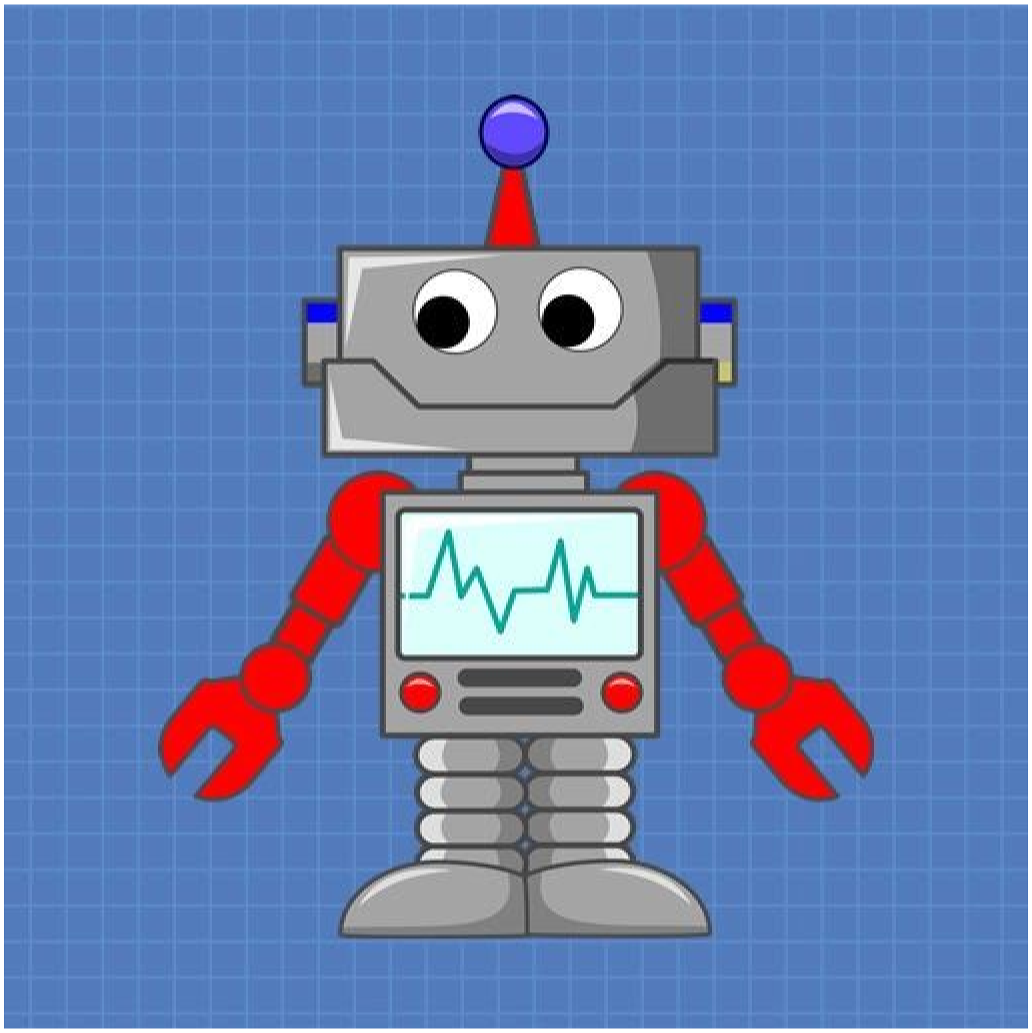
Build a Robot Car with Speed Sensors



DroneBot Workshop Tutorial

https://dronebotworkshop.com

Table of Contents

A rite of passage for every budding robotics experimenter is to build a small robot car. This task is simplified by using one of those robot car chassis kits which can be picked up on eBay and Amazon for less than 15 dollars. And while these kits are a great value they lack instructions, especially for using the small speed encoder disk that comes packaged with them. Let's see how to make use of that disk and build a robot car with speed sensors.

Introduction

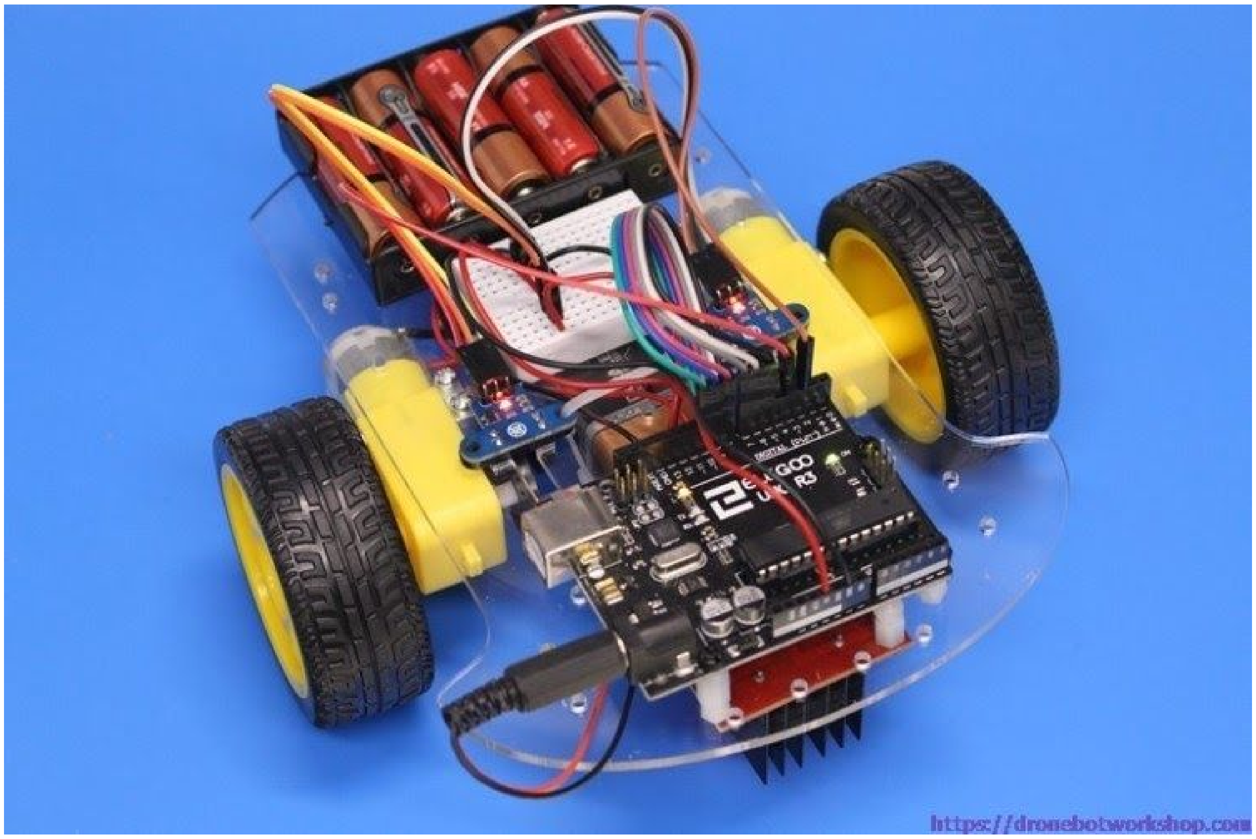
Many of the robotics projects that I've been working on lately have made use of a very common "Robot Car Chassis". These tiny units have a lot of advantages for robotics experimenters, especially ones like myself whose expertise lies with electronics and software and not with mechanics:

- They are very inexpensive. I've picked some of these kits up for less than 15 dollars.
- They are surprisingly powerful and can carry a decent payload
- They are pretty easy to assemble and modify
- They are quite durable so your robot can survive a few crashes while you are in the testing phase.

A few of the robotics projects that I'll be presenting to you here at the Dronebot Workshop rely upon these types of robot car chassis so I thought it would be a good idea to show you how to put one together. And I also want to use the opportunity to show you how to use those two strange black "slotted disks" that come with each of these kits.

Those disks are actually "encoder wheels" and they can be used with an opto interrupter-based sensor to measure speed and distance travelled. I haven't seen many instructions for using them with the robot car chassis so in this article (and the accompanying video) I'll give you all the information you need to put them to work.

So let's get rolling!

