBox.Show(message);

            //System.Windows.Forms.MessageBox.Show(message);
            // Add charge to total
            customersTableAdapter.AddToTotal(charge, track);
            // Get new total
            try
            {
                total = (decimal)customersTableAdapter.GetTotal(track);
                message = "Total retrieved for customer";
            }
            catch
            {
                message = "Could not retrieve total for customer";
            }
            message = "You've Just Added $" + (Math.Round(charge, 2)).ToString() + " For A Night's Total of $" + (Math.Round(total,
2)).ToString();

            data2 = Encoding.ASCII.GetBytes(message);
            newsock.SendTo(data2, data2.Length, SocketFlags.None, Remote);
            // send message back to user
            //System.Windows.Forms.MessageBox.Show(message);
            // Update DataGridView on form
            customersTableAdapter.Fill(bartenderDBDataSet.Customers);
            this.tableAdapterManager.UpdateAll(this.bartenderDBDataSet);
            data = new byte[1024];
```

```
            newsock.BeginReceiveFrom(data, 0, data.Length, SocketFlags.None, ref Remote, new AsyncCallback(ReceiveData), Remote);
        }

    }

}

delegate void updateLabelTextDelegate(string newText);
private void updateLabelText(string newText)
{
    if (DrinkQueue.InvokeRequired)
    {
        // this is worker thread
        updateLabelTextDelegate del = new updateLabelTextDelegate(updateLabelText);
        //label1.Invoke(del, new object[] {"You have poured " + newText + " ounces" });
        DrinkQueue.Invoke(del, new object[] { newText });
    }
    else
    {
        // this is UI thread
        DrinkQueue.Items.Add(newText);
    }
}

private void Form1_Load(object sender, EventArgs e)
{
    customersTableAdapter.ClearTable();
    // TODO: This line of code loads data into the 'bartenderDBDataSet.Customers' table. You can move, or remove it, as needed.
    this.customersTableAdapter.Fill(this.bartenderDBDataSet.Customers);

}

private void LoadButton_Click(object sender, EventArgs e)
{
    this.Refresh();
    Thread.Sleep(20);

    fileName.Visible = true;
    saveButton.Visible = false;

    // before loading new file, save file for the current date
    string filename = path + TodaysDate + ".txt";
    if (!(File.Exists(filename)))
    {
        // Save data to fileName in XML format
        bartenderDBDataSet.WriteXml(filename);
        bartenderDBDataSet.Dispose();
        // Delete all rows from table
        customersTableAdapter.ClearTable();
        // Update DataGridView
        customersTableAdapter.Fill(bartenderDBDataSet.Customers);
    }
    else
    {
        filename = path + TodaysDate + "_" + currentVersion.ToString() + ".txt";
        // Save data to fileName in XML format
        bartenderDBDataSet.WriteXml(filename);
        bartenderDBDataSet.Dispose();
        // Delete all rows from table
        customersTableAdapter.ClearTable();
        // Update DataGridView
        customersTableAdapter.Fill(bartenderDBDataSet.Customers);
        currentVersion = currentVersion + 1;

    }
```

```
    // Get current date from calender
    // day = CurrentDate.Value.Day;
    //month = CurrentDate.Value.Month;
    //year = CurrentDate.Value.Year;
    //FullDate = month.ToString() + "_" + day.ToString() + "_" + year.ToString();

    MessageBox.Show("Enter File Name In Text Box");


  }

  private void saveButton_Click(object sender, EventArgs e)
  {
    this.Refresh();
    Thread.Sleep(20);
    // Define file to write data to,
    string fileName = path + TodaysDate + ".txt";

