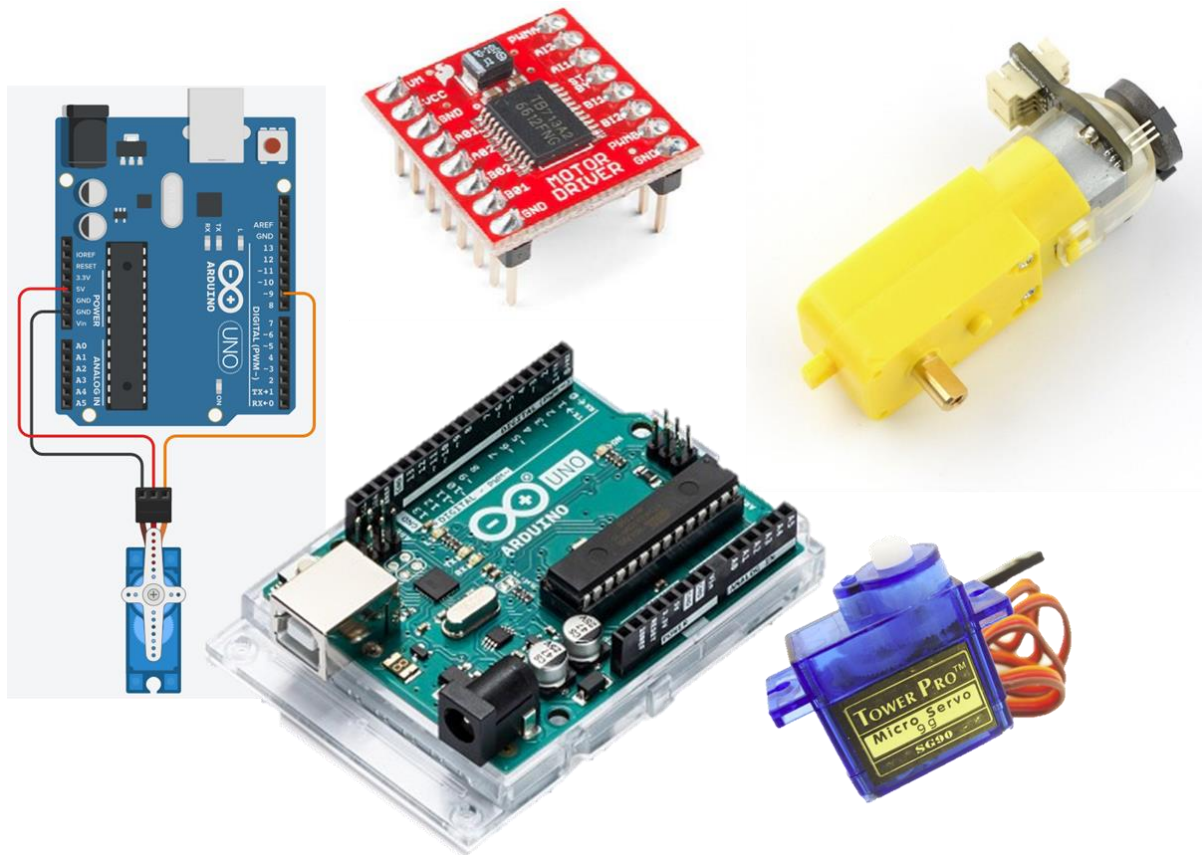


Introduction to Arduino – PWM Control of Actuators

Ben Taylor
Shuhe Miyashita and Dana Damian



Learning Outcomes

- | | |
|-----------|---|
| Practical | Program a microcontroller to control the average voltage supplied to an actuator, using a PWM signal and power electronic circuit. |
| Practical | Program the PWM module of an Arduino to generate a PWM signal with a specified period and on-time, to send a parameter signal to an external system. |
| Produce | Be able to write programs to instruct a microcontroller system to receive data from analogue/digital sensors, perform control algorithms to generate an |

appropriate PWM control signal to drive different actuator systems, to achieve the desired result

Introduction to Arduino – PWM control of Actuators: In-Laboratory Activities

1 Aims and Objectives

The aims of this laboratory session are to drive different type of actuators using a PWM output from an Arduino device, and use sensors to adjust the behaviour of a system.


During the laboratory session, you will use a PWM output of the Arduino in two different ways:

1. The PWM can be used to generate an average voltage, whose magnitude is the ratio of the t_{on}/t_s time, multiplied by the maximum output voltage of the switching device – see background section, below.
2. The PWM module is used to generate a pulse train with a specified period. The on-time, t_{on} , of the PWM has a specific range, and is used to send a parameter value to the device being controlled -see background section, below.

You will use the PWM output of the Arduino, as described above, to drive different types of actuator, building on knowledge you gained from the previous practical exercises.

2 Names of collaborators for the experiment

Use this space to record the member of the group you are conducting the experiment with.

Name of Lab partner(s)	Lab Desk Number
	

3 Background

Pulse Width Modulation, PWM, is a way of encoding information into a rectangular pulse train. Generally, the period, (and hence the frequency), of the pulse train is maintained constant, and the width of the rectangular pulses are varied – we will assume this definition for the remainder of these laboratory sessions.

Figure 1a illustrated the time domain waveform of the PWM signal, where the digital waveform has a rising edge at $t=t_0$, and a falling edge at $t=t_2$. The on-state of the PWM signal is defined as t_{on} , the off-state of the signal is defined as t_{off} , and the PWM period is defined as T_s .