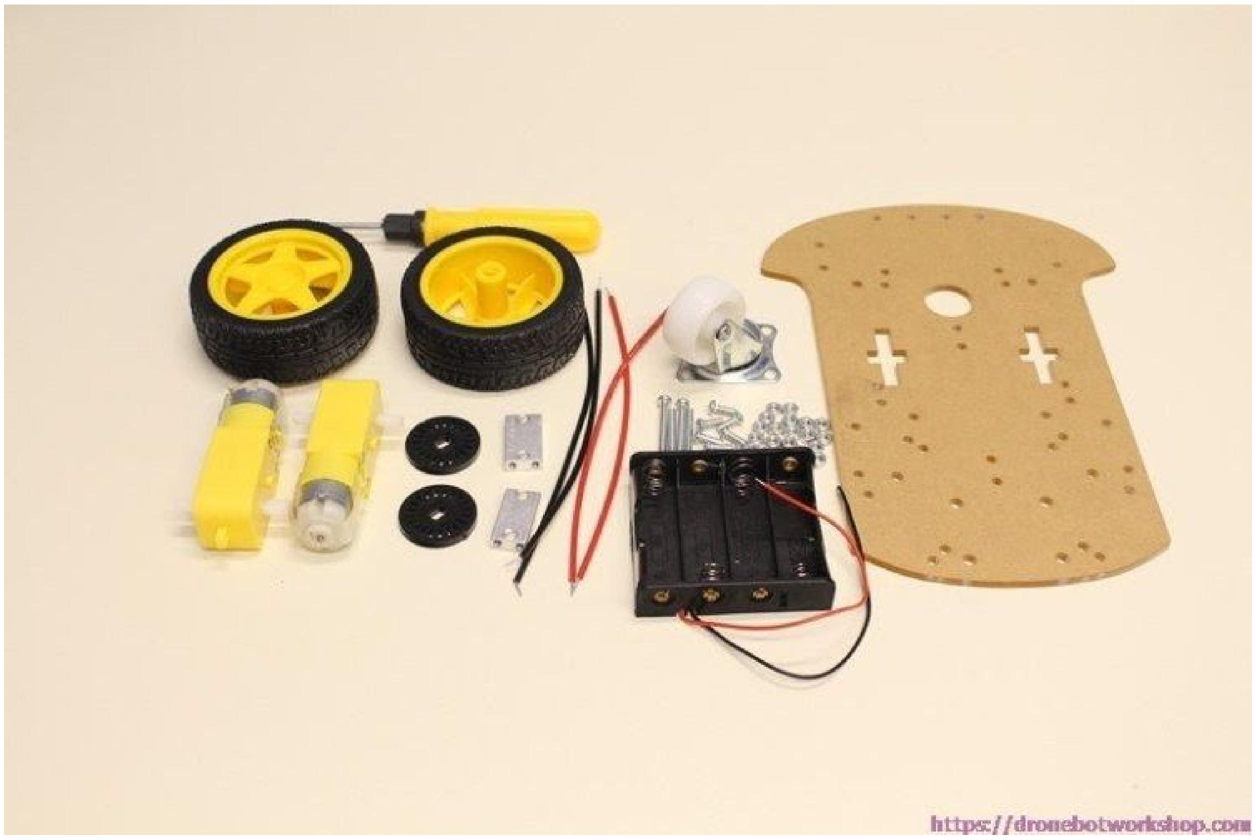


Robot Car Chassis

There are a few variations on the robot car chassis but for the most part they all consist of the following items:

- **The Chassis Base**. This is an acrylic plastic base, precut with several holes to mount components.
- **Motor Mounts**. Some of the kits use acrylic for this as well, others use aluminum brackets. Mounting the motors is probably the thing that confuses most users so I'll illustrate that in detail.
- **DC Motors**. These 6-volt motor have a surprising amount of torque for their size.

- Wheels. A car isn't much use without wheels! These chassis kits use two plastic wheels, with tire treads that work well on both smooth surfaces and carpet.
- **Rear Castor**. As there are only two wheels a castor is provided for balance and stability. The castor comes with mounting hardware.
- **Encoder Wheels**. These two plastic disks are meant to mount to the motor shafts, on the opposite sides from the wheels. They have a series of slots in them as they are designed to be used with an optical source-sensor to provide feedback on motor speed and wheel position. I'll show you how to use them today.
- **Battery Holder**. These kits come with a holder for 4 AA cells, to provide 6 volts for the motors. I often replace these with a 5 cell holder as the L298N H-Bridge controller that I usually use to drive the motors will drop 1.4 volts, so I consider the 4 cell holder to be "spare parts".
- **Mounting Hardware**. All of the nuts, bolts and spacers you will need to put the chassis together. You'll need to supply more hardware to mount your own components on the base.
- **Misc Parts** – It's amazing what else some of these kits come with. Common additions are power switches, wires and pan-and-tilt mechanisms for mounting sensors or cameras. One of my kits even supplied a small screwdriver.



Not bad for an under 15 dollar purchase!

Making Plans

Before starting any project it's a good idea to plan out what you'll be doing. In the case of the robot car chassis this means determining what components and sensors you'll be using in your design and where on the chassis you want to mount these items.

A couple of considerations you might want to take into account when planning;

- What components will your design use?
- What is the total weight of all of your components? Best to keep this under a kilogram (2.2 pounds) as heavier loads will cause performance issues and rapid battery drain.

- Are there any sensors (i.e. ultrasonic, optical) that will need to be mounted in specific positions? Are there already mounting holes on the chassis to attach these, or do you need to drill a few?
- Remember, you can mount components under the chassis as well as above it. For those under the chassis you need to be sure they are securely mounted and have enough clearance to not risk touching the ground.
- Weight distribution is very important. Try to avoid placing a lot of weight on one side of the car chassis. Ideally the weight should be equally distributed. If you have specifically heavy components it's a good idea to keep them need the center of the wheelbase.
- Will you be using the speed encoders? If so remember that you'll need to mount the optical source-sensor in the slots provided, so leave enough room for it.

Once you've determined how you'll mount your components it's time to begin assembling the robot car chassis. But before we do that let's first take a look at the optical sensor that is used to measure the wheel position and speed.

Optical Interrupter Sensor

Each of theses robot car chassis kits comes with a pair of encoder wheels, which are small black plastic disks with a series of slots cut through them. These are designed to mount onto each motor shaft opposite the wheel, so they will spin at the same speed as the wheel does.

What's missing from the kit is the sensor itself (to be fair most of these kits don't contain any electronic components, just the motors and hardware, so the omission is acceptable). The chassis, however, is precut to accept an H206 slot-type opto interrupter. This is the key to getting the speed sensor to work.

If you are not familiar with opto interrupters (also referred to as an “opto isolator” or “optical source-sensor”) don’t worry, they are actually pretty simple devices and they are used in a variety of applications. In fact you’re probably using one right now without even being aware of it, as a mouse with a scroll wheel and most non-laser printers make use of them.

An opto interrupter consists of a source of light, usually an infrared LED, and a phototransistor sensor. The light source is mounted facing the sensor with a gap between them. In the case of the H206 that gap is about 6 millimeters.

In operation, the LED is illuminated and it shines onto the phototransistor, which detects its light and allows current to pass from the collector to emitter. Essentially a phototransistor is like a regular bipolar transistor except instead of reacting to current applied to the base it reacts to photons of light.

If a solid non-transparent object is placed in the slot between the LED and phototransistor it will interrupt the light beam, causing the phototransistor to stop passing current.

In our application the opto interrupter will be positioned with the rotating encoder wheel in the gap between the LED and transistor. As the wheel spins the slots in the wheel will allow pulses of light to reach the phototransistor, causing it to switch on and off in time with the wheel rotation.

Each pulse will represent a slot in the encoder wheel, so if your encoder wheel has 20 equally-spaced slots (a pretty common value) then each pulse indicates that the wheel has turned 18 degrees (360 degrees divided by 20).

Add an LM393 Comparator

You could just purchase a couple of H206 opto interrupters and wire them up to your microcontroller, however you'd probably wouldn't be satisfied with the results. That's because in the real world the output pulses directly from the phototransistor are poorly formed and as a result your code would end up giving a lot of errors.

What is needed is a way to clean up the output a bit and generate some nice clean 5-volt pulses suitable for using with an Arduino or other microcontroller. And the perfect component to do that is a "comparator".