    if (getDate()) // if todays date and the day on the calendar are the same, save the file
    {
      if (File.Exists(fileName)) // if file exists, save to new file
      {
        fileName = path + TodaysDate + "_" + currentVersion.ToString() + ".txt";
        // Save data to fileName in XML format
        bartenderDBDataSet.WriteXml(fileName);
        bartenderDBDataSet.Dispose();
        currentVersion = currentVersion + 1;
        customersTableAdapter.ClearTable();
        // Update DataGridView
        customersTableAdapter.Fill(bartenderDBDataSet.Customers);
      }
      else // else save to file with current date
      {
        // Save data to fileName in XML format
        bartenderDBDataSet.WriteXml(fileName);
        bartenderDBDataSet.Dispose();
        // Delete all rows from table
        customersTableAdapter.ClearTable();
        // Update DataGridView
        customersTableAdapter.Fill(bartenderDBDataSet.Customers);
      }
    }
    else if (!getDate()) //if todays date and day on calendar are different, do not save current table
    {
      System.Windows.Forms.MessageBox.Show("You may not save data for a different date than the current date");
    }
  }

  private bool getDate()
  {
    day = CurrentDate.Value.Day;
    month = CurrentDate.Value.Month;
    year = CurrentDate.Value.Year;
    FullDate = month.ToString() + "_" + day.ToString() + "_" + year.ToString();

    CurrentDay = Date.Day;
    CurrentMonth = Date.Month;
    CurrentYear = Date.Year;
    TodaysDate = CurrentMonth.ToString() + "_" + CurrentDay.ToString() + "_" + CurrentYear.ToString();

    if (TodaysDate == FullDate)
    {
      return true;
    }
    else
    {
```

```
            return false;
        }
    }


    public void sendUDP(string messageToDroid, string ipAddress)
    {
        IPEndPoint ipep;
        //string ipAddress = "10.40.167.147";
        int portNumber = 9050;
        data = new byte[1024];
        ipep = new IPEndPoint(IPAddress.Parse(ipAddress), portNumber);
        Socket server = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);

        // convert amount to string to send to server
        data = Encoding.ASCII.GetBytes(messageToDroid);
        // try sending the order to the bartender
        try
        {
            server.SendTo(data, data.Length, SocketFlags.None, ipep);
        }
        catch (SocketException ex)
        {
            string message = ex.Message;
            System.Windows.Forms.MessageBox.Show(message);
        }
    }


    public void sendSMS(string textBody, string number)
    {
        /*if (number != "macomber.alex@gmail.com")
        {
            MailMessage message = new MailMessage();
            number = number + "@vtext.com";
            message.From = new MailAddress("AutoBev@gmail.com", "AutoBev");
            message.To.Add(number);
            message.Subject = "AutoBev: Your drink is ready";
            message.Body = textBody;

            SmtpClient smtp = new SmtpClient("smtp.gmail.com");
            smtp.UseDefaultCredentials = false;
            smtp.Credentials = new NetworkCredential("AutoBev@gmail.com", "NotreDame"); //NotreDame is the password of the
AutoBev@gmail.com

            smtp.EnableSsl = true;
            smtp.Port = 587;
            smtp.DeliveryMethod = SmtpDeliveryMethod.Network;
            try
            {
                smtp.Send(message);
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
            number = "eclark4@nd.edu";
        }*/

        MailMessage messageText = new MailMessage();
        messageText.From = new MailAddress("AutoBev@gmail.com", "AutoBev");
        messageText.To.Add(number);
        messageText.Subject = "AutoBev: Your drink is ready";
        messageText.Body = textBody;

        SmtpClient smtpText = new SmtpClient("smtp.gmail.com");
```

```
        smtpText.UseDefaultCredentials = false;
        smtpText.Credentials = new NetworkCredential("AutoBev@gmail.com", "NotreDame"); //NotreDame is the password of the
AutoBev@gmail.com

        smtpText.EnableSsl = true;
        smtpText.Port = 587;
        //smtp.DeliveryMethod = SmtpDeliveryMethod.SpecifiedPickupDirectory;
        // smtp.DeliveryMethod = SmtpDeliveryMethod.PickupDirectoryFromIis;//
        smtpText.DeliveryMethod = SmtpDeliveryMethod.Network;
        try
        {
           smtpText.Send(messageText);
           //  System.Windows.Forms.MessageBox.Show("message sent:"+message);

        }
        catch (Exception ex)
        {
           MessageBox.Show(ex.Message);
        }

     }


     public void updatePricesButton_Click(object sender, EventArgs e)
     {
        DrinkPrices dp = new DrinkPrices();
        dp.ShowDialog();
        decimal[] drinkPrices = new decimal[7];

