Assignment Two - Actuation and Sensing

48623 Mechatronics 2

William Rooke 12051342

September 3, 2020

1 Instructions

LCD Shield V1.1 was used in developing this assignment.

Peripheral Pin	Arduino Shield Pin
Stepper motor driver IN1	D13
Stepper motor driver IN2	D12
Stepper motor driver IN3	D11
Stepper motor driver IN2	D3
Stepper motor driver 5V	VCC
Stepper motor driver GND	GND
IR Sensor VCC	Analog Channel 2 5V
IR Sensor V_o	Analog Channel 2 S
IR Sensor GND	Analog Channel 2 GND

2 Code

```
/*
  MX2 Assignment 2
  Written by W Rooke
  SN: 12051342
  Date 2/9/2020
*/

// Include necessary libraries
#include <LiquidCrystal.h>
#include <avr/io.h>
#include <math.h>


// Define LCD shield button values
// These are the ideal values read from the ADC when a button is pressed
const uint16_t STEPS = 4096;
const uint16_t SEL_PB = 640;
const uint16_t UP_PB = 100;
const uint16_t DWN_PB = 257;
const uint16_t LFT_PB = 410;
const uint16_t RIT_PB = 0;
const uint16_t NONE_PB = 1023;

// Define range for button value
// This is used as a +/- value for the ideals above because the readings are inconsistent
const uint16_t PB_BOUND = 20;


// Define menu modes
// Used to display and select menus
const uint8_t MD_START = 1;

const uint8_t MD_DBG_IR = 20;
const uint8_t MD_DBG_CM = 21;
const uint8_t MD_DBG_PM = 22;
const uint8_t MD_DBG_SET = 23;
const uint8_t MD_DBG_E = 24;

const uint8_t MD_IR = 3;

const uint8_t MD_CM_START = 40;
const uint8_t MD_CM_EXIT = 41;
const uint8_t MD_CM_RUNNING = 42;

const uint8_t MD_PM = 4;

const uint8_t MD_DRV_INIT = 50;
const uint8_t MD_DRV_MOV = 51;

const uint8_t MD_SET = 60;
uint8_t menuState = MD_START;


// Define states for debug mode finite state machine
// Used to store current state of FSM
const uint8_t FSM_0Cor = 0;
const uint8_t FSM_1Cor = 1;
const uint8_t FSM_2Cor = 2;
const uint8_t FSM_3Cor = 3;
const uint8_t FSM_4Cor = 4;
const uint8_t FSM_5Cor = 5;
uint8_t FSMState = FSM_0Cor;
bool debugSel = false;

// Initialise LCD
LiquidCrystal lcd(8, 9, 4, 5, 6, 7);

// Initialise motor variables
int8_t Steps = 0;
bool clockwise = true;
uint16_t stepsLeft = STEPS;
uint16_t stepsSet = 100;
bool motorActive = false;
volatile bool stepped = false;
uint8_t motorSpeed = 2;
uint16_t motorDiv = 1;

// Initialise values for button checking and debouncing
uint16_t buttonVal = 1023;
uint16_t prevButton = 1023;
uint16_t debounceTime = 0;
uint16_t buttonElapsed = 0;
bool buttonRead = true;

// Initialise time variables
// Seconds and minutes overflow instantly to zero, only initialised to 255 to patch a bug
volatile uint16_t millisecs = 0;
volatile uint8_t seconds = 255;
volatile uint8_t minutes = 255;

// Variables used for distance calculations and sensor readings
uint16_t sensorReadings[31];
uint16_t sensorVal = 0;
uint8_t wheelSize = 20;
float numRevs = 0;
uint8_t distance = 0;

// Menu helper variables
bool menuUpdate = true;
uint16_t menuElapsed = 0;
uint16_t menuTime = 0;
volatile bool blocked = false;

void setup()
{
  // Set array of sensor readings to zero on startup
  for(uint8_t x = 0; x < (sizeof(sensorReadings)/sizeof(sensorReadings[0])); x++)
  {
    sensorReadings[x] = 0;
  }

  // Start LCD
  lcd.begin(16, 2);
  // Initialise ADC
  ADCInit();

  // Set digital pins 13, 12, 11 and 3 to output
  DDRB |= (1 << DDB5) | (1 << DDB4) | (1 << DDB3);
  DDRD |= (1 << DDD3);

  // Set timer 2 to CTC mode, set prescaler to 64, set overflow value to 250, enable overflow interrupt
  // Equates to approx 1ms period
  TCCR2A = (1 << WGM21);
  TCCR2B = (1 << CS22);
  OCR2A = 250;
  TIMSK2 = (1 << OCIE2A);

  // Initialise timer 1
  timer1Init();

  // Enable interrupts
  sei();
}

void loop()
{
  // Read and round button value
  buttonVal = buttonRound(ADCRead(0));

  // Check how much time has elapsed since last button read
  // If over 100ms, check if same as previous value
  // If same, set buttonRead flag, if not, save previous value and do nothing
  buttonElapsed = millisecs - debounceTime;

