Department of Electrical and Computer Engineering

Embedded Systems
EGC433-01

Cruise Control
Final Project

Date: 5/10/21 Semester: Spring 2021	
Group Members	Department
Kevin Nelson	EE

Lab Instructor: Prof. Mike Otis

Table of Contents

I. Abstract

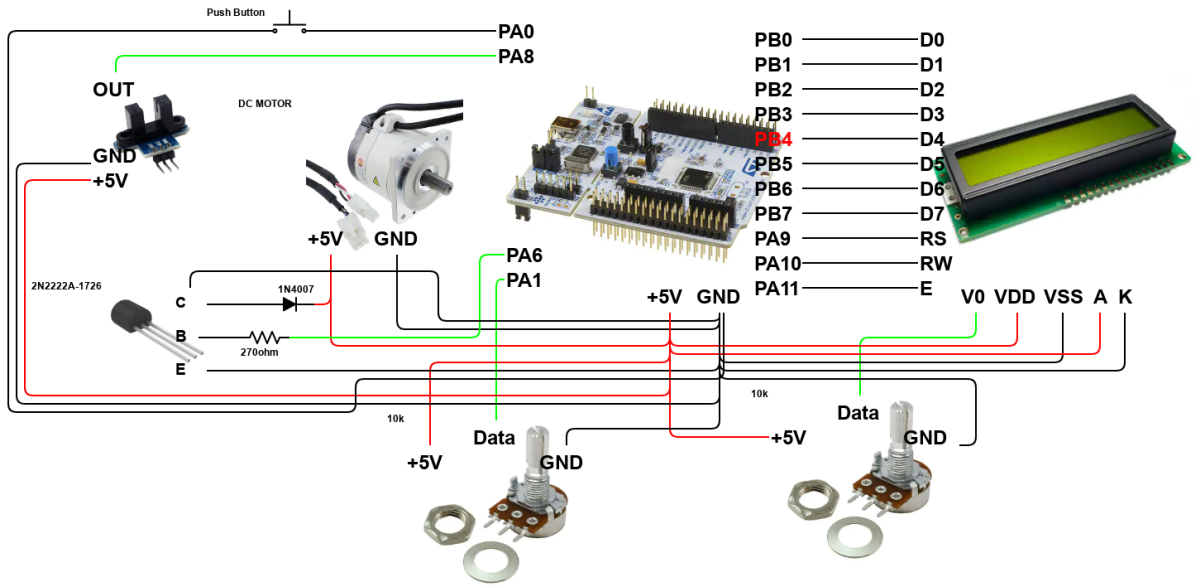
This project is a thorough demonstration on how one can utilize an STM32 board in conjunction with a multitude of peripherals in order to achieve a fully functional real world cruise control. It is through the use of coding language C, that the author demonstrates the approach to coding such a device. With real world applications being very complicated, approaching this problem through the use of traditional programming or interrupts, allows one to account for different situations, as one would expect in real world applications. It should be noted that neither method is perfect, and both maintain a number of setbacks, however it will be demonstrated that both can be effective.

This project includes a number of peripheral devices that one might expect to find in the real world, regardless of location. Some of these components are: LCD Display, Analog to Digital Converter(ADC), Digital to Analog Converter(DAC), Rotations Per Minute(RPM) sensor, 1-6V DC motor, Photoresistive Diode & LED pair for the RPM sensor, 9 volt battery, and a push button. Each component is crucial, as it allows a user to enable cruise control, and will allow the system to speed up and slow down based on the speed, when needed.

II. Design

Hardware Design

Figure 1.1 - Cruise Control



Software Design

Figure 2.1 - ADC Voltage & Distance Display

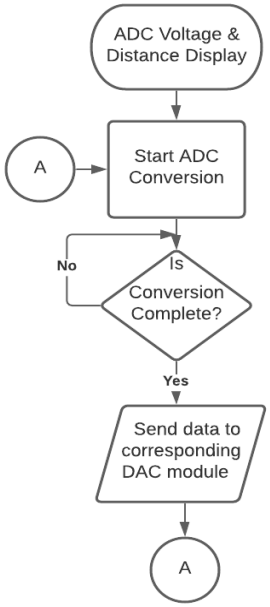


Figure 2.2 - LCD Display

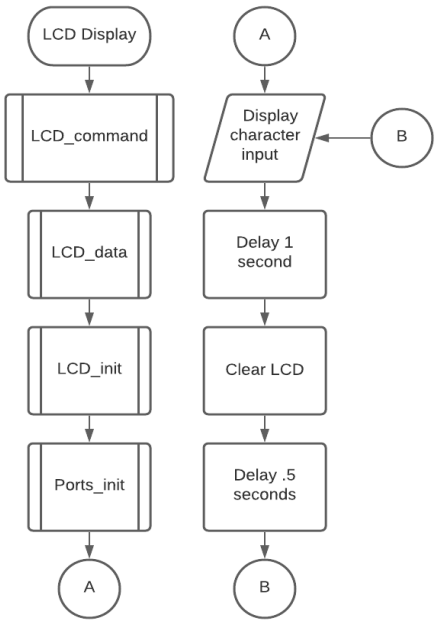


Figure 2.1 shows a very standard and simple flowchart for Analog to digital conversion. STM32 boards like the F446RE have an ADC converter built in, so the necessary steps would be to enable it in the proper register, wait for the conversion to complete, and then send the data to the proper function.