        TextReader tr = new StreamReader("C:\\Users\\lgarcia\\Desktop\\DrinkPrice.txt");

        //while (tr.Peek() > -1)
        //{
        // read a line of text
        char[] delimiterChar = { '/' };
        drinkPriceList = tr.ReadLine();
        drinkPriceArray = drinkPriceList.Split(delimiterChar);
        droidPriceList = "4#" + drinkPriceArray[0] + "#" + drinkPriceArray[1] + "#" + drinkPriceArray[2] + "#" + drinkPriceArray[3] + "#" +
drinkPriceArray[4] +
           "#" + drinkPriceArray[4] + "#" + drinkPriceArray[5] + "#" + drinkPriceArray[6] + "#" + drinkPriceArray[7] + "#" + drinkPriceArray[8] + "#"
+ drinkPriceArray[9] +
           "#" + drinkPriceArray[10] + "#" + drinkPriceArray[11] + "#";

        tr.Close();
     }

     public void initializePrices()
     {
        DrinkPrices dp = new DrinkPrices();
        dp.ShowDialog();
        decimal[] drinkPrices = new decimal[7];
        TextReader tr = new StreamReader(drinkPricePath);
        char[] delimiterChar = { '/' };
        drinkPriceList = tr.ReadLine();
        drinkPriceArray = drinkPriceList.Split(delimiterChar);
        droidPriceList = "4#" + drinkPriceArray[0] + "#" + drinkPriceArray[1] + "#" + drinkPriceArray[2] + "#" + drinkPriceArray[3] + "#" +
drinkPriceArray[4] +
           "#" + drinkPriceArray[4] + "#" + drinkPriceArray[5] + "#" + drinkPriceArray[6] + "#" + drinkPriceArray[7] + "#" + drinkPriceArray[8] + "#"
+ drinkPriceArray[9] +
           "#" + drinkPriceArray[10] + "#" + drinkPriceArray[11] + "#";
        tr.Close();
     }

     public void updateDrinks()
     {
        string message = drinkPriceList;
```

```
    data2 = Encoding.ASCII.GetBytes(message);
    data = new byte[1024];
    newsock.SendTo(data2, data2.Length, SocketFlags.None, Remote);
}

private void button2_Click(object sender, EventArgs e)
{
    this.Refresh();
}

private void clearButton_Click(object sender, EventArgs e)
{
    customersTableAdapter.ClearTable();
    // Update DataGridView
    customersTableAdapter.Fill(bartenderDBDataSet.Customers);
    clearButton.Visible = false;
    saveButton.Visible = true;
}



private void fileName_KeyDown(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Enter)
    {
        string FileName = path + fileName.Text + ".txt";
        // check to see if the file exists
        if (File.Exists(FileName))
        { // if it does, load it
            // Read data from fileName in XML format
            customersTableAdapter.ClearTable();
            customersTableAdapter.Fill(bartenderDBDataSet.Customers);
            bartenderDBDataSet.ReadXml(FileName);
            // Update DataGridView on form
            //  customersTableAdapter.Fill(db_4_1DataSet.customers);
            fileName.Visible = false;
            clearButton.Visible = true;
            saveButton.Visible = false;
        }
        //otherwise tell the user that the file does no exist
        else
        {
            System.Windows.Forms.MessageBox.Show("File For Selected Date Has Not Been Created Yet");
        }

    }


}

private void customersDataGridView_DataError(object sender, DataGridViewDataErrorEventArgs anError)
{

    //MessageBox.Show("Error happened " + anError.Context.ToString());

    if (anError.Context == DataGridViewDataErrorContexts.Commit)
    {
        MessageBox.Show("Commit error");
    }
    if (anError.Context == DataGridViewDataErrorContexts.CurrentCellChange)
    {
        MessageBox.Show("Cell change");
    }
    if (anError.Context == DataGridViewDataErrorContexts.Parsing)
    {
        MessageBox.Show("parsing error");
```

