Assignment Three - Embedded Integration

48623 Mechatronics 2

William Rooke 12051342

October 6, 2020

1 Instructions

LCD Shield V1.1 was used in developing this assignment. When in WallFollow or Navigation modes, the desired button must be held down for a period of time to register the input.
Baud rate: 9600
Line ending: '\r\n' as in demo code.

2 Code

```
/*
  MX2 Assignment 3
  Written by W Rooke
  SN: 12051342
  Date 6/10/2020
*/

// Include necessary libraries
#include <LiquidCrystal.h>
#include <avr/io.h>
#include <float.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_CON = 10;
const uint8_t MD_START_SWP = 11;
const uint8_t MD_START_WF = 12;
const uint8_t MD_START_NAV = 13;

const uint8_t MD_CON = 20;

const uint8_t MD_SWP = 30;

const uint8_t MD_WF = 40;

const uint8_t MD_NAV = 50;
const uint8_t MD_NAV_FIN = 51;

uint8_t menuState = MD_START_CON;

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

// 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
const uint8_t NUMREADINGS = 72;
float sensorReadings[NUMREADINGS];
float distance = 0.0;
uint16_t sensorRotation = 0;

// Menu helper variable
volatile bool blocked = false;

// String variable used for sending commands
String commandString = "";

// Wall follow and navigation mode booleans
bool wallFollow = false;
bool nav = false;

// Variables for tracking wall follow distances and angles
float prevWallDist = 2.0;
float currWallDist = 0.0;
float wallAngle = 90.0;
float wfDistance = 0.5;
uint8_t farCorrections = 0;
uint8_t nearCorrections = 0;

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

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

  // Init ADC
  ADCInit();

  // Initialise timer 1
  timer1Init();

  // 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);

  // Start serial
  Serial.begin(9600);

  // Send start command to sim
  PrintMessage("CMD_START");
  // 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;
  }
}

// Case switch statement which deals with the various menu states
switch (menuState)
{
  // Main menu with control mode flashing
  case MD_START_CON:
    // Print SN and flash con mode
    lcd.setCursor(0, 0);
    lcd.print("12051342");
    printHelp("", 0, 0);
    lcd.setCursor(0, 1);
    printHelp("Main Menu Con", 10, 3);

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

        // Down cycles menu
        case DWN_PB:
          menuState = MD_START_SWP;
          break;

        default:
          break;
      }
    }
    break;

  // Main menu with sweep mode flashing
  case MD_START_SWP:
    // Pring SN and flash sweep mode
    lcd.setCursor(0, 0);
    lcd.print("12051342");
    printHelp("", 0, 0);
    lcd.setCursor(0, 1);
    printHelp("Main Menu Sweep", 10, 5);

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

        // Down, cycle through menu
        case DWN_PB:
          menuState = MD_START_WF;
          break;
```

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

// Main menu with WF flashing
case MD_START_WF:
  lcd.setCursor(0, 0);
  lcd.print("12051342");
  printHelp("", 0, 0);
  lcd.setCursor(0, 1);
  printHelp("Main Menu WF", 10, 2);

  // Handle button press
  if (buttonRead)
  {
    switch (buttonVal)
    {
      case SEL_PB:
        // Select, start wall follow procedure
        // Update menu, then sweep and adjust for 2m gap, then begin following wall
        menuState = MD_WF;
        distance = sensorReadings[Sweep(true)];
        if (distance < 2.0)
        {
          commandString = String("CMD_ACT_LAT_0_" + String(2.0 - distance));
        }
        else
        {
          commandString = String("CMD_ACT_LAT_1_" + String(distance - 2.0));
        }
        PrintMessage(commandString);
        PrintMessage("CMD_ACT_ROT_1_90");
        lcd.clear();
        wallFollow = true;
        break;

      // Doum, cycle menu
      case DWN_PB:
        menuState = MD_START_NAV;
        break;

      default:
        break;
    }
  }
  break;

// Main menu with Nav flashing
case MD_START_NAV:
  lcd.setCursor(0, 0);
  lcd.print("12051342");
  printHelp("", 0, 0);
  lcd.setCursor(0, 1);
  printHelp("Main Menu Nav", 10, 3);
  // Handle button press
  if (buttonRead)
  {
    switch (buttonVal)
    {
      // Select, start navigating to goal
      case SEL_PB:
        menuState = MD_NAV;
        lcd.clear();
        nav = true;
        break;

