CARBURETOR AUTOMATOR

BRIDGING THE GAP BETWEEN CARBURETION AND EFI

TAYLOR ZINKE
XUECONG FAN

TAYLOR.A.ZINKE@GMAIL.COM
XUECONG.FAN@AGGIEMAIL.USU.EDU

1 MAY 2019

TABLE OF CONTENTS

1. EXECUTIVE SUMMARY

This report is written for the Electrical and Computer Engineering department of Utah State University. The report describes the Carburetor Automator system, which is a project designed to showcase the engineering aptitude of the students involved by developing a means to improve carbureted-engine performance.

Section 2 of this document introduces basic relevant information, discusses the impact of the project, and gives a general overview of the system.

Section 3 gives a detailed description of component selection and the work done in the scope of this project. Deeper and more technical information relevant to this project is found in this section.

Section 4 exhibits the results of the work done and the simulations performed during this project. Images of the system output can be seen in this section.

Section 5 provides a discussion of the results listed in Section 4. Here, the data seen in the images of Section 4 are clarified, and the future scope of the Carburetor Automator is considered.

Section 6 lists what we feel are the most important and impactful ideas/skills we learned through the scope of this project. This section concludes the report.

Sections 7 through 9 are appendices. The Carburetor Automator's microcontroller code and electrical design can be viewed in Sections 7 and 8, respectively. Section 9 lists a few websites and images that explain how our methods were derived.

2. INTRODUCTION

With ever-increasing stress on lowering environmental footprint, carburetors have been almost completely eliminated from the design of passenger and work vehicles. Starting in the 1970s, Electronic Fuel Injection (EFI) has replaced carburetion as a more accurate (and therefore "greener") solution. However, many vehicles with carburetors are still in use today. Most new, small engines, like those on motorcycles or lawn equipment, still use carburetion. This means that there are still many engines emitting greater quantities of unwanted gasses than they would with a better fuel-delivery system.

The Carburetor Automator aims to lower the disparity in emissions and performance between EFI and carburetion. Doing so will not only reduce harmful emissions, but it will also benefit the owner of the vehicle or gasoline-powered equipment financially. Use of the Carburetor Automator will reduce gasoline consumption, cause less wear in the combustion chamber(s) of the engine, and lengthen the life of components like sparkplugs and valves.

The goal of this project is to provide a baseline system that will achieve the above benefits to some degree. Decisions for this system were made under certain limitations: money, amount of time available, current knowledge of the mechanical and chemical processes in carburetion, and the type of engine available for system installation. We hope that we or some other interested entity will use the ideas behind our system, improve the methods, and use more accurate fuel-delivery models to more fully achieve the listed benefits.

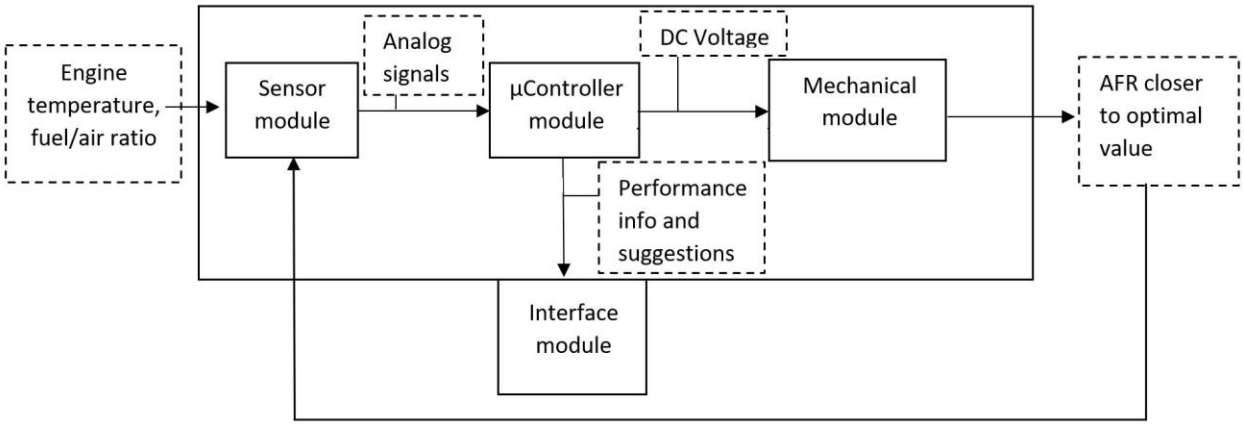


Figure 1. Block diagram with data flow

3. METHODS

To accomplish our goals for this system, we first needed to make some decisions for system components and interfaces. Several options were considered, but the criteria that each component needed to meet were straight forward enough that our conclusions were easily reached. All of the components were selected with the intent of simplifying the system, but a few of them lead to significant complications and postponement. This will be discussed further in the Results section of this document.

We used our knowledge of how EFI works to decide that we would use a Lambda sensor (the type of oxygen sensor used in modern EFI systems) in the engine's exhaust stream. This sensor creates a DC voltage signal proportional to the amount of oxygen that remains unburnt in the engine's exhaust. The system would then be able to make some adjustment to the carburetor to improve the air-to-fuel ratio (AFR) being fed to the engine.

The Lambda sensor allows the system to see how far it is from the desired AFR, but additional knowledge is needed to determine the desired AFR. The AFR should ideally change proportional to the engine temperature; that is, as the engine warms up, the desired AFR increases (to a certain extent). To capture the temperature of the engine, we decided to use a simple voltage divider using a fixed resistor coupled with a PTC thermistor. This sensor topology produces a voltage that varies proportionally to the temperature of the thermistor. We chose a PTC thermistor because they are durable, sensitive, and very inexpensive.

To read the signals from these sensors, we decided it would be easiest to use a printed circuit assembly with a programmable microcontroller, like an Arduino product, TI Launchpad, or STMicro discovery board. We wanted this project to be a challenge, thus giving us experience to benefit our careers and making the project look more impressive on a resume. This desire ruled out using an Arduino product, as they are designed to be used by hobbyists and beginners. Considering the alternatives, our decision was made simple by the fact that we already had an STMicro discovery board in our possession. The microcontroller we used to accomplish our project is an STM32L476VG, and the discovery board includes various GPIO pins and peripheral circuits to simplify the necessary hardware design.

For adjusting the AFR, our only viable options were to open/close the choke or make synchronized adjustments to the two pilot screws on the carburetor. Adjusting the pilot screws is an effective method of altering the AFR, but doing so would require the mechanical portion of our system to be in a spatially-restrictive and extremely hot area. Additionally, pilot screws are supposed to be in a fixed position as set by the manufacturer. Alternatively, the choke provides a singular interface to control the AFR to both cylinders and does not require the mechanical parts of the system to be near the engine, so our system uses the choke plunger to adjust the AFR. Given the motion of the choke plunger, we purchased the fastest linear-motion actuator that we could find for under $75.

Being able to adjust the AFR in both directions is a critical requirement for our system. For this to be possible, the carburetor has to be biased in such a way that the AFR is at its ideal value (around 14.7) with the choke somewhat engaged, whereas the ideal AFR should

normally be reached with the choke fully open (unengaged). This is done by carefully backing out the pilot screws while monitoring the AFR through the oxygen sensor until the desired point is reached. With the carburetor biased in this way, opening the choke will bring let more air into the engine than would be otherwise possible. This allows the Carburetor Automator to correct rich-running carburetor performance, which happens whenever the throttle is opened. A rich AFR is also what we want to avoid in order to achieve lower gas consumption, engine carbon build-up, and harmful emissions.

Finally, though it is not a critical requirement of the system, an LCD is included with the system to display sensor readings and give warning messages where necessary. The criteria for the LCD were as follows: it should add minimal overhead to system development, it should not use up many of the microcontroller GPIO pins, it should fit comfortably on the handlebar of a motorcycle, and it should not be so bright that it distracts the vehicle's operator. We found an LCD with four rows and 20 columns, is about six square inches, and includes an I^2C IO expander. The ability to control the display through I^2C was the most attractive feature, as that only requires two of the microcontroller GPIO pins and removes the complexity of addressing each data pin on the LCD. The LCD is comfortably bright, but since that is a subjective observation, the system includes an accessible killswitch to turn off the LCD.

A high-level functional overview of the system is found in the figure that follows.

