

# Build your Own Continuous Rotation Servo

---

## Learning Objectives

By the end of this laboratory experiment, the experimenter should be able to:

- Explain how an encoder operates and how it can be used to determine rotational speed and angle of a motor shaft
- Explain the concept of Pulse-Width Modulation (PWM) to control the speed of a DC motor
- Interface the Arduino microcontroller to an encoder
- Explain how to use an encoder with quadrature to obtain the direction of rotation
- Explain the difference between free-spin stopping of a motor and dynamic braking
- Design a simple continuous rotation servo.

## Components

<u>Qty.</u>	<u>Item</u>
1	Arduino with USB cable
1	DC motor mounted in laser-cut acrylic stand with encoder wheel and optical encoder pickup on the front and magnetic encoder with quadrature on the rear
1	Motor driver board (L298-based, such as item 251080674810 from <a href="http://stores.ebay.com/Chip-Partner-Store">http://stores.ebay.com/Chip-Partner-Store</a> )
1	Servo motor adapter board

## Introduction

In this lab you will learn how rotational speed and rotational angle can be determined using a rotary encoder. A rotary encoder consists of a disk with alternating opaque and clear radial regions. It operates by placing a light source on one side of the disk, and a photosensitive device such as a phototransistor on the other side. As the disk rotates, the passage of the opaque and clear regions of the encoder disk alternately block and allow light to impinge on the receiver, producing an alternating stream of voltage pulses. The rotational speed of the encoder disk can be determined by counting pulses on one track over a known period of time, since the number of pulses per revolution is constant. The angle of rotation corresponds directly to the number of pulses and the number of the tracks created along the radial direction of the disk -- more tracks would produce more accurate angle measurement.

You will also be using a magnetic encoder built into the rear of the motor to construct a continuous rotation servo (CRS). A CRS might be useful for such things as CNC (computer numerically controlled) machinery, incremental movement of a conveyor belt, and countless of other applications where rotating over a limited range of fixed angles is not an option. You will investigate some of the challenges and advantages of a closed loop system for precise motion control.

## Hardware and Setup

### **Motor with Encoder**

The mechanical hardware of this lab is shown in Figure 1. It consists of a DC motor with a gearhead to reduce the speed and increase the available torque, a magnetic encoder (integral to the motor), an optical encoder disk, and a slotted opto-interrupter. Roughly a third of the motor assembly length is in the gearhead. There is a magnetic encoder built into the tail end of the motor that uses a Hall effect sensor to determine

the rotation angle of the motor shaft. When a current flows through a conductor, the presence of a transverse magnetic field causes a transverse voltage to appear. This effect was discovered by physicist Edwin Hall in 1879 and bears his name. The encoder on the tail end of the motor has a permanent magnet attached to the motor shaft, which rotates relative to the Hall sensors, which produce voltage signals corresponding to the angle of rotation. The encoder signal uses what is called ‘quadrature’ encoding, which allows not only the speed of rotation to be measured, but also the direction of rotation. Figure 2 demonstrates an example of quadrature output.

