Automated Keurig

Automate a Keurig coffee machine with an Arduino and Raspberry Pi. Makes coffee on a user-defined schedule and on-demand.

John Hatfield, Role: Hardware and Analog Software

Tony Kalil, Role: Interfacing and Digital Software

Shane Ryan, Role: Networking and Program Logic

1. Executive Summary

The objective of the Automated Keurig group was to reverse engineer a coffee machine and control it on a schedule. Research led to the use of a Keurig K40, Raspberry Pi Model B+, and Arduino Uno to accomplish this task. The following includes technical details of every group member's contribution, broken down by related components of the project.

First the hardware needed to interface with the coffee machine is covered, followed by the software based in C used to ultimately make coffee. This includes technical details along with challenges that arose in the process. Next, the software used in the Raspberry Pi is covered along with the networking involved in connecting devices and automating the process.

2. Introduction

The goal in this project was to design a "smart" coffee maker, capable of making coffee on a schedule or via an ssh connection. The team drew on knowledge developed in CSE20133 alongside hardware skills developed through Notre Dame's electrical engineering curriculum to complete the project. One driving factor in picking this project was its relevance to daily student life: The automatic Keurig seemed appropriate considering the fact that statistically most students consume at least one cup of coffee daily, and this project could streamline the process.

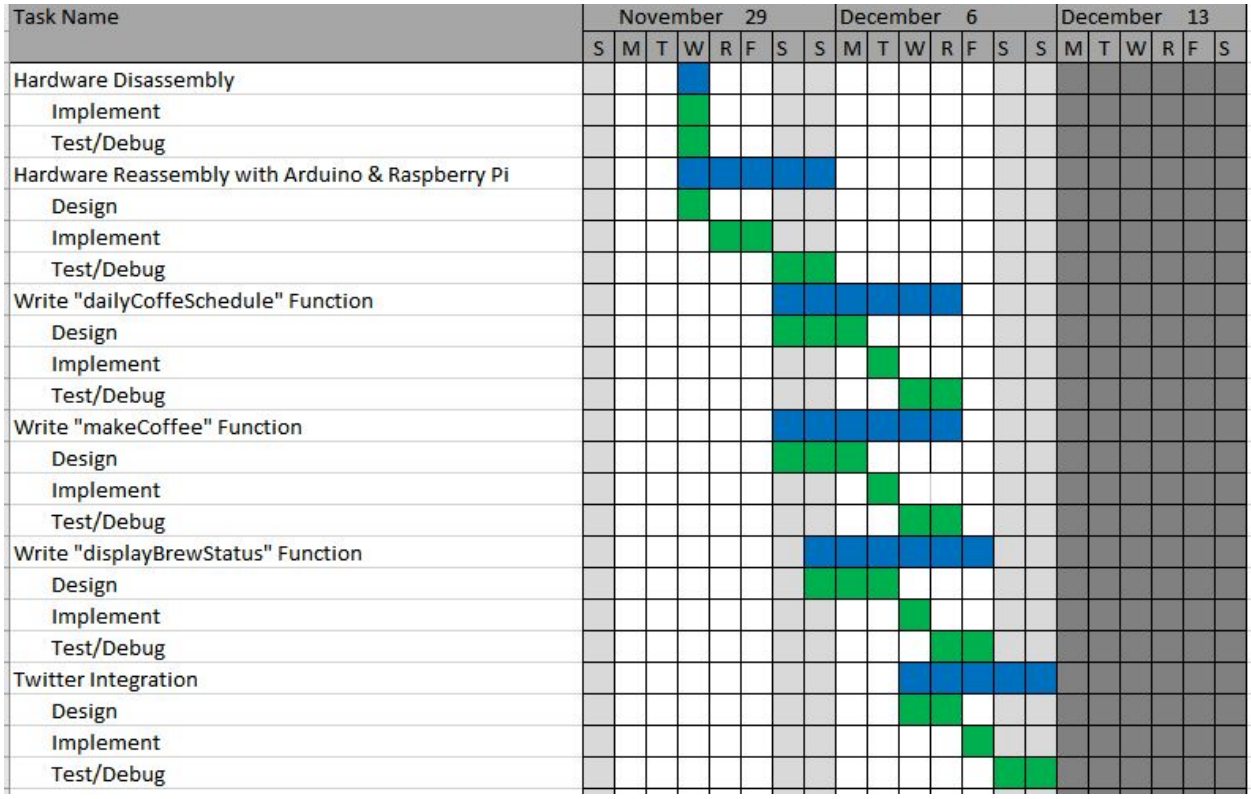
Some inspiration was drawn from the Rise and Dine project of last year's class. The Keurig can be set to a regular morning schedule, just like the aforementioned group's alarm clock could be programmed to the dining hall schedule. The Automated Keurig group similarly aimed to automate every function on the Keurig's inputs and outputs, including power, size selection, and status LEDs of add water, heating, descale, and powering off. This was realized through disassembling the Keurig and soldering control wires to the logic board of the Keurig K40 after careful study of its layout. We wired these into the input pins of an Arduino microcontroller.