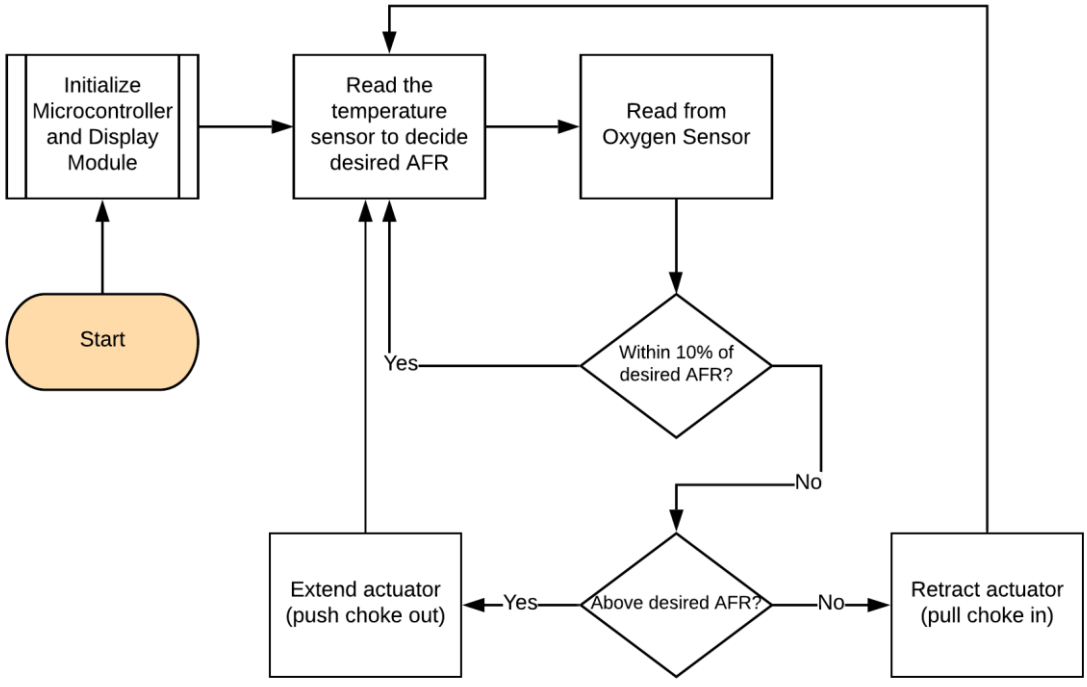


Figure 2. High-level functional diagram

The microcontroller configuration includes masking and writing several registers in the I^2C module to set timing, address mode, noise filters, and clock stretching. An analog-to-digital converter (ADC) is also configured by masking and setting register values to allow for two sequential, non-continuous ADC channels to be used through two of the GPIO pins. Finally, three interrupts are configured: one interrupt happens on a 50 ms period (the SysTick interrupt), another interrupt occurs when an ADC channel has finished a conversion, and the third interrupt is an edge-triggered (both rising- and falling-edge enabled) external interrupt through a GPIO pin to detect the power state of the Display Module.

In the microcontroller code, the main function runs through the configurations, assigns values to variables that are necessary for initialization, and sit in an infinite loop. Every 50 milliseconds, Systick interrupts the infinite loop. In the SysTick interrupt handler, a new pair of ADC conversions is completed. Based on the values produced and the state of the other modules, the microcontroller will make changes to the choke, print warning messages, and update the sensor values on the LCD in the scope of the SysTick interrupt handler. The interrupt is periodic and without drift, and the actuator that adjusts the choke moves at a set speed (16 mm per second, or 0.8 mm per SysTick period). This allows the microcontroller to determine the current position of the choke by tracking how long the actuator is turned on in either direction.

Since the microcontroller tracks choke position through timers, the choke must be fully open (actuator fully retracted) whenever the Mechanical Module switch is changed from OFF to ON. Otherwise, the microcontroller will think the choke is fully open when it is actually somewhat engaged. This situation will likely lead to the "Can't lean AFR" warning message, as the microcontroller will see that the AFR is too rich, but the choke cannot be opened any further. In this case, no adjustments will occur to correct the AFR. The full microcontroller code is found in Appendix A.

The hardware required for this project (additional to what is already provided by the discovery board) is also relatively simple. The actuator is controlled using an N-type MOSFET H-Bridge. N-type devices were chosen due to availability and cost. Since N-type MOSFETS make better pull-downs than pull-ups, two RTL-inverter networks drive each of the H-Bridge inputs. This forces the H-Bridge inputs to be active-high, which protects the actuator and MOSFET devices when the microcontroller is being configured. This inverter network also includes two normally-open push buttons, so the user can extend and retract the actuator manually. These buttons each drive one input high and ground the other, thus assuring that the H-Bridge will not be shorted. See Page 2 of the schematic under Appendix B.

Apart from the H-Bridge and its driver network, the rest of the hardware are simple voltage dividers for the temperature sensor and Mechanical-Module state pin, a linear 5V regulator with decoupling capacitors for overvoltage protection, the I^2C bus, and various other resistors to limit current flow to/from the microcontroller. We chose to have +12V_Sys drive the actuator H-Bridge instead of V_BAT because this requires the MAIN switch to be in the ON position, eliminating power leakage through the Mechanical Module's MOSFETs. The fixed resistance in the temperature sensor's voltage divider was chosen to minimize power

consumption. We wrote a C++ program that modeled the voltage divider with the thermistor at its lowest value (according to the datasheet).

In the C++ program, a for-loop altered the fixed-value resistor from one Ohm to one mega-Ohm in half-Ohm steps, and total power consumed was calculated for each case. With the fixed-value resistor at 1,894.50 Ohms, the worst-case power consumption of the voltage divider is 5.731 milliwatts, which is the lowest worst-case value found by the C++ program. Using E24-standard resistors, the combination that gets closest to this value is a two kilo-Ohm resistor in parallel with a 36 kilo-Ohm resistor for a value of 1,894.74.

This system will be installed on a 1982 Honda CB450SC motorcycle, but the motorcycle is currently not operational. Thus, to test our system, we created some voltage-divider circuits with potentiometers to simulate sensor data. These circuits are biased so that the output range includes the output range of the actual sensors for the project.

4. RESULTS

The LCD that we chose for the Display Module was selected for the simplicity that it offered – the I^2C bus requires only two GPIO pins and eliminates the need for synchronized data-setting functions. Configuring the I^2C bus on the microcontroller required us to go beyond our initial knowledge and embedded-system abilities. We had to read a significant amount of material detailing I^2C operations and the microcontroller's datasheet to learn its particular implementation. Once we calculated the required timing aspects and changed the other I^2C settings as needed, we used an oscilloscope and some known-value I^2C frames to confirm its operation (see Figure 3).

After verifying the I^2C bus, we wrote some functions to speak with the LCD. The LCD-drivers datasheet gives step-by-step instructions for configuring the LCD. Since the vendor of the LCD attached a break-out PCB with the I^2C IO expander, and the expander only has eight output pins, we had to configure the LCD-driver in 4-bit mode. This mode allows the LCD memory to be addressed with only the upper four (of eight) data pins, leaving the other four IO expander pins to control the read/$\overline{\text{write}}$, register select, and enable change pins.

The datasheet explains that each byte of data should be sent over two bytes, with the data bits in the upper four positions of the byte. The datasheet then states clearly that the upper four bits of the intended data byte should be sent before the lower four bits. We structured out code around this statement and thus spent about four weeks and many, many hours trying to get the LCD to work with faulty software.

Occasionally, a single, correct character would be printed in the position that we set it to, but the rest of the screen was printing garbage. We are unsure if this was a bad translation or some undiscovered mistake in our software's logic, but we noticed that sending palindromic bytes consistently produced the correct result, so we tried sending the lower four bits before the higher four bits. Upon doing so, the LCD finally accepted configuration commands and printed everything we were sending it. This set our project back significantly, as almost an entire month was dedicated solely to debugging this issue.