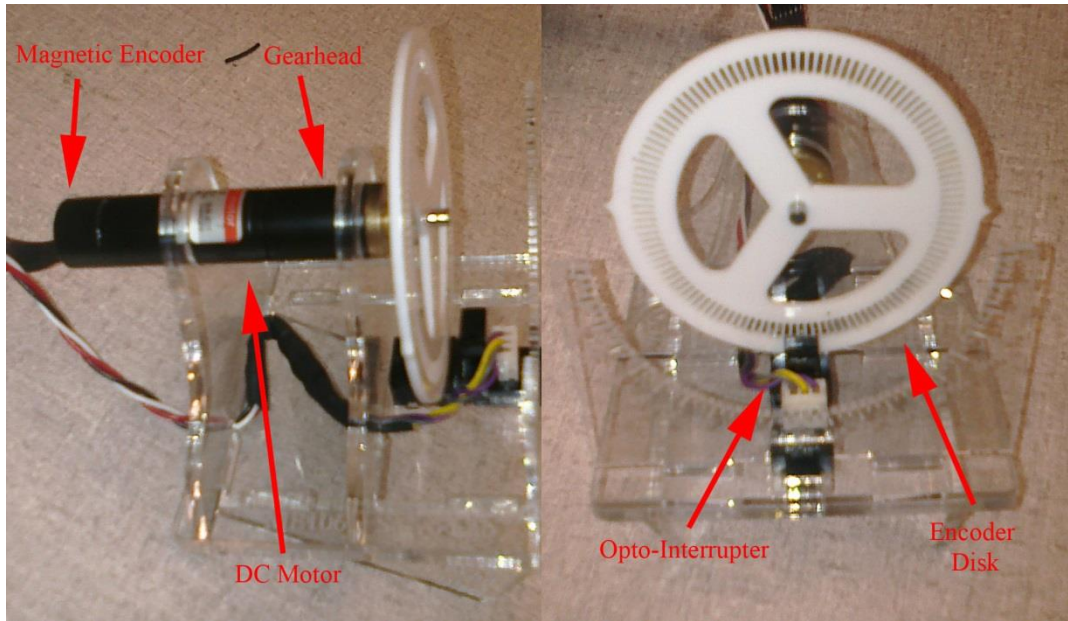


Figure 1: Motor with Encoder. The hardware setup consists of a permanent magnet DC motor (PMDC), a gearhead, a magnetic encoder (integral to the motor), an optical encoder disk, and slotted opto-interrupter

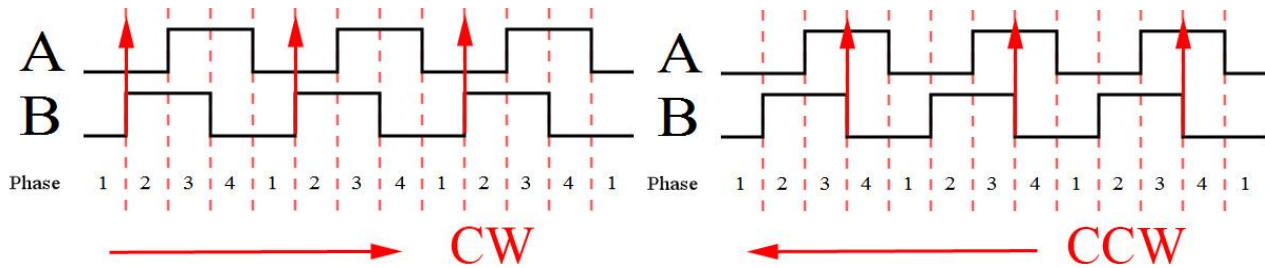


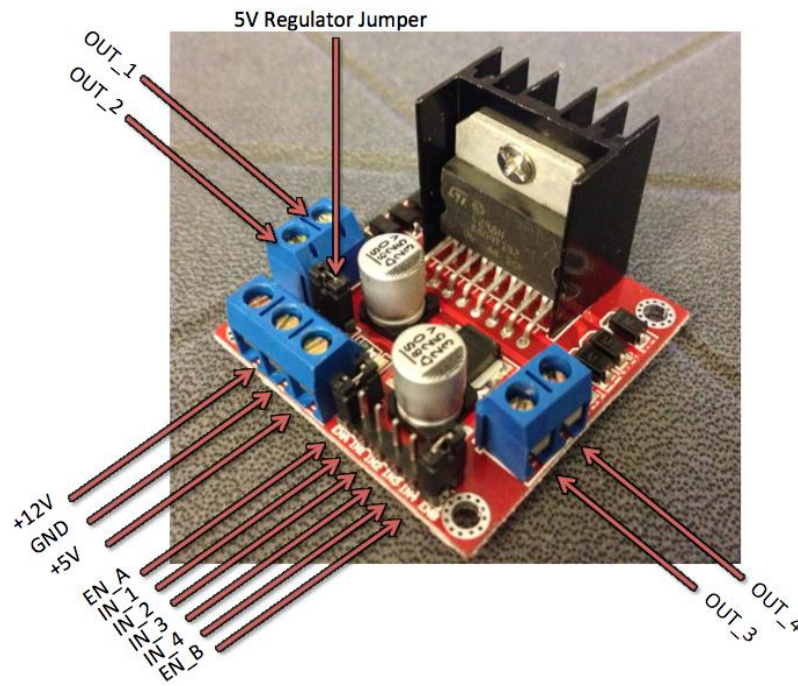
Figure 2 Encoder Quadrature Signal. Encoders typically output two digital signals that are 90 degrees out of phase. The quadrature signal allows the direction of rotation to be determined.