A comparator is a device that has two inputs and one output. One of the inputs is a "reference voltage" input, the other input is where you would connect the output of the phototransistor. The output of the comparator is digital, so it can go either high (5-volts) or low (Ground).

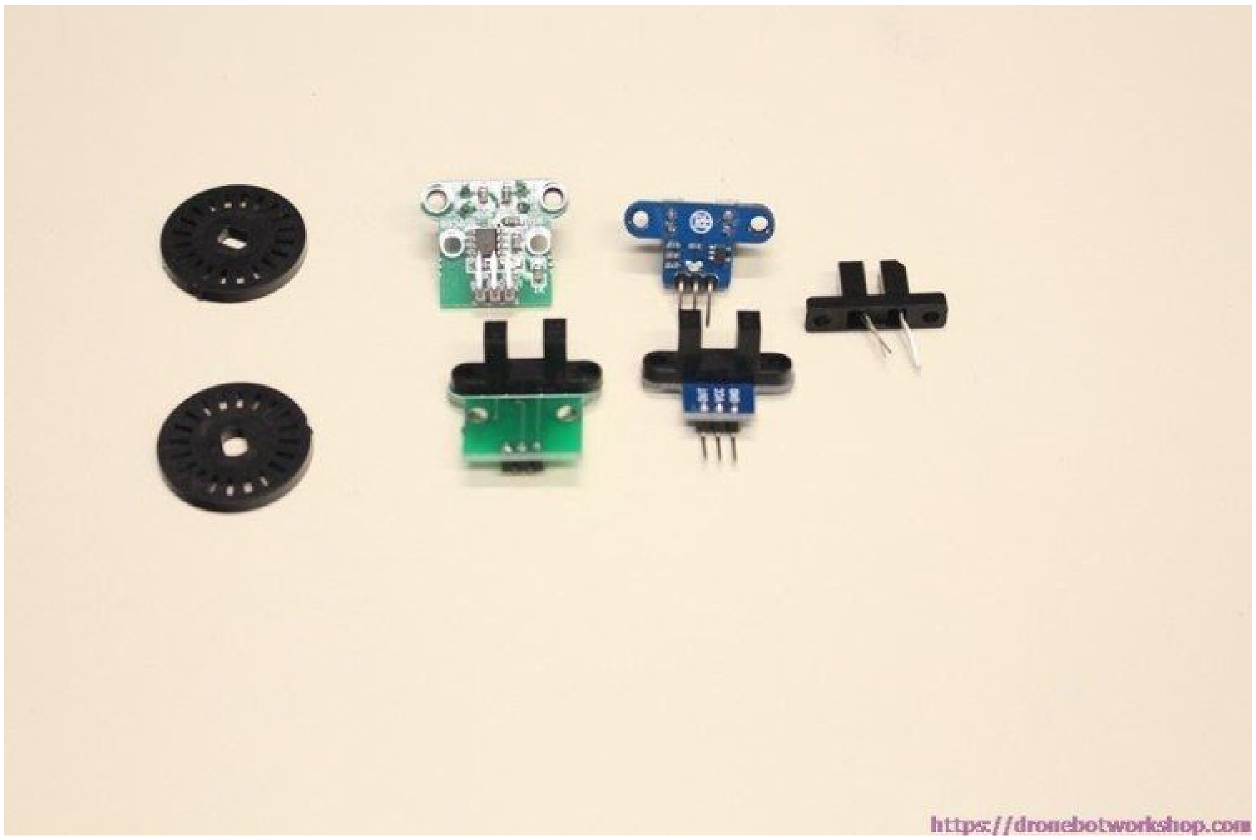
The key is the reference voltage. If the input (in our case from the phototransistor) is below the reference voltage then the digital output of the comparator remains low. If the input equals or exceeds the reference voltage then the output goes high.

To put it another way, a comparator is a good way to clean up a "weak" or "dirty" digital signal, as well as a way of determining if an input voltage has reached a preset threshold.

The LM393 is a dual comparator, meaning two independent comparators in the same tiny package. It's perfect for the job of cleaning up the output from the opto interrupter.

Because the combination of the H206 and LM393 is so common there are a number of small inexpensive sensor modules constructed with these two components (plus a handful of resistors and capacitors). These sensors are often called "LM393 Speed

Sensors" although the name is a bit of a misnomer as the LM393 is just one of the components. However as it's a common name I've chosen to stick with it.



Calculating Wheel Speed

Calculating the speed that the wheel spins is pretty simple really. If we use the example of an encoder wheel with 20 slots then for every 20 pulses from the LM323 Speed Sensor our motor has spun the wheel one revolution. Knowing that we can count how many pulses we get in a second and use that to determine the actual wheel speed.

Notice that I'm saying "wheel speed" and not "motor speed", they are different as the motor has internal gearing that slows it down.

As a simple example if I measure exactly 20 pulses every second then our wheel is spinning a one revolution per second. Multiplying that by 60 gives us the speed in RPM, which in this case is 60 RPM.

Once we determine the wheel speed we can use the wonderful power of mathematics to calculate a couple of other useful parameters:

- We can get the actual speed (in Kilometers or Miles per hour) of our robot car.
- We can determine how far our robot has travelled.
- We also know that the wheel is actually spinning, as opposed to being stuck and not moving.

To get those first two parameters we need to know one other thing – the circumference of the wheel itself. There are two ways to get that value:

1. **Measure it!** Use a flexible measuring tape or a piece of string wrapped around the outside of the wheel and see how long it is.
2. **Calculate it.** Multiply the diameter of the wheel by pi (3.14) to get the circumference.

You can probably see why this value is so important, the circumference of the wheel is equal to the distance that the wheel will travel in one rotation (assuming you have perfect traction with the surface you're travelling on).

I measured one of the wheels in the kit I'm working with and it had a diameter of 66.1 mm. So using the formula I arrive at the following:

$66.1 \times \pi = 207.6$

So my wheel has a circumference of 207.6 mm. A tape measure wrapped around the wheel confirms that this is the correct value.

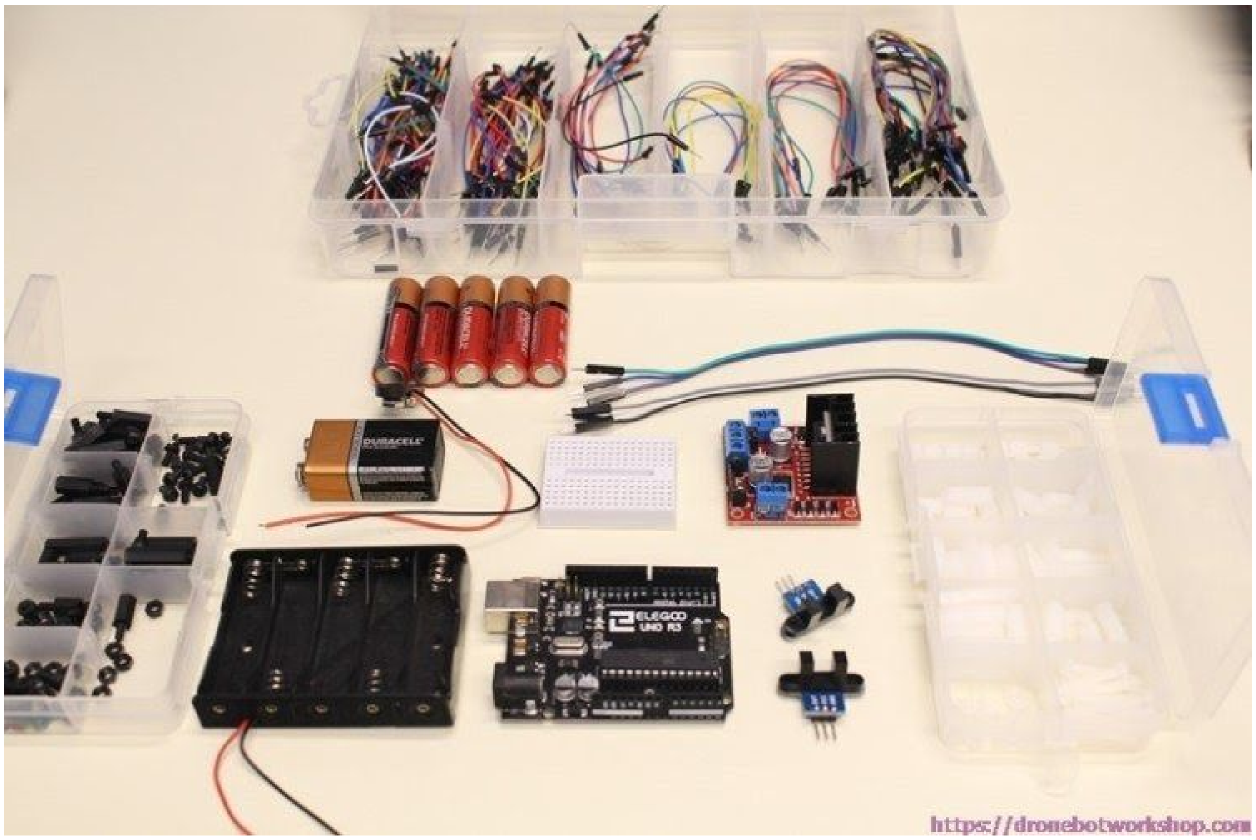
You could also do the math using imperial measurements if you really insist, but the metric measurements are much easier to work with.