The availability of well-documented, open-source hardware capable of controlling projects such as these was enormously helpful in breaking down the project into smaller components and realizing the final product.

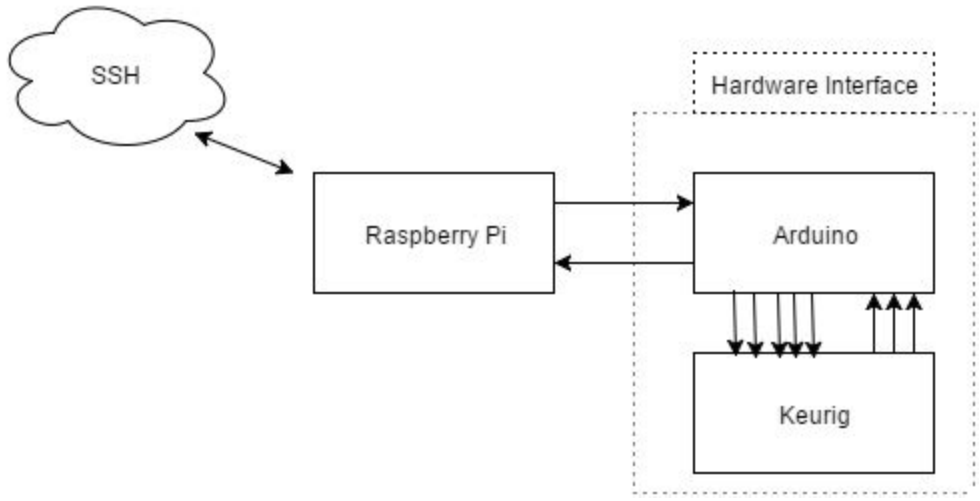
3. Methodology

3.1. Overview and Main Control Flow

At the project's conception, the first goal was verifying the plausibility of externally controlling a Keurig, and then developing a schedule for general implementation of the project. Below is the schedule developed that was mostly followed.

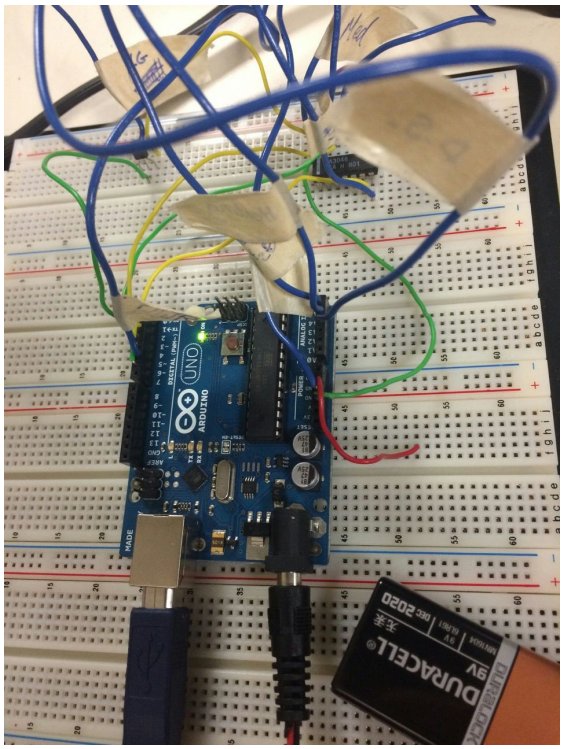


Here is a block diagram of the entire system that will help visualize the general functionality and allow exploration of specific tasks within each block:



The section within the dotted line contains the code for controlling the Keurig's hardware and properly brewing a cup of coffee. Five digital output pins on the Arduino are connected through transistor to the logic board in the Keurig to control Power, Open/Close State of the K-Cup chamber, and the size buttons Small, Medium, and Large. Three wires connected to the Arduino's analog input pins sample voltages at the positive leg of three status indication LEDs in the Keurig, which light when the machine is heating water, needs more water added, or needs to be descaled (cleaned).