  if (buttonElapsed > 100)
  {
    debounceTime = millisecs;
    if ((buttonVal == prevButton) && (buttonVal != NONE_PB))
    {
      buttonRead = true;
    }
    else
    {
      prevButton = buttonVal;
    }
  }

  // Check how much time has elapsed since menu was last updated
  // If over 250ms, update menu
  menuElapsed = millisecs - menuTime;
  if (menuElapsed > 250)
  {
    menuTime = millisecs;
    menuUpdate = true;
  }

  // Case switch statement which deals with the various menu states
  switch (menuState)
  {
    // Start up mode
    case MD_START:
      // Print minutes, seconds since startup and SN
      lcd.setCursor(0, 0);
      lcd.print(minutes);
      lcd.print(":");
      lcd.print(seconds);
      printHelp("",0,0);
      lcd.setCursor(0, 1);
      printHelp("12051342", 0, 0);

      // If a button has been read, handle it
      if (buttonRead)
      {
        switch (buttonVal)
        {
          // If the debug sequence has not been entered/completed, go to drive mode
          // Otherwise go to debug mode
          case SEL_PB:
            lcd.clear();
            if (debugSel)
            {
              menuState = MD_DBG_IR;
              debugSel = false;
            }
            else
            {
              lcd.setCursor(0,0);
              lcd.print("Drive Mode");
              motorSpeed = 3;
              menuState = MD_DRV_INIT;
            }
            break;

          case NONE_PB:
            break;

          // If anything other than select is pressed, handle it with the debug finite state machine
          default:
            debugFSM();
            break;
        }
      }
      break;

    // Debug mode with IR mode blinking
    case MD_DBG_IR:
      // Pring debug mode and menu string
      lcd.setCursor(0, 0);
      printHelp("DEBUG Mode", 0, 0);
      lcd.setCursor(0, 1);
      printHelp("IR CM PM SET E", 0, 2);

      // If a button has been read, handle it
      if (buttonRead)
      {
        switch (buttonVal)
        {
          // Select, go to IR mode
          case SEL_PB:
            lcd.clear();
            menuState = MD_IR;
            break;

          // Left and right navigate through the menu
          case LFT_PB:
            menuState = MD_DBG_E;
            break;

          case RIT_PB:
            menuState = MD_DBG_CM;
            break;

          // Anything else, do nothing
          default:
            break;
        }
      }
      break;

    // Debug mode with CM flashing
    case MD_DBG_CM:
      // Print debug menu
      lcd.setCursor(0, 0);
      printHelp("DEBUG Mode", 0, 0);
      lcd.setCursor(0, 1);
      printHelp("IR CM PM SET E", 3, 2);

      // Handle button press
      if (buttonRead)
      {
        switch (buttonVal)
        {
          // Select, go to CM mode
          // Set motor speed to 2, in case it changed somewhere else
          case SEL_PB:
            lcd.clear();
            motorSpeed = 2;
            menuState = MD_CM_START;
            break;

          // Left and right navigate menu
          case LFT_PB:
            menuState = MD_DBG_IR;
            break;

          case RIT_PB:
            menuState = MD_DBG_PM;
            break;

          default:
            break;
        }
      }
      break;

    // Debug mode with PM flashing
    case MD_DBG_PM:
      // Print debug menu
      lcd.setCursor(0, 0);
      printHelp("DEBUG Mode", 0, 0);
      lcd.setCursor(0, 1);
      printHelp("IR CM PM SET E", 6, 2);
      // Handle button press
      if (buttonRead)
      {
        switch (buttonVal)
        {
          case SEL_PB:
            // Go to PM mode
            // Set number of steps to default, motorspeed to maximum
            // PM Mode is printed here because it takes too long to print in PM mode itself, slowing the motor down considerably
            lcd.clear();
            menuState = MD_PM;
            lcd.setCursor(0,0);
            motorSpeed = 3;
            stepsSet = 100;
            lcd.print("PM Mode");