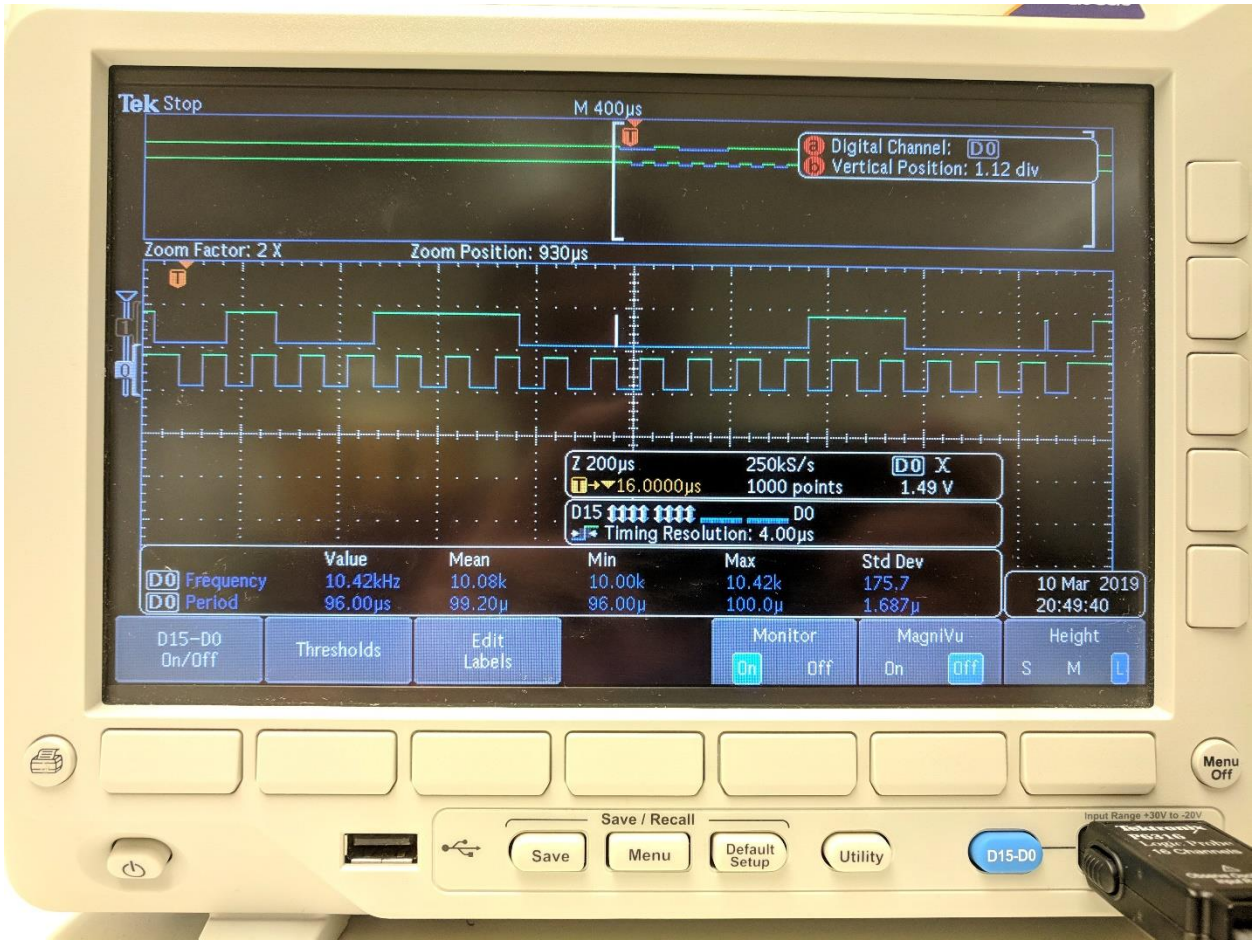


Figure 3. Oscilloscope monitoring the I^2C clock (lower waveform) and data (upper waveform) lines. There are two frames sent – 0x27 is the address of the LCD's IO expander, and 0x0C is the byte we sent to that address. The frames a separated by a stop/start condition as shown on the data line slightly left of the Y-axis.

Aside from this setback, we also had to significantly reevaluate our project upon discovering that the actual AFR cannot be confidently determined from the output Lambda oxygen sensors. Our system was originally intended to keep the AFR between the red and blue bands shown in Figure 11 (Appendix C). However, due to the limitation of Lambda sensors described above, our system is constrained to only use crude voltage thresholds without knowing the corresponding AFR values.

As described at the end of the Methods section, our system was tested using two potentiometer-driven voltage dividers as mock sensors. By turning the potentiometer knobs while running the microcontroller through our IDE's debugger, we can observe that the ADCs are able to read sensor data and perceive changes, the display is promptly and accurately updated, and the choke moves in the correct direction under the correct circumstances. Each of the six warning/error messages appear on the screen as expected and are erased when the condition is unmet. See Figures 4 through 9. Please note that Figure 4 shows the LCD displaying "???" for the choke position. This gets displayed whenever the

Mechanical Module switch is in the OFF position, as the microcontroller does not know if the user is manually adjusting the choke.

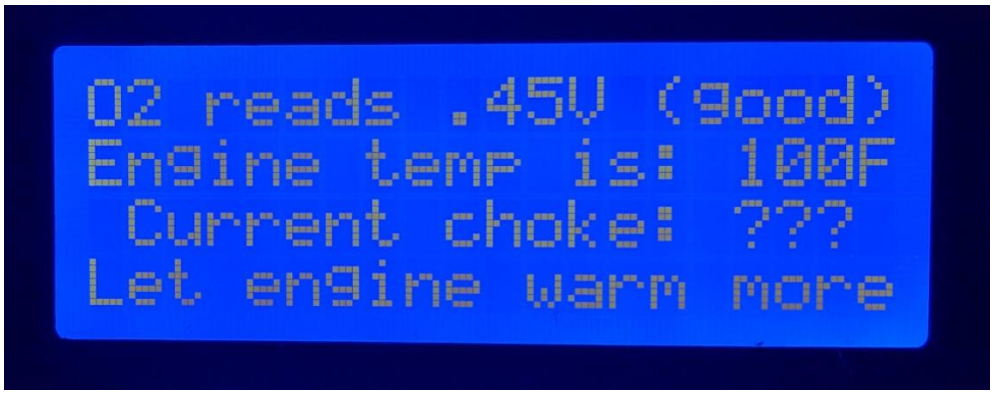


Figure 4. LCD showing the "Let engine warm more" message. This is a non-critical warning suggesting that the user should allow the engine a little more time to warm before operating the vehicle. The suggestion appears when the engine temperature is below 125 ˚F and the rest of the system is in a normal functional state. See Appendix C for relevant works.

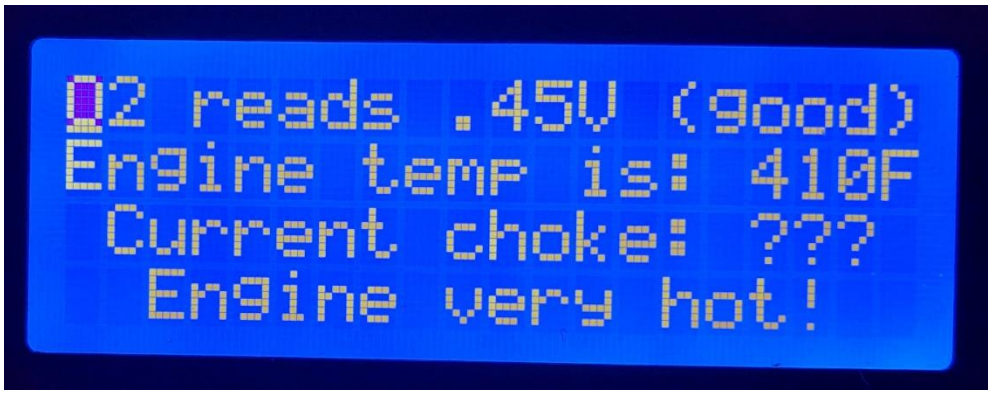


Figure 5. LCD showing the "Engine very hot!" warning message. This message is intended to inform the user that they should safely but quickly park the vehicle and turn off the engine. Otherwise, damage and danger may ensue. This warning appears when the engine temperature is above 260 ˚F. See Appendix C for relevant works.

