Toronto Metropolitan University – Chang School of Continuing Education
CKRE 150 – Issues in Robotics and Embedded Systems
Korey Francis 500501327
2024-05-31

Modular Articulated Tabletop-game System

Introduction

Board games typically require two or more players but assembling a play group for a game is not always possible due to timing and location constraints. Employing an autonomous robotic system to take the place of an absent human player can allow a single human to enjoy a gaming session at any time. Robotic systems also make it possible for human players who are not physically present or who may have mobility constraints to participate in tabletop gaming.

This project aims to develop a robotic platform that can allow a human to play a two-person board game such as Tic-Tac-Toe or Chess against an automated system. The Modular Articulated Tabletop-game System (M. A.T.S.) will be capable of seeing a game board, analyzing the board position, algorithmically determining the optimal next move based on user defined parameters, and operating a robotic arm to make the algorithmically determined move.

The project aims to be expandable to other board games, therefore a modular approach is taken in developing the code. As such, the first game the completed system is designed to play is Tic-Tac-Toe. The game of Tic-Tac-Toe is considered solved[1] and writing a program to play it perfectly is relatively trivial when compared with the challenge of developing a program to play Chess or Go at a human level or to play more complex games involving cards, dice, and pieces of different sizes, shapes and colours. More complex games can be added to the system in future versions once the proof-of-concept version playing Tic-Tac-Toe has been successfully implemented.

Function Description

M.A.T.S is able to perform many of the same functions as a human opponent. It has a computer vision system to examine a game board and interpret what it sees as a game state. An on-board game engine will determine the next best move. The game engine will be configurable to change the difficulty or playstyle of the system thus affecting the next-best-move algorithm. An articulated robotic platform will make moves based on the output of the game engine.

The user will be able to configure the game engine via two input buttons to select the game, difficulty and turn order. A switch will allow the user to turn the system on and off. LEDs will indicate the user's selection (future versions will employ LCD screens for improved usability). The LEDs will also indicate when the system is in a "waiting position" (i.e. when the system is waiting for the human to confirm it has made a move) and when the system is in a "running position" (i.e. when the system is analyzing the board, deciding which move to make, and operating the robotic arm).

The system will be consistently watching the game board for new game states to determine when the human has made a move. When it is in the waiting position it will read the state of the game board every five seconds and if it detects a new valid board configuration has been created it will switch to the running

1. A solved game is one where if both players play without making mistakes, the game is guaranteed to result in a either A) a draw or B) a win for a pre-determined player based on initial conditions such as the game's setup and the turn order. An elegant demonstration of perfect play strategy for Tic-Tac-Toe was drawn by Randall Munroe and can be seen here: https://xkcd.com/832/

position automatically. The user can also indicate that they have made a move or chosen to skip their turn via a button press. If the system detects an invalid game state this will be indicated to the human player via the user interface LEDs.

When M.A.T.S. has made a move, it will return to the waiting position. On shutdown, the system will return to an "off" position. The off position returns the robotic arm's encoders to a baseline position for accurate encoder reading in future activations. It also prevents the system from responding to sensor inputs. On startup, M.A.T.S. will run through a calibration cycle to ensure that all of its motors are properly connected and to establish a zero point for each joint's encoder. The calibration cycle will ensure that the platform operates consistently even if it was shut down improperly (i.e. if there was a sudden loss of power preventing the arm from entering an off position).

Project Requirements

Knowledge

Achieving full functionality of the system will require knowledge of the following topics:

1. Articulated Robotic Platforms
   The basic functionality of the system requires a robotic platform that will be operated based on the input from the game engine. The platform will be sufficiently flexible and powerful that it can:
   a. Smoothly transition through a series of positions
   b. Utilize a gripper or similar pick-and-place actuator to manipulate game pieces

   The user's experience with the overall system will also be improved by the robotic system remaining compact and robust (i.e. the system should occupy a small footprint on a desktop and be able to withstand small bumps or jostles from regular use. Likewise, cables should be securely fastened and kept free of moving components to avoid damage and jamming.

   The robotic platform will be an articulated robotic arm with multiple degrees of freedom. The arm will be electrically actuated by DC motors.

2. Computer Vision
   The system requires the game engine to be able to see the game board and translate the image to a game state. The system will capture an image of the game board. Pattern recognition algorithms will differentiate the game board, player one's pieces and player two's pieces. The vision system converts the captured image into a game-state from the identified patterns. The computer vision should operate at minimum in well-lit conditions. It will be required to distinguish between pieces of different players by analyzing the black/white gradients of the image. The initial functionality of the pattern recognition system will be to identify dark shapes (circles and squares) on a light background with a dark 3x3 grid superimposed on the light background. Future iterations can increase the complexity of the identified shapes, background, lighting conditions, colour, etc.

3. Game Engine
   The system's "mind" can be divided into two parts. The game engine is the module of the system which decides *where* to move the robotic platform to next. This is done by analyzing the game-

state captured by the vision system and determining the optimal next move based on a pre-programmed algorithm. The game engine will utilize a simple backtracking Min/Max approach. This approach assigns a score to all possible moves based on the board's state and determines which move is best for the opposing human player. It then selects the move that the minimizes the score for the human player's potential moves. There are many examples of such a program available for free and this project, rather than creating one from scratch, will adapt an existing program to take game states as an input rather than individual player moves.

The game engine's output will be a specific position or series of positions that is then passed to the motor controller module.

4. Motor Controller
   The second part of the system's "mind" is the motor controller. It decides *how* to move based on the output of the Game Engine module. The motor controller integrates feedback from digital encoders at each joint in the robotic arm into its control function to determine the position and speed of the robotic platform's joints as it moves in 3D space.

   A transfer function will be written for each joint to control the speed of the arm's motions. The transfer function will enable precise and consistent control of each joint position and prevent the arm from undershooting or overshooting its target position. Such functionality is crucial to ensure the overall system functions as intended.