A quadrature signal allows the direction of rotation to be determined if the signal level on one channel is checked when a rising edge occurs on the other channel. Using the example in Figure 2, if the motor is spinning clockwise, the voltage level of channel A will be low when a rising edge occurs on channel B. When the motor spins counterclockwise, the voltage level of channel A will be high when a rising edge occurs on channel B. The way this phase offset is produced is by having two sensors, which are offset in such a way so that the opaque and clear regions of the encoder disk pass by the sensors slightly offset from each other and produce a 90 degree phase shift. This 90 degree offset gives four distinct regions when the encoder is rotated as shown in Figure 2. This particular encoder produces 16 pulses per rotation.

The encoder that is located on the front of the motor consists of a laser-cut opaque acrylic disk that contains 120 windows. As the encoder rotates, pulses are produced using an opto-interrupter rather than magnetically as with the encoder on the end of the motor. **(Lab-report Q1) What advantage might**

there be to having the encoder located on the motor side of the gearhead instead of at the output shaft of the gearhead (in other words so that it would rotate at the speed of the armature rather than at the speed of the output shaft of the gearhead)?

Motor driver board



*Figure 3: Motor Driver Board.* The board uses an L298 Dual-H bridge chip that is capable of driving 2 A (ampere) continuous current up to a maximum power of 25 W.

The method that we will be using for driving the motor as well as interfacing with the Arduino is a motor board containing an H-bridge (L298). As you learned in the previous lab, an H-bridge is useful for bi-directionally driving high-current devices such as motors with a control signal from the Arduino pins. To drive the H-bridge, a high signal (or a PWM signal) on the enable pin is required as well as supplying signals of opposing polarity on the two input drive pins.

The Maxon motors used in this lab have a 2x5 female header for accessing the motor power, encoder logic power, and the two channels of the encoder. The pinout is detailed in **Figure 4**.

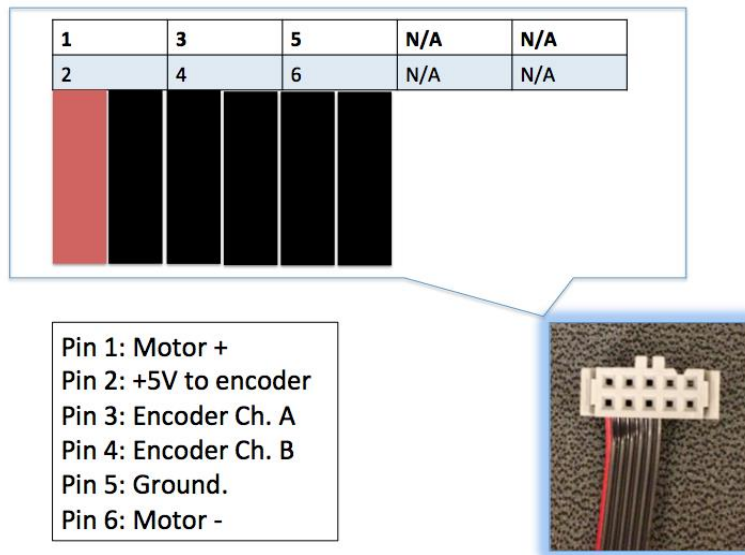


Figure 4: Motor Pinout

Notice that in Figure 4 the power for the motor (12V) is in very close proximity to the logic power of the encoder (5V). **It is critical that you do not have the 12V wires touch the 5V wires; this could result in permanent damage to the motor encoder or motor driver board!** A wiring harness will be used to help avoid this issue, as well as, assist with the setup. The wiring harness is shown in Figure 5 and its pinout in Table 1.