```
        }
        if (anError.Context == DataGridViewDataErrorContexts.LeaveControl)
        {
            MessageBox.Show("leave control error");
        }

        if ((anError.Exception) is ConstraintException)
        {
            DataGridView view = (DataGridView)sender;
            view.Rows[anError.RowIndex].ErrorText = "an error";
            view.Rows[anError.RowIndex].Cells[anError.ColumnIndex].ErrorText = "an error";

            anError.ThrowException = false;
        }
    }

    private void closeButton_Click(object sender, EventArgs e)
    {
        this.Close();
    }
  }
}
```

7.7 Android Application Main Code: AutoBev.java

```
// AutoBev.java
// AutoBev kiosk emulated on smartphone
// ND EE Senior Design May 2011
// Alex Macomber, Liz Clark, Lori Garcia, Mark Pomerenke

package autoBev.android;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.text.DecimalFormat;
import java.util.Enumeration;
import java.net.NetworkInterface;
import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.CheckBox;
import android.widget.EditText;
import android.telephony.TelephonyManager;
import android.app.AlertDialog;
import android.app.AlertDialog.Builder;
import android.content.Context;
import android.content.DialogInterface;

import android.content.SharedPreferences;

public class AutoBev extends Activity {
	Thread UDPserverThread ;
	// Preference file used for saving variables
   public static final String PREFS_NAME = "MyPrefsFile";

	// Define buttons,checkboxes and textboxes so they can be changed later
	public Button button2,button3,button4,button01,button02,button03;
	CheckBox checkBox1,checkBox2;
	private EditText username, pin;

	// Client/server vars
	DatagramSocket socket;
	DatagramPacket packet;
	InetAddress address;

	// global vars
	boolean receive = false;
	String[] rcvMessage = {"0"};

	// Define prices and drinks
	double[] prices = new double[] {2.00,2.50,2.50,3.00,1.99,1.00};
	String[] drinks = new String[] {"Pina Colada", "Long Island", "Margarita", "Shirley Temple", "Rootbeer Float", "H2O"};

	// Set decimal format so prices show up as $X.XX
	DecimalFormat priceFormatter = new DecimalFormat("$#0.00");

	@Override
	public void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);
		setContentView(R.layout.main);

		// Set buttons to show drink name and price
		button2 = (Button) this.findViewById(R.id.button2);button3 = (Button) this.findViewById(R.id.button3); button4 = (Button)
this.findViewById(R.id.button4);
		button01 = (Button) this.findViewById(R.id.Button01); button02 = (Button) this.findViewById(R.id.Button02); button03 = (Button)
this.findViewById(R.id.Button03);
```

```
        button2.setText(drinks[0]+"\n"+priceFormatter.format(prices[0])); button3.setText(drinks[1]+"\n"+priceFormatter.format(prices[1]));
button4.setText(drinks[2]+"\n"+priceFormatter.format(prices[2]));
        button01.setText(drinks[3]+"\n"+priceFormatter.format(prices[3]));
button02.setText(drinks[4]+"\n"+priceFormatter.format(prices[4])); button03.setText(drinks[5]+"\n"+priceFormatter.format(prices[5]));

        // Initialize checkboxes
        checkBox1 = (CheckBox) findViewById(R.id.checkBox1);
        checkBox2 = (CheckBox) findViewById(R.id.checkBox2);

        // Initialize textboxes
        username = (EditText) findViewById(R.id.editText1);
        pin = (EditText) findViewById(R.id.editText2);
    UDPserverThread = new Thread(new Runnable() { public void run() { UDPServer(); }});
    UDPserverThread.start();

    } // End onCreate

/////// Load previously saved username/password
    protected void onResume(){
      super.onResume();
      SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
      String savedUserName = settings.getString("username", "");
      String savedPin = settings.getString("pin", "");
      username.setText(savedUserName);
      pin.setText(savedPin);
    }


//////// These methods are called on button click ////////
    public void drink1_clicked(View view) { // User clicked Drink 1
        sendToServer(drinks[0],prices[0]);
    }
    public void drink2_clicked(View view) { // User clicked Drink 2
        sendToServer(drinks[1],prices[1]);
    }
    public void drink3_clicked(View view) { // User clicked Drink 3
        sendToServer(drinks[2],prices[2]);
    }
    public void drink4_clicked(View view) { // User clicked Drink 4
        sendToServer(drinks[3],prices[3]);
    }
    public void drink5_clicked(View view) { // User clicked Drink 5
        sendToServer(drinks[4],prices[4]);
    }
    public void drink6_clicked(View view) { // User clicked Drink 6
        sendToServer(drinks[5],prices[5]);
    }