5. Microcontroller
   If the Game Engine and Motor Controller comprise the system's "mind" then the Microcontroller can be considered the system's "brain. Communication protocols such as Serial Peripheral Interface (SPI) and I^2C are used to facilitate communication between the microcontroller and the peripherals selected for the project.

6. System Integration
   The modules and related hardware in topics 1 through 5 will be integrated in order to achieve full system functionality. This involves utilizing new code, existing code and 3rd party software as well as connecting components from different manufacturers.

Hardware

The project will utilize the following hardware components:

1. Romeo Development Board V1.3

   The Romeo Board V1.3 is an all-in-one development board with an Atmega328p microcontroller at its core. It has 16 digital I/O pins and an additional 6 analog IO pins (which may be configured as digital pins) and is capable of supporting peripherals via SPI and I^2C.

2. MCP23S17 Expander with Serial Interface
   The MCP23S17 provides a 16-bit general purpose I/O expansion via an SPI connection. Multiple MCP23S17 expanders can be connected via the same SPI connection to a master device. This project will utilize two expanders to achieve the necessary number of I/O pins for the peripherals.

If more pins are required, a third expander can be added at low cost and with minimal modification to the project's code.

3. Robotic Arm Edge (OWI 535)
   The base of the platform will be an articulated robotic arm manufactured by OWIKIT. The Robotic Arm is actuated by five DC motors. One motor is located in the "base", one is located in the "gripper", and there is one at each joint: the "shoulder, the "elbow" and the "wrist". The location of each motor is labelled in **Figure 1**.

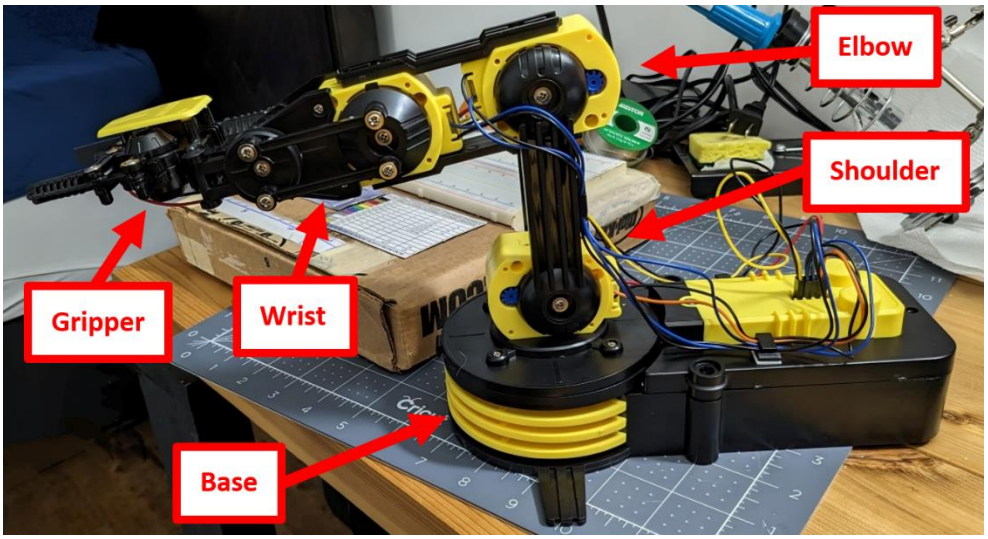

Figure 1: Fully assembled Robotic Arm Edge without any modifications.

4. L298 motor Driver
   The L298 motor driver is used to supply the necessary power to the Robotic Arm's motors. Signals from the microcontroller (two signals per motor) allows the motor driver to operate the motors forwards and backwards by reversing their supplied polarity. Each L298 is capable of operating 2 DC motors independently. As such, this project will require 3 motor drivers in total.

5. KY-040 Rotary Encoder
   The KY-040 rotary encoder reads the position of two switches and returns an angular position to indicate how much its knob has been turned. There are four encoders required for this project: one for each joint and the base. The position of the robotic arm's wrist will be determined by three limit switches due to space limitations.

6. OV7670 Camera Module
   The OV7670 Camera module is a low voltage, 0.3 Megapixel camera module. It is capable of outputting 30 frames per second (though this project will require a much lower framerate). It will be controlled via the Serial Camera Control Bus (SCCB) which is an I^2C interface.

Software

The project will not require any specialized or proprietary software to develop or run. The software will be written in C using free or open-source libraries and compilers. Third party code may be adapted to the project when necessary.

Component Selection

Components were selected for this project by the following criteria in order of decreasing importance: Functionality, Cost and Availability. For instance, more compact encoders likely exist which would improve the form and functionality of the robotic arm but these were either prohibitively expensive or had excessively long lead times. The Ky-040 Rotary Encoders met the requirements of having a physical knob that could be interfaced with the robotic arm's joints (with a little creativity), were available quickly and were inexpensive compared to alternatives.

Assembly

Wiring Diagram

The M.A.T.S. wiring is outlined in **Figure 2**. Some details have been omitted for clarity.