      // Doum, navigate menu
      case DWN_PB:
```

```
        menuState = MD_START_CON;
        break;

      default:
        break;
    }
  }
  break;

// Control mode
case MD_CON:

  lcd.setCursor(0, 0);
  lcd.print("12051342");
  lcd.setCursor(0, 1);
  lcd.print("Control");
  // Handle button press
  if (buttonRead)
  {
    switch (buttonVal)
    {
      // Select, go to main menu
      case SEL_PB:
        lcd.clear();
        menuState = MD_START_CON;
        break;

      // Left and right, rotate bot
      case LFT_PB:
        PrintMessage("CMD_ACT_ROT_0_5");
        break;

      case RIT_PB:
        PrintMessage("CMD_ACT_ROT_1_5");
        break;

      // Up and down, move backward and forward
      case UP_PB:
        PrintMessage("CMD_ACT_LAT_1_0.5");
        break;

      case DWN_PB:
        PrintMessage("CMD_ACT_LAT_0_0.5");
        break;

      default:
        break;
    }
  }
  break;

// Sweep mode
case MD_SWP:

  lcd.setCursor(0, 0);
  lcd.print("12051342");
  lcd.setCursor(0, 1);
  lcd.print("Sweep");
  // Handle button press
  if (buttonRead)
  {
    switch (buttonVal)
    {
      // Select returns to start up mode
      case SEL_PB:
        lcd.clear();
        menuState = MD_START_SWP;
        break;

      // Up, sweep
      case UP_PB:
```

```
        Sweep(true);
        break;

      default:
        break;
    }
  }
  break;

// WF Mode
case MD_WF:

  lcd.setCursor(0, 0);
  lcd.print("12051342");
  lcd.setCursor(0, 1);
  lcd.print("Wall follow");

  // Handle button press
  if (buttonRead)
  {
    switch (buttonVal)
    {
      // Select returns to main menu
      case SEL_PB:
        lcd.clear();
        menuState = MD_START_WF;
        wallFollow = false;
        break;

      // Up, stop following wall
      case UP_PB:
        wallFollow = false;
        break;
    }
  }
  break;

// Nav Mode
case MD_NAV:
  lcd.setCursor(0, 0);
  lcd.print("12051342");
  lcd.setCursor(0, 1);
  lcd.print("Navigation");

  // Handle button press
  if (buttonRead)
  {
    switch (buttonVal)
    {
      // Select, stop navigating and return to main menu
      case SEL_PB:
        lcd.clear();
        menuState = MD_START_NAV;
        nav = false;
        break;
    }
  }
  break;

// Navigation finished mode
case MD_NAV_FIN:
  lcd.setCursor(0, 0);
  lcd.print("Finished");
  lcd.setCursor(0, 1);
  lcd.print("Navigation");
  // Handle button press
  if (buttonRead)
  {
    switch (buttonVal)
    {
      // Select returns to main menu
```

```
        case SEL_PB:
          lcd.clear();
          menuState = MD_START_NAV;
          nav = false;
          break;
        }
      }
      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;

  // If bot is meant to follow wall, follow wall
  if (wallFollow)
  {
    FollowWall();
  }

  // If bot is meant to nav to goal, nav to goal
  if (nav)
  {
    NavToGoal();
  }
}


// Navigates around the map semi-randomly and pings for goal distance
// When goal is within range, find it
void NavToGoal()
{
  // Ping distance to goal
  PrintMessage("CMD_SEN_PING");
  float goalDistA = SerialRead();

  // If the goal is within range, search for it
  if (goalDistA != 0)
  {
    // If the goal is within 0.5m, consider the goal found and stop navigating
    if (goalDistA <= 0.5)
    {
      nav = false;
      menuState = MD_NAV_FIN;
    }

    // If goal is not within 0.5m, move back 0.5m and take new distance reading
    else
    {
      float goalDistB;
      PrintMessage("CMD_ACT_LAT_0_0.5");
      PrintMessage("CMD_SEN_PING");
      goalDistB = SerialRead();

      // If both goal readings are within range, execute FindGoal
      if (goalDistB != 0)
      {
        FindGoal(goalDistA, goalDistB);
      }

      // If second goal reading is not within range, move forward 0.5m, rotate 90deg and take IR measurement
      // If there's nothing within 0.5m move to that spot. The function will run again at this new point and will hopefully find t
      else
      {
        PrintMessage("CMD_ACT_LAT_1_0.5");
        PrintMessage("CMD_SEN_ROT_90");
        PrintMessage("CMD_SEN_IR");
        if (SerialRead() > 0.5)
        {
          PrintMessage("CMD_ACT_ROT_0_90");
          PrintMessage("CMD_ACT_LAT_1_0.5");
        }
```