Figure 2.2 shows the necessary steps in which one will have to take in order to turn on and output a character to an LCD display. The squares with two vertical lines represent the subroutines that Hitachi requires in order to turn on the LCD controller and allow it for use. It should be seen that there are no peripheral inputs that determine what is to be displayed, and the characters chosen for display come from the software.

Figure 2.3 -Measuring RPMs

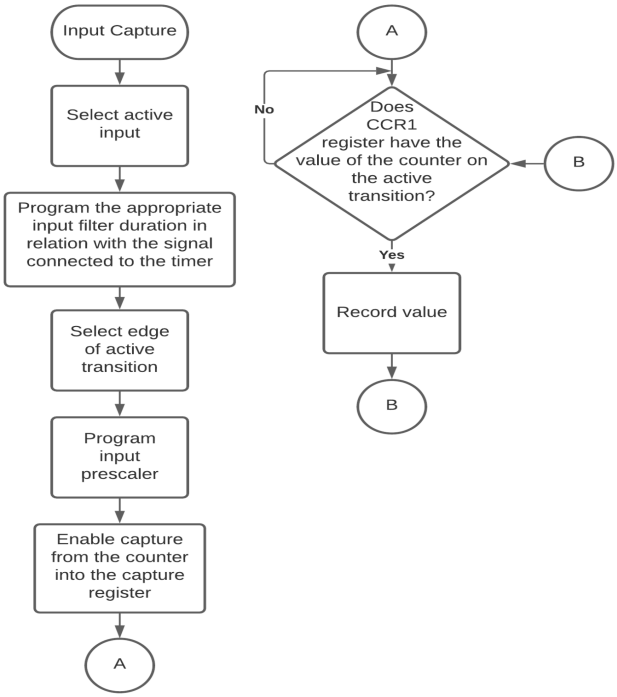


Figure 2.3 shows the standard way of approaching the input capture ability on an stm32 board, as these features are built in. There are a few in depth procedures here, however some of the most important lie in the edge detection, prescaler, and reading the value. It is important to set the appropriate number of edges one wishes to capture, as well as the falling or rising edge. Additionally, the prescaler divides the system clock which in turn speeds up or slows down the input capture. This will allow one to either get a better resolution, or wider range of values for input capture.

Figure 2.4 - Cruise Control

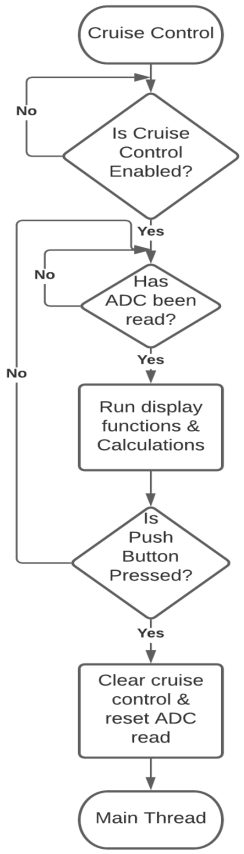


Figure 2.5 - Delay Function

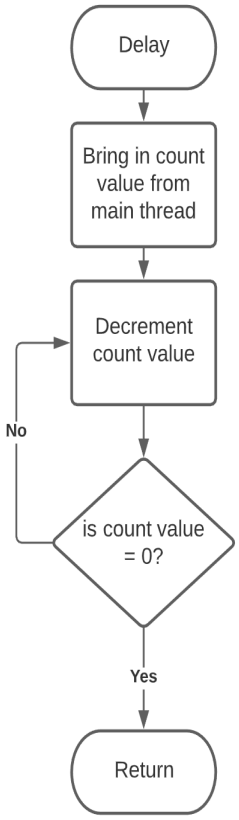


Figure 2.4 represents the steps one must take in creating a delay function. The way it works is by checking to see if cruise control has actually been enabled by a push button press. As long as Cruise control is true, the program can then check to see if the Analog-to-Digital converter has been read, since it is only to be read once. After the ADC is read, the program runs some predefined calculations and displays them, and then checks to see if the push button has been pressed again to see if it needs to head back to the main thread.

Figure 2.5 represents the steps one must take in creating a delay function. The way it works is by inputting the desired delay(in milliseconds) into the program, the program then passes that input parameter into the delay function, the function decrements the value by 1 until it reaches zero, and then it can return back to the main thread.

III. New Approaches

Interrupts

There is not much to change when creating an interrupt for only one action, all but one of the functions remain the same, however the way it is called and switched between has effectively been rewritten. With interrupts, the user can now press the button at any point in time and not worry about debounce, as the incorporation of the NVIC controller allows a press to be saved. However, due to the nature of interrupts and some of its difficulties, clearing the button press without disrupting the other functions has proven difficult. This is important because even though clearing the pending flags has been coded in, the interrupt function for some reason reads the rising edge as a key press as well(even though it is diabled in code) and does not clear its pending flag, thus making the cruise control function loop continuously.

Proportional-Integral-Derivative(PID) Control

Creating a PID controller for this project makes sense, considering that the current way the pulse width is coded, there is a lot of overshoot when the system tries to correct itself. A PID is a robust and ingenious way of eliminating or minimizing the overshoot, as well as the transient time and error from the desired step function the system provides. The appendix shows an array of different functions utilized in it's attempt, and even though at the time of writing the author was unable to have the system correct for RPM errors, the code is able to take an input speed and output the desired value. The compute function works based on timing, error, an integral and derivative term in order to calculate the appropriate output.