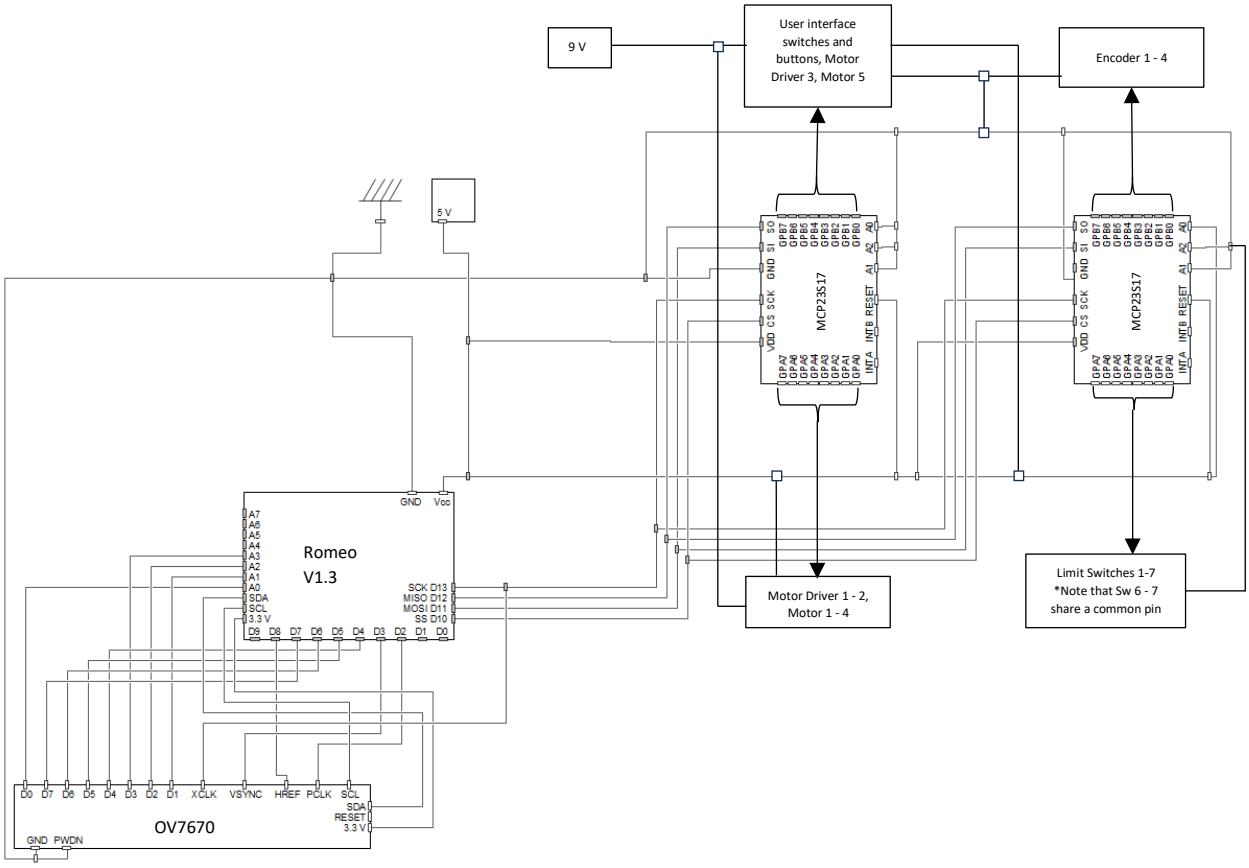

Figure 2: Wiring Diagram of the M.A.T.S.

Robotic Arm Edge

The Robotic Arm Edge will be assembled per the manufacturer's instructions. The encoders and limit switches will be attached to their respective joints via a combination of 3D printed and handcrafted components. Due to space constraints, two gears were 3D printed to interface the rotation of the base motor with the base encoder. The methods used to interface the KY-040 encoders to the Robotic Arm can be seen in **Figure 3**.

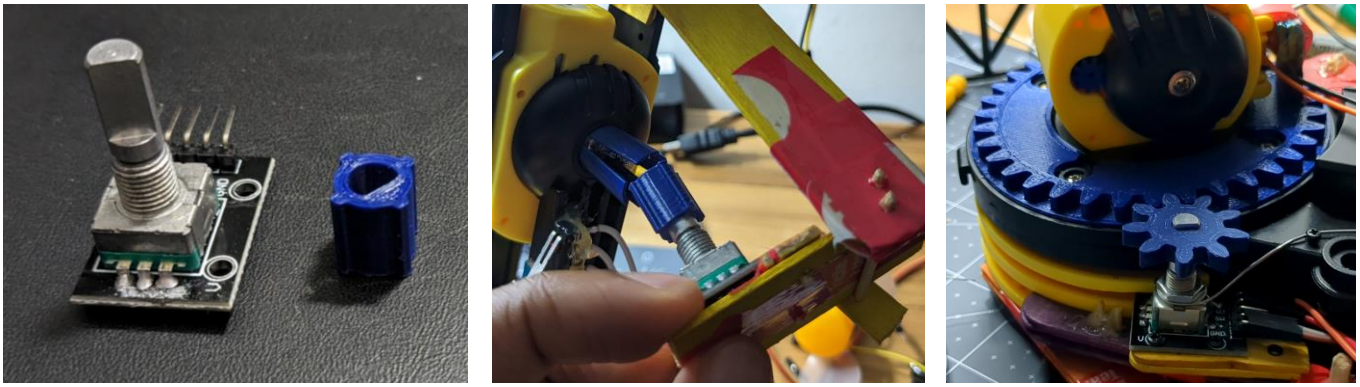

Figure 3: Interfacing the encoders with the Robotic Arm Edge

Module Testing

Unit tests and regression tests will be designed and implemented during the project's development.

Robotic Arm Edge

1. Basic robotic arm functionality including:
    a. Opening and closing the gripper;
    b. Rotating the base 270 degrees;
    c. Rotating the joints a minimum 180 degrees;

Motor Drivers

1. Powering motors from steady state DC supply.
2. Smooth operation of the motors via Pulse Width Modulation (PWM).

Limit Switches

1. Detecting limit switch button push.

Digital Encoders

1. Reading angular position of robotic arm joints.

MCP23S17

1. Reading inputs and writing to outputs of MCP23S17 via the SPI connection.
2. Controlling multiple MCP23S17 expanders via the SPI connection.

Tic-Tac-Toe Engine

1. Checking the current game state against the previous game state for validity.
2. Selecting the next best move based on a game state input.
3. Changing the parameters used to select the next best move based on a user input.

