Programming of hardware and robot technology for play and teaching purposes - Assignment 1

Anders larsen (larse19) and Theis Tengs (thten19)

Team 3

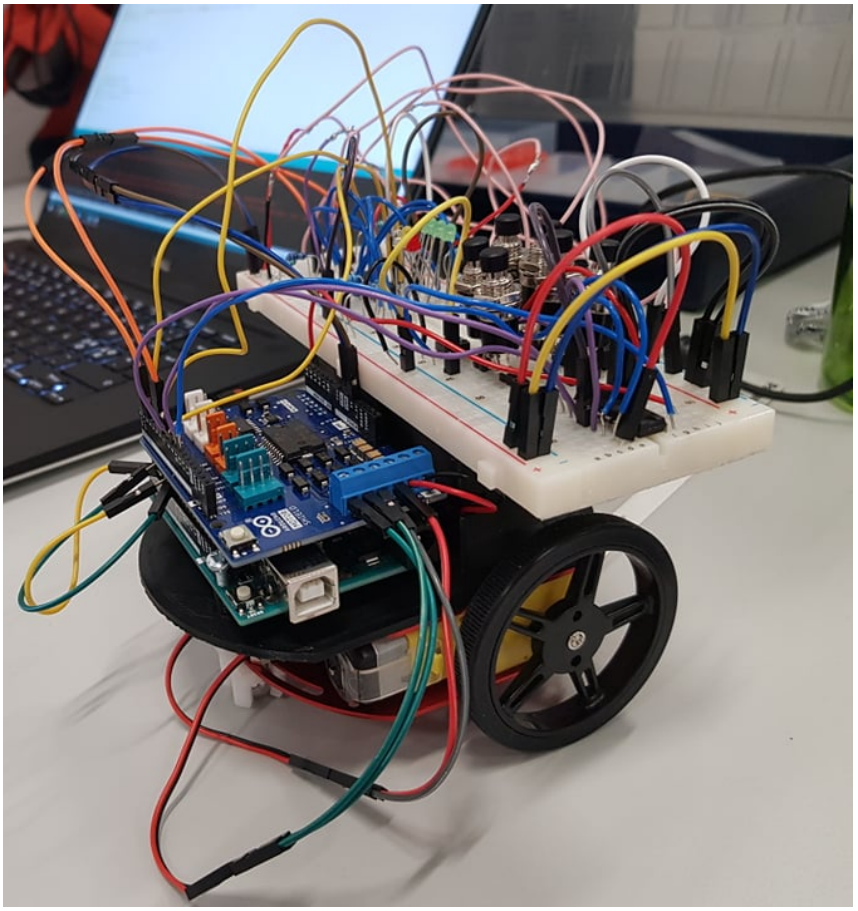


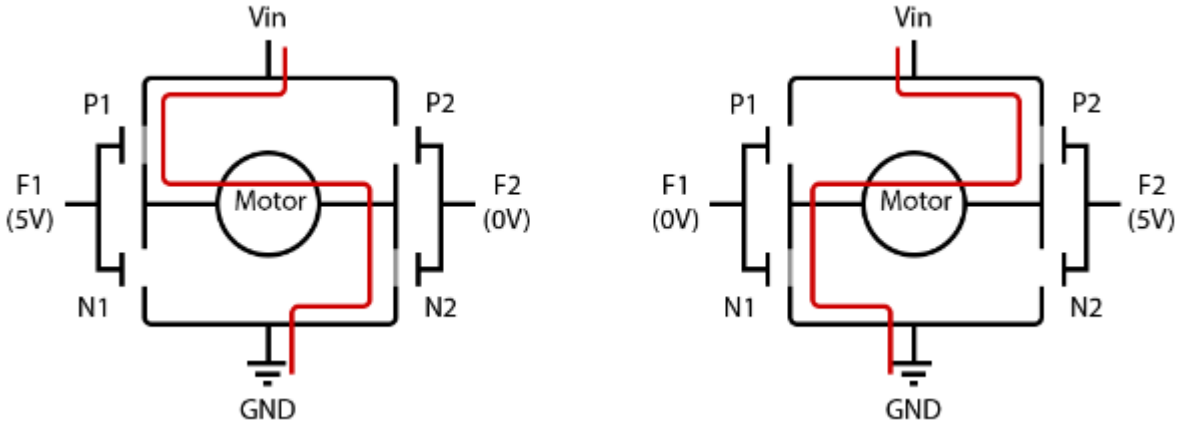
Table of contents

Introduction

This document describes the build process of developing a controllable robot for children to play with and learn. The robot is a small vehicle that can be programmed by giving it a series of commands to execute in a given order. The robot should be able to execute commands to move forward, backward, turn left and right, by pressing different buttons on the controller. These commands should be executed in the same order as they were pressed. The controller should also give feedback to the user about which command is currently being executed, as well as which state the robot is currently in; input, active or pause. It should also be possible to save a programmed route to persistent memory, so the user can rerun the last saved route when desired.