Putting the Chassis Together

Let's gather together all of our components and the robot car chassis parts and start assembly. In addition to the robot car chassis kit itself we will need the following items:

- **An Arduino Uno (or clone).** This is the "brains" of our simple robot. While other Arduino's would also suffice I'd really recommend you use an Uno so that the sketches I'm presenting will work "as-is". If you must use a different Arduino you may need to adjust the code to reflect the different pins used for the two interrupts.
- **An L298N Dual H-Bridge Motor driver.** This is a common way of driving DC motors with a microcontroller or microcomputer. I've covered the L298N before on the site.
- **Optical speed sensors based upon the H206 opto interrupter and LM393 dual comparator.** There are several suitable models, you'll find these on eBay as well as at your local electronics shop (if you have one).
- **Wires and Jumper Wires.** You'll need some 18 – 20 gauge wire to hookup the motors (some kits provide these) as well as some male to female jumper wires to hook the Arduino to the sensors and motor driver.
- **Hardware.** Nuts, bolts and spacers – the kits include all the hardware for mounting the main parts but you'll have to provide your own to hold down the Arduino and motor driver board.
- **A Solderless Breadboard** – This is optional but it's an easy way to wire up the 5-volt power distribution for your robot. Also, I plan on keeping this robot to add extra items to it so the breadboard will come in handy for future expansion.

You'll also need to do a small bit of soldering to attach the wires to the motors. You might also use a bit of heat-shrink tubing to insulate and strengthen the connections to the motor, but that's optional.



Step 1 – Preparing the Chassis

Our first step is to prepare the chassis for the components we'll be using. Lay out your components on the main chassis plate to determine the best arrangement. Then see if you need any additional mounting holes, if you do mark their positions on the protective backing that is affixed to the chassis plate.

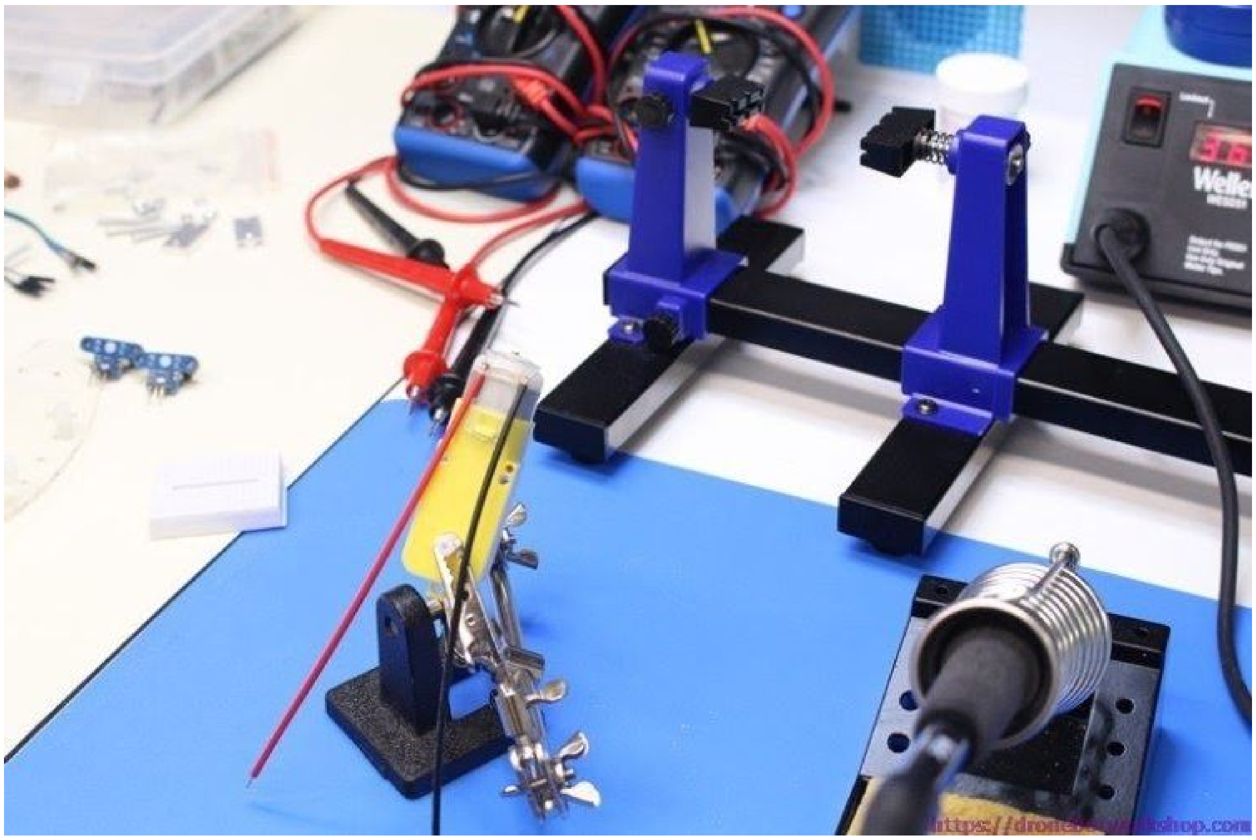
Now drill the holes (if required),

Once the plate has been prepared you can peel off the protective backing from both sides of the chassis plate. You can also remove the backing from the acrylic motor holders if your chassis uses them.

Step 2 – Prepare the Motors

Most of these robot car chassis kits have motors with no leads connected to them. Some of the chassis kits will provide wires for this purpose,with others you'll need to supply them yourself. If you do supply the wire use 18 or 20 gauge if you can to prevent any power loss. Ideally you'll use different color wires to distinguish between positive and negative.

The best way to attach the wires is to solder them. Mounting the motors in a clamp or vise might make it easier to hold the motor steady while you solder the wire to it. Be careful as the motor terminal connections are a bit fragile.



Again you may want to put a bit of heat-shrink tubing over the connections after they are soldered to make them more solid and to insulate them, but this step is optional.

Step 3 – Mount the Motors on the Chassis

There are two different styles of motor mounts used in these inexpensive robot car chassis kits. I'll describe both of them for you.

