S^3 Builds Together

Arduino Race Car

S^3 Spring 2020

*Professional driver on closed course. Do not attempt.

Introduction

Teams will build race cars of unparalleled agility and speed to compete for untold treasures! Your task is to combine multiple engineering disciplines to create an arduino-driven car.

Teams will be evaluated using a scoring rubric for car design, code design, and accuracy.

Teams will have the opportunity to request help from a mentor (using one and up to three available hearts on your rubric).

Step 1: Build the Car

Aesthetics are up to you and will come later. First: let's get the car built and running!

To build the car, follow the step-by-step guide provided with chassis, wheels, and smaller parts.

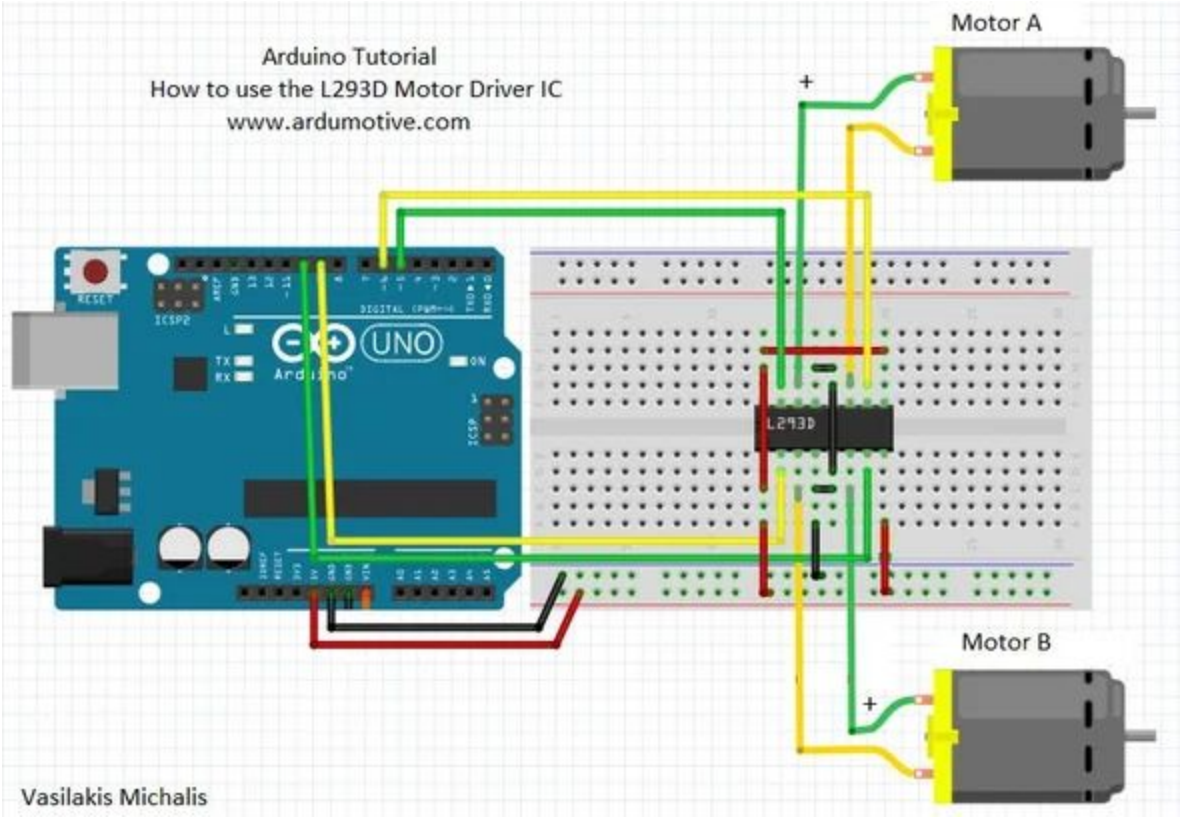
Let the mentors know if you get stuck here - these diagrams are not super technical.

*example car, parts not included

Now that your car is built, let's get it moving. This requires code!

Step 2: Build the Circuit

★ Just follow the wiring diagram provided; code comes next. Don't overthink it!



Tips and tricks:

★ Install batteries to check that your car moves!
★ Take the batteries out before connecting your car to your computer.
★ Load your code onto the Arduino following the steps in the next section.
★ Once your code is loaded onto the Arduino, you'll disconnect it from the laptop and attach the battery pack's wires to the VIN and second GND ports on the Arduino
★ DO NOT have Arduino connected to battery power and laptop power at the same time
★ The H-bridge is just an integrated circuit that controls how much power is sent to each motor's +/- terminals at a given time.