A circuit diagram describing the robot and it's electronics can be found in appendix 1.

Driving the motors

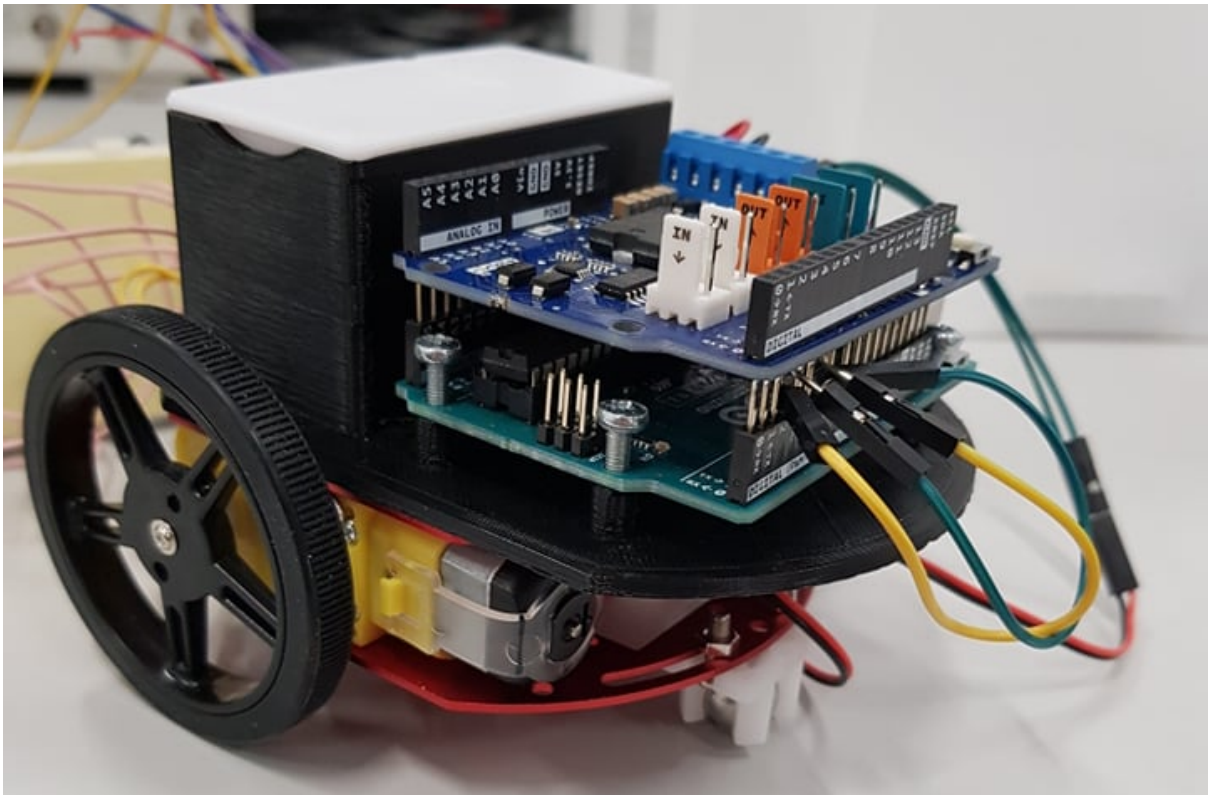
In order for the robot to drive, it is fitted with two dc motors with wheels attached to them. The arduino itself however, can't provide enough power for the motors to turn, so an external power supply is needed. To control the drive of the motors, a circuit called an H-bridge is needed. This circuit uses four transistors; two PNP and two NPN transistors. This circuit makes it possible to control which way the power flows through the motor, which controls the direction of the motor.



This diagram shows how an H-Bridge controls the flow of the power. If we power transistors P1 and N2, the power flows one way through the motor, and if we power P2 and N1 it flows the other way.