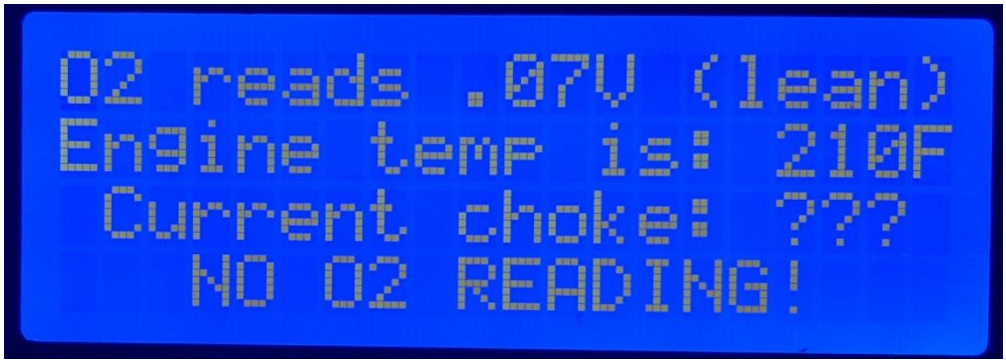


Figure 6. LCD showing the "NO O2 READING!" warning message. This message warns the user that the oxygen sensor has been disconnected or is no longer functional. This warning appears when the oxygen sensor reading is less than or equal to 0.75 V. The output range of the oxygen sensor is about 0.1 V to about 0.9 V.

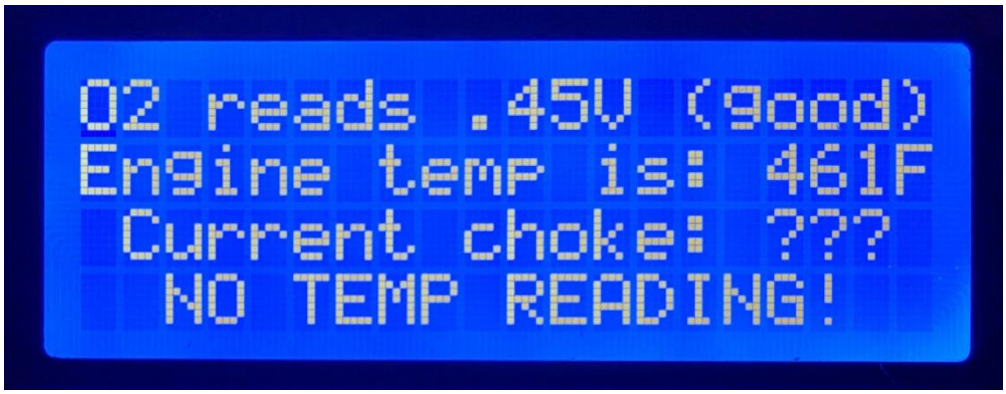


Figure 7. LCD showing the "NO TEMP READING!" warning message. This message warns the user that the temperature sensor has been disconnected. This warning appears when the temperature sensor reading is greater than or equal to 460 ˚F. This value is out of the range of the thermistor.

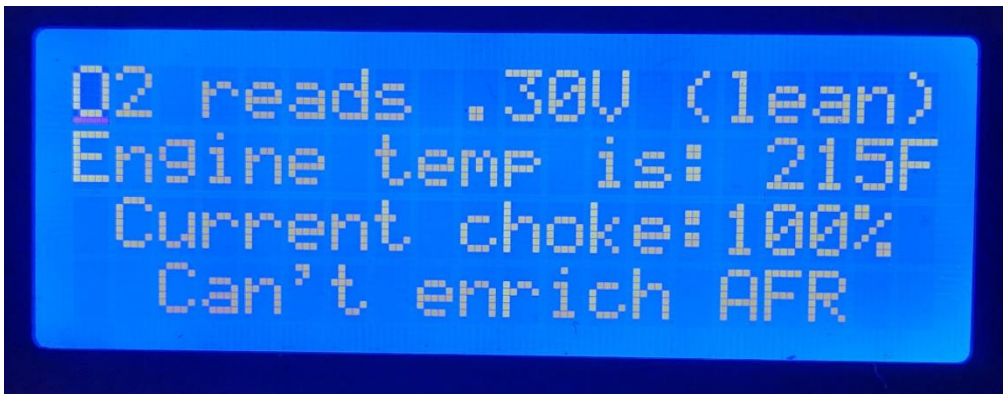


Figure 8. LCD showing the "Can't enrich AFR" warning message. This message informs the user that the carburetor is biased in such a way that the ideal AFR cannot be reached at some point in operation. This warning appears when the choke is fully closed, but the oxygen sensor output is still below 0.4 V (meaning the AFR is too lean).

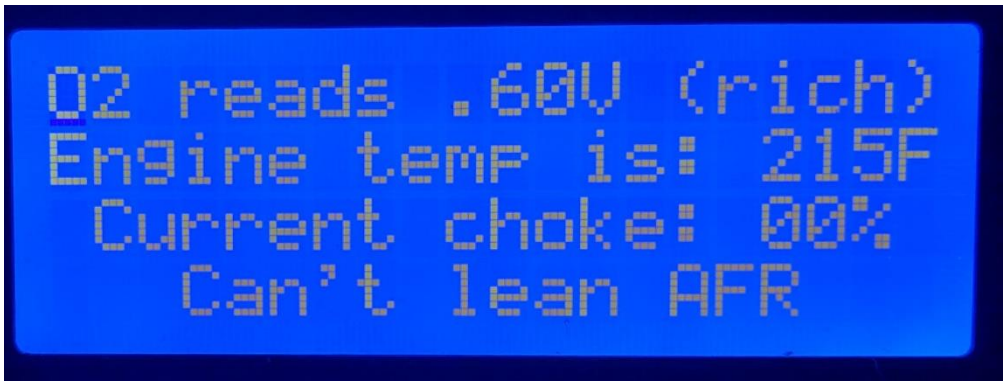


Figure 9. LCD showing the "Can't enrich AFR" warning message. This message informs the user that the carburetor is biased in such a way that the ideal AFR cannot be reached at some point in operation. This warning appears when the choke is fully open, but the oxygen sensor output is still above 0.5 V (meaning the AFR is too rich).

5. DISCUSSION

The ADCs on the microcontroller are Successive Approximation ADCs. This type of ADC works by charging a capacitor to the input voltage value and using a test voltage in a series of comparisons to determine that value. The input capacitance of these ADCs causes the ADC to remember an input voltage long after the input has been disconnected. That is, unless the input of the ADC is still connected to a closed loop, the value will hold for a significant amount of time. Because of this, the "NO TEMP READING!" and "NO O2 READING!" warning messages WILL NOT appear immediately after either sensor is disconnected.

When the temperature sensor outputs 0 V, the microcontroller calculates the temperature to be 1,110 ˚F. The code that calculates the temperature from the input voltage follows a trendline produced in Excel. This trendline was produced using the output of another C++ program that simulates the temperature sensor across the range of the thermistor. Because the thermistor used in the system is a PTC-type thermistor, lower voltages from the temperature sensor correspond to higher temperatures. Thus, the LCD will show "110F" when the sensor is disconnected and the ADC capacitor has fully discharged, as it displays only the last three numbers in "1110".

With those unintended design specifications considered, the results of our simulations were precisely as expected. When the oxygen sensor reading is below 0.4 V, the actuator extends. When the reading is above 0.5 V, the actuator retracts. The LCD is promptly and accurately updated. The choke position shown on the LCD has some error, but this was also expected. The actuator's speed occasionally changes, likely due to fluctuations in the power supplied.

To this point, the system has only been simulated. We have not been able to install the system on an engine to observe its actual performance and collect data thereof, as the target motorcycle is currently out-of-order. Our system *behaves* as intended, but we do not know that it *performs* as intended. However, having designed and created such a system opens the door for significant research and for improved generations to be developed. When the target motorcycle is repaired, the system will be installed, and data will be collected.

We expect that some moderate changes will be necessary upon analyzing the data. For this reason, we have soldered a mini-USB female connector to the discovery board, which will allow us to easily access and reflash the microcontroller without destroying the case/wiring. As we are able to test and tweak the performance of the Carburetor Automator, we are confident that we will observe significantly lower gas consumption, less frequent need of carburetor maintenance, and lower average emissions.

6. CONCLUSION

Although our project is incomplete, we are proud of the work we have done and of the system we have produced. As explained in the Methods section, we designed this project to challenge our abilities and provide a resume-worthy experience. We spent many hours writing configurations, structuring software for the microcontroller, and designing MOSFET and sensor networks.

Much of what we learned through the course of this project was learned through study and trial rather than by mistake and consequence (though certainly some was learned through mistake and consequence). Neither of us had fully configured and created an I^2C bus before, so we learned about different timing aspects, 7- versus 10-bit addressing, and a few of the various modes of operation. We also expanded our abilities in circuit design, as this was one of the few projects that afforded us unlimited choices.