Style 1 – Acrylic Mount

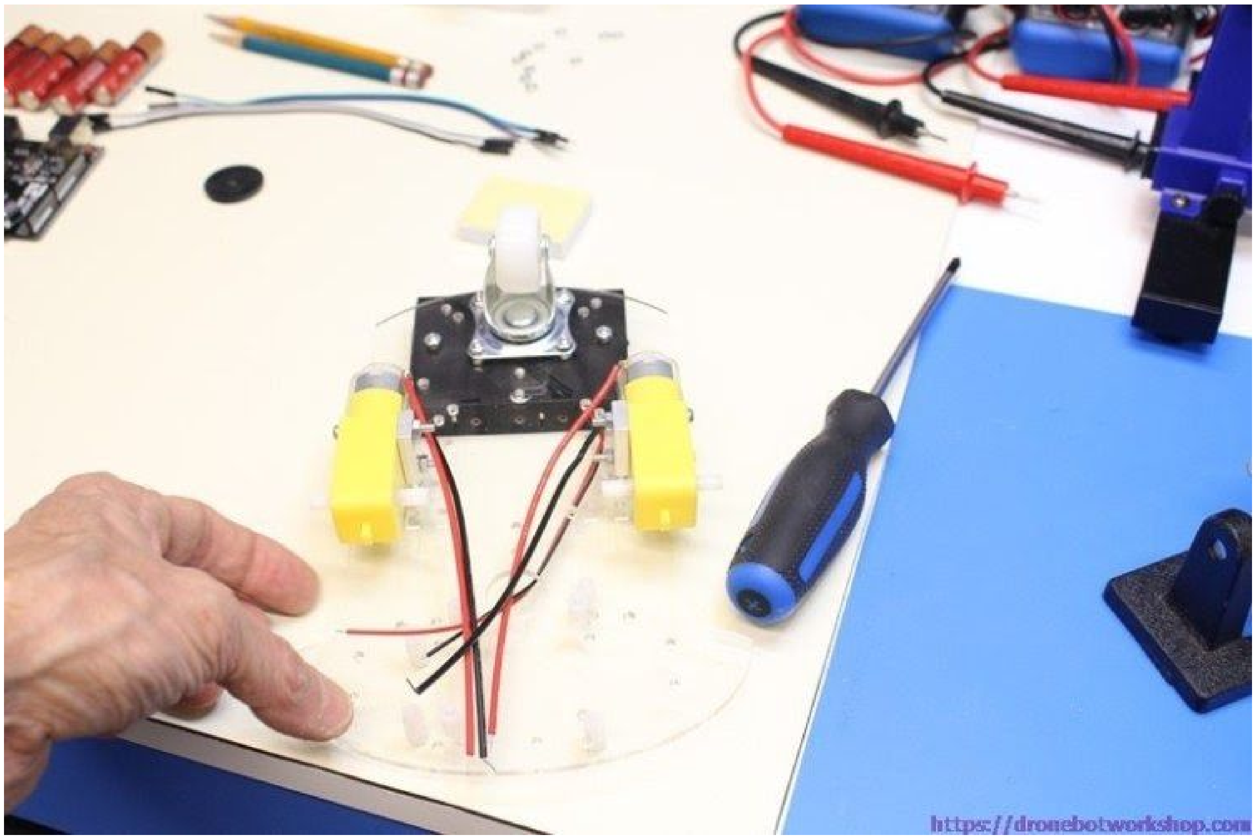
This style of mount uses a couple of “t-shaped” acrylic pieces, along with a couple of corresponding slots on the chassis base. To use them pass them through the slots and mount the motor between them. Then pass some screws (supplied with your kit) through the mounts and the motor to hold everything together.

On some kits you only get one acrylic piece for each motor, if this is the case with your kit then it mounts on the inside of the chassis. Again you secure it with a couple of screws.

Style 2 – Aluminum Mount

This type of mount makes use of a couple of aluminum blocks, one per motor. You attach them to the motors first and then fasten them to the chassis with a couple of small screws (the aluminum blocks have a coupe of threaded holes for this purpose).

You kit probably came with a minimal instruction sheet that should illustrate the motor mount arrangement. Have a peek at it if you aren’t sure how everything fits together.



Step 4 – Install the Wheels, Rotary Encoders and Sensors

We aren't going to get very far without wheels so now it's time to install them!

The wheels simply press-fit onto the motor shaft, note that both the motor shaft and the mounting hole on the wheels are "D shaped" so they need to be lined up correctly. Once you have them lined up press the wheels on as far as they will go.

Next we install the opto interrupter-based sensors. There are a couple of different styles of these but they all press-fit into the square slots provided on the base. Although they fit pretty snugly I usually drill a mounting hole in the proper position as well so that I can secure them with either a screw and nut or with a tie wrap (as I did in the demo robot in the video).

The rotary encoders also press-fit onto the motor shaft, on the opposite side from the wheels. You will need to slide the disk back and forth until you arrive at a position that lets it spin without binding on the robot car chassis or on the opto interrupters. I found them to be a bit loose so I secured mine in place with a dab of hot glue after I established the correct position. Epoxy would also work here as well.

So now we have wheels but our chassis won’t stand up correctly. To fix that we’ll need to install the support castor.

Step 5 – Install the Castor

The final step in assembling the basic robot car chassis is to instal the castor. This is mounted with four spacers, each spacer is threaded to accommodate a screw at each end.

Now that the castor is installed the robot car chassis is complete. However it really doesn’t do anything yet so you’ll want to perform one more step and install your components onto it.

Step 6 – Install your Components

Now that we have completed the robot car chassis itself it is time to install our components.

Obviously this step will be different for everybody, as it really depends upon what components you plan on using with your robot. I'll show you how I installed the Arduino, the motor controller and the LM393-based optical speed sensor.

Install the Battery Holder

I'm going to actually use a 5-cell battery holder instead of the 4-cell holder supplied with the kit, as I know I'll be losing 1.4 volts due to the voltage drop in my L298N H-Bridge driver. If you choose to use the original battery holder it installs in an identical fashion.

You'll also need to decide how you plan to power the Arduino Uno. The easiest (but not necessarily the best) way is to use a 9-volt battery plugged into the Arduino's 2.1 mm power jack. You can purchase 9-volt power clips already soldered to 2.1 mm plugs or you can just solder up one yourself.

If your kit came with a power switch (some do, some don't) you can use it to switch the Arduino power supply. The other batteries are used to power the motors so when the motors aren't being driven they shouldn't draw any current (assuming you use the Arduino supply for the L298N boards logic power).

Mounting the Arduino

The easiest way to install an Arduino Uno on the robot car chassis is to use some plastic or metal spacers. These can be purchased from eBay, Amazon or your local electronics store.

If you don't have spacers you can use a couple of different methods instead:

- A screw and three nuts for each mounting hole. This is a pretty good arrangement but it takes a lot of alignment and tightening to get it right.

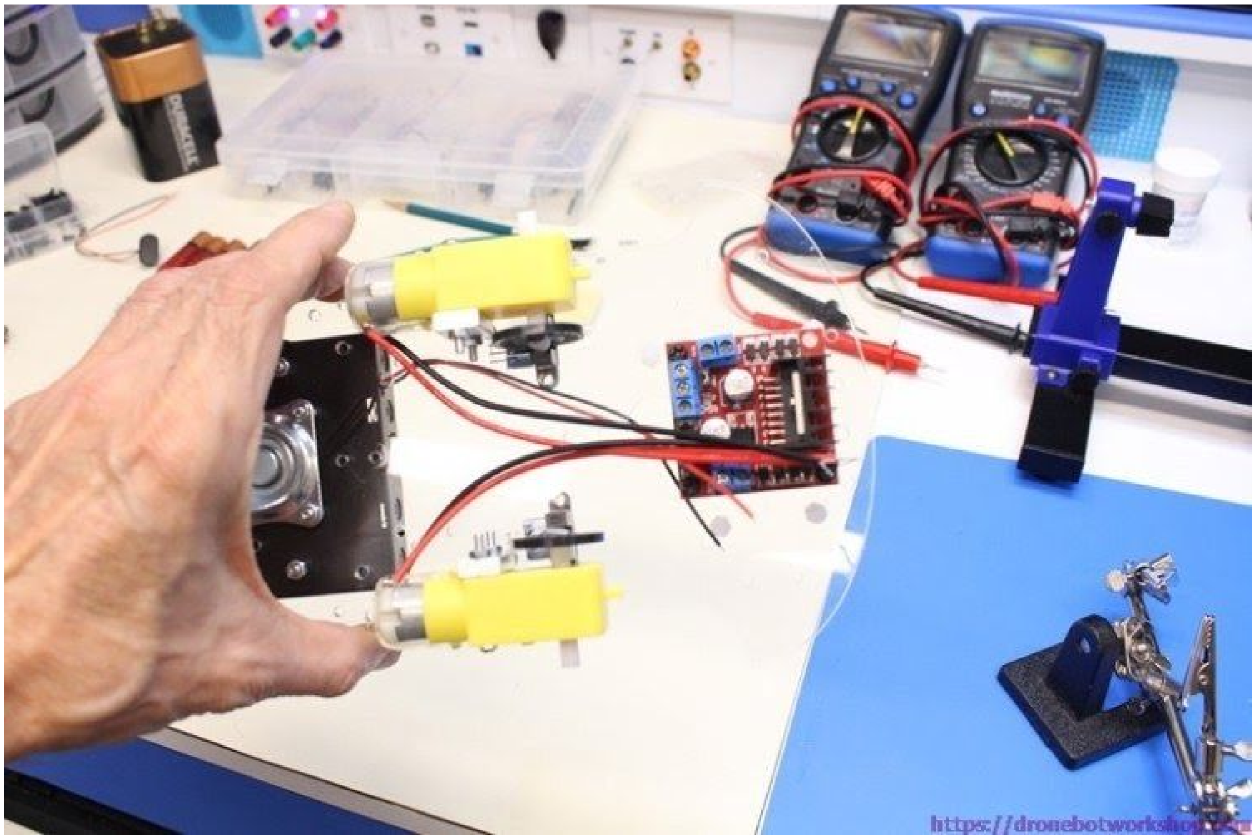
- Slice an old disposable ballpoint pen up into small spacers. You can mount these with a screw and a single nut.