Step 3: Code

3.1. Installation

If you don't already have it, install the version of Arduino appropriate to your operating system and follow that installation guide:

a. Windows
b. Linux
c. Mac

3.2. Getting Started

Once you've got Arduino up and running, make a new document and call it DC_motor_control. Notice that the document has two functions: `void setup()`and `void loop()`. The first function happens once when the program starts and the next function happens over and over while the program is running.

a) First, let's define values for the pins. We'll use preprocessor directives because these values will be global constants—they'll be accessible from anywhere in the program and they'll never change.

   Put this code at the top of your file:

```
#define motorPin1 9
#define motorPin2 10
#define motorPin3 5
#define motorPin4 6
```

b) Next, in your `setup()` function, we'll use the `pinMode()` function to designate which pins will be used for output later in the code.

```
void setup() {
      pinMode(motorPin1, OUTPUT);
      pinMode(motorPin2, OUTPUT);
      pinMode(motorPin3, OUTPUT);
      pinMode(motorPin4, OUTPUT);
}
```

That's everything we need in the `setup()` function!!

3.3. Let's Get Moving!

We have a chassis, we have a power source, we have motors, and we have wheels. What more do we need?! We've got to send the motors enough power so the wheels move as much as we need them to move.

a) After the `loop()` function, let's write a function that will control signals to the wheels. Call it whatever you want: go, move, drive, zoom, global_domination, whatever.
   i) It needs four `int` parameters, two for each DC motor.
   ii) Each motor is connected to two pins which is mediated by the H Board. One wire sends power to the motor's positive terminal, while the other sends power to the motor's negative terminal; this allows us to turn the wheels clockwise or anti-clockwise.
   iii) To reiterate, a DC motor's direction of rotation solely depends on the polarity of its power source (i.e. which direction current is flowing in/out of it via its two terminals). This is why we have two variables controlling each of the two wheels.

Your function and its parameters will look something like this:

```
void digi_go(int wheel1pos, int wheel1neg, int wheel2pos, int wheel2neg)
```

Our example function is called `digi_go()` (we'll talk about a different way to move your wheels using analog values later) and looks like this:

```
void digi_go(int wheel1pos, int wheel1neg, int wheel2pos, int wheel2neg)
{
  digitalWrite(motorPin1, HIGH);
  digitalWrite(motorPin2, LOW);
  digitalWrite(motorPin3, HIGH);
  digitalWrite(motorPin4, LOW);
}
```

This is the function we'll call each time we want to send a signal to the wheels. Now we have to call the function from the main `loop()`.

b) In the `loop()` function, call the function you just made for all the movement of your car. Provide parameters to the function depending on how much power you want to send each wheel.
   i) Use `HIGH` and `LOW`, which are macros: `LOW` is a small number that results in a low voltage, `HIGH` is a large number that results in a large voltage. This translates roughly to "off" and "on" signals to your wheels. If you just want your car to go forward, stop, and make sharp turns, using the `digitalWrite()` function with these off and on signals serves this purpose.
   ii) After that function, include a `delay()` — if we send too many commands too quickly, the motor won't be able to process them all.

Your final function will look something like this:

```
void digi_go(int wheel1pos, int wheel1neg, int wheel2pos, int wheel2neg){
      digitalWrite(motorPin1, HIGH);
      digitalWrite(motorPin2, LOW);
      digitalWrite(motorPin3, HIGH);
      digitalWrite(motorPin4, LOW);
      delay(20);
}
```

3.4. Compiling, Uploading, & Troubleshooting

Now that we've written our basic functionality, let's see if it works! First, save your file. Use any name—unlike some languages, Arduino doesn't require that the name of the function be the same as your file.

Use the .ino file extension so you and the compiler will know that this is an Arduino-type file.

Load the code onto the Arduino and troubleshoot errors like this:

a) Click the check-mark in the top left of the Arduino IDE screen. You'll notice a message that says "Compiling sketch ..." and then either error messages in the console (the black section underneath the code) or a message that says "Done compiling."

Your errors might look something like this:

The line the error is on will be highlighted and the message describing the error should be helpful (sometimes it's not!).

Most errors fall into a few categories:

1. A line is missing a semicolon at the end. This is the most common error, and the compiler will yell at you until there is a semicolon after every command.
2. A function is missing an ending curly brace. Since Arduino is C++ under the hood, every function must be encapsulated with a pair of curly braces: { }
3. A variable is misspelled.

This error can be a bit more ambiguous: to our human eyes, the variable *is* declared, but since it's not an exact match, the compiler thinks it's not.

Just keep in mind that errors are normal: you haven't done anything wrong, and almost no

one compiles perfect, error-free code the first time. Learning to recognize what an error refers to is a large part of what it means to do computer science!

Work through any errors, and when the compiler is happy and looks like this—

—upload the script to the car by clicking the arrow button to the right of the checkmark button.