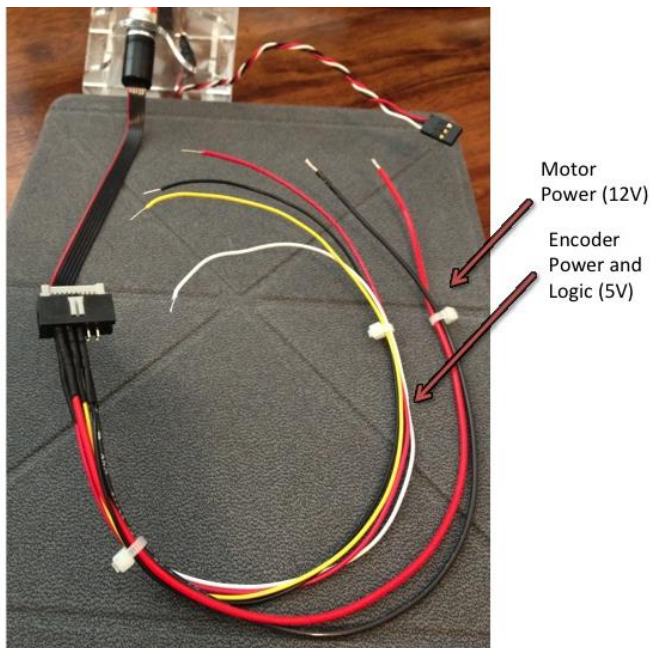


Figure 5: Wiring Harness

Table 1: Wiring harness pinout

Encoder Wires	Motor Wires
Red: +5V	Red: +12V
Black: Gnd	Black: Gnd
Yellow: Channel A	
White: Channel B	

**Introduction to Interrupts**

An *interrupt handler* (also called an *ISR*, or *interrupt service routine*) is a function created to handle a particular event, like the change in state of a pin from high-to-low, or low-to-high. Enabling interrupts allows the microcontroller itself, rather than your software, to monitor the state of an input pin. When the pin of interest changes state, the microcontroller freezes the execution of your main program and immediately executes the code contained within the interrupt handler, and then returns to running your mainline code. After the interrupt handler function has finished, control returns to the exact point in the main program where it was before the interrupt



handler was called. This makes it almost appear that two pieces of code are running in parallel, but in fact the main code is suspended for the interrupt handler to do its thing, without the original code knowing something else happened behind its back. The alternative to using interrupts is what you are used to, namely constantly monitoring one or more pins in a loop, doing nothing until a pin changes. This method of looping to check for (any or all) possible input changes is called *polling*.

An example of where interrupts (especially external interrupts) are useful is for an emergency stop button: No matter what the program is doing at the time that the emergency stop button is pressed, the specified interrupt handler is called, which in this case could be made to halt the machine immediately. This alleviates your mainline code from having to poll the emergency stop button often, which can greatly simplify the code for the rest of your application.

As it turns out, there are two types of interrupts supported by the ATmega328 used in the Arduino – *external* and *pin-change* interrupts. While both fundamentally work the same way, the only significant difference between the two types of interrupts are that external interrupts can occur only on a few select pins on the microcontroller (D2 and D3 on the Arduino Uno), whereas pin-change interrupts can occur on any pin. Pin-change interrupts are a relatively new thing on ATmega processors that were a necessary development, because having only two old-fashioned external interrupts was a huge limitation. If you ever find yourself needing more interrupt pins on your Uno, you can use a custom library called PinChangeInt. Suffice it to say that if you choose to use interrupts for your project, you need to understand that only D2 and D3 are directly usable with the standard Arduino library. For this lab we only need two interrupt pins, so we will not bother with the custom library.