The Arduino will also need some power. In this simple design I will be using a 9-volt battery to power the Arduino but of course there are better options available. Those 5-volt USB power banks work well and have the added advantage of being rechargeable.

Mounting the L298N Motor Driver

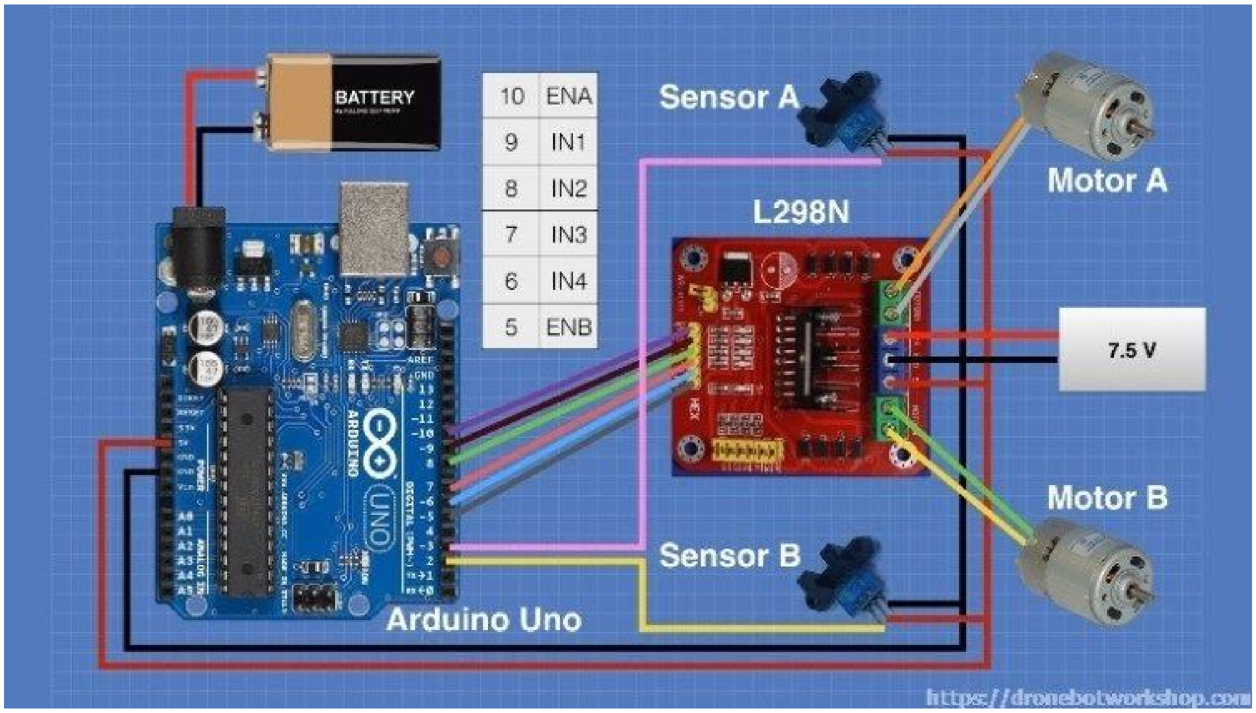
The motor driver mounts in the same way that the Arduino did, using spacers. Make sure to align the board so that the wires from the motors can reach the terminals on the driver board.

In my design I decided to place the motor driver boards under the chassis, even with the heatsink on the driver there is still adequate clearance. The chassis has a large hole in it that is ideal for passing the wires used to connect the L298N motor driver module to the Arduino.



Step 7 – Wire it Up!

Finally we can wire up the robot car. The following diagram shows all of the electronic components used in the car and how they are hooked together.



The design makes use of the Arduino Uno's two "hardware interrupt" pins. This is the key to getting everything to work so let's examine interrupts further.

Interrupts

Interrupts are a very important concept in coding for any computer, be it a tiny Arduino or a full sized desktop computer running Windows, Linux or OSX. You've probably used them a few million times, perhaps without knowing it.

A good example of using interrupts is getting data from a keyboard. One way of getting keyboard data would be to write a program that constantly keeps checking the keyboard to see if a key has been pressed. This will work, but not without disadvantages:

- Its wasteful. You spend a lot of time querying the keyboard only to find out there is no data.
- You can miss data. If a key is pressed while your program is performing another function it may miss a keypress altogether.

Interrupts provide a much better way of doing this.

Instead of querying (or "polling") the keyboard constantly the program for the most part ignores the keyboard. But when a key is pressed on the keyboard it generates a "hardware interrupt", a digital signal that tells the computer "stop whatever you are doing and take care of this". The computer then runs some code called an "interrupt handler", which in the case of the keyboard determines which key you actually pressed. It then takes whatever action is required to handle that keypress.

This is much more efficient because the computer doesn't waste time checking the keyboard until it is notified that a key has actually been pressed.

Types of Interrupts

You might notice that I used the term “hardware interrupt” in the previous example, and if it leads you to suspect that there may be other types of interrupts then you are correct.

First of all there are both Hardware and Software interrupts. The 8-bit AVR processors that power the Arduino boards are not capable of Software interrupts (unlike the processor in your computer which is) so we’ll limit our discussion to Hardware interrupts.

You can further divide hardware interrupts into two categories – internal and external. Internal hardware interrupts are generated by the internal timers in the AVR processors, you’ve probably used them without realizing it as timing functions like “delay()”, “millis()” and “micros()” make use of them. The “tone()” function also uses internal interrupts, as does the Servo library.

External hardware interrupts, as the name would imply, are generated externally from the processor. In an Arduino these interrupts can be generated in two ways.

- **External Interrupts** from a change in state of one of the predefined external interrupt pins.
- **Pin-change interrupts** from a change in state of any one of a predefined group of pins.

The term “change of state” simply means when a digital signal changes value from 0 to 1 or vice-versa.

In our design we will be using the first one, external interrupts from a change in state of one of the predefined external interrupt pins. Those “predefined external interrupt pins” differ depending upon which Arduino board you are using, as follows:

Arduino Model	**Digital Interrupt Pins**
Uno, Nano, Mini, other 328-based	2,3
Mega, Mega2560, MegaADK	2, 3, 18, 19, 20, 21
Micro, Leonardo, other 32u4-based	0, 1, 2, 3, 7
Zero	all digital pins, except 4

As our robot car chassis experiment will be using an Arduino Uno we have two interrupt pins available to us, pins 2 and 3.

Internally the AVR processors that the Arduinos are based upon use interrupt numbers starting at zero, as opposed to using the pin numbers. Here is a chart that explains how the pin numbers are related to some of the Arduino models:

Arduino	INT 0	INT 1	INT 2	INT 3	INT 4	INT 5
Uno	2	3				
Mega2560	2	3	21	20	19	18
Micro	3	2	0	1	7	

Coding for Interrupts

The key concept for coding for interrupts is that every interrupt needs an "Interrupt Service Routine" or "ISR", a special function that is run when an interrupt is detected. These are written like regular Arduino functions with the following caveats:

- An interrupt service routine (ISR) needs to run quickly, so it usually has a minimal amount of code.
- There are certain functions and commands that don't run well in an ISR, specifically functions that make use of the Arduino's internal timers. Examples are the tone and servo functions.
- Sometimes the variables used within an ISR are seen by the Arduino IDE compiler as being unused. If this happens the code won't run, the solution is to declare them to be "volatile"

In order to make the ISR run when an interrupt is detected it needs to be attached to the specific hardware interrupt that it services. You do this using the Arduino *attachInterrupt()* command. This command has the following format;

attachInterrupt(interrupt_number, ISR, mode)

The "*interrupt_number*" is (what else?) the number of the interrupt, on the Arduino Uno this would be either 0 or 1.