The car will either react or it won't! At this point, troubleshoot and experiment with values to see what they do to the car.

If you get stuck anywhere, flag one of us down and we can help. But don't worry if you're struggling!! That's the point and experimenting is the fun of it!

You can do it!!

Once you've got that down, let's try a different version of the `go()` function using analog signals:

b) First, comment out your original `digi_go()` function so we aren't uploading two versions of the function to the car. Do this by preceding the `digi_go()` function with two backslashes, like this:

```
//digi_go(HIGH, LOW, HIGH, LOW);
```

This function will use the same parameters, but it will need a different name from the other `go()` function.

```
void analog_go(int wheel1pos, int wheel1neg, int wheel2pos, int wheel2neg)
```

c) In the body of the function, to control the signals to the wheels, we're using what's called a "pulse with modulation" function, the `analogWrite()` function.

This function needs two values: which wire to pass a signal through (motorPin<val>) and how much to move (whatever value we pass to

wheel<1 or 2><pos or neg>). Your function will look something like this:

```
void analog_go(int wheel1pos, int wheel1neg, int wheel2pos, int
wheel2neg){
  analogWrite(motorPin1, wheel1pos);
  analogWrite(motorPin2, wheel1neg);
  analogWrite(motorPin3, wheel2pos);
  analogWrite(motorPin4, wheel2neg);
}
```

Keep close track of which wheel you're changing and whether you're sending it a positive or negative value!

For example:

```
analog_go(250, 0, 0, 250);
```

Try this and see what this does to each wheel!

Keep changing the wheel positions, change the delays, and see how much you can control your car!!

Once you feel confident about the series of function calls, add gorgeous flare to your car to win style points!!

And most of all: HAVE FUN!!

Additional Resources

1. Video L298N: https://www.youtube.com/watch?v=dyZolgNOomk
2. Using the H-bridge with two motors: https://www.instructables.com/id/How-to-use-the-L293D-Motor-Driver-Arduino-Tutorial/
3. How-to build the car: https://howtomechatronics.com/tutorials/arduino/arduino-dc-motor-control-tutorial-l298n-pwm-h-bridge/

Glossary of terms

1. **Arduino:** Is an open source electronic prototyping platform enabling users to create interactive electronic objects. It consists of both a physical programmable circuit board and a software, or IDE (Integrated Development Environment) that runs on your computer, where you can write and upload the computer code to the physical board.
2. **Breadboard:** A breadboard is a construction base of prototyping of electronics. It is a rectangular plastic board with a bunch of tiny holes in it. These holes let you easily insert electronic components to build and test an early version of an electronic circuit. These circuits can contain batteries, switches, resistors, LEDs, etc.
3. **Chassis:** The base frame of a motor vehicle or other wheeled conveyance
4. **Constants:** Is a value that cannot be altered by the program during normal execution, i.e., the value is a “constant”. This is contrasted with a variable, which is an identifier with a value that can be changed during normal execution, i.e., the value is variable.
5. **Directives:** Is a language construct that specifies how a compiler (or other translator) should process its input. They can be processed by a preprocessor to specify compiler behavior, or function as a form of in-band parametrization.
6. **H-bridge:** An H bridge is an electronic circuit that switches the polarity of a voltage applied to a load. These circuits are often used in robotics and other applications to allow DC motors to run forwards or backwards.
7. **High:** in terms of the coding, is a large number that results in a large voltage
8. **Low:** in terms of the coding, is a small number that results in a low voltage

****

Don’t look at this unless you need to!
All the code in one place, for debugging:

```
#define motorPin1 9 //wheel 1
#define motorPin2 10 //wheel 1
#define motorPin3 5 //wheel 2
#define motorPin4 6 //wheel 2

void setup() {
  pinMode(motorPin1, OUTPUT);
  pinMode(motorPin2, OUTPUT);
  pinMode(motorPin3, OUTPUT);
  pinMode(motorPin4, OUTPUT);
  Serial.begin(9600);
}

void loop() {
  digi_go(HIGH, LOW, HIGH, LOW);
  delay(20);
}

void digi_go(int wheel1pos, int wheel1neg, int wheel2pos, int wheel2neg) {
  digitalWrite(motorPin1, HIGH);
  digitalWrite(motorPin2, LOW);
  digitalWrite(motorPin3, HIGH);
  digitalWrite(motorPin4, LOW);
}

void analog_go(int wheel1pos, int wheel1neg, int wheel2pos, int wheel2neg)
{
  analogWrite(motorPin1, wheel1pos);
  analogWrite(motorPin2, wheel1neg);
  analogWrite(motorPin3, wheel2pos);
  analogWrite(motorPin4, wheel2neg);
}
```