IV. Discussion & Conclusion

There were quite a few considerations to be made throughout the scope of this project. The topics of main concern were: Cruise control implementation(Bang-Bang or Proportional Integral Derivative(PID)), and interrupts. In this project, the author chose to design the system in such a way that PORTB on the STM32 board was utilized for the output of the LCD Display, with a few pins found on PORTA. Additionally, the potentiometer that was implemented for the analog input value could be found in PORTA, as it only required 1 pin. The speed sensor, DAC(PWM for this project) output signal, and push button can also be found in PORTA. Lastly, the rest of this discussion assumes previous knowledge on how to set up and output values to an LCD, implementation of a PWM signal, and input capture using timers, as to make this paper concise to cruise control.

Addressing the appendix, one can see a relatively standard approach to cruise control using an approach slightly more elegant than bang-bang, however not as sophisticated as PID. The most important elements can be found under the functions Pulse_Width_Modulator, CruiseControlFunc, & CruiseControlSetup, as seen in Figures 4.1-4.3. It's important to note here that there is no extra setup needed for port initialization when incorporating these functions.

```
164 // Pulse Width Modulation
165 void Pulse_Width_Modulator(float pulse_width){
166   if((CruiseControl == 1) && (RPM < CruiseControlRPM)){
167     result2 = (CruiseControlRPM - RPM)*10;
168     TIM2->CCR1 = pulse_width + result2;}
169   else if((CruiseControl == 1) && (RPM > CruiseControlRPM)){
170     result3 = (RPM - CruiseControlRPM)*.11f;
171     TIM2->CCR1 = pulse_width - result3;}
172   else TIM2->CCR1 = pulse_width;
173 }
```

Figure 4.1 - Pulse Width Modulation

Figure 4.1 begins by an if statement that checks for two dependencies. The first dependency is a parameter called CruiseControl, and turns true when the user presses on the push button. The second parameter the if statement checks is whether or not the measured RPMs is less than the set cruise control value. If the if statement validates both input parameters, the statement drops into a new variable that calculates the difference between the current RPMs and the cruise control set value. Since this if statement only executes when the current speed is slower than what was set, we know to take the variable that calculated the difference, and add it to the current pulse width output in order to speed up the motor. In the situation where the motor is actually moving faster than the set value, the next if statement checks for cruise control, takes the difference of the two values, and subtracts the new variable from the pulse width since the idea is to slow the motor down. An important setback to discuss about this approach, is that a calculation can only be made when a new RPM value is read in, meaning that it is possible the pulse width will never change if a new RPM value is passed.

```
190 // Cruise Control Setup
191 void CruiseControlSetup(void){
192   CruiseControl = 1;
193   CruiseControlRPM = RPM;
194 }
195
196 // Manual Mode
197 void ManualMode(void){
198   Grab_Conversion_Data();           // Starts ADC & DAC, then reads data holding register
199   Voltage_Display();                // Displays voltage result to LCD from DAC
200   LCD_command(0xC0);                // Moves LCD cursor to 2nd line
201   //Compute();
202   Pulse_Width_Modulator(result);    // Output PWM value to motor
203   RPM_Display();                    // Displays RPM result to LCD from TIM1
204   LCD_command(0x02);                // Clear LCD & move cursor back to line 1
205 }
```

Figure 4.2 - Cruise Control Setup & Manual Mode

Figure 4.2 shows two critical functions, the first being CruiseControlSetup, and the other being ManualMode. CruiseControlSetup is a function that will allow the CruiseControl value to be true, which is needed in Pulse_width_Modulator as discussed above, and sets the current RPMs to a new value called CruiseControlRPM. It is important to create the new CruiseControlRPM variable here, so that there is a constant rpm value that can be compared to when reading in the new data. ManualMode is arguably the most important function in this code. Like a car driving on the road, it is not always in cruise control and can be described as being in a manual mode where the driver is accelerating and braking. ManualMode behaves exactly like the default setting of a car, where the user can freely control acceleration when cruise control is disabled.

```
207 // Cruise Control
208 void CruiseControlFunc(void){
209   if(j == 1){
210     Grab_Conversion_Data(); }       // Starts ADC & DAC, then reads data holding register
211   j=0;
212   Voltage_Display();                // Displays voltage result to LCD from DAC
213   LCD_command(0xC0);                // Moves LCD cursor to 2nd line
214   Pulse_Width_Modulator(result);    // Output PWM value to motor
215   RPM_Display();                    // Displays RPM result to LCD from TIM1
216   LCD_command(0x02);                // Clear LCD & move cursor back to line 1
217   if(CCButtonIsPressed){
218     CruiseControl = 0;
219     j=1;
220   }
221 }
```