The basic way to get started using interrupts is to attach a function to a specified type of state change on a specified pin. This is the function that is automatically called when the change occurs. Otherwise, there are two very basic things to do when using interrupts. First, data (variables) that are referenced in both the interrupt handler and your main body of code must be global and declared volatile<sup>1</sup>. You will see examples of this in the provided code for this lab. Second, the code within an interrupt handler should generally be kept very short, because the more time spent running your ISR code is time *not* spent running your main code, as well as code within other interrupt handlers. If you spend too much time running code in an ISR, you stand the chance of missing other interrupts, and this can be a huge problem for your system if it does not take this possibility into account. The most difficult thing to learn/understand when using interrupts is that it causes things to happen “behind the back” of your main code, so you have to keep that in mind when designing/implementing the remaining code in your application.

## Setup

1. Upload the example sketch, BareMinimum included in the Arduino IDE to the Arduino board. Located in File > Examples > Basic > BareMinimum. This is so that when you attach your motor driver, you will be sure that you are not telling it to do something you don't want it to be doing.
2. Be sure that the jumpers are removed from the 5V regulator AND Enable A headers (refer to Figure 3 if you are unsure where these are located).
3. Make connections as shown in Figure 6.
  - 3.1. Attach the *stranded* (**not** solid core) red and black wires from the wiring harness to the OUT1 and OUT2 terminal block of the motor driver board respectively.

---

<sup>1</sup> Why this is the case is somewhat beyond the scope of this course, because it has to do with compilers and CPU architecture, and we won't get into it here.