Vision System

1. Reading inputs from OV7670 camera module via the I^2C connection and save the inputs as a Bitmap image.
2. Detect board shape and piece shapes based on brightness values in the captured Bitmap image.
3. Detect when the board has been changed.
4. Create game state based on visual input.

User Interface

1. The switch powers off the system (enters a state where it does not respond to any inputs or outputs) and returns the robotic arm to an off position.
2. The buttons select the game mode and increase or decrease the difficulty setting of the game engine.
3. The game mode and difficulty setting are displayed via LEDs.

Regression Testing

Regression testing is a required step during development of the M.A.T.S. In order to ensure previous bugs do not recur as features are implemented. Regression testing also validates the system's primary functions.

Module Integration

The following regression test cases will validate modules have been successfully implemented to the project.

1. Resetting encoder angular position to zero when limit switch button push is detected.
2. Reading encoder inputs and limit switch inputs via the MCP23S17.
3. Controlling motor drivers via the MCP23S17 and a control system that limits motor speed and runtime based on the target encoder position.
4. Passing a game state from the vision system to the Tic-Tac-Toe Engine.
5. Passing a movement command from the Tic-Tac-Toe Engine to the motor controller to actuate the joints in pre-determined manner based on the game move selected.
6. The startup and shutdown cycles reset the encoder positions and turn the sensors on or off.

Typical Use Cases

Regression testing will be used to validate the system functions as intended.

1. On startup the robotic arm performs a calibration cycle that resets all encoder positions.
2. After the calibration cycle is complete, the user selects a game mode and difficulty setting via the user interface.
3. The vision system detects when the user has made a move and passes the game state to the game engine. The game engine determines the next best move based on the user selected difficulty setting and passes the required position of the robotic arm to the movement to the motor controller.
4. When the game engine detects the game state is a victory for either the human player or for the M.A.T.S., the motor controller runs either a "defeated shake" or "victory nod" cycle and returns to the "waiting" position.

Corner Cases

1. If the game engine detects an invalid game state, the LEDs light up in a specified pattern to indicate to the human player that an invalid move has been made.

Prototype Challenges

The current prototype of the MATS faces the following challenges:

1. Articulation of the robotic arm requires several wired connections. The wires presently used during development are moderately too stiff and occasionally bind on the robotic arm or detach from the component pins during operation. Development of future prototypes will use more flexible connections and/or an alternative method of securement.
2. During development of the prototype, dead pins were discovered on two components: One of the MCP23S17 expanders and one of the encoders. These components require replacement prior to further development.

Conclusion

The current prototype of the MATS is shown in **Figure 4**.

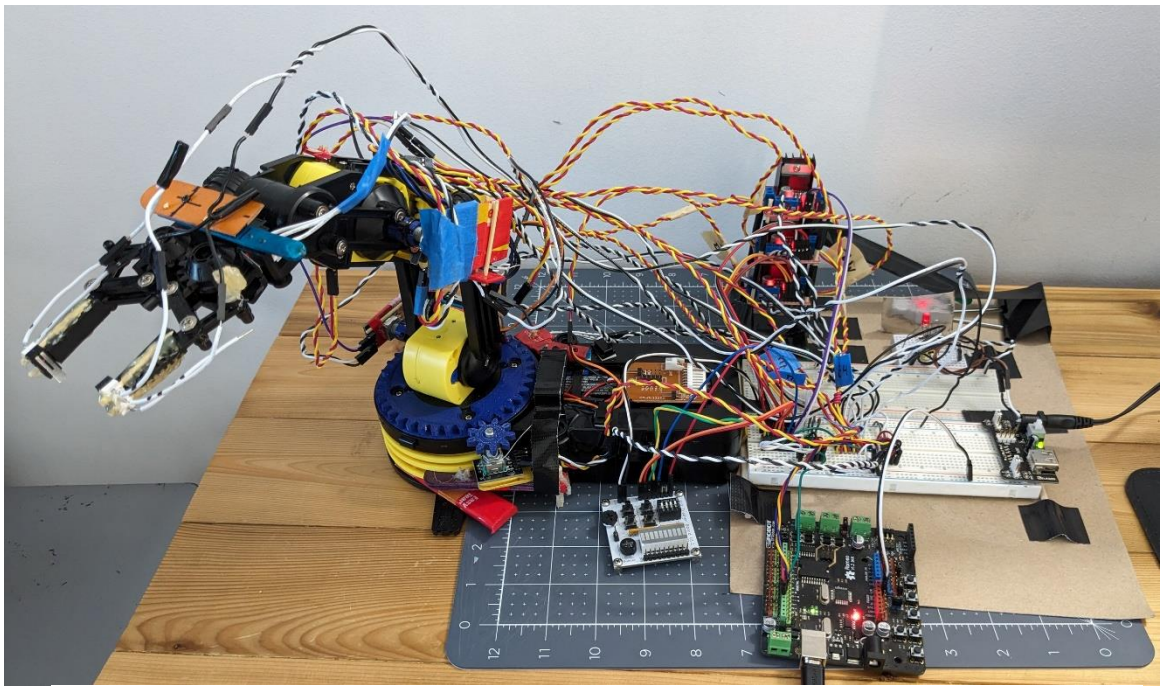

Figure 4: Fully assembled robotic arm prototype

Current Functionality

The current functionality of the M.A.T.S. is as follows:

1. The robotic arm is fully assembled with the requisite peripherals.
2. Two MCP23S17 expanders have been successfully integrated to the Romeo development board to increase the number of GPIO pins available for peripherals.

3. L298 Motor drivers and the associated motors can be successfully powered from a steady state 9V DC power supply. Five motors can be controlled by the user via push buttons and switches.
4. Seven limit switches have been integrated at the robotic arm's various joints.
5. A program has been developed to read the output of the encoders.