            break;

          // Left and right navigate menu
          case LFT_PB:
            menuState = MD_DBG_CM;
            break;

          case RIT_PB:
            menuState = MD_DBG_SET;
            break;

          default:
            break;
        }
      }
      break;

    // Debug mode with SET flashing
    case MD_DBG_SET:
      // Print debug menu
      lcd.setCursor(0, 0);
      printHelp("DEBUG Mode", 0, 0);
      lcd.setCursor(0, 1);
      printHelp("IR CM PM SET E", 9, 3);
      // Handle button press
      if (buttonRead)
      {
        switch (buttonVal)
        {
          // Select, go to settings mode
          case SEL_PB:
            lcd.clear();
            menuState = MD_SET;
            break;

          // Left and right navigate menu
          case LFT_PB:
            menuState = MD_DBG_PM;
            break;

          case RIT_PB:
            menuState = MD_DBG_E;
            break;

          default:
            break;
        }
      }
      break;

    // Debug mode with E flashing
    case MD_DBG_E:
      // Print debug menu
      lcd.setCursor(0, 0);
      printHelp("DEBUG Mode", 0, 0);
      lcd.setCursor(0, 1);
      printHelp("IR CM PM SET E", 13, 1);
      // Handle button press
      if (buttonRead)
      {
        switch (buttonVal)
        {
          // Select returns to start up mode
          case SEL_PB:
            lcd.clear();
            menuState = MD_START;
            debugSel = false;
            FSMState = FSM_OCor;
            break;

          // Left and right navigate menu
          case LFT_PB:
            menuState = MD_DBG_SET;
            break;

          case RIT_PB:
            menuState = MD_DBG_IR;
            break;

          default:
            break;
        }
      }
      break;

    // IR Mode
    case MD_IR:
      // Read ADC and store it in array sensorReadings
      arrayIncrement(sensorReadings,sizeof(sensorReadings),ADCRead(2));

      // Print IR mode
      lcd.setCursor(0,0);
      printHelp("IR Mode", 0, 0);
      lcd.setCursor(0,1);
      // Calculate distance in cm and print
      distance = IRFunc(arrayAverage(sensorReadings, sizeof(sensorReadings)));
      lcd.print(distance);
      printHelp("cm",0,0);

      // Handle button press
      if (buttonRead)
      {
        // Select, exit to debug mode
        if (buttonVal == SEL_PB)
        {
          lcd.clear();
          menuState = MD_DBG_IR;
        }
      }
      break;

    // CM Mode with start blinking
    case MD_CM_START:
      // Print CM mode and Start Exit menu
      lcd.setCursor(0,0);
      printHelp("CM Mode", 0, 0);
      lcd.setCursor(0,1);
      printHelp("Start Exit", 0, 5);

      // Handle button press
      if (buttonRead)
      {
        switch (buttonVal)
        {
          // Select, clear LCD, print CM mode, motor direction and start motor running
          // Printed here rather than in running mode because it takes too long to print and slows down the motor
          case SEL_PB:
            lcd.clear();
            lcd.setCursor(0,0);
            lcd.print("CM Mode");
            lcd.setCursor(0,1);
            lcd.print(" CW");
            menuState = MD_CM_RUNNING;
            motorActive = true;
            break;

          // Left nd right navigate menu
          case LFT_PB:
            menuState = MD_CM_EXIT;
            break;

          case RIT_PB:
            menuState = MD_CM_EXIT;
            break;

          default:
            break;
        }
      }
      break;

    // CM Mode with exit blinking
    case MD_CM_EXIT:
      // Print CM mode and Start Exit menu
      lcd.setCursor(0,0);
      printHelp("CM Mode", 0, 0);
      lcd.setCursor(0,1);
      printHelp("Start Exit", 6, 4);

      // Handle button press
      if (buttonRead)
      {
        switch (buttonVal)
        {
          // Select, returns to debug mode
          case SEL_PB:
            lcd.clear();
            menuState = MD_DBG_CM;