The Raspberry Pi is connected to the Arduino through a USB to Serial connection thanks to the Arduino Uno's onboard serial to USB capability. The Arduino's main function is activated when information is received via this serial connection, and the integer received represents a size to brew. The Raspberry Pi runs an ARM compatible build of Arch Linux due to it's lightweight and modular nature, and can be manipulated via SSH from anywhere on Notre Dame's network. Although the Arduino does receive power from the USB cable, the Raspberry Pi is a low-power device that cannot provide sufficient power for correct operation of the Arduino. To circumvent this, we have also connected a 9V battery to the Arduino.



On the software side, the Arduino's main control waits for input on the serial connection, and upon receiving input, saves the input as the brew size and activates the makeCoffee() function, passing the desired size as the argument. The makeCoffee() function calls press_button() and ready_status() multiple times internally to manipulate the machine. press_button() takes an integer corresponding to the desired digital pin for button press simulation, and ready_status() returns an integer corresponding to the Keurig's operating state. Upon completion, makeCoffee() returns a value caught in the main loop corresponding to the result of the brew. This is passed to the Raspberry Pi using the Arduino's built-in Serial.println() function, and a lightweight terminal called minicom monitors the output. This picture shows the Arduino along with the serial connection at the bottom left, 9V connection at the bottom right, and digital output / analog inputs connected to the blue wires running to the Keurig's mainboard. The transistor array used to isolate digital output pins from sourcing current needed to simulate button presses is also visible on the breadboard above the Arduino. Also, relevant status information is displayed in the SSH terminal window.

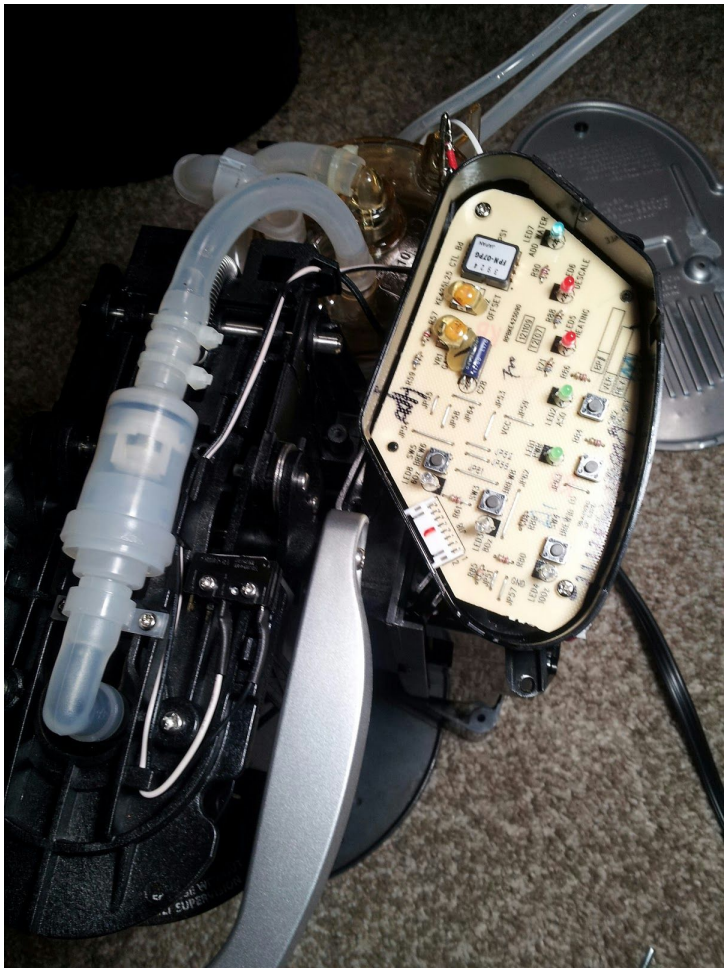
3.2. Main Technical Sub-Component #1: Hardware and Analog Software