Outstanding Work

The following work remains to be completed:

1. All encoders are to be integrated to the motor controller program via the MCP23S17 GPIO expanders. Testing for complete runtime functionality will ensure the encoders consistently read the same angular positions.
2. The robotic arm will move through set positions based on user input and the encoder values.
3. The OV7670 camera module is to be connected to the Romeo development board. The camera module's connection will be successful once it outputs a clear image of sufficient resolution for future processing.
4. Functions will be written to capture the output of the camera module as a bitmap image.
5. Functions will be written to analyze the light/dark values of the captured image for pre-determined patterns.
6. Functions will be written to output the identified patterns as a game state.
7. A game engine will be adapted from existing code to analyze a game state and select the optimal next move.
8. A function will be written to pass the optimal next move to the motor controller function as an input.

Appendix A – Source Code

Main_6.c

```
//Libraries used
#include <avr/io.h>
#include <util/delay.h>
#include <stdlib.h>
#include <stdio.h>
#include <avr/interrupt.h>

//Definitions
#define F_CPU 16000000UL
//SET STICTION HERE:
#define F_STICTION 50 //forward stiction threshold for PWM
#define R_STICTION 50 //reverse stiction threshold for PWM

#define R_ANGLE 0 //reset angle to zero degrees

//Definitions for MCP23S17 peripherals
#define SPI_PORT PORTB
#define SPI_DDR  DDRB
#define SPI_CS   PB2
// MCP23S17 SPI Slave Device
#define SPI_SLAVE_ID    0x40
#define SPI_SLAVE_ADDR_1  0x00      // A2=0,A1=0,A0=0
#define SPI_SLAVE_ADDR_2  0x01      // A2=0,A1=0,A0=1
#define SPI_SLAVE_WRITE 0x00
#define SPI_SLAVE_READ  0x01
// MCP23S17 Registers Definition for BANK=0
#define IODIRA 0x00
#define IODIRB 0x01
#define IOCONA 0x0A
#define GPPUA  0x0C
#define GPPUB  0x0D
#define GPIOA  0x12
#define GPIOB  0x13
#define GPB0   0x00
#define GPB1   0x01
//Function declarations

//Functions for MCP23S17
void SPI_Write_1(unsigned char addr,unsigned char data);
void SPI_Write_2(unsigned char addr,unsigned char data);
unsigned char SPI_Read_1(unsigned char addr);
unsigned char SPI_Read_2(unsigned char addr);


//Functions for Motor Control via PWM
void enablePowerControl(void); //Initialize counter for PWM output
void disablePowerControl(void); //Disable PWM counter

uint8_t M1(int16_t Mot_Power); //Change duty cycle M1
uint8_t M2(int16_t Mot_Power); //Change duty cycle M2
uint8_t M3(int16_t Mot_Power); //Change duty cycle M1
uint8_t M4(int16_t Mot_Power); //Change duty cycle M2
uint8_t M5(int16_t Mot_Power); //Change duty cycle M1
uint8_t lin_int(int16_t x); //linear interpolation

//Functions for debugging with USART
void transmitString(char *x);
void transmitByte(char data);
void printDec(float num);
void initUART(uint16_t baud);

void initTimer2(void);
uint8_t get_grayCode_1(void);
//uint8_t get_grayCode_2(void);
//uint8_t get_grayCode_3(void);
```

```
 66  //uint8_t get_grayCode_4(void);
 67  int get_angle(int16_t pulseCount);
 68
 69  //GLOBAL VARIABLES
 70  //uint8_t pinD;
 71  uint16_t dutyCycle;
 72  uint16_t baud = 9600; //for intefacing with UART
 73  char data; //for intefacing with UART
 74  //For encoder control
 75  uint8_t channelA, channelB; //, channelC, channelD, channelE, channelF, channelG, channelH;
 76  volatile uint8_t phase, phase_old, direction;
 77  volatile int_fast16_t pulseCount_1; //, pulseCount_2, pulseCount_3, pulseCount_4 = 0;
 78
 79  uint8_t inp_1, inp_2, inp_3;
 80
 81  int main(void)
 82  {
 83  // Motor control variables
 84      int angle_1; //, angle_2, angle_3, angle_4;
 85      uint8_t pwm_1, pwm_2, pwm_3, pwm_4, pwm_5;
 86      int16_t M1_Power = 60;
 87      int signal;
 88
 89  //    unsigned char inp_1,inp_2;
 90
 91    ADCSRA = (1<<ADEN) | (1<<ADPS2) | (1<<ADPS1);
 92    // Free running Mode
 93    ADCSRB = 0x00;
 94    // Disable digital input on ADC0 (PC0)
 95    DIDR0 = 0x01;
 96    ADMUX=0x00;                  // Select Channel 0 (PC0)
 97
 98    // Initial the AVR ATMega328P SPI Peripheral
 99    // Set MOSI (PB3),SCK (PB5) and PB2 (SS) as output, others as input
100    SPI_DDR = (1<<PB3)|(1<<PB5)|(1<<PB2);
101    // CS pin is not active
102    SPI_PORT |= (1<<SPI_CS);
103
104    // Enable SPI, Master, set clock rate fck/64
105    SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR1);
106
107    // Initial the MCP23S17 SPI I/O Expander
108      SPI_Write_1(IOCONA,0x28);     // I/O Control Register: BANK=0, SEQOP=1, HAEN=1 (Enable
Addressing)
109    SPI_Write_1(IODIRA,0x00);   // GPIOA 0-7 As Output
110    SPI_Write_1(IODIRB,0x3F);   // GPIOB 0-5 As Input
111    SPI_Write_1(GPPUB,0xFF);    // Enable Pull-up Resistor on GPIOB
112    SPI_Write_1(GPIOA,0x00);    // Reset Output on GPIOA
113
114      SPI_Write_2(IOCONA,0x28);     // I/O Control Register: BANK=0, SEQOP=1, HAEN=1 (Enable
Addressing)