            break;

          // Left and right navigate menu
          case LFT_PB:
            menuState = MD_CM_START;
            break;

          case RIT_PB:
            menuState = MD_CM_START;
            break;

          default:
            break;
        }
      }
      break;

    // CM mode with motor running
    case MD_CM_RUNNING:
      lcd.setCursor(0,1);
      lcd.print(motorSpeed);

      // Handle button press
      if (buttonRead)
      {
        switch (buttonVal)
        {
          // Select, stops the motor, goes back to CM mode with start mode flashing, and sets clockwise and speed to default value
          case SEL_PB:
            lcd.clear();
            menuState = MD_CM_START;
            motorSpeed = 2;
            clockwise = true;
            motorActive = false;
            break;

          // Left and right set and print direction
          case LFT_PB:
            clockwise = true;
            lcd.print(" CW ");
            break;

          case RIT_PB:
            clockwise = false;
            lcd.print(" CCW ");
            break;

          // Up and down increase and decrease the motor speed respectively
          case UP_PB:
            if (motorSpeed < 3)
            {
              motorSpeed++;
            }
            else
            {
              motorSpeed = 3;
            }
            break;

          case DWN_PB:
            if (motorSpeed > 0)
            {
              motorSpeed--;
            }
            else
            {
              motorSpeed = 0;
            }
          break;

        default:
          break;
      }
    }
    break;

  // PM Mode
  case MD_PM:
    // Print the number of steps set
    lcd.setCursor(0, 1);
    lcd.print(stepsSet);
    lcd.print(" ");

    // If the motor is not active, pad out the LCD with nothing so it doesn't have any leftover numbers
    if (!motorActive)
    {
      printHelp("",0,0);
    }

    // If the motor is active, print the number of steps remaining
    else
    {
      lcd.print(" ");
      lcd.print(stepsLeft);
      lcd.print(" ");
    }

    // Handle button press if motor is not running
    if (buttonRead && !motorActive)
    {
      switch (buttonVal)
      {
        // Up and down increase and decrease the number of steps
        case UP_PB:
          if (stepsSet < 65500)
          {
            stepsSet += 100;
          }
          break;

        case DWN_PB:
          if (stepsSet != 0)
          {
            stepsSet -= 100;
          }
          break;

        // Left sets the number of steps to default of 100
        case LFT_PB:
          stepsSet = 100;
          break;

        // Right starts the motor
        case RIT_PB:
          motorActive = true;
          stepsLeft = stepsSet;
          break;

        // Select stops the motor and returns to debug mode
        case SEL_PB:
          stepsSet = 100;
          motorActive = false;
          lcd.clear();
          menuState = MD_DBG_PM;
          break;

        default:
          break;
      }
  }
  // Handle button press if motor is running
  else if (buttonRead)
  {
    // Select returns to debug mode, stops motor
    if (buttonVal == SEL_PB)
    {
      stepsSet = 100;
      motorActive = false;
      lcd.clear();
      menuState = MD_DBG_PM;
    }
  }
  break;

// Settings mode
case MD_SET:
  // Print settings mode and wheel size
  motorActive = false;
  lcd.setCursor(0,0);
  printHelp("SETTINGS Mode", 0, 0);
  lcd.setCursor(0, 1);
  lcd.print("Wheel: ");
  lcd.print(wheelSize);
  printHelp("cm", 0, 0);

  // Handle button press
  if (buttonRead)
  {
    switch (buttonVal)
    {
      // Up and down increase and decreas the wheel size
      case UP_PB:
        if (wheelSize < 250)
        {
          wheelSize += 10;
        }
        break;

      case DWN_PB:
        if (wheelSize > 10)
        {
          wheelSize -= 10;
        }
        break;

      // Select returns to debug menu
      case SEL_PB:
        menuState = MD_DBG_SET;
        break;

      default:
        break;
    }
  }
  break;

// Drive mode
case MD_DRV_INIT:
  // If the motor is not runnning, take IR sensor reading, find distance and round to 1 DP
  // Then find number of revolutions and steps needed to reach distance, then print all information
  if (!motorActive)
  {
    lcd.setCursor(0, 1);
    arrayIncrement(sensorReadings,sizeof(sensorReadings),ADCRead(2));
    distance = IRFunc(arrayAverage(sensorReadings, sizeof(sensorReadings)));
    numRevs = round((((float)distance/(float)wheelSize) * 10)) / 10.0;
    stepsLeft = numRevs * (float)STEPS;
    lcd.print(distance);
    lcd.print("cm ");
    lcd.setCursor(6,1);
```

```
          lcd.print(numRevs, 1);
        }

        // If the menu needs to be updated (250ms), print the number of steps left
        // This is done because printing it in real time takes too long and slows down the motor
        if (menuUpdate)
        {
          lcd.setCursor(10,1);
          lcd.print(stepsLeft);
          lcd.print("   ");
        }

        // Handle button press when motor is not running
        if (buttonRead && !motorActive)
        {
          switch (buttonVal)
          {
            // Up starts motor clockwise
            case UP_PB:
              motorActive = true;
              clockwise = true;
              break;

            // Down starts motor counter clockwise
            case DWN_PB:
              motorActive = true;
              clockwise = false;
              break;

            // Select returns to start up mode, stops motor
            case SEL_PB:
              motorActive = false;
              menuState = MD_START;
              debugSel = false;
              FSMState = FSM_OCor;
              break;

            default:
              break;
            }
        }

        // Handle button press when motor is running
        else if (buttonRead)
        {
          // Select returns to start up mode, stops motor
          if (buttonVal == SEL_PB)
          {
              motorActive = false;
              menuState = MD_START;
              debugSel = false;
              FSMState = FSM_OCor;
          }
        }
        break;
    }

    // Buttons have been handled and menu has been updated, set to false to ensure they don't get read again until necessary
    buttonRead = false;
    menuUpdate = false;


    // If the motor has not stepped this cycle and is active, and also has steps remaining, step motor
    if ((!stepped && motorActive) && (stepsLeft > 0))
    {
      // Set stepped flag as the motor has stepped
      stepped = true;
      // Run stepper function
      stepper();

      // If the motor is in any mode other than continuous, decrement step count
      if (menuState != MD_CM_RUNNING)
      {
        stepsLeft--;
      }
    }

    // If the number of steps remaining is zero, stop the motor
    else if (stepsLeft == 0)
    {
      motorActive = false;
    }

  }

  /* This function handles the digital pin toggles to drive the motor */
  // Function operates by switching off all outputs in each case, then turning on the required ones
  // It probably isn't the most efficient, but it makes sense for bidirectional operation
  void stepper(void)
  {
    switch (Steps)
    {
      // Turn on D13
      case 0:
        PORTB = 0;
        PORTD = 0;
        PORTB |= (1 << PORTB5);
        break;

      // Turn on D13 and D12
      case 1:
        PORTB = 0;
        PORTD = 0;
        PORTB |= (1 << PORTB5);
        PORTB |= (1 << PORTB4);
        break;

      // Turn on D12
      case 2:
        PORTB = 0;
        PORTD = 0;
        PORTB |= (1 << PORTB4);
        break;

      // Turn on D12 and D11
      case 3:
        PORTB = 0;
        PORTD = 0;
        PORTB |= (1 << PORTB4);
        PORTB |= (1 << PORTB3);
        break;

      // Turn on D11
      case 4:
        PORTB = 0;
        PORTD = 0;
        PORTB |= (1 << PORTB3);
        break;