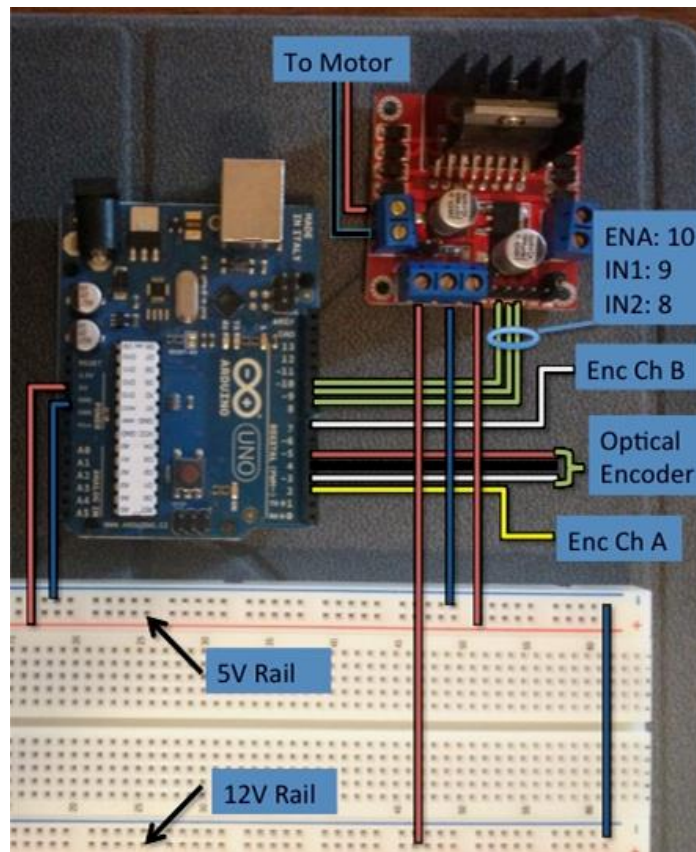


Figure 6: Wiring Diagram

- 3.2. Use the female-to-male ribbon cable to connect the pins ENA, IN1, and IN2 on the driver board to pins 10, 9, and 8 on the Arduino respectively.
- 3.3. Connect Arduino pins “5V” and “GND” to the 5V rail of the breadboard.
- 3.4. Connect the white, black, and red wires from the **optical** encoder attached to the motor stand to pins 3, 4, and 5 of the Arduino respectively. Note: pin 4 will be written LOW, and pin 5 will be written HIGH to serve as a ground and power source.
- 3.5. Attach the *solid core* red and black wires from the wiring harness (the ones bundled to the white and yellow wires) to your 5V rail on the breadboard. Now attach the yellow wire to pin 2 and the white wire to pin 7 of the Arduino.
- 3.6. **With the Arduino and power supply powered off**, run wires from the 5V and GND terminals of the motor driver to the 5V rail of the breadboard.
  - 3.6.1. Run a wire from the +12 terminal of the driver board to your 12V rail of the breadboard.
  - 3.6.2. Use a wire to jump the grounds together from each rail of the breadboard (see the lower right corner of Figure 6).
- 3.7. Turn on the DC power supply, and turn the +-20 V knob until the display reads 12V on the +20V terminal. **Return the power supply to the off position** and connect leads to your breadboard’s 12V rail.
- 3.8. **Have the TA approve your setup before proceeding.**

**Exercise 1**

To test to make sure that all of the encoders are working properly, download the code into the Arduino from Appendix A. Be sure to take note of the pin-change interrupt code (include files, global data, and related functions), as you will be using them for the remainder of the lab.

Once the code is downloaded, open the Serial Monitor, and manually turn the front encoder wheel in one direction, until 1000 counts on the front encoder is reached. **(Lab-report Q2) Use the counts from the front and rear encoders to calculate the gear ratio of the gearhead on the motor, and record the value in your lab report.** (Keep in mind that there are 120 counts per rotation on the front encoder and 16 counts per rotation on the back encoder.)

**Exercise 2**

**Write a short program that will calculate and print the speed of the motor in RPM based on the readings from the front encoder ( (Lab-report Q3) include only the parts you wrote for your lab report).** Also have your program record the maximum speed achieved and **state it in your report.**

**Exercise 3**

Run the motor at different duty cycles and **(Lab-report Q4)** record the speed from the serial monitor

<b>% Duty Cycle</b>	<b>Speed in RPM</b>
<b>100</b>	
<b>80</b>	
<b>60</b>	
<b>40</b>	
<b>20</b>	

**(Lab-report Q5) Explain why the motor speeds vary with the duty cycle (if you don't know ask your TA).**

**Exercise 4**

Many times it is important to know what the minimum power required is to turn a motor under a known load. Write a short program that will sweep through a duty cycle at short intervals from 0 to 20 percent to determine the torque required to get the motor to start moving. **(Lab-report Q6) Record the lowest value at which the motor begins to rotate reliably.**

**Exercise 5**

Another important factor to consider when programming a motor to do precise movements is the amount that the motor will overshoot its intended target due to inertia in the rotor as well as other factors, such as the load that it is driving.

There are several methods of braking that can be used to stop a motor. The first and most obvious is just shutting off the power to the motor (setting the enable pin low) and letting it coast to a stop. The second method of braking, involves basically shorting the coils of the motor together to use the back EMF to slow the motor down. This method of braking involves setting the enable pin to high and setting both motor drive inputs to high. The last method of braking involves actively reversing the motor for a short period of time to more aggressively provide a counter torque, yet not causing the motor to move in the reverse direction. It is more difficult to use this reversing technique, because

the duration of the reversal period varies with the actual momentum of the motor at the time of braking. As such, it is *very* difficult to get right without using some pretty involved software and a lot of experimental tweaking.

**In this exercise, write a program that will bring the motor up to speed, then impose coasting and active braking techniques. (Lab-report Q7) Record the overshoot in (quadrature *not* window) counts.**

In summary:

Coasting = Setting enable LOW

Active Braking = Keeping enable HIGH AND setting both IN pins HIGH

% Duty Cycle	Coasting	Active braking
<b>100</b>		
<b>30</b>		
<b>Your lowest duty cycle</b>		

## Exercise 6

**In this exercise, write a function that takes an angle as a parameter and rotates the shaft the correct number of degrees and include the code in your report (Lab-report Q8).** Just like the *Servo.write(angle)* function in the Arduino library.