Led by John Hatfield, assisted by Shane Ryan

The first task in the project was to make sure that the Keurig could be controlled electronically. This was accomplished by disassembling the casing of the Keurig and gaining access to its circuit board. Using a multimeter and a small power supply, we found that the functionality of the Keurig was controlled primarily by four buttons and a switch, and feedback could be obtained from three LEDs. The power button is connected to a status indicating LED, as are the three size selection buttons. Additionally. the handle that opens the Keurig triggers a

small limit switch when a new cup is inserted. This needs to be cycled after the water heating is complete for the Keurig to be able to brew correctly. We bypassed these buttons and wired the LEDs directly to our Arduino microcontroller, which allowed our software to control these hardware functions.

One of the hardware challenges we faced was figuring out where the legs of the relevant LEDs were on the back of the circuit board. We figured this out, and figured out which way the LEDs faced using a voltmeter. Small wires were soldered to all these connection points on the circuit board. The Arduino correctly provided 5V to all these buttons except for the Power button at first. This is because the power button needed a larger instantaneous amount of current than the Arduino could provide. Using a 9V battery to power the Arduino solved this. Later, the group switched to using an NPN transistor connected to each of the digital output pins to control the buttons. The Arduino pulsed the transistors which supplied 5V to the LEDs and enabled the hardware to function correctly.



Besides the button functionality, the Keurig has other native functions. After the Keurig is powered on, it heats for roughly a minute. While the Keurig is heating, a red “heating” LED is powered up. This is one of the analog inputs to our arduino. The arduino analog input can sense the voltage it is receiving from the heating LED input. While the heating LED is activated, there

is a dip in the voltage, and the voltage going into the analog input is decreased. This allows our software to sense when the Keurig is heating and output this information to the serial terminal.

We used the exact same approach with the other two Analog inputs. One of these was the "needs water LED". This was activated by a sensor when the water level in the Keurig dropped too low. The other analog input was the "needs descaling" LED. This basically lights up when the Keurig needs to be cleaned. We included all these analog inputs in our project for robustness of design. If the Keurig were operated on a schedule continuously, then eventually it would need to be refilled with water and descaled.

3.3. Main Technical Sub-Component #2: Interfacing and Digital Software

Led by Tony Kalil, assisted by Shane Ryan and John Hatfield

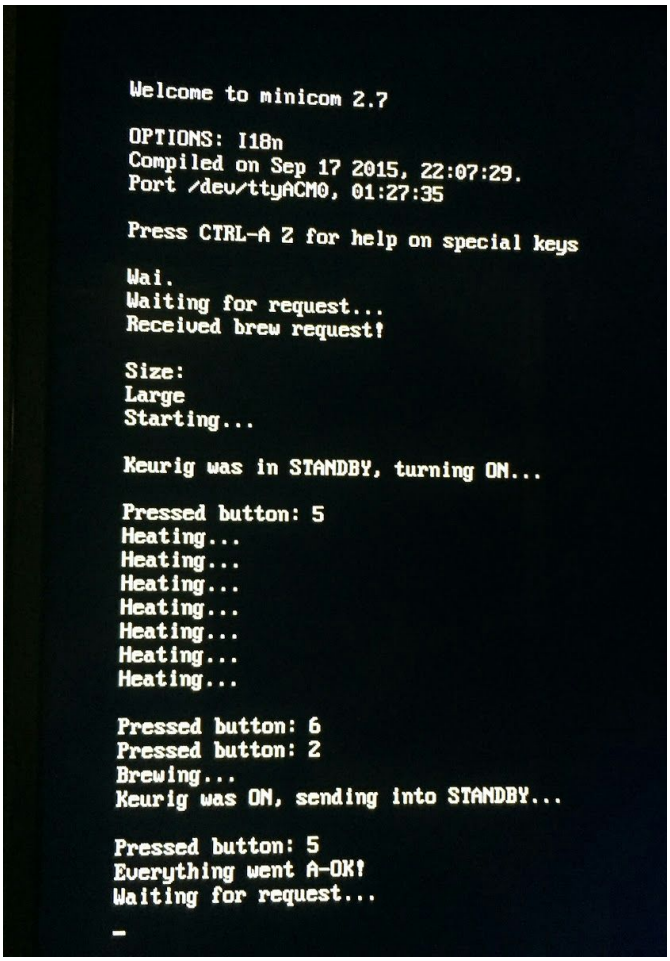
The next step in making the automated Keurig was to write the base Arduino code. This code would allow the Arduino to interface directly with the Keurig and have it perform a set of actions in a specific order, namely: turning the machine on/off, waiting for the machine to heat up the water, cycling the "open/close" switch, choosing the brew size, and pouring the coffee. One of the defining characteristics of the Arduino is that its main function is a continuously running loop--as long as the Arduino has power, it'll loop through the function. We therefore decided to use a while loop that simply checks for any serial data. Upon receiving it, the while loop goes to the makeCoffee() function and brews the cup of coffee. After finishing that function, the Arduino prints the exit status of the process to the terminal window and goes back to the beginning of the while loop again to continually check for new serial data.