      // Turn on D11 and D3
      case 5:
        PORTB = 0;
        PORTD = 0;
        PORTB |= (1 << PORTB3);
        PORTD |= (1 << PORTD3);
        break;

      // Turn on D3
      case 6:
        PORTD = 0;
        PORTB = 0;
        PORTD |= (1 << PORTD3);
        break;
      // Turn on D3 and D13
      case 7:
        PORTD = 0;
        PORTB = 0;
        PORTD |= (1 << PORTD3);
        PORTB |= (1 << PORTB5);
        break;
    }
    SetDirection();
  }

  /* This Function handles the resetting of direction, when reaching the end of the rotation */
  void SetDirection()
  {
    if (clockwise == true)
    {
      Steps++;
    }
    else
    {
      Steps--;
    }
    if (Steps > 7)
    {
      Steps = 0;
    }
    if (Steps < 0)
    {
      Steps = 7;
    }
  }

  // Initialises the ADC
  void ADCInit()
  {
    // Use interval voltage reference
    ADMUX |= (1 << REFS0);

    // Set 8-bit resolution
    // ADMUX |= (1 << ADLAR);

    // 128 prescale for 16Mhz (maybe change this, I dunno what the fuck it means)
    ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);

    // Enable the ADC
    ADCSRA |= (1 << ADEN);
  }

  // Reads from the ADC
  uint16_t ADCRead(uint8_t channel)
  {
    // If the channel is out of range, return zero
    if ((channel < 0) || (channel > 7))
    {
      return 0;
    }

    // Set ADCMux to zero and select VCC as reference
    ADMUX = (1 << REFS0);

    // Mask and select ADC channel to read from
    ADMUX |= (0b00001111 & channel);

    // Start ADC read
    ADCSRA |= (1 << ADSC);

    // Do nothing while reading
    while ((ADCSRA & (1 << ADSC)));

    // Return read ADC value
    return ADC;
  }

  // Initialised timer 1
  void timer1Init()
  {
    // Clear timer control registers, enusre correct values are being set
    TCCR1B = 0;
    TCCR1A = 0;

    // Set overflow clear value, will clear at 1s
    // f=f_io/(1024*(1+OCR1A))
    OCR1A = 15625;

    // Set timer to have 1024 prescaler and run in CTC mode
    TCCR1B = (1 << WGM12) | (1 << CS12) | (1 << CS10);

    // Ensure timer is not disabled
    PRR &= ~(1 << PRTIM1);

    // Enable compare interrupt
    TIMSK1 = (1 << OCIE1A);
  }

  // Takes a string, pads it to 16 characters and blocks characters from index to index+numToBlock
  // Used to make menus blink and ensure no stray characters are left printed to the LCD
  void printHelp(char inString[], uint8_t index, uint8_t numToBlock)
  {
    // Find size of string
    size_t arraySize = strlen(inString);

    // Create new 16 char long string
    char outString[16];

    // Copy inString to outString and pad with spaces
    for (uint8_t x = 0; x < 16; x++)
    {
      if (x < arraySize)
      {
        outString[x] = inString[x];
      }
      else
      {
        outString[x] = ’ ’;
      }
    }

    // If the menu is to have block chars instead of regular characters, block out the desired chars and print to LCD
    if (blocked)
    {
      for (uint8_t x = index; x < (index + numToBlock); x++)
      {
        outString[x] = 0xFF;
      }
      lcd.print(outString);
    }
    else
    {
      lcd.print(outString);
    }
  }

  // Rounds the button values
  // They can be inconsistent so this just makes life easier
  int buttonRound(int checkValue)
  {
    // Check if the value is within the given range for a given button value
    // If it is, return the ideal value
    if ((checkValue >= (LFT_PB - PB_BOUND)) && (checkValue <= (LFT_PB + PB_BOUND)))
    {
      return LFT_PB;
    }
    if (checkValue <= (RIT_PB + PB_BOUND))
    {
      return RIT_PB;
    }
    if ((checkValue >= (UP_PB - PB_BOUND)) && (checkValue <= (UP_PB + PB_BOUND)))
    {
      return UP_PB;
    }
    if ((checkValue >= (DWN_PB - PB_BOUND)) && (checkValue <= (DWN_PB + PB_BOUND)))
    {
      return DWN_PB;
    }
    if ((checkValue >= (SEL_PB - PB_BOUND)) && (checkValue <= (SEL_PB + PB_BOUND)))
    {
      return SEL_PB;
    }
    if ((checkValue >= (NONE_PB - PB_BOUND)) && (checkValue <= (NONE_PB + PB_BOUND)))