You must use the magnetic encoder on the back of the motor for this exercise. You will need to make use of everything you have done earlier to accomplish this task. Hint: you should run the motor near the lowest possible measured duty cycle, employ active braking, and use the quadrature values to determine direction and the actual angle of the shaft to correct for overruns.

Demonstrate your function to your TA by running the test code below.

```

Loop() {
  MyServoFunc(180);
  delay(2000);
  MyServoFunc(360);
  delay(2000);
  MyServoFunc(0);
  delay(2000);
  MyServoFunc(-180);
  delay(2000);
  MyServoFunc(0);
  delay(2000);
}

```

**(Lab-report Q9) Summary of this lab.**

Repeat exercise 5 using the active reversing technique discussed.



<b>% Duty Cycle</b>	<b>Active Reversing</b>
<b>100</b>	
<b>30</b>	
<b>Your lowest duty cycle</b>	

## **Appendix A**

```

// encoder pins
#define PIN_ENC0 2
#define PIN_ENC1 7
#define PIN_OPTO 3

//power and ground for opto enc
#define OPTO_GND 4
#define OPTO_POWER 5

//H-bridge pins
#define ENB 10
#define IN1 9
#define IN2 8

//Encoder counting variables
volatile uint16_t optoCount=0;
volatile int16_t encCount=0;

void opto_isr()
{
  optoCount++;
}

void enc_isr()
{
  if (digitalRead(PIN_ENC1) == HIGH)
    encCount++;
  else encCount--;
}

void setup()
{
  pinMode(PIN_OPTO, INPUT);
  pinMode(PIN_ENC0, INPUT);
  pinMode(PIN_ENC1, INPUT);

  // setup power and ground
  pinMode(OPTO_POWER,OUTPUT);
  pinMode(OPTO_GND,OUTPUT);
  digitalWrite(OPTO_POWER,HIGH);
  digitalWrite(OPTO_GND,LOW);

  // attach interrupts
  attachInterrupt(1, opto_isr, RISING);
  attachInterrupt(0, enc_isr, RISING);

  Serial.begin(9600);

```

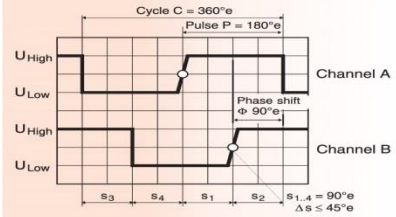
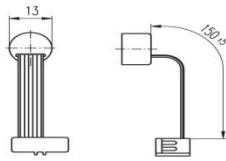
## Encoder Laboratory

```
}  
  
void loop()  
{  
  Serial.print ("Opto >> ");  
  Serial.print (optoCount);  
  Serial.print (" Motor Enc >> ");  
  Serial.println(encCount);  
}
```

**APPENDIX B**

**Encoder MEnc 13, 16 Counts per turn, 2 Channels**

maxon tachometer



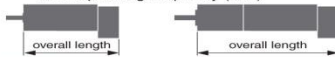
- Stock program
- Standard program
- Special program (on request)