    // Function that sends drink to server
    public void sendToServer(final String drink, double price){
        // Check if username/pin is of correct format
        boolean correctInfo = true;
        // Check if username is all letters
        if ((!isLetters(username.getText().toString()) || username.getText().toString().length() == 0)) {
            alertbox("Your username must contain letters only.");
            correctInfo = false;
        }
        // Check if pin is a 4 digit number
        if ((correctInfo) && ((!isInteger(pin.getText().toString())) || (numDigits(pin.getText().toString()) != 4))) {
            alertbox("Your pin must be 4 numbers.");
            correctInfo = false;
        }

        // If the information is correct open a dialog box and send order to server
        if (correctInfo) {
              // Open dialog to confirm order
```

```
Builder builder = new AlertDialog.Builder(this);
builder.setMessage("This will charge your credit card " + priceFormatter.format(price) + ".  Proceed?")
    .setCancelable(true)

    // if they click yes below code runs
    .setPositiveButton("Yes, order a " + drink,
            new DialogInterface.OnClickListener() {
                @Override

                public void onClick(DialogInterface dialog,
                        int which) {
                    // Save username and password
                saveInformation();

                    // Send drink to server
                    UDP_Send(drink);

                // wait til receive from bartender
                    int n = 0; double y; int limit = 100000000;
                if (!receive) {
                    while (!receive & n < limit) {
                        n++; y = 3*3*2.3+56; // this is used to timeout if no response
                    } // timeout after certain amount of itme
                }

                // If no response, start over
                if (n >= limit) {
                    alertbox("There was no response from the bartender.  Please try again.");
                }
                // if there is a response
                else {
                    // If order went through
                    if (rcvMessage[0].equals("1")) {
                        alertbox("Thank you for your purchase!  You're order #" + rcvMessage[1] +
                                ".\nCurrently serving order #" + rcvMessage[2] +
                                ".\nYou'll receive a text message when your drink is ready.");
                    }
                    // If username was wrong
                    else if(rcvMessage[0].equals("2")) {
                        alertbox("The username \"" + username.getText().toString() +
                                "\" does not exist.  Please setup an account at an " +
                                "AutoBev kiosk and try ordering a " + drink + " again.  " +
                                "Your credit card has not been charged.");
                    }
                    // If password was wrong
                    else if(rcvMessage[0].equals("3")) {
                        alertbox("Your password did not match the saved password for username " +
                                "\"" + username.getText().toString() + "\".  Please try ordering a " +
                                drink + " again.  Your credit card has not been charged.");
                    }
                } // end else
                // reset receive flag
                receive = false;
                }
    })
    // If they click no, nothing happens
    .setNegativeButton("No",
            new DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog,
                        int which) {
                    ////////////////////////////////////////
                    // if they click no this code runs //
                    ////////////////////////////////////////
                }
            }
```

```
                        );

                AlertDialog dialog = builder.create();
                dialog.show();
            }
        }   // end sentToServer function


// Thread that is constantly receiving packets
   public void UDPServer()
   {
      while(true) {
            final String TAG = "UDP playback";
            try {
                        byte[] buffer = new byte[140];
                        int port = 9050;
                // Create new UDP-Socket and packet
                DatagramSocket socket = new DatagramSocket(port);
                        DatagramPacket packet = new DatagramPacket(buffer, buffer.length );
              // Receive the UDP-Packet
                socket.receive(packet);
                // Set flag to tell main code that message has been received
                receive = true;

                rcvMessage = new String(buffer).split("#");
                }

                catch (Exception e) {
              }
            }
      }

//***************************** Functions *****************************//

////////// Function that sends UDP message to C# server //////////////
   public void UDP_Send(String drink){
            try {
            socket = new DatagramSocket();
               address = InetAddress.getByName("10.40.167.6");
               String port = "9050";
               int pnum = Integer.parseInt(port);
               byte[] messageBytes = new byte[0];