Figure 4.3 - Cruise Control main function

Figure 4.3 represents the main purpose for this paper. CruiseControlFunc shows the behavior of the system when no longer in manual mode and the push button is pressed. At first glance, one might notice that many of the functions that are in ManualMode are also in CruiseControlFunc, and that would be correct. Like ManualMode, CruiseControlFunc does all the same actions required of that of a normal driving mode, however it checks how often it needs to read the Analog-to-Digital Converter(which is once). After the ADC is read once, the j variable is immediately set to 0, to ensure it no longer reads that input. From here, the function will loop(shown in Figure 4.4) until the button is

pressed again, leading CruiseControl to now read as false, and will reset j back to 1 so that the next time CruiseControlFunc is called, the ADC can be read once again.

```
44 int main (void) {
45   Port_init();
46   LCD_init();
47
48   while (1) {
49     if(CCButtonIsPressed == true) CruiseControlSetup();
50     while(CruiseControl == true) CruiseControlFunc();
51     ManualMode();}
52 }
```

Figure 4.4 - Main Thread

Figure 4.4 shows the execution of the main thread and the always executing while(1). Lines 49 & 50 clearly show that when the button is pressed the conditional statements will execute the necessary cruise control functions as discussed above. Line 51 shows the default position of this thread, which is manual mode.

In summary, it is clear from the code that understanding how to critically think about the implementation of cruise control is the most important and difficult task. There are no built in board functions or registers that simply enable something like this, and so one must rely on intuition, wit, and trial and error when tackling a task like this. In the author's opinion going forward, it is clear that making as many functions as possible in conjunction with some redefining of statements to make the code more readable is of the utmost importance. When tackling a project that makes use of an array of technologies and techniques, it can be quickly lost on an outsider looking in what this code is actually doing and how it may behave given a particular input, thus clear and concise programming is crucial. Additionally, in the future one may wish to add a couple pieces of hardware to the design. A good example of this would be a capacitor in series with the button, as to account for debounce from each press. In the current design and without interrupts, it is easy for the system to think a single button press was actually two, and thus it will automatically re enter cruise control even when undesired. Lastly, a better implementation of cruise control would be a strong consideration for the future. As discussed in the previous section, this enhanced version of bang-bang control provides way too much overshoot and error when trying to correct, thus an implementation such as PID would be a good fit. Indeed a future research project in which one can utilize these alternate methods, as well as Direct Memory Access, would be extremely worthwhile, especially when attempting to read an extremely high speed motor in conjunction with a high accuracy shaft encoder and sensor.

V. Citations

1. M. A. Mazidi, S. Chen, and E. Ghaemi, *STM32 arm programming for embedded systems using C Language with STM32 Nucleo*. Place of publication not identified: Mazidi, 2018.

