# Automated Espresso Machine

Isabella Gilman, Alexander Hudson, Ethan Melvick, Dawson Mildon
Computer Engineering, University Of Utah

***Abstract*—This report explains the work done to create an automated coffee machine. The original goal was to build an automated espresso machine, but after further exploration, it was discovered this task was too complicated to complete within the time restrictions. This document outlines the original design, changes made to the design to meet completion upon delivery date, and the engineering work that had to be done.**

***Index Terms*—coffee, espresso, automation, computer engineering, systems engineering**

## I. Introduction

Drinking coffee is a morning ritual for many Americans. It is customary for people to turn to caffeine as a way to help wake up in the morning, and a popular choice is coffee. According to the National Coffee Association, it is estimated that two-thirds of Americans drink coffee on a daily basis, most having 3 or more cups of coffee throughout the day [1]. With Americans consuming so much coffee daily, our group will seek to design and build a fully automated coffee machine, reducing the time Americans spend brewing their own coffee or waiting in line at a coffee shop.

With hundreds of different types of coffee drinks (especially when considering the different beans, roasts, grounds, and brews), it was important to limit the scope of this project while leaving room for expansion. Since espresso, and espresso-based drinks, are the most popular in the U.S. [1], our original goal was to create an automated espresso machine. However, once development started, it was discovered the project was more comprehensive than could be accomplished during the time frame. As such, the project was changed to an automated coffee machine.

This paper explains what our original design was, and what changes were made to it to limit the overall work requirements. This paper also goes into detail about the engineering work done by each member of the team which resulted in the final product.

## II. Related Work

We believe that our system is pretty unique. While there are fully automatic espresso machines such as the Bravilor Bonamat – Sego [2], we believe our system is largely not related to these machines. Our system is more similar to devices like vending machines, pinball machines, ATMs, and any other heavily mechanical device. Several mechanical subsystems will have to work in harmony with each other.

Some may argue the current fully automatic espresso machines are smaller and systematically superior to our proposed solution; we would argue that our solution is far more interesting than the fully automatic espresso machines found in offices and gas stations. The movement and system complexity will entertain users of our system providing novelty. This novelty provides value similar to the Makr Shakr (A robotic bar tender) [3]. While our system will not be using expensive robots, it will ultimately be extremely similar to the Reis & Irvy's frozen yogurt robot. Reis & Irvy's – The Future of Frozen Yogurt! [4]. This device has a moving arm that collects a cup, fills it with frozen yogurt, gets toppings, then dispenses it to the user. A system like this is exactly what we are looking for.

While Reis & Irvy do not have a publicly available datasheet, we have been able to find a few academic papers relevant to our topic. One such academic reference is a text book: Design of Embedded Robust Control Systems Using MATLAB / Simulink [5]. The topics in this text book are useful for validating our approach and are for the most part language agnostic. Working using the philosophy in the textbook will help us build a stable system.

Our system is fairly different because of the use of LabVIEW. While the graphical programming language is no stranger to control systems, including its use in the Hadron Collider, it is a language some members of the team are fairly unfamiliar with. We were able to identify academic writing on a LabVIEW based traffic controller [6].The article uses explicit examples for data communication, system design, and other useful code samples. Referencing this article will help the team understand what LabVIEW is, as well as how to use it.

A combined understanding of what is currently on the market, similar systems, and academic reviews of control systems is vital for our project success. Using each of these resources will help us develop a robust, reliable machine. These pieces of related work do not reflect all the related work in our industry. As we continue our development, we believe we will continue a find a large variety of systems related to espresso machines.

## III. Implementation

The implementation of our automated espresso machine is drastically different than that of our design in our intermediate project proposal. In our original design, we had planned to make the coffee completely from scratch starting from coffee beans and use a mechanical arm to ferry the beans to different the stations necessary to process them. A rough drawing of this original design is shown in Fig. 1. After deliberating on our design during the summer, we decided that this would be too mechanically challenging, given that none of us have extensive experience in mechanical engineering. Instead, we decided that we would automate an existing coffee machine. For this project we used a Keurig®K-Classic®and altered it so that it could be used by simply pressing a button on a digital user interface.