115    SPI_Write_2(IODIRA,0xFF);   // GPIOA As 0-7 Input
116    SPI_Write_2(IODIRB,0xFF);   // GPIOB As 0-7 Input
117    SPI_Write_2(GPPUA,0xFF);    // Enable Pull-up Resistor on GPIOB
118    SPI_Write_2(GPPUB,0xFF);    // Enable Pull-up Resistor on GPIOB
119  //  SPI_Write_2(GPIOA,0x00);    // Reset Output on GPIOA
120
121    initUART(baud);
122    initTimer2();
123
124            transmitString("\rSignal = \n");
125
126      while(1){
127
128          inp_1=SPI_Read_1(GPIOB); // Button 1-4 and Sw 1 and Sw 2
129          inp_2=SPI_Read_2(GPIOB); // Encoder Inputs
130          inp_3=SPI_Read_2(GPIOA); // Limit Switches. Note that pin 7 is dead
131
132  //Motor 1 and Motor 2
133      if((!((inp_1 & (1 << 4)) == 0)) & (!((inp_1 & (1 << 5)) == 0))){
```

```
134          if(signal==0) {
135              if((inp_1 & (1 << 0)) == 0) {
136                  signal = 1;
137                  enablePowerControl();
138                  pwm_1 = M1(100);
139                  angle_1 = get_angle(pulseCount_1);
140              }
141              if((inp_1 & (1 << 1)) == 0) {
142                  signal = 2;
143                  enablePowerControl();
144                  pwm_1 = M1(-M1_Power);
145                  angle_1 = get_angle(pulseCount_1);
146              }
147              if((inp_1 & (1 << 2)) == 0) {
148                  signal = 3;
149                  enablePowerControl();
150                  pwm_2 = M2(M1_Power);
151                  angle_2 = get_angle(pulseCount_2);
152              }
153              if((inp_1 & (1 << 3)) == 0) {
154                  signal = 4;
155                  enablePowerControl();
156                  pwm_2 = M2(-70);
157                  angle_2 = get_angle(pulseCount_2);
158              }
159          }
160
161          else {
162              if(!((inp_1 & (1 << 0)) == 0)) {
163                  signal = 0;
164                  M1_Power = 1;
165                  disablePowerControl();
166              }
167              if(!((inp_1 & (1 << 1)) == 0)) {
168                  signal = 0;
169                  M1_Power = 1;
170                  disablePowerControl();
171              }
172              if(!((inp_1 & (1 << 2)) == 0)) {
173                  signal = 0;
174                  M1_Power = 1;
175                  disablePowerControl();
176              }
177              if(!((inp_1 & (1 << 3)) == 0)) {
178                  signal = 0;
179                  M1_Power = 1;
180                  disablePowerControl();
181              }
182          }
183      }
184  // Motor 3 and Motor 4
185      if(((inp_1 & (1 << 4)) == 0) & (!((inp_1 & (1 << 5)) == 0))){
186          if(signal==0) {
187              if((inp_1 & (1 << 0)) == 0) {
188                  signal = 5;
189                  enablePowerControl();
190                  pwm_3 = M3(100);
191                  angle_3 = get_angle(pulseCount_3);
192              }
193              if((inp_1 & (1 << 1)) == 0) {
194                  signal = 6;
195                  enablePowerControl();
196                  pwm_3 = M3(-M1_Power);
197                  angle_3 = get_angle(pulseCount_3);
198              }
199              if((inp_1 & (1 << 2)) == 0) {
200                  signal = 7;
201                  enablePowerControl();
202                  pwm_4 = M4(M1_Power);
203                  angle_4 = get_angle(pulseCount_4);
```

```
204             }
205             if((inp_1 & (1 << 3)) == 0) {
206                 signal = 8;
207                 enablePowerControl();
208                 pwm_4 = M4(-70);
209                 angle_4 = get_angle(pulseCount_4);
210             }
211         }
212 
213         else {
214             if(!((inp_1 & (1 << 0)) == 0)) {
215                 signal = 0;
216                 M1_Power = 1;
217                 disablePowerControl();
218             }
219             if(!((inp_1 & (1 << 1)) == 0)) {
220                 signal = 0;
221                 M1_Power = 1;
222                 disablePowerControl();
223             }
224             if(!((inp_1 & (1 << 2)) == 0)) {
225                 signal = 0;
226                 M1_Power = 1;
227                 disablePowerControl();
228             }
229             if(!((inp_1 & (1 << 3)) == 0)) {
230                 signal = 0;
231                 M1_Power = 1;
232                 disablePowerControl();
233             }
234         }
235     }
236 
237 // Motor 5
238     if(((inp_1 & (1 << 4)) == 0) & ((inp_1 & (1 << 5)) == 0)){
239         if(signal==0) {
240             if((inp_1 & (1 << 0)) == 0) {
241                 signal = 9;
242                 enablePowerControl();
243                 pwm_5 = M5(100);
244             }
245             if((inp_1 & (1 << 1)) == 0) {
246                 signal = 10;
247                 enablePowerControl();
248                 pwm_5 = M5(-M1_Power);
249             }
250         }
251 
252         else {
253             if(!((inp_1 & (1 << 0)) == 0)) {
254                 signal = 0;
255                 M1_Power = 1;