VI. Appendix

Code for Cruise Control in C(No New Approaches):

```
1.  // Kevin Nelson
2.  // Embedded Systems - Final Project - Cruise Control
3.
4.  // Spring 2021
5.
6.  #include "stm32f4xx.h"
7.  #include <stdio.h>
8.  #include <stdbool.h>
9.
10. #define RS 0x200                                                    // PA9 mask for reg select
11. #define RW 0x400                                                    // PA10 mask for read/write
12. #define EN 0x800                                                    // PA11 mask for enable
13. #define CCButtonIsPressed ((GPIOA->IDR & 0x0001) != 0x0001)          // PA0 button press
14.
15. void delayMs(int n);
16. void LCD_command(unsigned char command);
17. void LCD_data(char datawrite);
18. void LCD_init(void);
19. void Port_init(void);
20. void Send_A_String(char *StringOfCharacters);
21. void Voltage_Display(void);
22. void RPM_Display(void);
23. void Grab_Conversion_Data(void);
24. void ManualMode(void);
25. void CruiseControlSetup(void);
26. void CruiseControlFunc(void);
27. void Pulse_Width_Modulator(float pulse_width);
28. void EXTI0_IRQHandler(void);
29. static int i, j=1, CruiseControl;
30. static char Voltage_Buffer[18], RPM_Buffer[18];
31. static float voltage, RPM, result, Period, Current, Frequency, Last = 0.0, CruiseControlRPM, result2, result3;
32.
33. // PID Functions & Variables
34. void Compute(void);
35. void SetTunings(double Kp, double Ki, double Kd);
36. void SetSampleTime(int NewSampleTime);
37. void SetOutputLimits(double Min, double Max);
38. void Initialize(void);
39. unsigned long millis(void);
40. static unsigned long lastTime, time;
41. static double ITerm, lastInput, kp, ki, kd, outMin, outMax;
42. static int SampleTime = 1000;
43.
44. int main (void) {
45. Port_init();
46. LCD_init();
47. Initialize();
48. SetOutputLimits(CruiseControlRPM,10);
49. SetSampleTime(1000);
50. SetTunings(100,100,50);
51.
52. while (1) {
53. if(CCButtonIsPressed == true) CruiseControlSetup();
```

```
54.  while(CruiseControl == true) CruiseControlFunc();
55.  ManualMode();}
56.  }
57.
58.  // Port Initializations for LCD. PA9-R/S, PA10-R/W, PA11-EN, PB0-PB7 for D0-D7, respectively.
59.  void Port_init(void) {
60.  // Enable Clocks
61.  RCC->AHB1ENR = 3;                  // enable GPIOA/B clocks
62.  RCC->APB1ENR |= 0x200000003;   // enable TIM2 clock & DAC
63.  RCC->APB2ENR |= 0x00004101;    // enable TIM 1, ADC1, & SYSCFG clock
64.
65.  // Enable Modes
66.  GPIOA->MODER = 0;                          // clear pin mode
67.  GPIOA->MODER = 0x55566B5C;             // set pins output mode, PA1 to analog(capture pot value), PA5(output PWM) to
     alternate function, PA8(input capture) alternate function
68.     GPIOA->BSRR  = 0x0C000000;     // turn off EN and R/W
69.  GPIOA->PUPDR  = 0x1;
70.  GPIOA->AFR[0] = 0;                                  // clear alt mode
71.     GPIOA->AFR[0] = 0x02100000;          // set pin to AF1 for TIM2 CH1, configure PA6 as input of TIM3 CH1
72.  GPIOA->AFR[1] = 0;                                  // clear alt mode
73.     GPIOA->AFR[1] = 0x00000001;          // set alt mode TIM1_CH1
74.  GPIOB->MODER = 0;                                        // clear pin mode
75.     GPIOB->MODER = 0x55555555;          // set pins output mode
76.
77.  // Setup ADC1
78.     ADC1->CR2 = 0;              // SW trigger
79.     ADC1->SQR3 = 1;             // conversion sequence starts at ch 1
80.     ADC1->SQR1 = 0;             // conversion sequence length 1
81.     ADC1->CR2 |= 1;             // enable ADC1
82.
83.  // Setup TIM1 for input capture - PA8
84.  TIM1->PSC = 16000;                         // divided by 16000
85.     TIM1->CCMR1 = 0x01;      // Input capture mode, validates transition on TI1 with 1 consecutive sample, input prescaler
     disabled(IC1PS bits to 0)
86.     TIM1->CCER = 1;             // CC1S bits are writable only when the channel is OFF(CC1E = '0' in TIMx_CCER), rising edge
     selected by keeping CC1P & CC1NP bits to 0
87.     TIM1->CR1 = 1;              // enable timer
88.
89.  // Setup TIM2 for output PWM - PA5
90.     TIM2->PSC = 1000;                          // divided by 1000
91.     TIM2->ARR = 750;                           // counts up to 750
92.  TIM2->CNT = 0;              // clear counter
93.  TIM2->CCMR1 = 0x0068;         // set output to PWM
94.                 TIM2->CCR1 = 0;                  // set match value
95.                 TIM2->CCER = 1;                  // enable ch 1 compare mode
96.                             TIM2->CR1 = 1;              // enable TIM2
97.
98.  // Setup TIM3 for system clock
99.  TIM3->PSC = 16000;                         // divided by 16000
100. TIM2->CNT = 0;              // clear counter