Still, this project is heavily mechanical and required much effort to put together. The following sections will go through the mechanical aspects of the implementation, including motions automated and motors used; the motor controller board, which facilitates control of the motors and onboard buttons of the Keurig®; the microcontroller, which contains the main control loop of the project; the user interface, which gathers input from the user; and finally the last section will describe the final working state of the project.

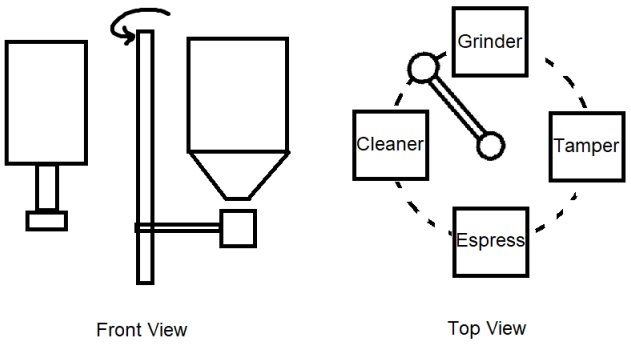


Fig. 1: Original project design

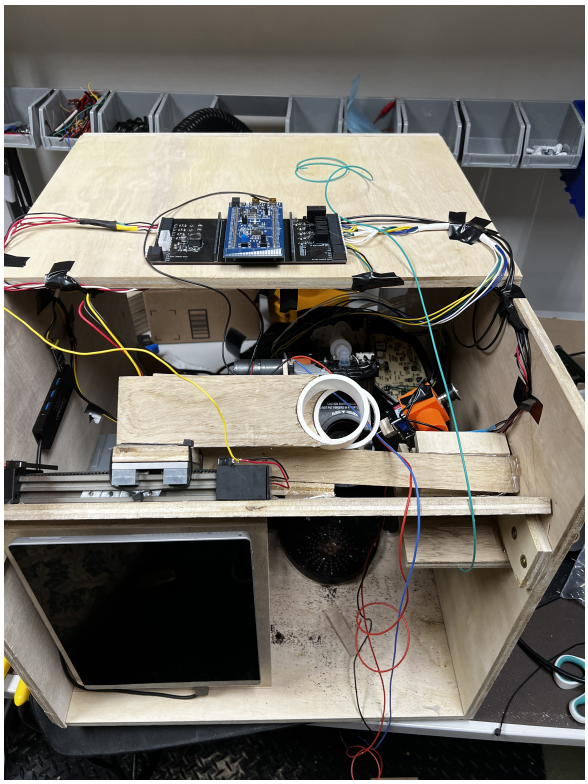

Fig. 2: Actual Implementation

### A. Mechanical Design

A Keurig®is convenient for our purposes because it comes with pre-packaged coffee flavors and a water-tight machanism for properly brewing the coffee. It also, however, requires the user to manually load the pod, open and close the lid, press the brew button, and finally unload the pod. To fully automate the Keurig®, there were four features we had to implement opening/closing the lid, loading the pod, unloading the pod, and brewing the coffee.

- Opening/Closing the Lid
  In the Keurig®'s unmodified form there was a handle that the user used to open and close the lid. This handle spun a small knob that we were able to connect a motor to on the side of the Keurig®after we removed the handle. The knob itself raised and lowered along with the lid, so we had to use the lid of the Keurig®as an anchor point for the motor. The lid is spring-loaded to force the lid into either an open or closed state, and because of this the force needed to open or close the machine was greater than we initially expected and our first motor was not strong enough. We ended up using a 499:1 Metal Gearmotor (Polulu item #1591). This gave us enough force to overcome the spring and open/close the Keurig®.
- Loading the Pod
  The pod loading mechanism utilized a linear actuator powered by a 488:1 Metal Gearmotor (Polulu item #3711). The linear actuator was connected to a short PVC pipe that contained the desired coffee pod. When the pod needed to be loaded into the machine, the linear actuator would push the PVC pipe (with the pod inside) over until the pipe lined up with a small chute. The pod could then fall into the chute, landing in the opened Keurig®. The force of landing in the Keurig®would not be enough to puncture the pod on the needle inside the Keurig®, but the force of closing the lid with the pod inside would be enough to do this. In the future, the automated coffee machine would be able to hold more than one pod by simply adding a longer PVC pipe above where the existing one sits in resting position, so the pods could fall into place sequentially.
- Unloading the Pod
  The unloading mechanism was much more complicated and we were not able to get it working reliably. Originally we tried to use a solenoid to kick the pod out, but the solenoid we were using did not have enough travel to completely remove the pod. Instead we put together a two-phase system, in which a needle was attached the solenoid, which spun about the axis of another brushed motor. When it was time to remove the cup, the solenoid and needle spun into place, the solenoid would fire, and the needle would attempt to pierce the cup, before the needle and cup would spin back out of the way. Unfortunately, in either design we were unable to remove the cup.
- Brewing the Coffee
  The Keurig®was already able to brew coffee at the push of a button. For us to control this process we simply hijacked their control board and simulated the button presses that would trigger the brewing process. We accomplished this by shorting out the buttons with our own relays located on our motor controller board.