After initializing all constants and variables and setting digital pins to OUTPUT mode in the void setup() function, the Arduino checks to see if the program is currently waiting. If not, it will print "Waiting for request…" on the serial terminal, and flip the waiting variable so as to not spam the output. Now data is sought on the serial connection, and if it is available, the program begins, using the data as a size variable. Due to the transmission of data in the serial terminal defaulting to character strings instead of integers, the input is converted to the appropriate integer selection by subtracting the ASCII char equivalent value for '0'. This makes the interface extremely modular, as now any device reporting a single value via serial can be used to control

the coffee machine. In our case, this device, the Raspberry Pi, is also capable of connecting to the internet, so there is no foreseen end to the variety of events or processes that could trigger.

The Arduino updates the terminal, saying “Received new brew request!” and reporting the size before starting. Now, the makeCoffee() function is called, using the size as an argument and capturing the return value to later be used as an end status at the output.

The function makeCoffee() relies on the functions press_button() and ready_status() heavily to operate. press_button() pulses the digital pin equal to the argument with which it was called in a way to emulate a physical button press. It also updates the terminal window with the current activity. brew_status() takes no arguments and returns the state of the Keurig via an integer. The current state is determined by sampling the voltage on both the descale and add_water pins, mapping the value of 0 to 1023 to a more useable voltage level between 0.0and 5.0. If either are less than 4V, the LED is on and the machine is currently in need of manual interaction to either fill water or clean the internals.



For actual brewing, makeCoffee() calls press_button(power_pin) to turn the machine from STANDBY to ON mode, and brew_status() to verify the Keurig is ready to brew. Upon completion, assuming the Keurig is in the ready state, the program enters a while loop checking the heating status of the water to update the user and pause the function until the water is properly heated. Now, the NC limit switch used to identify the open / closed state of the machine’s loading bay is pulsed with logic opposite than that of the press_button() function to move the Keurig into a size selection mode. At this point, the brew size button is pressed with press_button(selection), corresponding to the desired size passed from

serial input, and the program waits for 60 seconds before checking the brew_status() again. Now the power pin is pulsed to turn the machine back off, and the appropriate exits status is sent back to the void loop() function to be printed to the screen.

3.4. Main Technical Sub-Component #3: Networking and Program Logic

Led by Shane Ryan

Another aspect of this project that was largely outside the scope of our computing class was setting up the Raspberry Pi B+, installing the correct Linux packages to manage networking over SSH (openssh), connecting to the Arduino (minicom), creating a bash script to open and manipulate the serial connection, and automatically running such a script at a given time every morning (crontab). The group had some practical knowledge and background in this area, but Shane broke the problem into controllable tasks and implemented the final design.

Minicom was configured so as to operate at the correct baud rate and use required communication standards by default, and the Arduino was assigned a static serial port identity (/dev/ttyACM0), so simply opening minicom in the terminal with the Arduino connected would begin the serial connection. Minicom was largely used to troubleshoot and design the final functionality of scheduled brewing, which is realized via a bash script writing and reading from /dev/ttyACM0 like a file, but it can also be used to manually start the brewing process on-demand and present feedback in the terminal.