101. TIM3->ARR = 0xFFFFFFFF;                                          // Max value
102.  TIM3->CR1 = 1;              // enable TIM3
103.
104. }
105.
106. // Initialize port pins then initialize LCD controller
107. void LCD_init(void) {
```

```
    delayMs(30);          // initialization sequence
    LCD_command(0x30);
    delayMs(10);
    LCD_command(0x30);
    delayMs(1);
    LCD_command(0x30);
    LCD_command(0x38);    // set 8-bit data, 2-line, 5x7 font
    LCD_command(0x06);    // move cursor right after each char
    LCD_command(0x01);    // clear screen, move cursor to home
    LCD_command(0x0F);    // turn on display, cursor blinking
}

// Send command to LCD
void LCD_command(unsigned char command) {
    GPIOA->BSRR = (RS | RW) << 16;  // RS = 0, R/W = 0
    GPIOB->ODR = command;          // put command on data bus
    GPIOA->BSRR = EN;              // pulse E high
    delayMs(0);
    GPIOA->BSRR = EN << 16;        // clear E
    if (command < 4) delayMs(2);   // command 1 and 2 needs up to 1.64ms
    else delayMs(1);
}

// Write data to the LCD
void LCD_data(char datawrite) {
    GPIOA->BSRR = RS;              // RS = 1
    GPIOA->BSRR = RW << 16;        // R/W = 0
    GPIOB->ODR = datawrite;        // put data on data bus
    GPIOA->BSRR = EN;              // pulse E high
    delayMs(0);
    GPIOA->BSRR = EN << 16;        // clear E
    delayMs(1);
}

// Start & Grab ADC conversion Data
void Grab_Conversion_Data(void){
ADC1->CR2 |= 0x40000000;                                    // start a conversion
while(!(ADC1->SR & 2)) {}                                   // wait for conv complete
result = ADC1->DR;                                          // read conversion result
}

// Display Voltage Function
void Voltage_Display(void){
voltage = result *(3.3f/4095.0f);                           // convert ADC output to voltage
sprintf(Voltage_Buffer, "Voltage: %.2f V", voltage);
Send_A_String(Voltage_Buffer);                  // Display Voltage
}

// Pulse Width Modulation
void Pulse_Width_Modulator(float pulse_width){
if((CruiseControl == 1) && (RPM < CruiseControlRPM)){
result2 = (CruiseControlRPM - RPM)*10;
TIM2->CCR1 = pulse_width + result2;}
else if((CruiseControl == 1) && (RPM > CruiseControlRPM)){
result3 = (RPM - CruiseControlRPM)*.11f;
TIM2->CCR1 = pulse_width - result3;}
else TIM2->CCR1 = pulse_width;
```

```
}

// Display RPM Function
void RPM_Display(void){
if (!(TIM1->SR & 2)) {}                         // wait until input edge is captured
else{
Current = TIM1->CCR1;                           // Read CCR1 input capture into Current
Period = Current - Last;                        // calculate the period
Last = Current;
Frequency = 1000.0f / Period;
RPM = Frequency * 60.0f; }

// Display Data
sprintf(RPM_Buffer, "RPM: %.1f", RPM);
Send_A_String(RPM_Buffer);                                  // Display RPM
}

// Cruise Control Setup
void CruiseControlSetup(void){
CruiseControl = 1;
CruiseControlRPM = RPM;
}

// Manual Mode
void ManualMode(void){
Grab_Conversion_Data();                         // Starts ADC & DAC, then reads data holding register
Voltage_Display();                              // Displays voltage result to LCD from DAC
LCD_command(0xC0);                                  // Moves LCD cursor to 2nd line
//Compute();
Pulse_Width_Modulator(result);                  // Output PWM value to motor
RPM_Display();                                      // Displays RPM result to LCD from TIM1
LCD_command(0x02);                                  // Clear LCD & move cursor back to line 1
}

// Cruise Control
void CruiseControlFunc(void){
if(j == 1){
Grab_Conversion_Data(); }               / Starts ADC & DAC, then reads data holding register
j=0;
Voltage_Display();                      // Displays voltage result to LCD from DAC
LCD_command(0xC0);                      // Moves LCD cursor to 2nd line
Pulse_Width_Modulator(result);          // Output PWM value to motor
RPM_Display();                          // Displays RPM result to LCD from TIM1
LCD_command(0x02);                      // Clear LCD & move cursor back to line 1
if(CCButtonIsPressed){
CruiseControl = 0;
j=1;
}
}

// Sending a String of Characters
void Send_A_String(char *StringOfCharacters){
while(*StringOfCharacters){
LCD_data(*StringOfCharacters++);}
}

// Delay timer
```

```
void delayMs(int n) {
    SysTick->LOAD = 16000;  // reload with number of clocks per millisecond
    SysTick->VAL = 0;       // clear current value register
    SysTick->CTRL = 0x5;    // Enable the timer
    for(i = 0; i < n; i++){
                                    1.      while((SysTick->CTRL & 0x10000) == 0){}} // wait until the COUNTFLAG is set{}
    SysTick->CTRL = 0;      // Stop the timer (Enable = 0)
}