In addition to these mechanical processes, we also needed a way to calibrate our motors and track position. We had two ways to do this: current sense and limit switches. The premise of current sense is to use spikes in current to gauge whether or not we've hit a wall and need to stop the motor. Initially, we tried to use current sense for opening and closing the lid, however, we found that in opening the lid the spike in current was actually greater while overriding the spring loaded mechanism than it was when hitting the limit when fully opening the lid. This observation is demonstrated in Fig. 3. Because of this, limit switches were used to determine stopping points for the motors while opening the lid, closing the lid, and moving the linear actuator. The outputs of the limit switches were plugged directly into inputs of the microcontroller.

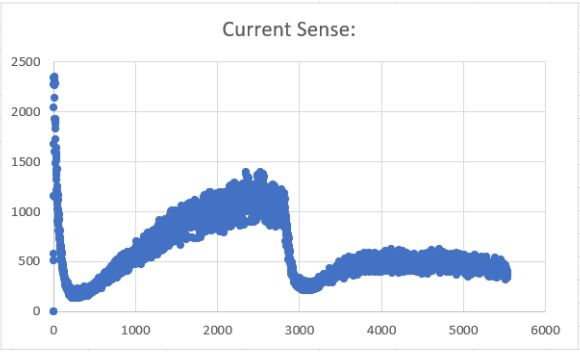


Fig. 3: Graph of current sense output over time

A full list of all mechanical components used is shown below:

TABLE I: Mechanical Components Used

| Component: | Type: | # Used: | Purpose: |
|---|---|---|---|
| Polulu Metal Gearmotor #1591 | Brushed Motor | 1 | Open/Close the lid |
| Polulu Metal Gearmotor #3711 | Brushed Motor | 2 | Drive linear actuator and rotate pod removal mechanism |
| 70155K76 | Linear Solenoid | 1 | Remove Pod |
| DBWDKG-FT01 | Limit Switch | 4 | Track position of motors |

## B. Motor Control Board

The motor control board allows our microcontroller to interface with the motors. The board is capable of controlling four brushed motors, one solenoid, and four relays. The motor control board was powered by a computer power supply that provided a 12V and a 5V rail. The PCB of the motor control board was designed so that the microcontroller could be plugged directly into the center of the board, with corresponding pin headers directly to the side of every microcontroller pin for debugging. The PCB is shown in Fig. 4.

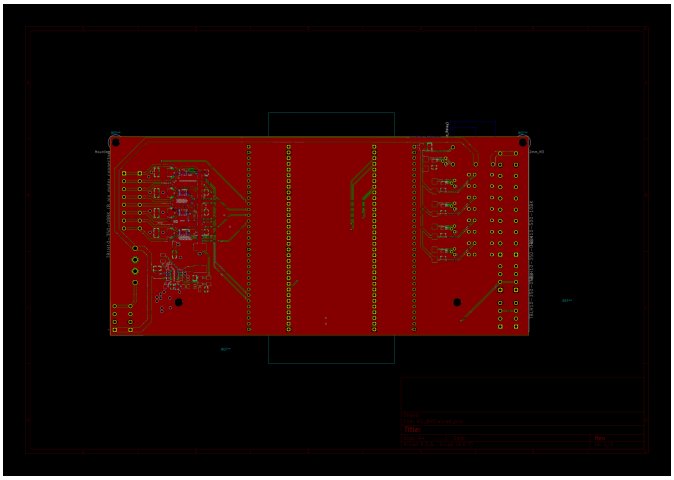

Fig. 4: Motor controller PCB

*1) Brushed Motors:* The four motors are powered by a 6V rail derived by stepping down the power supply's 12V using a buck converter. Each of the motors are controlled via an H-bridge, which allows us directional control in addition to simply being turned on or off. Each motor also generates a current sense output, whose voltage is amplified through an op-amp before being sent back to the microcontroller. Each amplifier is powered through the 3V rail on the microcontroller. In total the microcontroller has two motor control output channels and one input current sense channel per motor. Reserved pins will be described in a later section.