               // Get phone number of Android
                  String phoneNumber = "null";
                  try {
                        TelephonyManager tm = (TelephonyManager)
                        getSystemService(Context.TELEPHONY_SERVICE);
                       phoneNumber = tm.getLine1Number();
                  }
                  catch (Exception e) {
                  }

                  // Get IP address of droid
                  String IP = getIpAddress();

                  // Create complete message with num,username,pin, and order
                  String drinkComplete = "&" + phoneNumber + "#" + IP + "#" + username.getText() +
                                         "#" + pin.getText() + "#" + drink + "#";
                  // Convert message to bytes
                  messageBytes = drinkComplete.getBytes();

                  // Create datagram packet
                  packet = new DatagramPacket(messageBytes, messageBytes.length, address, pnum);
                  packet.setLength(messageBytes.length);
```

```
                // send packet to server
                socket.send(packet);
                // close socket
                socket.close();
  }
                catch (IOException io) {
                }
  }
////////// Function that returns true if string is made up of all letters //////////
    public boolean isLetters(String str)
    {
        for(int i = 0; i < str.length(); i++)
        {
            int Ascii = (int)str.charAt(i);
            if(Ascii < 65 || Ascii > 122){ // check if it not a letter
                return false; }
        }
        return true;  // if all letters
    }

////////// Function that returns true if string is made up of all numbers //////////
    public boolean isInteger(String i)
    {
        try
        {
            Integer.parseInt(i);
            return true;
        }
        catch(NumberFormatException nfe)
        {
            return false;
        }
    }

////////// Function that returns number of digits in integer //////////
    public int numDigits(String str){
        int x = Integer.parseInt(str);
        int count = 0;
        while(x > 0)
        {
          x  /= 10;
          count++;
        }
        return count;
    }

////////// Function that displays alert box ///////////////
    protected void alertbox(String mymessage)
      {
      new AlertDialog.Builder(this)
        .setMessage(mymessage)
        .setCancelable(true)
        .setNeutralButton(android.R.string.ok,
          new DialogInterface.OnClickListener() {
          public void onClick(DialogInterface dialog, int whichButton){}
          })
        .show();
      }

//////// Function that gets Ip address of device
  public String getIpAddress() {
    try {
      for (Enumeration<NetworkInterface> en = NetworkInterface.getNetworkInterfaces(); en.hasMoreElements();) {
        NetworkInterface intf = en.nextElement();
        for (Enumeration<InetAddress> enumIpAddr = intf.getInetAddresses(); enumIpAddr.hasMoreElements();) {
          InetAddress inetAddress = enumIpAddr.nextElement();
```

```
                if (!inetAddress.isLoopbackAddress()) {
                    return inetAddress.getHostAddress().toString();