    {
      return NONE_PB;
    }
  }

  // Deals with the debug button sequence
  void debugFSM(void)
  {
    if (buttonVal != NONE_PB)
    {
      // Case switch for the current state of the FSM
      switch (FSMState)
      {
        // Initial state with zero correct inputs
        case FSM_0Cor:
          // Left pressed advances to next state
          if (buttonVal == LFT_PB)
          {
           FSMState = FSM_1Cor;
          }
          // Anything else returns to zero correct state
          else
          {
           FSMState = FSM_0Cor;
          }
          break;

        // One correct input
        case FSM_1Cor:
          // Left pressed advances to next state
          if (buttonVal == LFT_PB)
          {
           FSMState = FSM_2Cor;
          }
          // Anything else returns to zero correct state
          else
          {
           FSMState = FSM_0Cor;
          }
          break;

        // Two correct inputs
        case FSM_2Cor:
          // Left pressed remains in same state
          if (buttonVal == LFT_PB)
          {
           FSMState = FSM_2Cor;
          }
          // Up pressed advances to next state
          else if (buttonVal == UP_PB)
          {
           FSMState = FSM_3Cor;
          }
          // Anything else returns to zero correct state
          else
          {
           FSMState = FSM_0Cor;
          }
          break;

        // Three correct inputs
        case FSM_3Cor:
          // Right marks the next select to enter debug menu
          if (buttonVal == RIT_PB)
          {
           FSMState = FSM_4Cor;
           debugSel = true;
          }
          // Anything else returns to zero correct state
          else
          {
           FSMState = FSM_0Cor;
          }
          break;

        // If this state is reached, return to zero correct
        case FSM_4Cor:
          FSMState = FSM_0Cor;
          debugSel = false;
      }
    }
   }

   // Takes an array, pushes oldest value out, moves all values down and inserts the new value
   void arrayIncrement(uint16_t inArray[], size_t arraySize, uint16_t newValue)
   {
    for (size_t x = (arraySize / sizeof(inArray[0])); x > 0; x--)
    {
      inArray[x] = inArray[x-1];
    }
    inArray[0] = newValue;
   }

   // Finds average of an array
   uint16_t arrayAverage(uint16_t inArray[], size_t arraySize)
   {
    // Iterate through array and find sum
    uint32_t sum = 0;
    for (size_t x = 0; x <= (arraySize / sizeof(inArray[0])); x++)
    {
      sum += inArray[x];
    }
    // Return sum divided by number of elements
    return (sum / (arraySize / sizeof(inArray[0])));
   }

   // Calculates the distance from the IR sensor ADC value
   uint8_t IRFunc (uint16_t inVal)
   {
    // Expression to calculate the distance, found from excel line of best fit when calibrating
    uint8_t outVal = round(pow((float)inVal/18109,1.0/-1.09));

    // Useful distance of the IR sensor is 150cm, so anything above will be an unstable reading
    // If the reading is over 150, round it to 150
    // Otherwise return the calculated value
    if (outVal >= 150)
    {
      return 150;
    }
    else
    {
      return outVal;
    }
   }
```

```
// Timer 1 ISR, runs every 1s
ISR(TIMER1_COMPA_vect)
{
  // Increment seconds and minutes
  if (seconds >= 59)
  {
    seconds = 0;
    minutes++;
  }
  else
  {
    seconds++;
  }
  // Clear interrupt flag, not strictly necessary because it gets cleared when the ISR runs
  TIFR1 = (1 << OCF1A);

  // Flip value which blocks out menu selection
  blocked = !blocked;
}

// Timer 2 ISR, runs ever 1ms
ISR(TIMER2_COMPA_vect)
{
  // Increment millisecond count
  millisecs++;

  // Increment motor divider, used to control the speed of the motor
  motorDiv++;

  // If the divider is greater than the scaled required speed, set divider to zero and flag the motor for stepping
  if (motorDiv > ((3 - motorSpeed) * 2))
  {
    motorDiv = 0;
    stepped = false;
  }

}
```