256                 disablePowerControl();
257             }
258             if(!((inp_1 & (1 << 1)) == 0)) {
259                 signal = 0;
260                 M1_Power = 1;
261                 disablePowerControl();
262             }
263         }
264     }
265 
266         if((inp_3 & (1 << 0)) == 0){
267             pulseCount_1 = 0;
268         }
269         if((inp_3 & (1 << 1)) == 0){
270             pulseCount_1 = 0;
271         }
272 
273         transmitString("\r");
```

```
274         printDec(angle_1);
275         transmitString(" Degrees");
276 
277 //        transmitString("\r");
278 //        printDec(signal);
279 //        transmitString(" <- Motor control test");
280 
281 //        transmitString("\r");
282 //        printDec(inp_3);
283 //        transmitString(" <- Limit Switch Test");
284   }
285 
286     return 0;
287 }
288 
289 void enablePowerControl(void) { //Counter 0 PWM mode
290      TCCR0A = (1 << COM0A1)|(1 << COM0B1)|(1 << WGM01)|(1 << WGM00); //Set Fast PWM mode 3,
normal operation on digital pins
291     TCCR0B = (1 << CS01)|(1 << CS00); //Set divider to 64
292 }
293 
294 void disablePowerControl(void) { //Counter 0 normal mode
295     TCCR0A &= ~(1 << COM0A1);
296     TCCR0A &= ~(1 << COM0B1);
297     TCCR0A &= ~(1 << WGM01);
298     TCCR0A &= ~(1 << WGM00);
299     TCCR0B &= ~(1 << CS01);
300     TCCR0B &= ~(1 << CS00);
301     SPI_Write_1(GPIOA,0b00000000);
302     SPI_Write_1(GPIOB,0b00000000);
303 }
304 
305 uint8_t M1(int16_t Mot_Power){
306 
307     int16_t setting;
308 
309     if(Mot_Power > 0){ // control the motor direction
310         SPI_Write_1(GPIOA,0b00000001);
311     }else if(Mot_Power < 0){
312         SPI_Write_1(GPIOA,0b00000010);
313     }
314 
315     OCR0A = Mot_Power;
316 
317     return setting;
318 
319 }
320 
321 uint8_t M2(int16_t Mot_Power){
322 
323     int16_t setting;
324 
325     if(Mot_Power > 0){ // control the motor direction
326         SPI_Write_1(GPIOA,0b00000100);
327     }else if(Mot_Power < 0){
328         SPI_Write_1(GPIOA,0b00001000);
329     }
330 
331     OCR0A = Mot_Power;
332 
333     return setting;
334 
335 }
336 
337 uint8_t M3(int16_t Mot_Power){
338 
339     int16_t setting;
340 
341     if(Mot_Power > 0){ // control the motor direction
342         SPI_Write_1(GPIOA,0b00010000);
```

```
343      }else if(Mot_Power < 0){
344          SPI_Write_1(GPIOA,0b00100000);
345      }
346
347      OCR0A = Mot_Power;
348
349      return setting;
350
351  }
352
353  uint8_t M4(int16_t Mot_Power){
354
355      int16_t setting;
356
357      if(Mot_Power > 0){ // control the motor direction
358          SPI_Write_1(GPIOA,0b01000000);
359      }else if(Mot_Power < 0){
360          SPI_Write_1(GPIOA,0b10000000);
361      }
362
363      OCR0A = Mot_Power;
364
365      return setting;
366
367  }
368
369  uint8_t M5(int16_t Mot_Power){
370
371      int16_t setting;
372
373      if(Mot_Power > 0){ // control the motor direction
374          SPI_Write_1(GPIOB,0b01000000);
375      }else if(Mot_Power < 0){
376          SPI_Write_1(GPIOB,0b10000000);
377      }
378
379      OCR0A = Mot_Power;
380
381      return setting;
382
383  }
384
385
386  int get_angle(int16_t pulseCount){
387
388      float x;
389
390  //    x = pulseCount;
391      x = pulseCount*1.35;
392
393      return x;
394
395  }
396
397  /*
398  Encoder Timer Functions
399  */
400  void initTimer2(void){
401
402      TCCR2A |= (1<<COM2A1)|(1<<COM2A0);
403      TCCR2A |= (1<<WGM21);
404  //    TCCR2B |= (1<<CS22)|(1<<CS21)|(1<<CS20);
405      TCCR2B |= (1<<CS21);
406      TIMSK2 |= (1<<OCIE2A);
407      TIFR2 |= (1<<OCF2A);
408      OCR2A = 200;
409
410      sei();
411  }
412
```

```
413  /*
414  Functions related to interrupts for encoders
415  */
416
417  ISR(TIMER2_COMPA_vect){
418
419      phase = get_grayCode_1();
420
421      if(phase != phase_old){
422          if((phase == 0 && phase_old == 3)
423              ||(phase == 1 && phase_old ==0)
424              ||(phase == 2 && phase_old ==1)
425              ||(phase == 3 && phase_old ==2)
426             ){
427              pulseCount_1 += 1;
428              direction = 1;
429              }
430          else if((phase == 0 && phase_old == 1)
431              ||(phase == 1 && phase_old ==2)
432              ||(phase == 2 && phase_old ==3)
433              ||(phase == 3 && phase_old ==0)
434             ){
435              pulseCount_1 -= 1;
436              direction = 0;
437             }
438      phase_old = phase;
439      }
440  }
441
442  uint8_t get_grayCode_1(void){
443
444      uint8_t x;
445
446      if((inp_2 & (1 << 0)) == 0){
447          channelA = 0;
448      }
449      else {
450          channelA = 1;
451      }
452
453      if((inp_2 & (1 << 1)) == 0){
454          channelB = 0;
455      }
456      else {
457          channelB = 1;
458      }
459
460      if((channelA == 0)&&(channelB == 0)){
461          x = 0;
462      } else if((channelA == 0)&&(channelB == 1)){
463          x = 1;