```

Code for Cruise Control in C(New Approaches):

```
// Kevin Nelson
// Embedded Systems - Final Project - Cruise Control

// Spring 2021

#include "stm32f4xx.h"
#include <stdio.h>
#include <stdbool.h>

#define RS 0x200                                                    // PA9 mask for reg select
#define RW 0x400                                                    // PA10 mask for read/write
#define EN 0x800                                                    // PA11 mask for enable
#define CCButtonIsPressed ((GPIOA->IDR & 0x0001) != 0x0001)         // PA0 button press

void delayMs(int n);
void LCD_command(unsigned char command);
void LCD_data(char datawrite);
void LCD_init(void);
void Port_init(void);
void Send_A_String(char *StringOfCharacters);
void Voltage_Display(void);
void RPM_Display(void);
void Grab_Conversion_Data(void);
void ManualMode(void);
void CruiseControlSetup(void);
void CruiseControlFunc(void);
void Pulse_Width_Modulator(float pulse_width);
void EXTI0_IRQHandler(void);
static int i, j=1, CruiseControl;
static char Voltage_Buffer[18], RPM_Buffer[18];
static float voltage, RPM, result, Period, Current, Frequency, Last = 0.0, CruiseControlRPM, result2, result3;

// PID Functions & Variables
void Compute(void);
void SetTunings(double Kp, double Ki, double Kd);
void SetSampleTime(int NewSampleTime);
void SetOutputLimits(double Min, double Max);
void Initialize(void);
unsigned long millis(void);
static unsigned long lastTime, time;
static double ITerm, lastInput, kp, ki, kd, outMin, outMax;
static int SampleTime = 1000;

int main (void) {
Port_init();
```

```
LCD_init();
Initialize();
SetOutputLimits(CruiseControlRPM,10);
SetSampleTime(1000);
SetTunings(100,100,50);

while (1) {
if(CCButtonIsPressed == true) CruiseControlSetup();
while(CruiseControl == true) CruiseControlFunc();
ManualMode();}
}

// Port Initializations for LCD. PA9-R/S, PA10-R/W, PA11-EN, PB0-PB7 for D0-D7, respectively.
void Port_init(void) {
// Enable Clocks
RCC->AHB1ENR = 3;                  // enable GPIOA/B clocks
RCC->APB1ENR |= 0x200000003;   // enable TIM2 clock & DAC
RCC->APB2ENR |= 0x00004101;    // enable TIM 1, ADC1, & SYSCFG clock

// Enable Modes
GPIOA->MODER = 0;                          // clear pin mode
GPIOA->MODER = 0x55566B5C;              // set pins output mode, PA1 to analog(capture pot value), PA5(output PWM) to alternate function, PA8(input capture) alternate function
   GPIOA->BSRR  = 0x0C000000;    // turn off EN and R/W
GPIOA->PUPDR  = 0x1;
GPIOA->AFR[0] = 0;                                  // clear alt mode
   GPIOA->AFR[0] = 0x02100000;          // set pin to AF1 for TIM2 CH1, configure PA6 as input of TIM3 CH1
GPIOA->AFR[1] = 0;                                  // clear alt mode
   GPIOA->AFR[1] = 0x00000001;          // set alt mode TIM1_CH1
GPIOB->MODER = 0;                                                  // clear pin mode
   GPIOB->MODER = 0x55555555;          // set pins output mode

// Setup ADC1
   ADC1->CR2 = 0;              // SW trigger
   ADC1->SQR3 = 1;             // conversion sequence starts at ch 1
   ADC1->SQR1 = 0;             // conversion sequence length 1
   ADC1->CR2 |= 1;             // enable ADC1

// Setup TIM1 for input capture - PA8
TIM1->PSC = 16000;                         // divided by 16000
   TIM1->CCMR1 = 0x01;      // Input capture mode, validates transition on TI1 with 1 consecutive sample, input prescaler disabled(IC1PS bits to 0)
   TIM1->CCER = 1;             // CC1S bits are writable only when the channel is OFF(CC1E = '0' in TIMx_CCER), rising edge selected by keeping CC1P & CC1NP bits to 0
   TIM1->CR1 = 1;              // enable timer

// Setup TIM2 for output PWM - PA5
   TIM2->PSC = 1000;                                // divided by 1000
   TIM2->ARR = 750;                                 // counts up to 750
TIM2->CNT = 0;              // clear counter
TIM2->CCMR1 = 0x0068;         // set output to PWM
            TIM2->CCR1 = 0;                  // set match value
            TIM2->CCER = 1;                  // enable ch 1 compare mode
                        TIM2->CR1 = 1;              // enable TIM2

// Setup TIM3 for system clock
   TIM3->PSC = 16000;                         // divided by 16000
```

```
329. TIM2->CNT = 0;               // clear counter
330. TIM3->ARR = 0xFFFFFFFF;                                          // Max value
331.                              TIM3->CR1 = 1;              // enable TIM3
332.
333. // Setup Interrupts
334. __disable_irq();                                                 // global disable IRQs
335. SYSCFG->EXTICR[0] = 0;                                          // select port A for EXTI0
336. EXTI->IMR  = 1;                                                 // unmask EXTI0
337. EXTI->FTSR = 1;                                                 // select falling edge trigger
338. NVIC_SetPriority(EXTI0_IRQn, 1);                               // set priority interrupt for push button PA0
339. NVIC_EnableIRQ(EXTI0_IRQn);                                    // enable interrupt in NVIC
340. //__enable_irq();                    // global enable IRQs
341. }
342.
343. // Initialize port pins then initialize LCD controller
344. void LCD_init(void) {
345.     delayMs(30);          // initialization sequence
346.     LCD_command(0x30);
347.     delayMs(10);
348.     LCD_command(0x30);
349.     delayMs(1);
350.     LCD_command(0x30);
351.     LCD_command(0x38);     // set 8-bit data, 2-line, 5x7 font
352.     LCD_command(0x06);     // move cursor right after each char
353.     LCD_command(0x01);     // clear screen, move cursor to home
354.     LCD_command(0x0F);     // turn on display, cursor blinking
355. }
356.
357. // Send command to LCD
358. void LCD_command(unsigned char command) {
359.     GPIOA->BSRR = (RS | RW) << 16;  // RS = 0, R/W = 0
360.     GPIOB->ODR = command;          // put command on data bus
361.     GPIOA->BSRR = EN;              // pulse E high
362.     delayMs(0);
363.     GPIOA->BSRR = EN << 16;        // clear E
364.     if (command < 4) delayMs(2);    // command 1 and 2 needs up to 1.64ms
365.     else delayMs(1);
366. }
367.
368. // Write data to the LCD