When a user logs in to the Raspberry Pi via SSH, they receive a shell session on the host that has permissions to run access needed files. Issuing the command “minicom” will open the serial connection to the Arduino and print “Waiting…”, and then sending a 2, 3, or 4 will brew a small, medium, or large. Relevant status information will be constantly updated in the terminal window.

The user can also modify the existing crontab run time by issuing the command “crontab” in the command line. For example, the following lines would brew coffee on weekdays at 7:00am and on weekends at 10:00am:

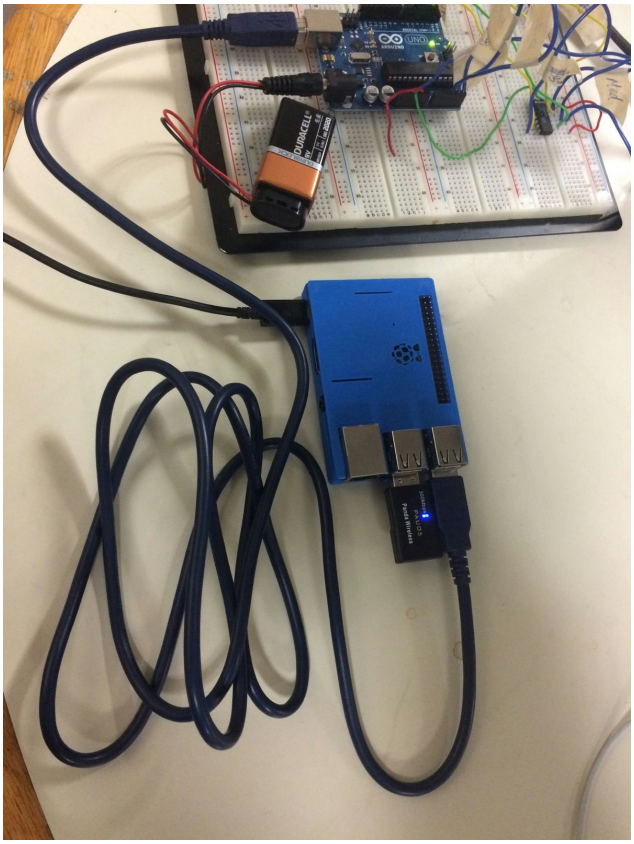
*** 7 * * 1-5 ./Make_Large.sh >/dev/null 2>&1**

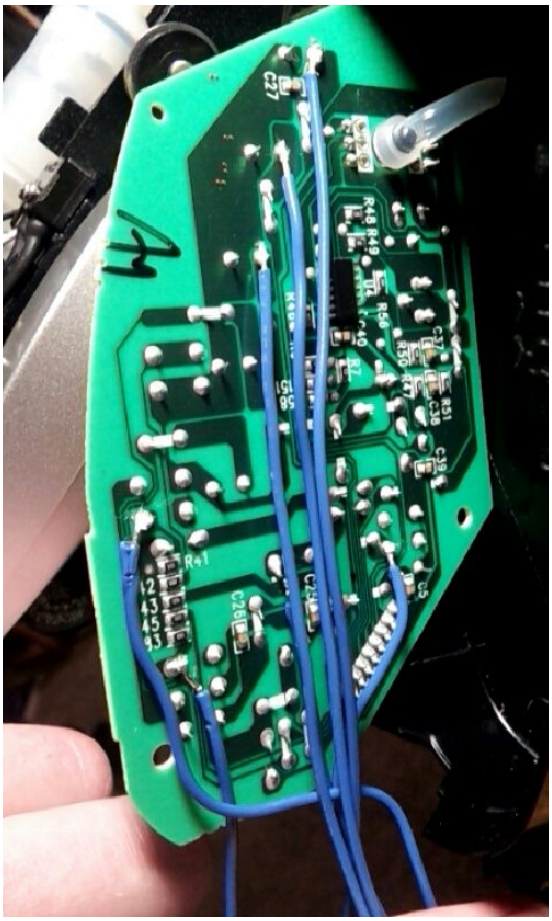
*** 10 * * 6-7 ./Make_Large.sh >/dev/null 2>&1**