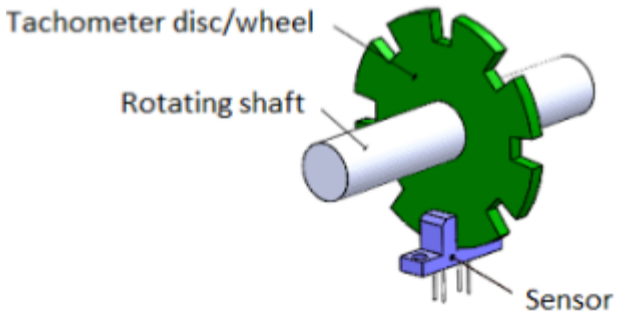
Arduino motor shield

While it would be possible to build the robot using a H-Bridge for each motor, a more simple solution exists; an arduino motor shield. This is a component that fits on top of an Arduino Uno and has two built-in H-Bridges. This means that by attaching the shield to the top of the arduino, attaching the motors as well as an external power supply, 6V in our case, we can drive the motors through the arduino.



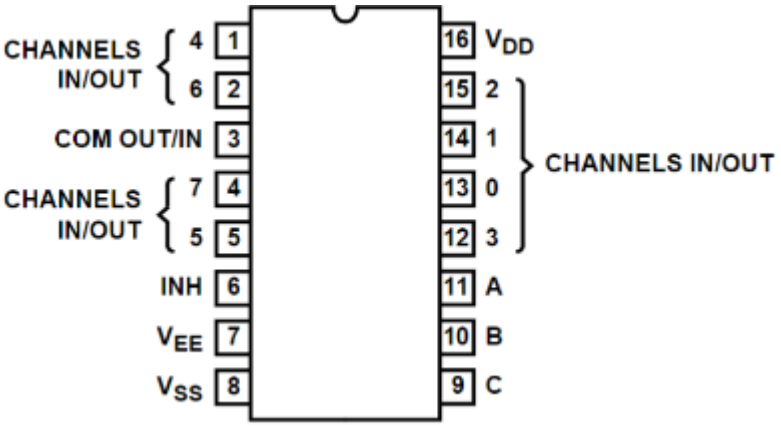
Tachometer

DC motors are inherently unpredictable when it comes to speed. Since the speed of the motor is based on the amount of power supplied along with resistance, we can't be sure that both motors will be equally fast when supplied with the same amount of power, since one motor might have more resistance than the other.
In order to circumvent this, we can attach a tachometer to the motors. A tachometer is an instrument that can measure the rotational speed of a motor or shaft. On this robot the tachometer consists of a 3D printed disc with holes in it, that is attached to the motor. When the motor turns, the disc turns with it. The disc then turns between a photointerrupter. This is a component that consists of an infrared emitter and phototransistor. When the space between the two isn't obstructed, the phototransistor registers the infrared waves that then turns it on, allowing power to flow through. So when the disc spins around, we get a signal every time one of the holes passes through, which allows us to calculate the rotational speed of the motor, and adjust the power accordingly.



Expanding the I/O with a multiplexer

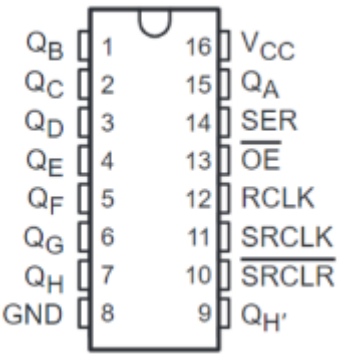
An arduino uno only has 13 I/O pins of which 2 are used for the inner workings of the arduino and at least 4 are used for the motor shield (it normally uses two more for braking, but those have been disabled). That leaves us with only 7 pins to work with for the rest of the electronics. Since we need 7 input buttons; one for forward, backwards, left, right, play, pause and stop, as well as an LED indicator for each of those inputs, we need more I/O. One way to increase the number of I/O pins is with a multiplexer. A multiplexer is a component that has multiple channels that connect to a given pin, that can be both read and written to. The multiplexer we use has 8 channels. The multiplexer chooses which channel to connect to the I/O pin, based on a 3 bit input, given to 3 of its pins; A, B and C. So if you write high to pin A and B, and low to pin C (011 in binary), it connects to channel 3. The multiplexer can only read on one channel at a time, but since an arduino can read and write many times a second, we can cycle through the channels quickly, and it will “simulate” the possibility to read from all of them at a time.