The "*ISR*" refers to the name of the Interrupt Service Routine that we coded to handle this interrupt.

The "mode" defines when the interrupt is triggered, and it can have the following values:

- **LOW** – triggered when the interrupt whenever the pin is low,
- **CHANGE** – triggered when the interrupt whenever the pin changes value
- **RISING** – triggered when the pin goes from low to high,
- **FALLING** – triggered when the pin goes from high to low.

So the following statement will trigger an ISR called "My_ISR" when interrupt zero goes from low to high:

attachInterrupt(0, MY_ISR, RISING);

Better Interrupt Code

The previous example is perfectly valid but it requires us to know which interrupt number our interrupt pin is attached to. Since this varies depending upon which Arduino model we use it isn't necessarily transportable between Arduino types.

A better way to code this would be to also use the Arduino "*digitalPinToInterrupt()*" function. This function translates the pin number to the Interrupt number, taking into account which Arduino model you are compiling code for.

The "*digitalPinToInterrupt*" function has one input parameter, the pin number you're using. It outputs the Interrupt number. Using it our previous code statement would read as follows, assuming that we are coding for an Arduino Uno.

attachInterrupt(digitalPinToInterrupt(2), MY_ISR, RISING);

As the Uno has Interrupt 0 attached to Pin 2 the *digitalPinToInterrupt* function will return a zero in this case.

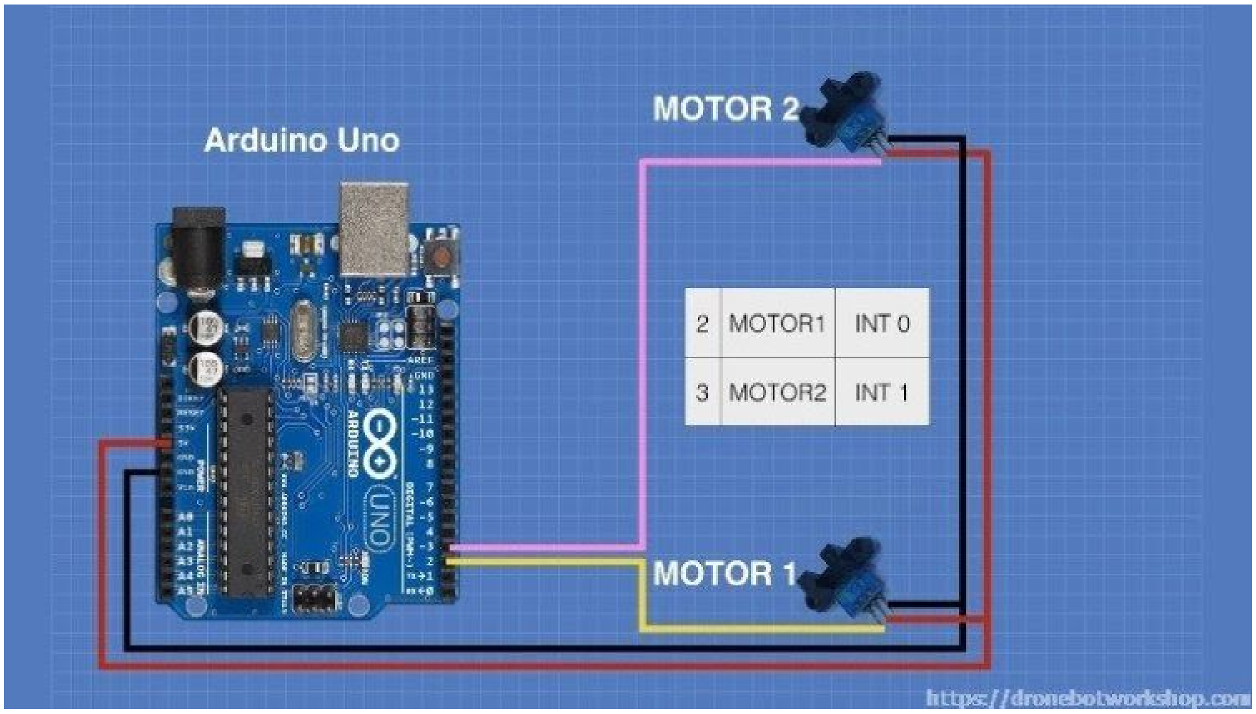
I've barely scratched the surface when it comes to interrupts, it's a very important code concept that you should try and learn more about as it will open up a whole new world of applications you can design. One excellent source of information regarding interrupts is the forum maintained by Nick Gammon in Australia. I highly suggest you check it out

Testing the Speed Sensors

Ok enough talk, it's time for some code!

The following code will test the speed sensors and display the motor rotation speed in RPM on the Arduino serial monitor. You can watch the experiment in the YouTube video associated with this article or give it a try yourself. In the video I just hooked the motors directly to a 6-volt battery to spin them at full speed and I attached the sensors as follows:

- Sensor 1 output to pin 2 (Interrupt 0) of the Arduino Uno
- Sensor 2 output to pin 3 (interrupt 1) of the Arduino Uno



The two sensors are powered from the 5-volt output from the Uno. If you use something other than an Uno you’ll need to consult the Interrupt pinout chart to get the correct pins

```
/*
       Optical Sensor Two Motor Demonstration
       DualMotorSpeedDemo.ino
       Demonstrates use of Hardware Interrupts
       to measure speed from two motors

       DroneBot Workshop 2017
       http://dronebotworkshop.com
*/

// Include the TimerOne Library from Paul Stoffregen
#include "TimerOne.h"

// Constants for Interrupt Pins
// Change values if not using Arduino Uno

const byte MOTOR1 = 2;  // Motor 1 Interrupt Pin - INT 0
const byte MOTOR2 = 3;  // Motor 2 Interrupt Pin - INT 1

// Integers for pulse counters
unsigned int counter1 = 0;
unsigned int counter2 = 0;

// Float for number of slots in encoder disk
float diskslots = 20;  // Change to match value of encoder disk

// Interrupt Service Routines

// Motor 1 pulse count ISR
```

```
void ISR_count1()
{
  counter1++;  // increment Motor 1 counter value
}

// Motor 2 pulse count ISR
void ISR_count2()
{
  counter2++;  // increment Motor 2 counter value
}

// TimerOne ISR
void ISR_timerone()
{
  Timer1.detachInterrupt();  // Stop the timer
  Serial.print("Motor Speed 1: ");
  float rotation1 = (counter1 / diskslots) * 60.00;  // calculate RPM for Motor 1
  Serial.print(rotation1);
  Serial.print(" RPM - ");
  counter1 = 0;  //  reset counter to zero
  Serial.print("Motor Speed 2: ");
  float rotation2 = (counter2 / diskslots) * 60.00;  // calculate RPM for Motor 2
  Serial.print(rotation2);
  Serial.println(" RPM");
  counter2 = 0;  //  reset counter to zero
  Timer1.attachInterrupt( ISR_timerone );  // Enable the timer
}

void setup()
```

```
{
  Serial.begin(9600);

  Timer1.initialize(1000000); // set timer for 1sec
  attachInterrupt(digitalPinToInterrupt (MOTOR1), ISR_count1, RISING);  // Increase counter 1 when speed sensor pin goes High
  attachInterrupt(digitalPinToInterrupt (MOTOR2), ISR_count2, RISING);  // Increase counter 2 when speed sensor pin goes High
  Timer1.attachInterrupt( ISR_timerone ); // Enable the timer
}

void loop()
{
  // Nothing in the loop!
  // You can place code here
}
```

After compiling the code and uploading it to the Arduino open your serial monitor. If everything is working you'll see the motor speeds displayed in RPM.

One thing you will notice about this sketch is that the loop doesn't contain any code! This might seem surprising as a standard Arduino Sketch has most of its code within the loop. I did this intentionally to show you that the interrupts are not dependent upon any code within the loop. If you wish try adding some of your own code inside the loop and watch it run while the speed sensors still display the motor speeds.