```
      }
    }
  }

  // If the goal is not within range, sweep for largest distance and move towards that
  // Provides a semi-random way of navigating the map
  else
  {
    distance = sensorReadings[Sweep(false)];
    if (distance < 5)
    {
      commandString = String("CMD_ACT_LAT_1_" + String(distance - 0.5));
      PrintMessage(commandString);
    }
    else
    {
      PrintMessage("CMD_ACT_LAT_1_4");
    }
  }
}

// Function for finding bearing and distance of goal from 2 distance readings
// Used trilateration to find the goal
void FindGoal(float distanceA, float distanceB)
{
  // Ensure distanceA is always greater than distanceB
  // Helped with some weird angle NaN errors
  if (distanceB > distanceA)
  {
    float temp = distanceB;
    distanceB = distanceA;
    distanceA = temp;
  }

  // Find X and Y coordinates
  // Formulae from https://en.wikipedia.org/wiki/True-range_multilateration#Two_Cartesian_dimensions,_two_measured_slant_ranges_(
  // This method gives two "points of interest" (POIs) at (x,y) and (x,-y) so both must be checked
  float x = (pow(distanceA, 2) - pow(distanceB, 2) + pow(0.5, 2)) / (2 * 0.5);
  float y = sqrt(pow(distanceA, 2) - (pow(x, 2)));

  // If y is NaN, abort function to avoid crashes
  if (y != y)
  {
    return;
  }

  // Find bearing and distance of goal using pythagorus
  float GoalAngle = RadsToDegrees(atan(y/x));
  float goalRange = sqrt(pow(x, 2) + pow(y, 2));

  // Rotate to first POI and move to it
  commandString = String("CMD_ACT_ROT_0_" + String(GoalAngle));
  PrintMessage(commandString);
  commandString = String("CMD_ACT_LAT_1_" + String(goalRange));
  PrintMessage(commandString);

  // Ping goal distance, if it isn't 0 and within 0.5m, consider it found and stop navigating
  PrintMessage("CMD_SEN_PING");
  float findGoalDist = SerialRead();
  if ((findGoalDist <= 0.5) && (findGoalDist > 0))
  {
    nav = false;
    menuState = MD_NAV_FIN;
  }

  // If the goal was not at the first POI, check the second by moving back to the initial point,
  // rotating 2x the initial angle in the opposite direction, and moving towards the second POI
  else
  {
    commandString = String("CMD_ACT_LAT_0_" + String(goalRange));
    PrintMessage(commandString);
```

```
    commandString = String("CMD_ACT_ROT_1_" + String(2 * GoalAngle));
    PrintMessage(commandString);
    commandString = String("CMD_ACT_LAT_1_" + String(goalRange));
    PrintMessage(commandString);

    // Ping goal distance, if it isn't 0 and within 0.5m, consider it found and stop navigating
    PrintMessage("CMD_SEN_PING");
    findGoalDist = SerialRead();
    if ((findGoalDist <= 0.5) && (findGoalDist > 0))
    {
      nav = false;
      menuState = MD_NAV_FIN;
    }
  }
}

// Reads from the serial port and puts the read value into float form
float SerialRead()
{
  // Wait until there is data available
  while (Serial.available() == 0);

  // Read string until terminator carriage return and newline are found
  String inString = Serial.readStringUntil('\r\n');

  // If the string starts with an N, its a NAN and should be considered the largest number
  if (inString[0] == 'N')
  {
    return FLT_MAX;
  }

  // If not NaN, return the float value
  else
  {
    return inString.toFloat();
  }
}

// Sweeps for distance readings and rotate to desired distance
// Takes bool to check for minimum or maximum distance, returns uint8_t of the distance index
uint8_t Sweep(bool min)
{
  uint8_t rotIndex = 0;

  // Sweep 360deg in 5deg intervals and read distances into sensorReadings array
  for (int16_t sensorRotation = 355; sensorRotation >= 0; sensorRotation -= 5)
  {
    commandString = String("CMD_SEN_ROT_" + String(sensorRotation));
    PrintMessage(commandString);
    PrintMessage("CMD_SEN_IR");
    sensorReadings[(sensorRotation * 2) / 10] = SerialRead();
  }