We can use this component to connect our 7 buttons to the arduino. If we connect the COM IN/OUT pin to the arduino and make use of the built in pullup resistor, we can read from up to 8 buttons and only use 4 pins on the arduino.

Providing feedback with a shift register

Another way of expanding the output of the arduino is with a shift register. This component can store data and write it to its output pins. As opposed to a multiplexer, a shift register can, however, only write data, not read it, so we can not attach buttons to this component, but we can use it to provide feedback to the user in the form of LEDs.

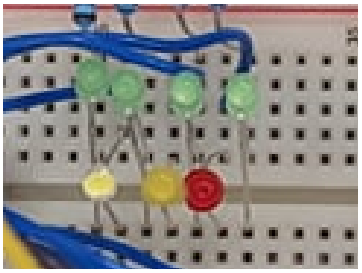


A shift register works by sending bits of data to it, and then store internally. The component is controlled by a clock (SRCLK). When the clock goes from low to high, the component reads the value of the data pin(SER) and stores it. When the latch pin(RCLK) goes from low to high, it tells the component that all the data has been sent, and it writes the stored data to the output pins (QA - QH).

The Shift register in our robot can store a byte of data, or 8 bits. This means that we have 8 output pins that can all be controlled at the same time, giving us $2^8 = 256$ different outputs, using only 3 pins on the arduino. Since we only need 7 LEDs, as mentioned earlier, we can attach one to each output pin. So if we want to, for example, make the forward LED indicator light up, we can write 00000010 to the shift register, and it will output on pin QB, which is connected to that LED, and 00001000 if we want to make the backwards LED indicator light, and so on.
If we want to make more than one LED light up, we can use the bitwise OR operator to combine more than one byte together. For example, if we want both the “active state” LED to light up, along with the forward LED, in order to show that the robot is currently performing the “forward” command, we combine their representative bytes like so:

```
00000010
00010000 OR
--------
00010010
```

Now the register will send a signal to both the active- and the forward LED lighting them up.



The LED indicators. The top LEDs represent the four commands, and the bottom represents the 3 states

The software

In order to actually control the robot's behavior, we need to program it.

Finite state machine

The core of the software for the robot is what is called a finite state machine. A finite state machine describes a series of states, along with the conditions for transitioning between them.
The state machine for the robot consists of three states: Input-, active- and pause mode.

- Input mode
    - When in this mode, the robot waits for the commands from the user, which are added to a list.
    - When the user presses “play” go to active mode
- Active mode
    - Executes the commands one after the other.
    - When all commands are executed, go to input mode
    - When the user presses pause, go to pause mode
    - When the user presses stop, clear the list of commands and go to input mode

- Pause mode
    - Waits for user input
    - When the user presses play, go back to active mode and continue executing the commands
    - When stop is pressed, clear the list of commands and go to input mode.

This state machine is implemented using a series of if statements in the loop() function. If the variable; state == “input” we are in input state, and so on.
When in input mode, we listen for user input on the buttons connected to the multiplexer. If any of the programming buttons are pressed, we append that command to a LinkedList. We also provide feedback to the user, by flashing the corresponding LED whenever a command is inputted. If the user then presses the play button, the state variable is set to “active” and the next if statement in the for loop is now true, and we have transitioned to active state.
In the active state, we go through the LinkedList and remove the first element, and execute that command. Once that command is done, the next command is now the first, and we execute that and so on. Whilst a command is being executed, we listen for presses on the stop and pause buttons, in case the user wants to switch state.

Variable motor speed

As mentioned earlier, we make use of tachometers on each wheel to adjust the speed of the rotation dynamically. In order to do this, we use interrupts. An interrupt is a function that can fire regardless of which code is currently being run. That means that the function will always be called when it is told to, and won’t wait until some other code has been executed. It can be activated in different ways, but we activate it whenever the photointerrupter sends a signal to the arduino, ie. whenever a hole on the disc of the tachometer passes through.
This wheel has 20 holes in it, so when we have received 20 pulses from the tachometer, the wheel has made a complete revolution. The speed of the motor is then set to 1 round per second, so there has to be 1/20 seconds between each pulse. So in the interrupt function, we measure the time between each pulse using the c++ millis() function, and if there is more than 1/20 of a second between each pulse, we increase the speed, and if there is less, we decrease the speed. This ensures that we always drive at the desired speed.