                }
            }
        }
    } catch (Exception E) {
    }
    return null;
  }

////////// Function that saves username and password
  public void saveInformation() {
        if (checkBox1.isChecked()) {
            SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
                SharedPreferences.Editor editor = settings.edit();
                editor.putString("username",username.getText().toString());
                editor.commit();
        }
        // Save pin if save checkbox is checked
        if (checkBox2.isChecked()) {
            SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
                SharedPreferences.Editor editor = settings.edit();
                editor.putString("pin",pin.getText().toString());
                editor.commit();
        }
  }
} //end class
```

Layout Code: main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_height="fill_parent" android:layout_width="wrap_content" android:orientation="vertical">
   <TextView android:layout_width="wrap_content" android:textSize="30sp" android:text="@string/welcome" android:id="@+id/textView3"
android:layout_height="wrap_content" android:layout_gravity="center"></TextView>
   <TextView android:layout_height="wrap_content" android:textSize="16sp" android:id="@+id/welcomeText"
android:layout_width="fill_parent" android:layout_weight="1" android:gravity="center_vertical|center_horizontal"
android:text="@string/enterInfo"></TextView>
   <RelativeLayout android:layout_height="wrap_content" android:id="@+id/relativeLayout2" android:layout_width="fill_parent"
android:layout_weight="1">
      <TextView android:id="@+id/textView1" android:textSize="17sp" android:layout_alignParentLeft="true" android:layout_height="50dip"
android:text="@string/username" android:layout_width="85dip" android:gravity="center"></TextView>
      <EditText android:layout_height="50dip" android:id="@+id/editText1" android:layout_toRightOf="@+id/textView1"
android:layout_alignTop="@+id/textView1" android:layout_alignBottom="@+id/textView1" android:layout_width="155dip"></EditText>
      <CheckBox android:layout_height="wrap_content" android:text="Save" android:id="@+id/checkBox1"
android:layout_toRightOf="@+id/editText1" android:layout_alignTop="@+id/editText1" android:layout_alignBottom="@+id/editText1"
android:layout_gravity="left" android:layout_width="fill_parent"></CheckBox>
   </RelativeLayout>
   <RelativeLayout android:layout_height="wrap_content" android:id="@+id/relativeLayout3" android:layout_weight="1"
android:layout_width="wrap_content">
      <TextView android:id="@+id/textView2" android:textSize="17sp" android:layout_alignParentLeft="true" android:layout_height="50dip"
android:text="@string/pin" android:layout_width="85dip" android:gravity="center"></TextView>
      <EditText android:layout_height="50dip" android:id="@+id/editText2" android:layout_toRightOf="@+id/textView2"
android:layout_alignTop="@+id/textView2" android:layout_alignBottom="@+id/textView2" android:inputType="textPassword"
android:layout_width="155dip"></EditText>
      <CheckBox android:layout_height="wrap_content" android:text="Save" android:id="@+id/checkBox2"
android:layout_toRightOf="@+id/editText2" android:layout_alignTop="@+id/editText2" android:layout_alignBottom="@+id/editText2"
android:layout_width="wrap_content"></CheckBox>
   </RelativeLayout>
   <RelativeLayout android:id="@+id/relativeLayout1" android:layout_width="fill_parent" android:layout_height="wrap_content"
android:gravity="center" android:layout_weight="1">
      <Button android:id="@+id/button2" android:textSize="15dip" android:layout_alignParentTop="true"
android:layout_width="wrap_content" android:layout_height="wrap_content" android:onClick="drink1_clicked"
android:text="@string/drink1"></Button>
      <Button android:id="@+id/button3" android:textSize="15dip" android:layout_width="wrap_content"
android:layout_toRightOf="@+id/button2" android:layout_height="wrap_content" android:layout_alignTop="@+id/button2"
android:layout_alignBottom="@+id/button2" android:text="@string/drink2" android:onClick="drink2_clicked"></Button>
      <Button android:id="@+id/button4" android:textSize="15dip" android:layout_width="wrap_content"
android:layout_toRightOf="@+id/button3" android:layout_height="wrap_content" android:layout_alignTop="@+id/button3"
android:layout_alignBottom="@+id/button3" android:text="@string/drink3" android:onClick="drink3_clicked"></Button>
   </RelativeLayout>
   <RelativeLayout android:layout_height="wrap_content" android:id="@+id/RelativeLayout01" android:layout_width="fill_parent"
android:onClick="drink4_clicked" android:gravity="center" android:layout_weight="1">
      <Button android:layout_height="wrap_content" android:text="@string/drink4" android:onClick="drink4_clicked"
android:layout_width="wrap_content" android:layout_alignParentTop="true" android:id="@+id/Button01"></Button>
      <Button android:layout_height="wrap_content" android:layout_toRightOf="@+id/Button01" android:text="@string/drink5"
android:onClick="drink5_clicked" android:layout_width="wrap_content" android:id="@+id/Button02"
android:layout_alignTop="@+id/Button01" android:layout_alignBottom="@+id/Button01"></Button>
      <Button android:layout_height="wrap_content" android:textSize="15dip" android:layout_toRightOf="@+id/Button02"
android:text="@string/drink6" android:onClick="drink6_clicked" android:layout_width="wrap_content" android:id="@+id/Button03"
android:layout_alignTop="@+id/Button02" android:layout_alignBottom="@+id/Button02"></Button>
   </RelativeLayout>
</LinearLayout>
```