  // If looking for minimum distance, call arrayMin to find it
  if (min)
  {
    rotIndex = arrayMin(sensorReadings);
  }

  // If looking for maximum distance, call arrayMax to find it
  else
  {
    rotIndex = arrayMax(sensorReadings);
  }

  // Rotate sensor to forward position, then rotate bot to required bearing for distance
  PrintMessage("CMD_SEN_ROT_0");
  commandString = String("CMD_ACT_ROT_0_" + String((rotIndex * 10) / 2));
  PrintMessage(commandString);

  // Return the index of the min/max distance
```

```
  return rotIndex;
}

// Function to check distance, adjust and move along closest wall
void FollowWall()
{
  // Check distance from parallel wall
  PrintMessage("CMD_SEN_ROT_90");
  PrintMessage("CMD_SEN_IR");
  currWallDist = SerialRead();

  // If robot is further out than 2m (+0.15m buffer, otherwise it just adjusts every time)
  if ((currWallDist - 2.0) > 0.15)
  {
    // Rotate towards wall, move to 2m distance and rotate to parallel
    PrintMessage("CMD_ACT_ROT_0_90");
    commandString = String("CMD_ACT_LAT_1_" + String(currWallDist - 2.0));
    PrintMessage(commandString);
    PrintMessage("CMD_ACT_ROT_1_90");

    // Increment farCorrections, used to adjust angle
    farCorrections++;
  }
  // If robot is closer than 2m
  else if ((currWallDist - 2.0) < -0.15)
  {
    // Rotate towards wall, move to 2m distance and rotate to parallel
    PrintMessage("CMD_ACT_ROT_0_90");
    commandString = String("CMD_ACT_LAT_0_" + String(2.0 - currWallDist));
    PrintMessage(commandString);
    PrintMessage("CMD_ACT_ROT_1_90");

    // Increment farCorrections, used to adjust angle
    nearCorrections++;
  }

  // If the robot is tracking to move away from the wall, adjust 5deg inwards and reset correction counter
  if (farCorrections > 1)
  {
    PrintMessage("CMD_ACT_ROT_0_5");
    farCorrections = 0;
  }

  // If the robot is tracking to move toward the wall, adjust 5deg outwards and reset correction counter
  if (nearCorrections > 1)
  {
    PrintMessage("CMD_ACT_ROT_1_5");
    nearCorrections = 0;
  }

  // Check distance from upcoming wall
  PrintMessage("CMD_SEN_ROT_0");
  PrintMessage("CMD_SEN_IR");
  wfDistance = SerialRead();

  // If distance is maxed, move 3m forward
  if (wfDistance == FLT_MAX)
  {
    wfDistance = 3.0;
    commandString = String("CMD_ACT_LAT_1_" + String(wfDistance));
    PrintMessage(commandString);
  }

  // otherwise move to be 2m out from upcoming wall
  else
  {
    commandString = String("CMD_ACT_LAT_1_" + String(wfDistance - 2.0));
    PrintMessage(commandString);
    PrintMessage("CMD_ACT_ROT_1_90");
  }
}
```

```
// Converts radians to degrees
double RadsToDegrees(double radAngle)
{
  return radAngle * (180.0 / M_PI);
}

// 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;
  }
}

// Initialise 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;
}

// Finds index of minimum value in an array
uint8_t arrayMin(float inArray[])
```

```
{
  // Iterate through array and find minimum value
  float min = FLT_MAX;
  uint8_t index = 0;
  for (uint8_t x = 0; x < NUMREADINGS; x++)
  {
    if (inArray[x] < min)
    {
      index = x;
      min = inArray[x];
    }
  }
  // Return index of minimum value
  return index;
}

// Finds index of max value in an array
uint8_t arrayMax(float inArray[])
{
  // Iterate through array and find max value
  float max = 0;
  uint8_t index = 0;
  for (uint8_t x = NUMREADINGS; x > 0; x--)
  {
    if (inArray[x] > max)
    {
      index = x;
      max = inArray[x];
    }
  }
  // Return index of max value
  return index;
}

// Outputs serial command followed by terminators for MATLAB reading
void PrintMessage(String message)
{
  Serial.print(message);
  Serial.write(13); //carriage return character (ASCII 13, or '\r')
  Serial.write(10); //newline character (ASCII 10, or '\n')
}

// 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 every 1ms
ISR(TIMER2_COMPA_vect)
{
  // Increment millisecond count
  millisecs++;
}
```