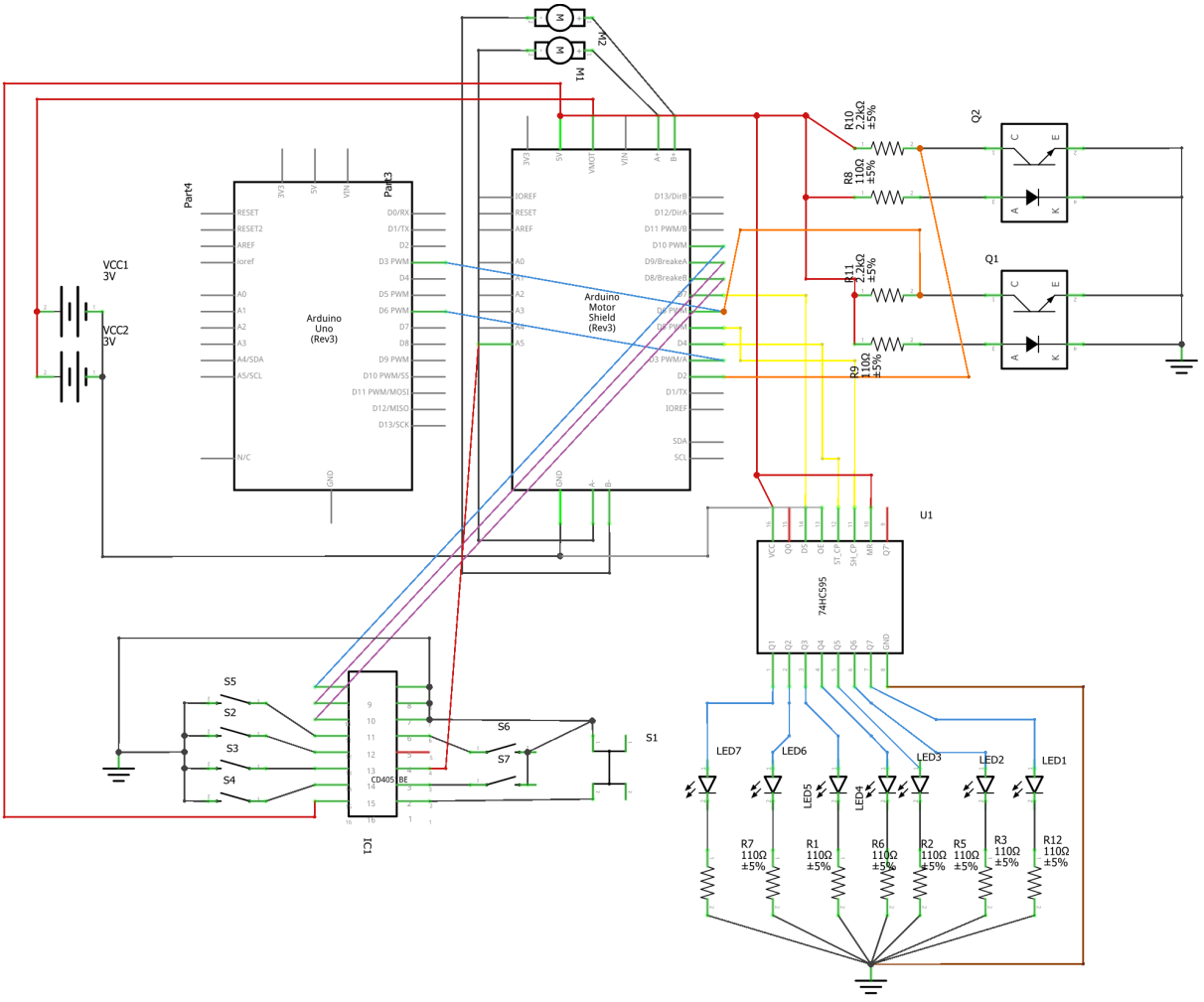
Persistent memory

In order to be able to save a list of commands between uses, we can’t use the LinkedList, because it is only saved in memory, and will therefore be reset when the arduino is turned off. The arduino has 1024 bytes of onboard persistent memory called EEPROM, which we can utilize to save the list of commands. Whenever the user presses the play button, the current list of commands is saved to the EEPROM. In order to save to the EEPROM you have to manually specify an address, where you can store 1 byte of data. Our list of commands consists of integers which describe each command, so we can only write one command per address. So we need to store each command on it’s own address, to do so we start at a given address: 10, and store the first command. We then move onto the next address and store the next there and so on. This is done in a for loop for the size of the array. When we then need to read the data back in again, we need to know how many

addresses we need to read from, so we store the length of the list on address 9, which we can then use to know the length of the list of commands the next time we start the robot.
The user can play the last list of commands stored, by simply pressing play without entering any commands.
The entire code for the project can be found in appendix 2.