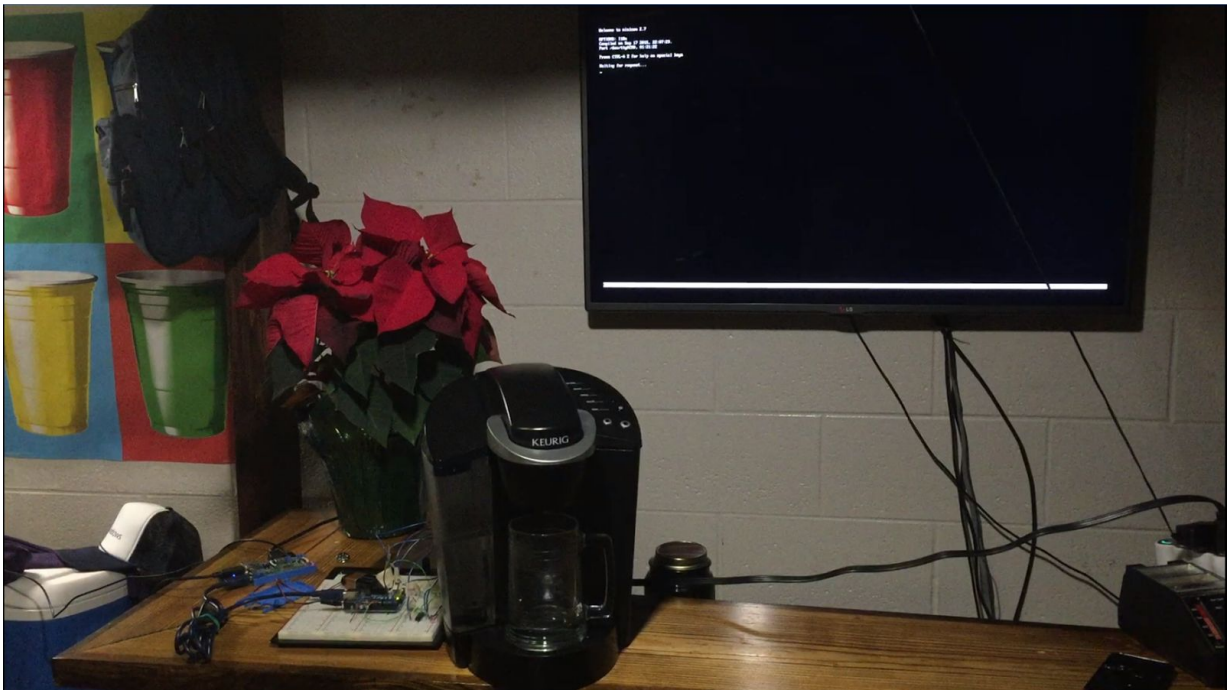
The bash script used to automate the process was called Make_Large.sh, which reads and writes to the serial port to appropriately initiate the coffee brew using shell commands. This fits well with the Arduino side of the project due to the simplicity of the communication between the devices.

4. Results and Discussion

Overall the group is satisfied with the project, and has met the main goals. The Keurig is able to operate remotely on demand, and on a pre-programmed schedule, and currently does so daily in one of the group member's rooms. Issues with the Raspberry Pi's uptime and reliability were addressed by switching from debian to Arch Linux, using a reliable power supply, and an authentic microSD card as a hard drive. A live demonstration of the final product can be seen here: https://www.youtube.com/watch?v=fpo8ZUwcHW0







5. Discussion and Future Work

One extension that time did not allow for this project was on-demand functionality via Twitter. This may be explored further over Christmas break. Another possible extension of the project could be to host a lightweight LAMP server on the Raspberry Pi to host a web app and port forward the address, effectively creating an extremely modular interface framework that a full-fledged GUI could be based on. Future groups could use this overarching networking idea to control a variety of hardware projects over the internet. Especially as Electrical Engineers, we need to be able to implement code in parallel with hardware knowledge.

Another possible hardware extension to our project is the use of the peristalsis pump from a Keurig as a component in some sort of drink-mixing machine. It could be programmed to

measure out precise amounts of liquid. The options for mixes could be set up as a user-friendly GUI, so users could mix and match drinks.