**Order Number**

110778

**Type**

Counts per turn	16
Number of channels	2
Max. operating frequency (kHz)	20



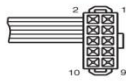
**Combination**

+ Motor	Page	+ Gearhead	Page	Overall length [mm] / see: + Gearhead
RE 13, 0.75 W	58/59			27.0 / 29.4
RE 13, 0.75 W	59	GP 13, 0.05 - 0.15 Nm	217	•
RE 13, 0.75 W	59	GP 13, 0.2 - 0.35 Nm	218	•
RE 13, 2 W	62/63			39.2 / 41.6
RE 13, 2 W	63	GP 13, 0.05 - 0.15 Nm	217	•
RE 13, 2 W	63	GP 13, 0.2 - 0.35 Nm	218	•
RE 13, 1.5 W	66/67			30.3 / 32.7
RE 13, 1.5 W	67	GP 13, 0.05 - 0.15 Nm	217	•
RE 13, 1.5 W	67	GP 13, 0.2 - 0.35 Nm	218	•
RE 13, 3 W	70/71			42.5 / 44.9
RE 13, 3 W	71	GP 13, 0.05 - 0.15 Nm	217	•
RE 13, 3 W	71	GP 13, 0.2 - 0.35 Nm	218	•
RE 16, 3.2 W	74			46.5
RE 16, 3.2 W	74	GP 16, 0.06 - 0.18 Nm	223	•
RE 16, 3.2 W	74	GP 16, 0.1 - 0.3 Nm	224	•
RE 16, 4.5 W	76			49.5
RE 16, 4.5 W	76	GP 16, 0.06 - 0.18 Nm	223	•
RE 16, 4.5 W	76	GP 16, 0.1 - 0.3 Nm	224	•
A 2516, 0.8 W	88			24.7
A 2516, 0.8 W	88	GS 16, 0.01 - 0.03 Nm	219	•
A 2516, 0.8 W	88	GS 16, 0.01 - 0.03 Nm	220	•
A 2516, 0.8 W	88	GS 16, 0.06 - 0.1 Nm	221/222	•
A 2520, 1.2 W	89			25.6
A 2520, 1.2 W	89	GP 19, 0.1 - 0.3 Nm	226	•
GM 20, 1.2 W	90	GS 20 (integriert)	90	28.5
A-max 16	104/106			33.5
A-max 16	104/106	GS 16, 0.01 - 0.03 Nm	219/220	•
A-max 16	104/106	GS 16, 0.06 - 0.1 Nm	221/222	•
A-max 16	104/106	GP 16, 0.06 - 0.18 Nm	223	•
A-max 16	104/106	GP 16, 0.1 - 0.3 Nm	224	•
A-max 19	108/110			36.4 / 39.0
A-max 19	108/110	GP 19, 0.1 - 0.3 Nm	226	•
A-max 19	108/110	GS 20, 0.06 - 0.25 Nm	227	•
A-max 19	108/110	GP 22, 0.5 - 2.0 Nm	230/231	•
A-max 19	108/110	GS 24, 0.1 Nm	234	•
A-max 22	112/114			39.0
A-max 22	112/114	GP 22, 0.1 - 0.6 Nm	228/229	•
A-max 22	112/114	GP 22, 0.5 - 2.0 Nm	230/231	•
A-max 22	112/114	GS 24, 0.1 Nm	234	•
A-max 26	115-121			51.8
A-max 26	115-121	GP 26, 0.5 - 2.0 Nm	235	•
A-max 26	115-121	GS 30, 0.07 - 0.2 Nm	236	•
A-max 26	115-121	GP 32, 0.4 - 2.0 Nm	237	•
A-max 26	115-121	GP 32, 0.75 - 6.0 Nm	237/238	•
A-max 26	115-121	GS 38, 0.1 - 0.6 Nm	243	•

**Technical Data**

Supply voltage V <sub>CC</sub>	3.8 - 24 V
Output signal V <sub>CC</sub> = 5 VDC	TTL compatible
Phase shift	90° ± 45°e
Power input at V <sub>CC</sub> 5 VDC	max. 8 mA
Inertia of the magnetic disc	0.07 gcm <sup>2</sup>
Operating temperature range	-20 ... +80°C

**Pin Allocation**



- 1 Motor+
  - 2 V<sub>CC</sub>
  - 3 Channel A
  - 4 Channel B
  - 5 GND
  - 6 Motor-
- Pin type DIN 41651  
(Type SM 89110-0101HA)  
flat band cable AWG 28