Beyond honing our technical skills, we also learned various time- and project-management skills. Our original Gantt chart was somewhat vague under certain aspects of the project development, which lead to a lack of pressure to complete specific goals. From this, we learned the importance of frequently updating the project schedule and resetting relevant goals. As we began to do so later in the project, our accomplishments became more dense.

This project presented a challenge in design specification. We were required to completely lay out the design specifications early on in the project, and many ideas became irrelevant or unattainable as new information was gained. We both agree that we should have spent more time considering the intricacies of our system as we were writing the design specifications. This would have made our goals clearer, and we would not have spent as much valuable time reevaluating our system.

Another hugely important skill we gained came from our frustrations with the I^2C address registers and with the LCD's IO expander. To resolve the issues we were facing, we had to dedicate a significant amount of time to becoming familiar with the datasheets of the microcontroller and IO expander. Finding information strictly-technical documents can be difficult and monotonous, but it is necessary to a concise and accurate development cycle in embedded system design. We learned both how to read datasheets and the importance of reading them early on in a project.

7. APPENDIX A

This section is comprised of the microcontroller code for the Carburetor Automator.

```
#include "stm32l476xx.h"
#include <string.h>

#define true 1
#define false 0
#define MAX_MOTOR 40
#define MOTOR_FWD 0x40000000
#define MOTOR_BWD 0x80000000

uint32_t adc_count = 0;
uint8_t sysTickCallCount = 0;
float o2Sensor = 14.7f;
signed int tempSensor = 215;
uint8_t chokePosition = 0;
uint8_t backlight;

char line1[] = "O2 reads .45V (good)";
char line2[] = "Engine temp is: 000F";
char line3[] = " Current choke: 00%";

char line4[20] = "";

uint8_t EoC = false;
uint8_t backlightval;
uint8_t line4clear = true;

void delay(int milliseconds)
{
	int i = 0;
	int endTime = milliseconds * 3150;
	while(i++ < endTime) ;
}

void Configure_Pins()
{
	RCC->AHB2ENR |= RCC_AHB2ENR_GPIOAEN | RCC_AHB2ENR_GPIOBEN | RCC_AHB2ENR_GPIOEEN;

	GPIOB->MODER &= 0xFFFF0F0F;
	GPIOB->MODER |= 0x0000A000; //Set PB2, 3 to input, PB6, 7 to alternate function

	GPIOB->OTYPER |= 0x000000C0; //Set bits 6 and 7 for PB6, 7 to open-drain

	//Set pins PB2 as pull down, PB6, 7 as pull-up
	GPIOB->PUPDR &= 0xFFFF0F0F;
```

```
	GPIOB->PUPDR |= 0x00005020;

	GPIOB->AFR[0] |= 0x44000000; // Set PB6, 7 to AF4

	// Configure PA1, 2 for ADC inputs
	GPIOA->MODER |= 0x03E; // Set PA0 to output (debug), 1 & 2 to analog
	GPIOA->ASCR |= 0x06; // Connect PA1, 2 to ADC input

	GPIOE->MODER &= 0x0FFCFFFF; // Clear PE8, 14, 15
	GPIOE->MODER |= 0x50000000; // Set PE14, 15 to output (8 input)
	GPIOE->ODR &= 0; // Clear ODR for good measures

	//Set PE8 to pull down
	GPIOE->PUPDR &= 0xFFFCFFFF;
	GPIOE->PUPDR |= 0x00020000;

	NVIC_EnableIRQ(EXTI2_IRQn);
	NVIC_SetPriority(EXTI2_IRQn, 1);

	RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;
	//Set the external interrupt sources
	SYSCFG->EXTICR[0] &= 0x0;
	SYSCFG->EXTICR[0] |= 0x0100; //PB2 Display
	//interrupt mask register
	EXTI->IMR1 |= EXTI_IMR1_IM2;
	//Enable rising AND falling edge triggers
	EXTI->RTSR1 |= EXTI_RTSR1_RT2;
	EXTI->FTSR1 |= EXTI_FTSR1_FT2;
}

void I2C_Setup()
{
	RCC->APB1ENR1 |= RCC_APB1ENR1_I2C1EN; //Enable I2C Clock

	RCC->CCIPR &= ~RCC_CCIPR_I2C1SEL;
	RCC->CCIPR |= RCC_CCIPR_I2C1SEL_1; //Select HSI as I2C clock (16MHz)

	RCC->APB1RSTR1 |= RCC_APB1RSTR1_I2C1RST;
	RCC->APB1RSTR1 &= ~RCC_APB1RSTR1_I2C1RST;

	I2C1->CR1 &= ~I2C_CR1_PE;
	I2C1->CR1 &= ~I2C_CR1_ANFOFF;
	I2C1->CR1 &= ~I2C_CR1_DNF;
	I2C1->CR1 |= I2C_CR1_ERRIE;
	I2C1->CR1 &= ~I2C_CR1_NOSTRETCH;

	//I2C1->TIMINGR = 0x3 << 28 | 011 << 20 | 0x11 << 16 | 0xBE << 8 | 0xC0; //I2C ~ a little less than 100 kb/s
	I2C1->TIMINGR = 0x3 << 28 | 0x11 << 20 | 0x11 << 16 | 0x17 << 8 | 0x18; //I2C ~ 100 kb/s
```

```
	I2C1->OAR1 &= ~I2C_OAR1_OA1EN;
	I2C1->OAR1 = I2C_OAR1_OA1EN | 0x52;
	I2C1->OAR1 &= ~I2C_OAR2_OA2EN;

	I2C1->CR2 &= ~I2C_CR2_ADD10;
	I2C1->CR2 |= I2C_CR2_AUTOEND;
	I2C1->CR2 |= I2C_CR2_NACK;
	I2C1->CR1 |= I2C_CR1_PE;
}

void I2C_Start(uint8_t size)
{
	uint32_t tempReg = I2C1->CR2;

	tempReg &= (uint32_t)~((uint32_t)(I2C_CR2_SADD | I2C_CR2_NBYTES |

                                                I2C_CR2_RELOAD | I2C_CR2_AUTOEND |

                                                I2C_CR2_RD_WRN | I2C_CR2_START |

                                                I2C_CR2_STOP));

	tempReg &= ~I2C_CR2_RD_WRN; //Write to slave

	tempReg |= (uint32_t)(((uint32_t) (0x27 << 1) & I2C_CR2_SADD) | (((uint32_t) size << 16) &
		I2C_CR2_NBYTES)) | I2C_CR2_START;

	I2C1->CR2 = tempReg;

	if( (I2C1->ISR & I2C_ISR_NACKF) == 1 && (GPIOB->IDR & GPIO_IDR_ID2))
		I2C_Start(size);
}

void I2C_Stop()
{
	I2C1->CR2 |= I2C_CR2_STOP;

	int watchdog = 0;
	//Wait for stop flag
	while( (I2C1->ISR & I2C_ISR_STOPF) == 0 && (GPIOB->IDR & GPIO_IDR_ID2) && ++watchdog < 3150);

	I2C1->ICR |= I2C_ICR_STOPCF;
}

void I2C_Write(uint8_t data)
{
	if((GPIOB->IDR & GPIO_IDR_ID2))
```

```
	{
		I2C_Start(1);

		I2C1->TXDR = data;

		int watchdog = 0;
		while( (I2C1->ISR & I2C_ISR_TXE) == 0 && (GPIOB->IDR & GPIO_IDR_ID2) && ++watchdog <
			3150);

		watchdog = 0;
		//Wait until TC flag set
		while( (I2C1->ISR & I2C_ISR_TC) == 0 && (I2C1->ISR & I2C_ISR_NACKF) == 0 && (GPIOB->IDR
			& GPIO_IDR_ID2) && ++watchdog < 3150);

		I2C_Stop();
	}
}

void pulseEnable(uint8_t data)
{
	//I2C_Write(data | backlight);
	I2C_Write(data | 0x04 | backlight);    // En high
	delay(2);		// enable pulse must be >450ns

	I2C_Write((data | backlight) & ~0x04);		// En low
	delay(5);		// commands need > 37us to settle
}

void write4bit(uint8_t value, uint8_t RW, uint8_t RS)
{
	uint8_t highnib=value&0xf0;
	uint8_t lownib=(value&0x0f) << 4; //Shift left 4 to put data @ D4-D7 (LCD pins)

	pulseEnable(highnib | RS | RW);
	pulseEnable(lownib | RS | RW);
	delay(10);
}

void setCursor(uint8_t col, uint8_t row){