*2) Solenoid:* The solenoid is powered directly from the 12V power supply and only requires one channel to activate. The solenoid is activated when a fifth relay - reserved only for the solenoid - connects the solenoid to the 12V source.

*3) Relays:* The relays (excluding the one connected to the solenoid) are responsible for shorting out the buttons located on the Keurig®. They are powered through the 5V supply of the molex connector on the power supply. Each relay uses up one channel on the microcontroller.

## C. Microcontroller

The microcontroller we use for this project is the STM32F072R8 Discovery Board from ST Microelectronics. It facilitates communication between the user interface and the motors at the command of the LabVIEW code.

*1) USART Communication and Control Flow:* The main control flow of this project is fully dependent on the communication between the LabVIEW-based user interface and the microcontroller. The LabVIEW code enforces the order of operations that the Keurig®should perform and requests the microcontroller perform each function. Before the LabVIEW code sends a command over USART, the microcontroller remains completely inactive, besides flashing an LED light to signal a "heartbeat". When the LabVIEW code sends a command, it triggers an interrupt that will then decode the command in the C code's comm's module, perform the requested task if valid, and then return a status code. Because of this interrupt based approach, the commands run asynchronously and the microcontroller will not become locked

up while performing a task, enabling the GUI to query the microcontroller on the state of the system at any time. The implementation of the C code will be described in more detail in the rest of this section.

As stated above, the process begins when the LabVIEW code sends an instruction to the microcontroller via their UART connection. Each instruction is four bytes long, the first byte being the opcode, the second two any payload contents, and the last being a checksum value. The interrupt the incoming UART message triggers is handled by the USART3_4_IRQHandler which calls our rxHanlder function with the data from our UART connection passed into it. The interrupt handler only allows us to access one byte at a time so each full instruction is pieced together through four calls to the rxHandler function, with each byte being written into its corresponding section of a struct called "receivedBytes". If we have successfully received all four bytes we can move onto the decode stage.

The decode function starts by checking the quality of the incoming data using the checksum value. The checksum that was sent by the LabVIEW code is determined mathematically by the values contained within the other bytes. The C code recalculates this and compares it to the sent checksum. If the recalculated and sent checksum values do not match, then the data has been corrupted and the microcontroller will send an error back to the labVIEW code. If the data is determined to be intact, then the operating state (a global variable) will be updated to the opcode of the incoming command. The five possible opcodes are shown in Table II.

The machine will finally act on an instruction from the main while loop. The main while loop is continually checking for the operating state to change. Once it does, it will run the appropriate subroutine. The composition of these subroutines are heavily reliant of the functionality of our motor driver, which provides functions for control of the motors and buttons. The motor driver will be discussed in the "Motor Board Driver" subsection.

TABLE II: UART communication standards

| LabVIEW ->Microcontroller | | | |
|---|---|---|---|
| Function | Opcode (8-bit) | Contents (16-bit) | Notes: |
| Reset() | 0x01 | None, pad with 0xFF | |
| FeedPod() | 0x02 | None, pad with 0xFF | |
| UnloadPod() | 0x03 | None, pad with 0xFF | |
| BrewCoffee(int size) | 0x04 | 1, 2, or 3 (for sm, med, lg) | |
| GetStatus() | 0x05 | None, pad with 0xFF | Expects 0x90-0xAF |

The final thing the microcontroller has to do after running the appropriate subroutine is to report back to the LabVIEW code. The transmit function is much simpler than the receive function and only sends two bytes. The first being the status report opcode, and the second being the checksum value. Because there are only two bytes, the checksum value is simply a copy of the opcode. Sending is done by simply loading the transmit data register (USART3_TDR). All possible return values are listed in Table III.