Appendix

1. Circuit diagram



2. Code

```
#include <LinkedList.h>
#include <EEPROM.h>

#define latchPin 4
#define dataPin 7
#define clockPin 5
```

```
#define forwardLED B00000010
#define reverseLED B00001000
#define leftLED B00100000
#define rightLED B10000000
#define inputLED B00000100
#define activeLED B00010000
#define pauseLED B01000000

const byte interruptPinA = 3;
const byte interruptPinB = 2;

#define dirA 12
#define spA 6
#define dirB 13
#define spB 11

#define A 8
#define B 9
#define C 10

#define multiplexerIO A5

volatile int numA = 0;
volatile int numB = 0;
volatile int speedA = 150;
volatile int speedB = 150;
int interval = 1000/20; // Interval between each tick of the tachometer (1000 millis / 20 slits)
volatile int oldMilisA = 0;
volatile int oldMilisB = 0;

int listAddr = 10;
int listSizeAddr = 9;

bool readFromMulti(int chnl){
  int a = bitRead(chnl,0); //Take first bit from binary value of i channel.
  int b = bitRead(chnl,1); //Take second bit from binary value of i channel.
  int c = bitRead(chnl,2); //Take third bit from value of i channel.

  digitalWrite(A, a);
  digitalWrite(B, b);
  digitalWrite(C, c);
  //Serial.println(digitalRead(multiplexerIO));
  return digitalRead(multiplexerIO);
}

class Button {
    private:
```

```
        bool _state;
        uint8_t _pin;

    public:
        Button(uint8_t pin) : _pin(pin) {}

        void begin() {
            _state = readFromMulti(_pin);
        }

        bool isReleased() {
            bool v = readFromMulti(_pin);
            if (v != _state) {
                _state = v;
                if (_state) {
                    return true;
                }
            }
            return false;
        }
};

Button forwardButton(0);
Button reverseButton(1);
Button leftButton(2);
Button rightButton(3);
Button playButton(4);
Button pauseButton(5);
Button stopButton(6);

LinkedList<int> commands;
String state = "input";
int commandIndex = 0;

void setup() {
  // put your setup code here, to run once:
  pinMode(multiplexerIO, INPUT_PULLUP);
  pinMode(A, OUTPUT);
  pinMode(B, OUTPUT);
  pinMode(C, OUTPUT);
  pinMode(latchPin, OUTPUT);
  pinMode(dataPin, OUTPUT);
  pinMode(clockPin, OUTPUT);

  forwardButton.begin();
  reverseButton.begin();
  leftButton.begin();
  rightButton.begin();
```

```
  playButton.begin();
  pauseButton.begin();
  stopButton.begin();

  pinMode(interruptPinA, INPUT_PULLUP);
  pinMode(interruptPinB, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPinA), incrementA, RISING);
  attachInterrupt(digitalPinToInterrupt(interruptPinB), incrementB, RISING);
  Serial.begin(9600);
}


void loop() {
  // Input state
  if(state == "input"){
    writeToRegister(inputLED);
    if(forwardButton.isReleased()){
      commands.add(0);
      Serial.println("forward pressed");
      writeToRegister(forwardLED | inputLED);
    }
    if(reverseButton.isReleased()){
      commands.add(1);
      Serial.println("reverse pressed");
      writeToRegister(reverseLED | inputLED);
    }
    if(leftButton.isReleased()){
      commands.add(2);
      Serial.println("left pressed");
      writeToRegister(leftLED | inputLED);
    }
    if(rightButton.isReleased()){
      commands.add(3);
      Serial.println("right pressed");
      writeToRegister(rightLED | inputLED);
    }

    if(playButton.isReleased()){
      if(commands.size() == 0){
        // If no command is imputted, run the last command from EEPROM
        int commandArr[EEPROM.read(listSizeAddr)];
        readIntArrayFromEEPROM(listAddr, commandArr, sizeof(commandArr));
        addArrayToCommands(commandArr);
        Serial.println("reading from EEPROM");
      }else{
        // If a command is inputted, write it to EEPROM