Robot Car Sketch

Finally we come to the sketch to run our robot car!

```
/*
  Robot Car with Speed Sensor Demonstration
  RobotCarSpeedSensorDemo.ino
  Demonstrates use of Hardware Interrupts
  to control motors on Robot Car

  DroneBot Workshop 2017
  http://dronebotworkshop.com
*/

// Constants for Interrupt Pins
// Change values if not using Arduino Uno

const byte MOTOR_A = 3;  // Motor 2 Interrupt Pin - INT 1 - Right Motor
const byte MOTOR_B = 2;  // Motor 1 Interrupt Pin - INT 0 - Left Motor

// Constant for steps in disk
const float stepcount = 20.00;  // 20 Slots in disk, change if different

// Constant for wheel diameter
const float wheeldiameter = 66.10; // Wheel diameter in millimeters, change if different

// Integers for pulse counters
volatile int counter_A = 0;
volatile int counter_B = 0;


// Motor A
```

```
int enA = 10;
int in1 = 9;
int in2 = 8;

// Motor B

int enB = 5;
int in3 = 7;
int in4 = 6;

// Interrupt Service Routines

// Motor A pulse count ISR
void ISR_countA()
{
  counter_A++;  // increment Motor A counter value
}

// Motor B pulse count ISR
void ISR_countB()
{
  counter_B++;  // increment Motor B counter value
}

// Function to convert from centimeters to steps
int CMtoSteps(float cm) {

  int result;  // Final calculation result
  float circumference = (wheeldiameter * 3.14) / 10; // Calculate wheel
circumference in cm
```

```
  float cm_step = circumference / stepcount;  // CM per Step

  float f_result = cm / cm_step;  // Calculate result as a float
  result = (int) f_result; // Convert to an integer (note this is NOT rounded)

  return result;  // End and return result

}

// Function to Move Forward
void MoveForward(int steps, int mspeed)
{
   counter_A = 0;  //  reset counter A to zero
   counter_B = 0;  //  reset counter B to zero

   // Set Motor A forward
   digitalWrite(in1, HIGH);
   digitalWrite(in2, LOW);

   // Set Motor B forward
   digitalWrite(in3, HIGH);
   digitalWrite(in4, LOW);

   // Go forward until step value is reached
   while (steps > counter_A && steps > counter_B) {

    if (steps > counter_A) {
    analogWrite(enA, mspeed);
    } else {
```

```
    analogWrite(enA, 0);
    }
    if (steps > counter_B) {
    analogWrite(enB, mspeed);
    } else {
    analogWrite(enB, 0);
    }
   }

  // Stop when done
  analogWrite(enA, 0);
  analogWrite(enB, 0);
  counter_A = 0;  //  reset counter A to zero
  counter_B = 0;  //  reset counter B to zero

}

// Function to Move in Reverse
void MoveReverse(int steps, int mspeed)
{
   counter_A = 0;  //  reset counter A to zero
   counter_B = 0;  //  reset counter B to zero

   // Set Motor A reverse
  digitalWrite(in1, LOW);
  digitalWrite(in2, HIGH);

  // Set Motor B reverse
  digitalWrite(in3, LOW);
  digitalWrite(in4, HIGH);


   // Go in reverse until step value is reached
   while (steps > counter_A && steps > counter_B) {

    if (steps > counter_A) {
    analogWrite(enA, mspeed);
    } else {
    analogWrite(enA, 0);
    }
    if (steps > counter_B) {
    analogWrite(enB, mspeed);
    } else {
    analogWrite(enB, 0);
    }
    }

  // Stop when done
  analogWrite(enA, 0);
  analogWrite(enB, 0);
  counter_A = 0;  //  reset counter A to zero
  counter_B = 0;  //  reset counter B to zero

}

// Function to Spin Right
void SpinRight(int steps, int mspeed)
{
   counter_A = 0;  //  reset counter A to zero
   counter_B = 0;  //  reset counter B to zero

   // Set Motor A reverse
  digitalWrite(in1, LOW);
  digitalWrite(in2, HIGH);

  // Set Motor B forward
  digitalWrite(in3, HIGH);
  digitalWrite(in4, LOW);

   // Go until step value is reached
   while (steps > counter_A && steps > counter_B) {

    if (steps > counter_A) {
    analogWrite(enA, mspeed);
    } else {
    analogWrite(enA, 0);
    }
    if (steps > counter_B) {
    analogWrite(enB, mspeed);
    } else {
    analogWrite(enB, 0);
    }
   }

  // Stop when done
  analogWrite(enA, 0);
  analogWrite(enB, 0);
  counter_A = 0;  //  reset counter A to zero
  counter_B = 0;  //  reset counter B to zero

}

// Function to Spin Left
void SpinLeft(int steps, int mspeed)
{
   counter_A = 0;  //  reset counter A to zero
   counter_B = 0;  //  reset counter B to zero

   // Set Motor A forward
  digitalWrite(in1, HIGH);
  digitalWrite(in2, LOW);

  // Set Motor B reverse
  digitalWrite(in3, LOW);
  digitalWrite(in4, HIGH);

   // Go until step value is reached
   while (steps > counter_A && steps > counter_B) {

    if (steps > counter_A) {
    analogWrite(enA, mspeed);
    } else {
    analogWrite(enA, 0);
    }
    if (steps > counter_B) {
    analogWrite(enB, mspeed);
    } else {
```

```
    analogWrite(enB, 0);
    }
  }

  // Stop when done
  analogWrite(enA, 0);
  analogWrite(enB, 0);
  counter_A = 0;  //  reset counter A to zero
  counter_B = 0;  //  reset counter B to zero

}

void setup()
{
  // Attach the Interrupts to their ISR's
  attachInterrupt(digitalPinToInterrupt (MOTOR_A), ISR_countA, RISING);  // Increase counter A when speed sensor pin goes High
  attachInterrupt(digitalPinToInterrupt (MOTOR_B), ISR_countB, RISING);  // Increase counter B when speed sensor pin goes High

  // Test Motor Movement  - Experiment with your own sequences here

  MoveForward(CMtoSteps(50), 255);  // Forward half a metre at 255 speed
  delay(1000);  // Wait one second
  MoveReverse(10, 255);  // Reverse 10 steps at 255 speed
  delay(1000);  // Wait one second
  MoveForward(10, 150);  // Forward 10 steps at 150 speed
  delay(1000);  // Wait one second
  MoveReverse(CMtoSteps(25.4), 200);  // Reverse 25.4 cm at 200 speed
  delay(1000);  // Wait one second
  SpinRight(20, 255);  // Spin right 20 steps at 255 speed
  delay(1000);  // Wait one second
  SpinLeft(60, 175);  // Spin left 60 steps at 175 speed
  delay(1000);  // Wait one second
  MoveForward(1, 255);  // Forward 1 step at 255 speed


}


void loop()
{
  // Put whatever you want here!


}
```

This sketch demonstrates how to run the car and take the input from the two speed sensors to control how far it moves, we can also control the car's direction. Note the use of the "volatile" integers in the pulse counters, this makes sure that the Arduino IDE doesn't "throw away" these variables when it compiles the code.

The code has four functions to control the car:

- **MoveForward** – this moves the car forward
- **MoveReverse** – this moves the car in reverse
- **SpinRight** – this spins the car right
- **SpinLeft** – this spins the car left

Each of the functions has the same input parameters:

- **Steps** – the number of steps in the encoder disk to move the wheel
- **MSpeed** – the speed of the motors, from 0 to 255

The care movement is controlled within the “while” loop inside each of the functions (aside from the motor direction commands all four functions are identical). You’ll note that the movement of each wheel is calculated independently. This is done to compensate for speed differences between the motors.

There is also a function called “CMtoSteps” that can be used to convert centimeters to the number of steps required to move that distance. This will simplify moving the car to a specific location.

Playing with the code

The code at the end of the setup routine is just a series of statements to move the car in various directions, using the previously described functions. You can experiment by adding more functions or by changing the parameters (steps, speed) existing code.

You’ll note that once again I didn’t put anything in the loop, all of the code runs in the setup routine and as such it will run once and finish. You can repeat it by pressing the reset button on the Arduino Uno. If you would prefer that the code runs forever then put the car movement statements into the loop instead.

The code is basically for demonstration purposes and you can build upon it to improve it and to add more functions, such as functions to steer the car at a specific angle. Let me know in the comments if you come up with anything interesting!

The End – Not Really!

This brings us to the end of the article but it's really just the beginning . I'm planning to add more features to our little Robot Car so keep tuned to the site to see the next evolution of the design. A few obvious ones are collision avoidance, line following and remote control.

If there are features you would like to see please let me know in the comments and I'll see what I can do to include them.

Until we meet next time happy motoring!