	int row_offsets[] = { 0x00, 0x40, 0x14, 0x54 };
	if ( row > 4 ) {
		row = 3;    // we count rows starting w/0
	}
	write4bit(0x80 | (col + row_offsets[row]), 0x00, 0x00);
}

void printString(char* firstChar)
{
```

```
	while(*firstChar != '\0' && (GPIOB->IDR & GPIO_IDR_ID2))
	{
		write4bit(*firstChar, 0x00, 0x01);
		firstChar++;
	}
}

void updateDisplay()
{
	//Modify line1 bits 10, 11 O2 voltage, 15-18 for rich/lean/good
	//Modify line2 bits 16, 17, 20 for engine temp
	//Modify line3 bits 15, 16, 17, 18 for choke percentage

	line1[10] = (int)((o2Sensor - (int)o2Sensor) * 10) + '0';
	line1[11] = ((int)(((o2Sensor - (int)o2Sensor) * 100)) % 10) + '0';
	if(o2Sensor > 0.5f)
		strncpy(&line1[15], "rich", 4);
	else if (o2Sensor < 0.4f)
		strncpy(&line1[15], "lean", 4);
	else
		strncpy(&line1[15], "good", 4);

	setCursor(10, 0);
	printString(&line1[10]);

	//Temp update
	if(tempSensor > 0)
	{
		line2[16] = (((int)tempSensor/100) % 10) + '0';
		line2[17] = (((int)tempSensor/10) % 10) + '0';
		line2[18] = ((int)tempSensor % 10) + '0';
	}
	else
	{
		line2[16] = '-';
		line2[17] = (((int)(tempSensor * -1 ) / 10) % 10) + '0';
		line2[18] = ((int)(tempSensor * -1 ) % 10) + '0';
	}
	setCursor(16,1);
	printString(&line2[16]);

	//Choke pos update
	if((GPIOE->IDR & GPIO_IDR_ID8) != 0)
	{
		if(chokePosition == 40)
		{
			line3[15] = '1';
			line3[16] = '0';
```

```
                line3[17] = '0';
                line3[18] = '%';
        }
        else
        {
                line3[15] = ' ';
                line3[16] = (chokePosition * 10 / 40) + '0';//(int)((chokePosition / 40.0f) * 10) + '0';
                line3[17] = ((chokePosition * 100 / 40) % 10) + '0';//(int)(((chokePosition / 40.0f) * 1000) /
                        10) + '0';
                line3[18] = '%';
        }
}
else
{
        line3[15] = ' ';
        line3[16] = '?';
        line3[17] = '?';
        line3[18] = '?';
}

setCursor(15, 2);
printString(&line3[15]);

//All warning messages have a non-space character at index 6.
//line4[6] indicates that the warning line is blank.
if(line4[6] == ' ')
{
        if(tempSensor >= 260)
                strncpy(line4, "  Engine very hot!  ", 20);
        if(tempSensor <= 125)
                strncpy(line4, "Let engine warm more", 20);
}

//This condition block prevents constant blank-string writing
if(line4[6] == ' ')
{
        if(!line4clear)
        {
                line4clear = true;
                setCursor(0, 3);
                printString(line4);
        }
}
else
{
        setCursor(0, 3);
        printString(line4);
        line4clear = false;
```

```
        }
}

void ADC_Setup()
{
        RCC->AHB2ENR |= RCC_AHB2ENR_ADCEN; //Step 1 (book pg 498)
        ADC1->CR &= ~ADC_CR_ADEN; //Step 2
        SYSCFG->CFGR1 |= SYSCFG_CFGR1_BOOSTEN; //Step 3

        ADC123_COMMON->CCR |= ADC_CCR_VREFEN; //Step 4
        ADC123_COMMON->CCR &= 0xFFC3FFFF; //Step 5
        //1111 1111 1100 0011
        //1111 1111 1111 1111
        ADC123_COMMON->CCR &= ~ADC_CCR_CKMODE;
        ADC123_COMMON->CCR |= 0x00010000;//step 6
        ADC123_COMMON->CCR &= ~ADC_CCR_DUAL;//step 7

  //ADC Wakeup
        int wait_time = 0;

        if((ADC1->CR & ADC_CR_DEEPPWD) == ADC_CR_DEEPPWD)
                ADC1->CR &= ~ADC_CR_DEEPPWD;

        ADC1->CR |= ADC_CR_ADVREGEN;
        //NVIC->ISER[0] |= 1 <<30; // Something with selecting interrupt source

        ADC1->IER |= ADC_IER_EOCIE; // Enables End of Conversion interrupt

        while(wait_time++ != 1601) ;
        //end wakeup
        ADC1->CFGR &= 0xFFFFFFC7; //Set data to left-aligned and 12-bits

        ADC1->SQR1 &= ~ADC_SQR1_L; //This sets it to do one conversion per ADC_start
        ADC1->SQR1 |= 0x01; // 2 conversions per ADC_Start

        ADC1->SQR1 &= 0xFFFFF83F; // Clear 1st channel select
        ADC1->SQR1 |= 0x000000180; //step 12 SETS channel 6 to 1st conversion

        ADC1->SQR1 &= 0xFFFE0FFF; // Clear 2nd channel select
        ADC1->SQR1 |= 0x00007000; // Set channel 7 to 2nd conversion

        ADC1->DIFSEL &= 0xFFFFFF3F; // Set channel 6, 7 to single-ended
        ADC1->SMPR1 |= 0x00EC0000; // Set channel 6, 7 sample time to 640.5 ADC clock cycles

        ADC1->CFGR &= ~ADC_CFGR_CONT; // Set ADC1 to single conversion mode
        ADC1->CFGR &= ~ADC_CFGR_EXTEN; //Sets software trigger

        ADC1->CR |= ADC_CR_ADEN; //step 17
```

```
	while((ADC1->ISR & ADC_ISR_ADRDY) != ADC_ISR_ADRDY) ; //step 18

	NVIC_EnableIRQ(ADC1_IRQn); //Enable handler

	RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;
}

void InitializeLCD()
{
	delay(1000); //Expander needs >=40ms after power up before receiving commands
	backlight = 0;
	write4bit(0x00, 0x00, 0x00);

	//Weird initialization stuff
	write4bit(0x03, 0x00, 0x00);
	delay(10);

	write4bit(0x03, 0x00, 0x00);
	delay(5);

	write4bit(0x03, 0x00, 0x00);
	delay(3);
	//End weird initialization stuff


	write4bit(0x02, 0x00 ,0x00); //Put it in 4-bit interface. After this, use write4bit
	delay(5);

	write4bit(0xC2, 0x00, 0x00); //0x2 keeps 4-bit, 0xC sets 2 line, 5x10
	delay(5);

	write4bit(0x80, 0x00, 0x00); //this made cursor go down
	delay(5);

	write4bit(0x10, 0x00, 0x00); //Clear display
	delay(15);

	write4bit(0x60, 0x00, 0x00); //Set entry mode how you want
	delay(5);
	write4bit(0x02, 0x00, 0x00);
	delay(20);

	backlight = 0x08;

	setCursor(0, 0);
	printString(line1);

	setCursor(0, 1);
```

```
	printString(line2);

	line3[19] = '\0';
	setCursor(0, 2);
	printString(line3);
}

void EXTI2_IRQHandler(void) //Disp_Switch on
{
	if ((EXTI->PR1 & EXTI_PR1_PIF2) != 0)
	{
			I2C_Setup();
			InitializeLCD();

		EXTI->PR1 |= EXTI_PR1_PIF2;
	}
}

void ADC1_IRQHandler()
{
	//Vref ~= 3.12V -> 4095/3.12 = 1,312.5 ~= 1,313
	//Dividing adc data register by 1313 gives value in volts

	if(adc_count % 2 == 0) //First conversion
		o2Sensor = ADC1->DR / 1313.0f;
	else //Second conversion
	{
		if(tempSensor != 0)
		{
			tempSensor = ADC1->DR / 1313;
			tempSensor *= (-559);
			tempSensor += 1110; //According to trendline produced by Excel
		}

		EoC = true;
	}

	adc_count++;
}

void SysTick_Handler()
{
	EoC = false;
	ADC1->CR |= ADC_CR_ADSTART;
	while(!EoC); //Wait for ADC to finish both conversions

	if(!(GPIOE->IDR & GPIO_IDR_ID8))
	{
```