        writeIntArrayIntoEEPROM(listAddr, commandsToArray(), listSizeAddr);
        Serial.println("writing to EEPROM");
```

```
    }
      state = "active";
      Serial.println("play pressed");
    }


  // Active state
  }else if(state == "active"){
    if (commands.size() > 0){
      doCommand(commands.shift());
    }else{
      state = "input";
      commandIndex = 0;
      Serial.println("input mode:");
    }


  // Pause state
  }else if(state == "pause"){
    Serial.println("paused");
    writeToRegister(pauseLED);
    if(playButton.isReleased()){
       state = "active";
      }
    if(stopButton.isReleased()){
      state = "input";
      commands.clear();
    }
  }
    //small delay to avoid bouncing
    delay(50);
}

void doCommand(int command){
  switch(command){
      case 0:
        writeToRegister(forwardLED | activeLED);
        forward();
        break;
       case 1:
        writeToRegister(reverseLED | activeLED);
        reverse();
        break;
       case 2:
        writeToRegister(leftLED | activeLED);
        left();
        break;
       case 3:
```

```
      writeToRegister(rightLED | activeLED);
      right();
      break;
    }
}

// General drive function with pause/stop listener
void drive(int ticksA, boolean directionA, int ticksB, boolean directionB){
  numA = 0;
  numB = 0;
  while(numA < ticksA || numB < ticksB){
    if(numA < ticksA){
      analogWrite(spA, speedA);
      digitalWrite(dirA, directionA);
    }else{
      analogWrite(spA, 0);
    }
    if(numB < ticksB){
      analogWrite(spB, speedB);
      digitalWrite(dirB, directionB);

    }else{
      analogWrite(spB, 0);
    }

    if(pauseButton.isReleased()){
       state = "pause";
    }
    if(stopButton.isReleased()){
      state = "input";
      commands.clear();
    }

  }
  analogWrite(spA, 0);
  analogWrite(spB, 0);
}

// Incrementers for tachometer
void incrementA(){
  numA += 1;
  if(millis() - oldMilisA > interval){
    if(speedA < 250){
      speedA += 5;
    }
  }else if(millis() - oldMilisA < interval){
    if(speedA > 100){
      speedA -= 5;
```

```
    }
  }
  oldMilisA = millis();
}

void incrementB(){
  numB += 1;
  if(millis() - oldMilisB > interval - 20){
    if(speedB < 250){
      speedB += 5;
    }
  }else if(millis() - oldMilisB < interval + 20){
    if(speedB > 100){
      speedB -= 5;
    }
  }
  oldMilisB = millis();
}

// Directional drive functions
void forward(){
  Serial.println("forward");
  drive(40, HIGH, 40, HIGH);
  Serial.println("done");
}

void left(){
  Serial.println("left");
  drive(15, HIGH, 0, HIGH);
  Serial.println("done");
}

void right(){
  Serial.println("right");
  drive(0, HIGH, 15, HIGH);
  Serial.println("done");
}

void reverse(){
  Serial.println("reverse");
  drive(40, LOW, 40, LOW);
  Serial.println("done");
}

// Write a byte to shift register
void writeToRegister(byte data) {
  digitalWrite(latchPin, LOW);
  shiftOut(dataPin, clockPin, MSBFIRST, data);
```

```
  digitalWrite(latchPin, HIGH);
}

void writeIntArrayIntoEEPROM(int address, int numbers[], int arraySizeAddr)
{
  arraySize = sizeof(numbers);
  EEPROM.write(arraySizeAddr, arraySize)
  for (int i = 0; i < arraySize; i++)
  {
    EEPROM.write(address + i, numbers[i]);
  }
}

void readIntArrayFromEEPROM(int address, int numbers[], int arraySize)
{
  for (int i = 0; i < arraySize; i++)
  {
    numbers[i] = (EEPROM.read(address + i));
  }
}

void addArrayToCommands(int arr[]){
  for(int i = 0; i < sizeof(arr); i++){
    commands.add(arr[i]);
  }
}

int * commandsToArray(){
  int arr[commands.size()];
  for(int i = 0; i < commands.size(); i++){
    arr[i] = commands.get(i);
  }
  return arr;
}
```