TABLE III: Microcontroller response codes

| Microcontroller ->LabVIEW | | |
|---|---|---|
| Status Meaning: | Opcode: | Notes: |
| General Statuses | | |
| Task Started | 0xFA | |
| Check Value Fault | 0x01 | |
| Receiver Fault | 0x02 | Not enough bytes received |
| Blocked | 0x03 | Device is already running |
| Unresolved Fault | 0x04 | Device encountered critical error and has not been reset |
| Bad Payload | 0x05 | |
| FeedPod() Faults | | |
| Out of Pods | 0x31 | |
| BrewCoffee(int size) Faults | | |
| Out of Water | 0x71 | |
| GetStatus() Responses | | |
| Device in Undefined Fault State | 0x90 | |
| Device Idle | 0x91 | |
| Device Reset | 0x92 | |
| Loading Pod | 0x93 | |
| Unloading Pod | 0x94 | |
| Brewing Coffee | 0x95 | |

*2) Motor Board Driver:* The motor board driver contains many macros to control simple motor movement by changing settings in the GPIO pin's control registers. Pin assignments can be seen in Table IV. The driver offers seven functions for controls the components connected to the PCB.

- MBD_initFunctions()
- MBD_getMotorChannelFeedback(uint8_t channel)
- MBD_setRelay(uint8_t relayNumber)
- MBD_resetRelay(uint8_t relayNumber)
- MBD_toggleRelay(uint8_t relayNumber)
- MBD_popSolenoid()
- MBD_motorCommand(uint8_t channel, uint8_t command)

The initFunctions method needs to be run prior to using any functions provided by the driver. This function sets up all registers and timers needed by the driver.

The second method, getMotorChannelFeedback, reports back the current sense value. The channel parameter represents which motor you are requesting information from.

SetRelay, resetRelay, and toggleRelay are responsible for turning a relay on, off, or toggling its current setting, respectively. The relayNumber parameter specifies which relay you are modifying.

Recall that the solenoid is connected to a fifth relay. The popSolenoid method utilizes similar macros as those used in the relay functions to activate the solenoid for a brief period of time to create a "popping" action. This is achieved by turning on the solenoid's relay and then beginning a timer.

When the timer ends its handler resets the relay, completing the solenoid's popping action.

The motorCommands function is used to control the brushed motors. The function has channel and command parameters. The channel parameter specifies which motor you would like to control, while the command parameter specifies the type of action. Command 1 is forward, 2 is backward, 3 is stop, and 4 is stop all motors.

TABLE IV: Pin Assignments

| Pin | Alias | Purpose |
|---|---|---|
| I2C Pins | | |
| PB13 | I2C2_SCL | Clock Signal |
| PB14 | I2C2_SDA | Data Signal |
| UART Pins | | |
| PB10 | USART3_TX | Transmit |
| PB11 | USART3_RX | Receive |
| Analog (Current Sense) | | |
| PC0 | M1IS | Motor 1 Current Sense |
| PC1 | M2IS | Motor 2 Current Sense |
| PC2 | M3IS | Motor 3 Current Sense |
| PC3 | M4IS | Motor 4 Current Sense |
| GPIO (Motors) | | |
| PB0 | M1I1 | Motor 1 Input 1 |
| PB1 | M1I2 | Motor 1 Input 2 |
| PB3 | M2I1 | Motor 2 Input 1 |
| PB4 | M2I2 | Motor 2 Input 2 |
| PB5 | M3I1 | Motor 3 Input 1 |
| PB6 | M3I2 | Motor 3 Input 2 |
| PB8 | M4I1 | Motor 4 Input 1 |
| PB9 | M4I2 | Motor 4 Input 2 |
| GPIO (Buttons) | | |
| PC4 | R1T | |
| PC5 | R2T | |
| PC10 | R3T | |
| PC11 | R4T | |
| PC12 | SRT | |
| LED GPIO | | |
| PC6 | LD3 | Red LED |
| PC7 | LD6 | Blue LED |
| PC8 | LD4 | Orange LED |
| PC9 | LD5 | Green LED |

### D. LabVIEW

LabVIEW was used to create the user interface, with the interface itself running on a normal Windows 10 machine. The LabVIEW code connected to the UART pins on the microcontroller through one of the USB serial ports on the computer. The interface had 3 radio buttons for choosing the size (small, medium, or large) and a "Begin Brewing" button. In a display window to the right the GUI also displayed the current status of the coffee machine. A picture of the graphical user interface is shown in Fig. 5.