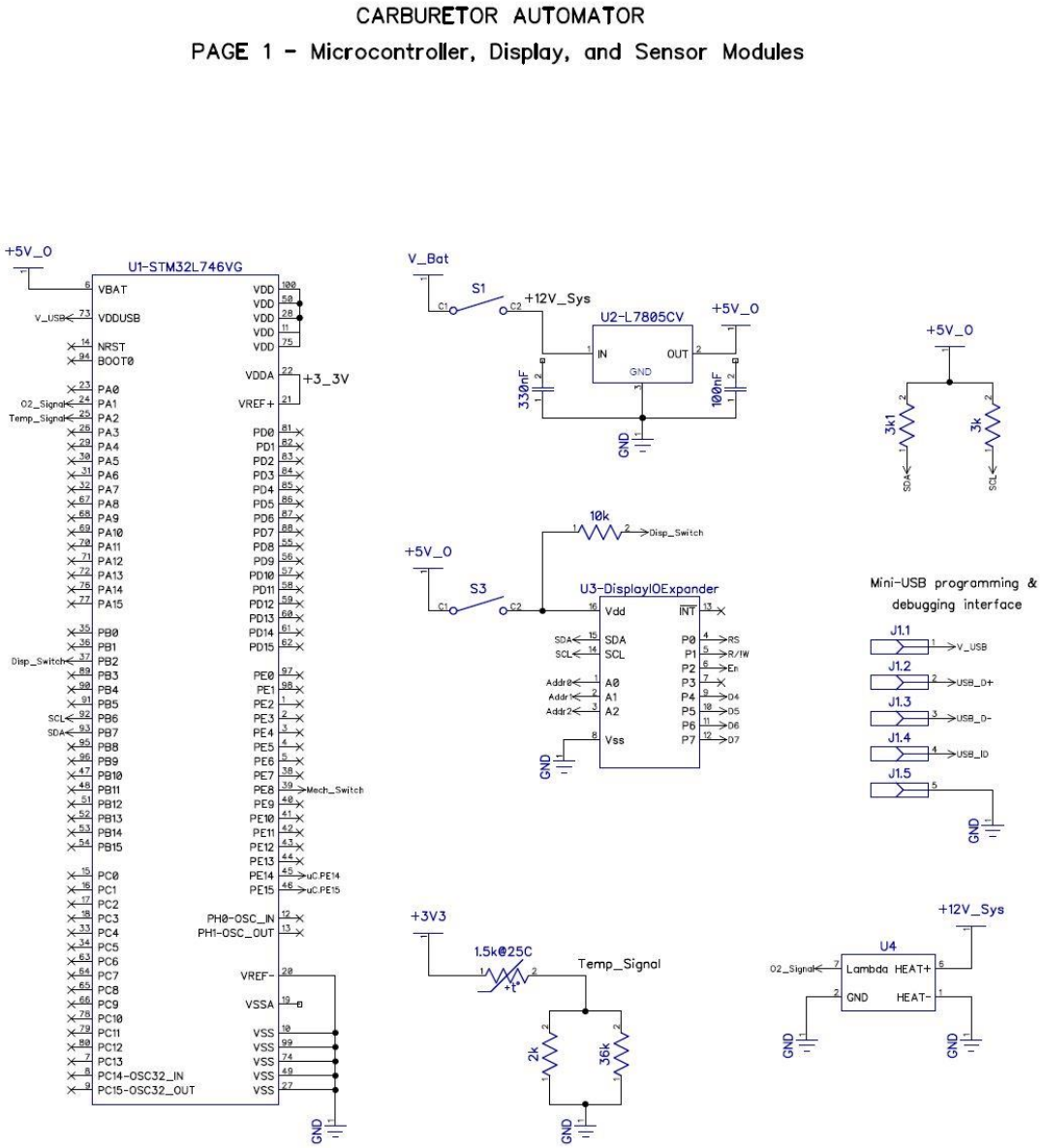
```
		GPIOE->ODR &= ~MOTOR_FWD; //Actuator should be motionless h-bridge is off

		GPIOE->ODR &= ~MOTOR_BWD;
}

strncpy(line4, "                    ", 20);
if(o2Sensor < 0.075f)
		strncpy(line4, "   NO O2 READING!   ", 20);
else if(tempSensor > 460) //Beyond range of the thermistor
		strncpy(line4, "  NO TEMP READING!  ", 20);
else
{
		if((GPIOE->IDR & GPIO_IDR_ID8))
		{
			if(tempSensor > 125)
			{
				if(o2Sensor < 0.4f)
				{
					if(chokePosition != MAX_MOTOR)
					{
						GPIOE->ODR &= ~MOTOR_BWD; //Need to make sure other
							half of h-bridge is off
						GPIOE->ODR |= MOTOR_FWD; //Set motor forward
						chokePosition++;
					}
					else
					{
						GPIOE->ODR &= ~MOTOR_FWD;
						strncpy(line4, "  Can't enrich AFR  ", 20);
					}
				}
				else if(o2Sensor > 0.5f)
				{
					if(chokePosition != 0)
					{
						GPIOE->ODR &= ~MOTOR_FWD; //Need to make sure other half
							of h-bridge is off
						GPIOE->ODR |= MOTOR_BWD; //Set motor backward
						chokePosition--;
					}
					else //AFR rich, but choke is fully open
					{
						GPIOE->ODR &= ~MOTOR_BWD;
						strncpy(line4, "   Can't lean AFR   ", 20);
					}
				}
				else
				{
```

```
                                        GPIOE->ODR &= ~MOTOR_FWD; //Actuator should be motionless h-
                                                bridge is off
                                        GPIOE->ODR &= ~MOTOR_BWD;
                                }
                        }
                        else
                        {
                                if(chokePosition < MAX_MOTOR)
                                {
                                        GPIOE->ODR &= ~MOTOR_BWD; //Need to make sure other half of h-
                                                bridge is off
                                        GPIOE->ODR |= MOTOR_FWD; //Want to set the choke fully closed
                                        chokePosition++;
                                }
                                else
                                        GPIOE->ODR &= ~MOTOR_FWD; //Stop at full choke
                        }
                }
                else
                {
                        chokePosition = 0;
                }
        }

        if(++sysTickCallCount == 5 && (GPIOB->IDR & GPIO_IDR_ID2))
        {
                sysTickCallCount = 0;
                updateDisplay();
        }
}

int main()
{
        // Switch system clock to HSI here
        RCC->CR |= RCC_CR_HSION;
        while( (RCC->CR & RCC_CR_HSIRDY) == 0) ; //Wait for HSI to be ready
        RCC->CFGR |= 0x00000001; //Switch system clock to HSI
        RCC->CR &= 0xFFFFFFFE; //Turn MSI off

        Configure_Pins();
        ADC_Setup();

        ADC1->CR |= ADC_CR_ADSTART;
        while(!EoC);
        EoC = false;

        line1[10] = (int)((o2Sensor - (int)o2Sensor) * 10) + '0';
        line1[11] = ((int)(((o2Sensor - (int)o2Sensor) * 100)) % 10) + '0';
```

```
	if(o2Sensor > 0.5f)
		strncpy(&line4[15], "rich", 4);
	else if (o2Sensor < 0.4f)
		strncpy(&line4[15], "lean", 4);
	else
		strncpy(&line4[15], "good", 4);

	//Temp update
	if(tempSensor > 0)
	{
		line2[16] = (((int)tempSensor/100) % 10) + '0';
		line2[17] = (((int)tempSensor/10) % 10) + '0';
		line2[18] = ((int)tempSensor % 10) + '0';
	}
	else
	{
		tempSensor *= -1;
		line2[16] = '-';
		line2[17] = (((int)tempSensor/10) % 10) + '0';
		line2[18] = ((int)tempSensor % 10) + '0';
	}

	if((GPIOB->IDR & GPIO_IDR_ID2))
	{
		I2C_Setup();
		InitializeLCD();
	}

	SysTick->CTRL &= 0;
	NVIC_SetPriority(SysTick_IRQn, 2); //ADC interrupt needs higher priority than SysTick
	SysTick->VAL = 0;
	SysTick->LOAD = 799999; //Period of 50ms
	SysTick->CTRL |= 0x07;

	while(1) {}
}
```

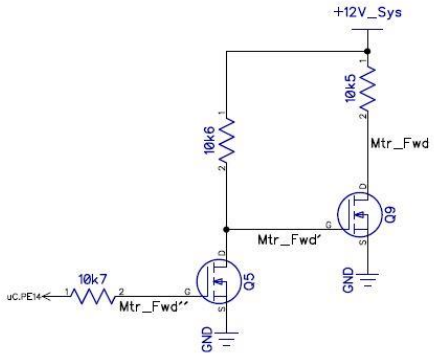
8. APPENDIX B

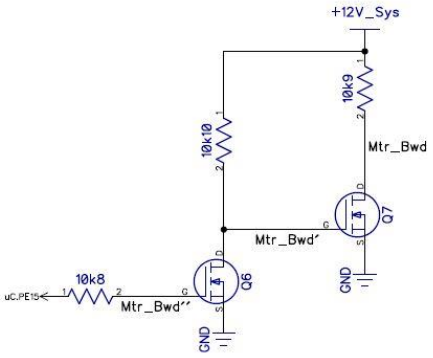
This section is comprised of the two-page hardware schematic of the Carburetor Automator.