464      } else if((channelA == 1)&&(channelB == 1)){
465          x = 2;
466      } else if((channelA == 1)&&(channelB == 0)){
467          x = 3;
468      }
469
470      return x;
471  }
472
473  //MCP23S17 Functions
474
475  void SPI_Write_1(unsigned char addr, unsigned char data)
476  {
477    // Activate the CS pin
478    SPI_PORT &= ~(1<<SPI_CS);
479    // Start MCP23S17 OpCode transmission
480    SPDR = SPI_SLAVE_ID | ((SPI_SLAVE_ADDR_1 << 1) & 0x0E)| SPI_SLAVE_WRITE;
481    // Wait for transmission complete
482    while(!(SPSR & (1<<SPIF)));
```

```
483    // Start MCP23S17 Register Address transmission
484    SPDR = addr;
485    // Wait for transmission complete
486    while(!(SPSR & (1<<SPIF)));
487
488    // Start Data transmission
489    SPDR = data;
490    // Wait for transmission complete
491    while(!(SPSR & (1<<SPIF)));
492    // CS pin is not active
493    SPI_PORT |= (1<<SPI_CS);
494  }
495
496  void SPI_Write_2(unsigned char addr,unsigned char data)
497  {
498    // Activate the CS pin
499    SPI_PORT &= ~(1<<SPI_CS);
500    // Start MCP23S17 OpCode transmission
501    SPDR = SPI_SLAVE_ID | ((SPI_SLAVE_ADDR_2 << 1) & 0x0E)| SPI_SLAVE_WRITE;
502    // Wait for transmission complete
503    while(!(SPSR & (1<<SPIF)));
504    // Start MCP23S17 Register Address transmission
505    SPDR = addr;
506    // Wait for transmission complete
507    while(!(SPSR & (1<<SPIF)));
508
509    // Start Data transmission
510    SPDR = data;
511    // Wait for transmission complete
512    while(!(SPSR & (1<<SPIF)));
513    // CS pin is not active
514    SPI_PORT |= (1<<SPI_CS);
515  }
516
517  unsigned char SPI_Read_1(unsigned char addr)
518  {
519    // Activate the CS pin
520    SPI_PORT &= ~(1<<SPI_CS);
521    // Start MCP23S17 OpCode transmission
522    SPDR = SPI_SLAVE_ID | ((SPI_SLAVE_ADDR_1 << 1) & 0x0E)| SPI_SLAVE_READ;
523    // Wait for transmission complete
524    while(!(SPSR & (1<<SPIF)));
525
526    // Start MCP23S17 Address transmission
527    SPDR = addr;
528    // Wait for transmission complete
529    while(!(SPSR & (1<<SPIF)));
530
531    // Send Dummy transmission for reading the data
532    SPDR = 0x00;
533    // Wait for transmission complete
534    while(!(SPSR & (1<<SPIF)));
535
536    // CS pin is not active
537    SPI_PORT |= (1<<SPI_CS);
538    return(SPDR);
539  }
540
541  unsigned char SPI_Read_2(unsigned char addr)
542  {
543    // Activate the CS pin
544    SPI_PORT &= ~(1<<SPI_CS);
545    // Start MCP23S17 OpCode transmission
546    SPDR = SPI_SLAVE_ID | ((SPI_SLAVE_ADDR_2 << 1) & 0x0E)| SPI_SLAVE_READ;
547    // Wait for transmission complete
548    while(!(SPSR & (1<<SPIF)));
549
550    // Start MCP23S17 Address transmission
551    SPDR = addr;
552    // Wait for transmission complete
```

```
  while(!(SPSR & (1<<SPIF)));

  // Send Dummy transmission for reading the data
  SPDR = 0x00;
  // Wait for transmission complete
  while(!(SPSR & (1<<SPIF)));

  // CS pin is not active
  SPI_PORT |= (1<<SPI_CS);
  return(SPDR);
}

//-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-

/*
Functions for debugging via UART below
*/
//Convert number to ASCII Decimal representation and write to UART
void printDec(float num) {

    char stringDec[40];
    sprintf(stringDec,"%f",num);

    transmitString(stringDec);
}
//Write to UART one byte at a time
void transmitString (char *x) {

    int i = 0;

    while(i>=0){
        data = x[i];

        if (data == 0){
            return;
        }
        transmitByte(data);
        i++;
    }
}
//Write one byte to UART
void transmitByte (char data) {
    while ( !(UCSR0A & (1 << UDRE0)) ); // Wait for empty transmit buffer
    UDR0 = data; // Start transmission by writing to UDR0 register
}

void initUART(uint16_t baud) {
    unsigned int ubrr = F_CPU/16/(baud)-1;
    //Normal speed mode UBRR formula
    UBRR0H = (unsigned char) (ubrr >> 8);
    // shift MSB and store in UBRR0H
    UBRR0L = (unsigned char) ubrr;
    // store LSB in UBRR0L
    UCSR0B = (1 << RXEN0) | (1 << TXEN0);
    // Enable transmitter/receiver
    UCSR0C = (1 <<UCSZ00) | (1 <<UCSZ01);
    //8-Bit Characters, 0 Stop bits, No parity
    _delay_ms(10);
}
```

Appendix B – References

TicTacToe.c

- https://www.geeksforgeeks.org/tic-tac-toe-game-in-c/

Capturing Camera Images with CircuitPython

- https://learn.adafruit.com/capturing-camera-images-with-circuitpython

Using Serial Peripheral Interface (SPI) Master and Slave with Atmel AVR Microcontroller

- https://www.ermicro.com/blog/?p=1050

Image Processing in C

- Dwayne Phillips, Image Processing in C, Second Edition, R & D Publications, 1994

Hacking the OV7670 camera module (SCCB cheat sheet inside)

- https://embeddedprogrammer.blogspot.com/2012/07/hacking-ov7670-camera-module-sccb-cheat.html

Make Your Own Camera

- https://www.instructables.com/Make-Your-Own-Camera/

How to Use OV7670 Camera Module with Arduino

- https://circuitdigest.com/microcontroller-projects/how-to-use-ov7670-camera-module-with-arduino