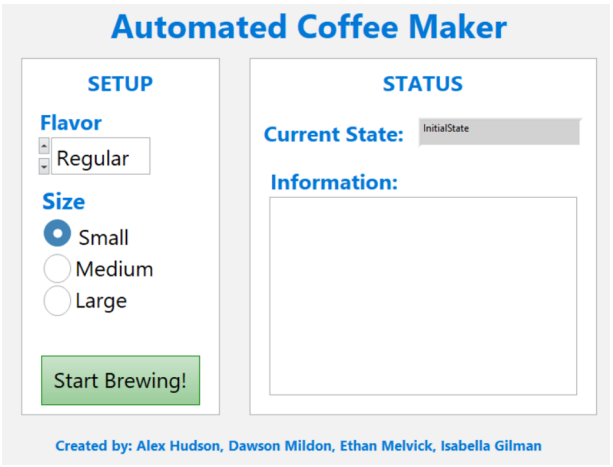


Fig. 5: User Interface

Besides the user interface, the LabVIEW code was also responsible for the control flow of the coffee machine (i.e. what steps had to be taken in what order). So when a user interacted with the user interface, it would initiate the coffee making process and the LabVIEW would instruct the microcontroller to take the first step, then the LabVIEW code would decide what to do next based on the responses from the microcontroller. This process will be discussed in more detail in the "Control Flow" subsection.

*1) Hardware Abstraction Layer:* The Testeract Hardware Abstraction Layer (HAL) is designed to address all of the common needs of a hardware abstraction layer. These include modularity, scalability, and abstracting commonly used parts into pre-packaged modules so the end user and developers of any extensible plugins to the HAL are required to do minimal work with maximal code reuse and can deploy the update separately from the rest of the HAL. The HAL was designed so it could be used in a way that would allow the state of Devices to be shared anywhere in the application space, including sharing between NI LabVIEW and NI TestStand. The HAL is not dependent on any specific connection type, and is not limited to any specific communication protocols or Device types. Any Device, with any well-defined interface can be integrated into the HAL, which made it perfect for our coffee machine. The HAL is built using LabVIEW Object Oriented Programming (LVOOP). It is class based and uses inheritance and containment to dynamically specify class and method definitions. Its ease of use, adaptability, and support made it perfect for our project.

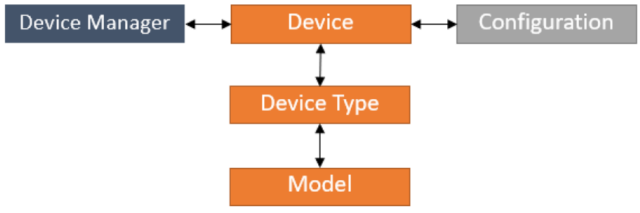


Fig. 6: Testeract HAL Architecture

We used the Testeract HAL by implementing a device

type called ”CoffeeMachine” to define all of the methods needed to interact with the micro controller code. This includes other methods such as initalize and close which managed the VISA/UART bus to support our protocol. After all of the methods were defined, two child / model classes were created: “Hardware” and “Simulation”. By using the already created Testeract device config editor, we could easily switch between the two implementation classes. Simulation was used when hardware was not available but work on the GUI needed to be done. It used a series of weights to emulate the hardware. The Hardware model classes implemented all the code for sending and receiving with hardware, This includes opcodes, managing device state, etc.

*2) Control Flow:* This GUI utilizes what is called a producer-consumer loop. This is two while loops running in parallel with each other. The producer loop is listening for user input, and the consumer loop is acting on those instructions. When the user presses “Start Brewing” the producer loop receives that command and queues the UnloadPod state. The consumer loop is constantly watching the queue, so when it receives the UnloadPod state, it begins the sequence to brew the coffee. First, it disables the controls so the user cannot send another command while the machine is busy. Then it sends the command to the machine to unload the pod. Once the device is idle, it can send the next command, which is LoadPod. Again, the LabVIEW code waits for the device to be idle before sending the BrewCoffee command. Then, the machine waits for the device to be idle before enabling the controls again.