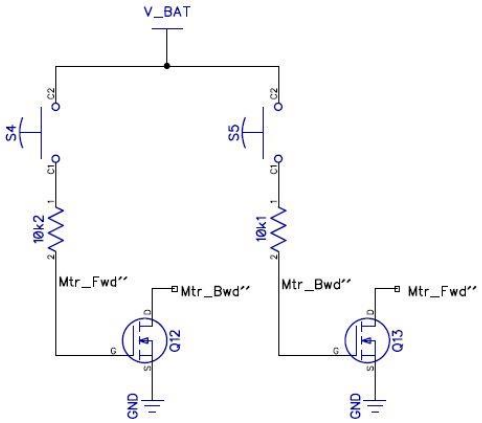


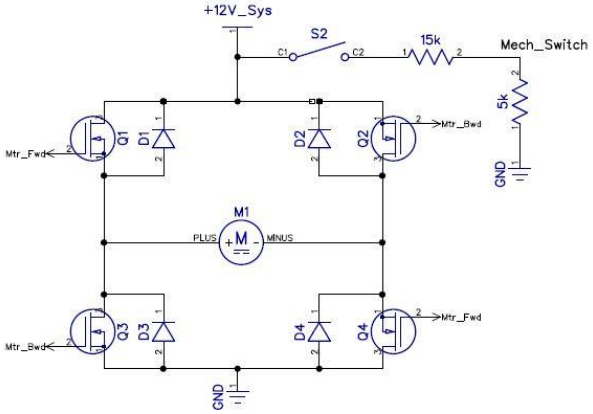
CARBURETOR AUTOMATOR

Page 2 - Mechanical Module









9. APPENDIX C

This section contains a few relevant articles, graphics, and websites to further explain how the methods and threshold values were derived for the Carburetor Automator.

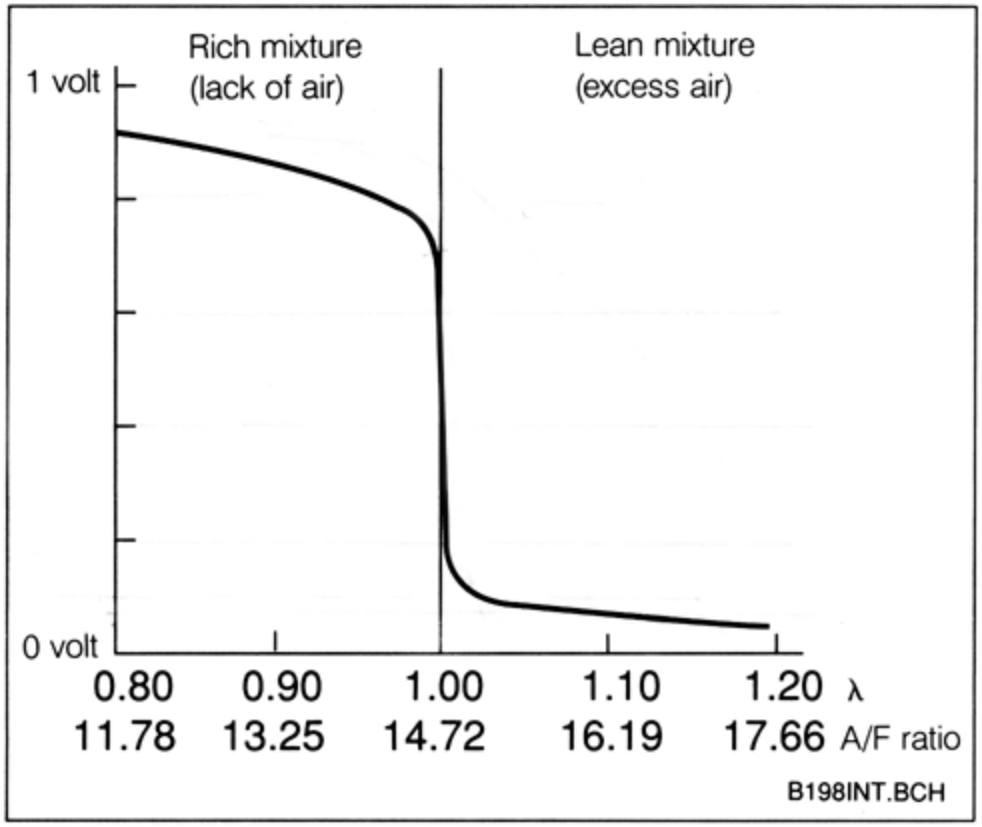


Figure 10. Output curve of a Lambda oxygen sensor. This curve shows that the AFR cannot be (accurately) quantitatively determined from the output voltage. For this reason, the Carburetor Automator only considers the sensor's output voltage in reference to 0.45 V rather than using AFR numbers. Image taken from: https://www.google.com/search?q=voltage+vs+lambda&safe=active&rlz=1C1CHBF_enUS813US813&source=lnms&tbm=isch&sa=X&ved=0ahUKEwig3fK5rvrhAhUOvKwKHelNAnsQ_AUIDigB&biw=1440&bih=818#imgrc=ReN6emCu98Op9M:

For information/details regarding the I^2C communication protocol, see https://i2c.info on the web.

For information/details regarding the operation of Lambda oxygen sensors, see https://mechanics.stackexchange.com/questions/23933/how-do-lambda-sensors-work on the web.

For information/details regarding carburetion and Air-to-Fuel Ratios, see https://www.enginebuildermag.com/2008/07/carburetor-tuning-the-airfuel-equation/ on the web.

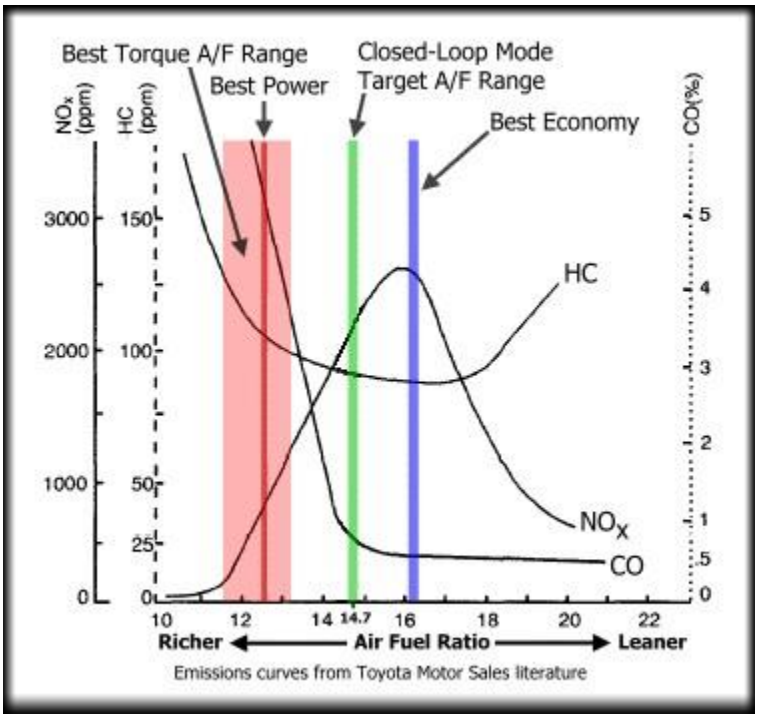


Figure 11. Example target range for and emission curves for AFR. Image taken from: https://mechanics.stackexchange.com/questions/58229/relationship-between-torque-power-specific-fuel-consumption-and-afr