369. void LCD_data(char datawrite) {
370.     GPIOA->BSRR = RS;              // RS = 1
371.     GPIOA->BSRR = RW << 16;        // R/W = 0
372.     GPIOB->ODR = datawrite;        // put data on data bus
373.     GPIOA->BSRR = EN;              // pulse E high
374.     delayMs(0);
375.     GPIOA->BSRR = EN << 16;        // clear E
376.     delayMs(1);
377.  }
378.
379. // Start & Grab ADC conversion Data
380. void Grab_Conversion_Data(void){
381. ADC1->CR2 |= 0x40000000;                                                  // start a conversion
382. while(!(ADC1->SR & 2)) {}                                                 // wait for conv complete
383. result = ADC1->DR;                                                       // read conversion result
384. }
385.
```

```
// Display Voltage Function
void Voltage_Display(void){
voltage = result *(3.3f/4095.0f);                                   // convert ADC output to voltage
sprintf(Voltage_Buffer, "Voltage: %.2f V", voltage);
Send_A_String(Voltage_Buffer);                  // Display Voltage
}

// Pulse Width Modulation
void Pulse_Width_Modulator(float pulse_width){
if((CruiseControl == 1) && (RPM < CruiseControlRPM)){
result2 = (CruiseControlRPM - RPM)*10;
TIM2->CCR1 = pulse_width + result2;}
else if((CruiseControl == 1) && (RPM > CruiseControlRPM)){
result3 = (RPM - CruiseControlRPM)*.11f;
TIM2->CCR1 = pulse_width - result3;}
else TIM2->CCR1 = pulse_width;
}

// Display RPM Function
void RPM_Display(void){
if (!(TIM1->SR & 2)) {}                 // wait until input edge is captured
else{
Current = TIM1->CCR1;                   // Read CCR1 input capture into Current
Period = Current - Last;                // calculate the period
Last = Current;
Frequency = 1000.0f / Period;
RPM = Frequency * 60.0f; }

// Display Data
sprintf(RPM_Buffer, "RPM: %.1f", RPM);
Send_A_String(RPM_Buffer);                          // Display RPM
}

// Cruise Control Setup
void CruiseControlSetup(void){
CruiseControl = 1;
CruiseControlRPM = RPM;
}

// Manual Mode
void ManualMode(void){
Grab_Conversion_Data();                 // Starts ADC & DAC, then reads data holding register
Voltage_Display();                      // Displays voltage result to LCD from DAC
LCD_command(0xC0);                              // Moves LCD cursor to 2nd line
//Compute();
Pulse_Width_Modulator(result);          // Output PWM value to motor
RPM_Display();                                  // Displays RPM result to LCD from TIM1
LCD_command(0x02);                              // Clear LCD & move cursor back to line 1
}

// Cruise Control
void CruiseControlFunc(void){
if(j == 1){
Grab_Conversion_Data(); }       / Starts ADC & DAC, then reads data holding register
j=0;
Voltage_Display();              // Displays voltage result to LCD from DAC
LCD_command(0xC0);              // Moves LCD cursor to 2nd line
```

```
443.  Pulse_Width_Modulator(result);          // Output PWM value to motor
444.  RPM_Display();                          // Displays RPM result to LCD from TIM1
445.  LCD_command(0x02);                     // Clear LCD & move cursor back to line 1
446.  if(CCButtonIsPressed){
447.  CruiseControl = 0;
448.  j=1;
449.  }
450.  }
451.
452.  //// Interrupt for Cruise Control
453.  //void EXTI0_IRQHandler(void){
454.  //                    if(CCButtonIsPressed == true) CruiseControlSetup();
455.  //                    while(CruiseControl == true) CruiseControlFunc();
456.  //}
457.
458.  // Sending a String of Characters
459.  void Send_A_String(char *StringOfCharacters){
460.  while(*StringOfCharacters){
461.  LCD_data(*StringOfCharacters++);}
462.  }
463.
464.  // Amount of time that has passed since program started
465.  unsigned long millis(void){
466.  return time = TIM3->CCR1;
467.  }
468.
469.  // Compute output
470.  void Compute(){
471.    unsigned long now = millis();
472.    int timeChange = (now - lastTime);
473.    if(timeChange>=SampleTime){
474.      //Compute all the working error variables
475.      double error = CruiseControlRPM - RPM;
476.      ITerm += (ki * error);
477.      if(ITerm > outMax) ITerm= outMax;
478.      else if(ITerm < outMin) ITerm= outMin;
479.      double dInput = (RPM - lastInput);
480.
481.      //Compute PID Output
482.  TIM2->CCR1 = kp * error + ITerm - kd * dInput;
483.      if(TIM2->CCR1 > outMax) TIM2->CCR1 = outMax;
484.      else if(TIM2->CCR1 < outMin) TIM2->CCR1 = outMin;
485.
486.      //Remember some variables for next time
487.      lastInput = RPM;
488.      lastTime = now;
489.    }
490.  }
491.
492.  // Set K values
493.  void SetTunings(double Kp, double Ki, double Kd){
494.    if (Kp<0 || Ki<0|| Kd<0) return;
495.    double SampleTimeInSec = ((double)SampleTime)/1000;
496.     kp = Kp;
497.     ki = Ki * SampleTimeInSec;
498.     kd = Kd / SampleTimeInSec;
499.  }
```

```
500.
501.  // Set how often the PID samples
502.  void SetSampleTime(int NewSampleTime){
503.    if (NewSampleTime > 0){
504.      double ratio  = (double)NewSampleTime/(double)SampleTime;
505.      ki *= ratio;
506.      kd /= ratio;
507.      SampleTime = (unsigned long)NewSampleTime;
508.    }
509.  }
510.
511.   // Set the Minimum & Maximum output for each term
512.  void SetOutputLimits(double Min, double Max){
513.    if(Min > Max) return;
514.    outMin = Min;
515.    outMax = Max;
516.    if(TIM2->CCR1 > outMax) TIM2->CCR1 = outMax;
517.    else if(TIM2->CCR1 < outMin) TIM2->CCR1 = outMin;
518.    if(ITerm > outMax) ITerm= outMax;
519.    else if(ITerm < outMin) ITerm= outMin;
520.  }
521.
522.  void Initialize(){
523.    lastInput = RPM;
524.    ITerm = TIM2->CCR1;
525.    if(ITerm > outMax) ITerm= outMax;
526.    else if(ITerm < outMin) ITerm= outMin;
527.  }
528.
529.  // Delay timer
530.  void delayMs(int n) {
531.     SysTick->LOAD = 16000;  // reload with number of clocks per millisecond
532.     SysTick->VAL = 0;      // clear current value register
533.     SysTick->CTRL = 0x5;   // Enable the timer
534.     for(i = 0; i < n; i++){
                                    1.      while((SysTick->CTRL & 0x10000) == 0){}} // wait until the COUNTFLAG is set{}
535.     SysTick->CTRL = 0;     // Stop the timer (Enable = 0)
536.  }
537.
```