If at any point the device returns an error, the error is caught by the LabVIEW code and reported to the user. This puts the consumer loop in the error state. The user then has the option to reinitialize the device, in which case the consumer loop goes to the Reset state, or the user can opt to close the program, which puts the consumer loop in the Close state. Utilizing the HAL made these states fairly basic, because each state was just making a call to the HAL which was doing all of the communication with the device.

### *E. Final Project State*

We managed to get our software and hardware completed early on, but struggled to finish the mechanical aspects of the project. In, hindsight we underestimated the risks that such a mechanically-intensive posed for us. As a result, we were able to open and close the lid and simulate pressing the brew and sizing options buttons, but had unreliable loading and unloading of the pod. Further because of these issues, we had trouble with the final steps of integration; namely, combining each of working discrete steps into one sequential process. Despite this, we are proud of the underlying infrastructure we created. By the end, we were able to create a sleek user interface, a reliable communication protocol, and were able to control an assortment of motors through the use of a custom made PCB and motor driver.

## IV. Acknowledgements

As a team, we are lucky to have an interested financer for our project. Sam Roundy, the financer is a founder of a local test and verification company called Testeract [7]. Testeract specializes in hardware control and testing specifically with LabVIEW. Sam is offering to fund the majority of the project if we use LabVIEW and integrate our project with some of Testeract’s tools. In order to develop in LabVIEW every group member who currently does not have access to the LabVIEW development environment will be given a license by Testeract if needed.

LabVIEW is an extremely niche, yet powerful programming language. It is a graphical programming language that was originally designed for electrical engineers that did not know how to program, and as a result looks like circuit diagrams. It allows us to easily create GUI’s and interface with hardware, as well as eliminate a large amount of GUI design overhead. Using LabVIEW will also allow us to use Testeracts internal tools; this includes using their Hardware Abstraction Layer (HAL) and their GUI debug tool called TASC. This will involve creating a custom device type and model for the HAL and creating custom GUI’s for TASC. As these are things that we would need to be doing anyways, using these tools will be beneficial. Furthermore, Testeract employs two of our group members (Dawson Mildon and Alex Hudson), who will lead the LabVIEW portion of development for our project and integration with Testeract’s tools.

Funding will be secured by the end of the semester after our bill of materials is created and our proposal is finalized. In return for the non-refundable engineering (NRE) costs, Sam will own the hardware and software created for the project. Our team will be able to re-use and redistribute any code created for the project through outlets like Github for career development. This excludes Testeract intellectual property like the HAL core and TASC. While unlikely, if there is a problem financing the project, Alex Hudson is willing to fund the majority of the project.

In addition to getting help with funding, Aria Burk and Chance Endo have agreed to be mechanical mentors to help with the mechanical engineering that will need to be done for this project. None of us have much experience with mechanical engineering, so it will be helpful to have people with experience to turn to and ask questions.

A huge thanks to Sam Roundy, Ryan Humbert, Aria Burk, and Chance Endo for their help on this project.

## References

[1] Nca releases atlas of american coffee. [Online]. Available: https://www.ncausa.org/Newsroom/NCA-releases-Atlas-of-American-Coffee

[2] Bean to cup systems coffee service–bean to cup. [Online]. Available: https://goldcupservices.com/bean-to-cup

[3] Your drink with a splash of robotics. [Online]. Available: https://www.makrshakr.com/

[4] (2021, Jan) The future of frozen yogurt! [Online]. Available: https://www.reisandirvys.com/

[5] P. Petko Hristov, S. Tsonyo Nikolaev, and K. Jordan Konstantinov, *Design of Embedded Robust Control Systems Using MATLAB® / Simulink®.*, ser. IET Control, Robotics and Sensors Series. The Institution of Engineering and Technology, 2018, no. Vol. 113. [Online]. Available: https://search-ebscohost-com.ezproxy.lib.utah.edu/login.aspx?direct=true&db=nlebk&AN=2000713&site=ehost-live

[6] A. Sachenko, O. Osolinskyi, P. Bykovyy, M. Dobrowolski, and V. Kochan, "Development of the flexible traffic control system using the labview and thingspeak," in *2020 IEEE 11th International Conference on Dependable Systems, Services and Technologies (DESSERT)*, 2020, pp. 326–330.
[7] Testeract home page. [Online]. Available: https://testeract.com/