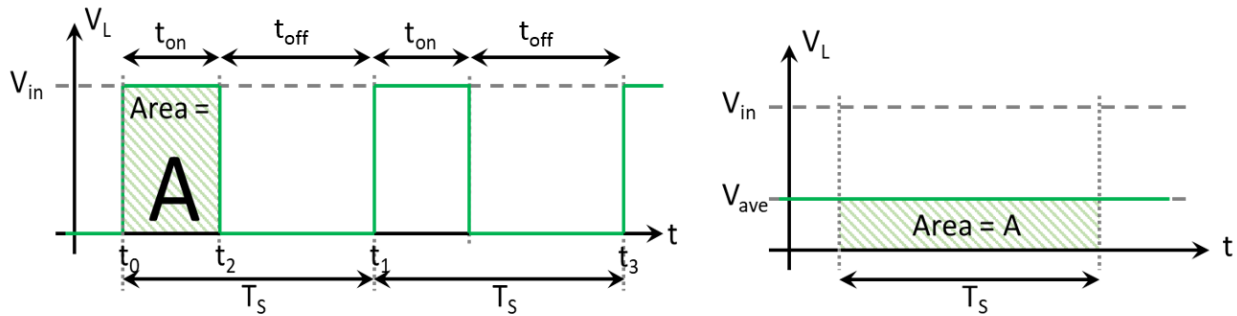


Figure 1. Pulse Width Modulation, (left) time domain waveforms, (right) average voltage of PWM signal.

PWM can be used in a number of different ways; the most common being to ‘chop’ a DC voltage to reduce its average value, as illustrated graphically in Figure 1. The input voltage, V_{in} is ‘chopped’, using a converter circuit, into pulses, as shown in Figure 1 (left), where the time integral of the on-state of the waveform is $t_{on} \times V_{in}$, and has a graphical area of A . The filtering effect of the system we are driving will effectively spread the area under the pulse across in, Figure 1 (left), across the switching period, T_s , as illustrated in Figure 1 (right). This can be expressed as an integral equation:

$$V_{ave} = \frac{1}{T_s} \int_{t_0}^{t_2} V_{in} dt = \frac{t_{on}}{T_s} V_{in} \quad \text{Equation 1}$$

where: V_{in} is the supply voltage, and V_{ave} is the average value of the waveform.

This method of operation is used in most electrical power converters to control the power supply to downstream system, such as a motor.

Another way in which PWM can be used, is to encode some information as a function of the on-state time, t_{on} , whilst maintaining a constant switching period, T_s . The PWM signal is then decoded by the downstream system and a decision is made on its value.

During the laboratory session, you will be driving two different types of actuators:

1. A DC motor - to control the speed of rotation, you will need to control the PWM to regulator the average voltage being fed to the DC motor. To interface the motor to the Arduino, you will need to use a motor drive circuit, to supply the current requirements of the motor.
2. A standard servomotor - this servomotor will move between 0 and 180 degrees and the angle can be controlled through a 5V pulse signal between 1-2ms and an embedded potentiometer

The DC motor is different from the standard servomotor, in terms of the type of control. For servomotors, the average value of the PWM signal is not being used to regulate the average supply voltage to the device, like it is for DC motors, but instead, the PWM on-time is bounded between 2 values, (1ms and 2ms in this case), and is an encoded signal relating to a rotational position demand parameter for the servomotor, closed loop, controller.

4 Laboratory Exercises

4.1 Exercise 1: Control LED Intensity Example

In the Arduino Digital I/O practical, you coded LEDs to turn on and off using the *digitalWrite* function.

In this example, we will use the `analogWrite()` function to generate a PWM signal, which we will use to vary the voltage supplied to an LED, and hence the LED intensity.

The PWM signal generated by the `analogWrite()` function is illustrated in Figure 2. When a variable value between 0 and 255 is passed to the `analogWrite()` function, and the resulting PWM signal is modified between 0% and 100% duty cycle.

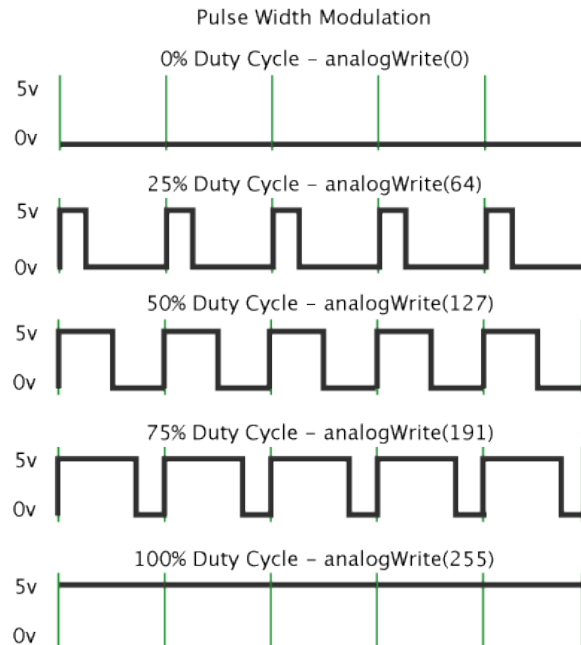


Figure 2. Illustration of the use of `analogWrite()` function, and the resulting PWM output.

(Figure 2 is taken from: https://commons.wikimedia.org/wiki/File:Pwm_5steps.gif)

The Arduino language reference page for the `analogWrite` command can be found at:

<https://www.arduino.cc/reference/en/language/functions/analog-io/analogwrite/>

The circuit shown in Figure 3 will be used for this exercise, where:

- The anode of the LED is connected to pin 11 of the Arduino.
- The cathode of the LED is connected to the ground, through a 470Ω resistor.

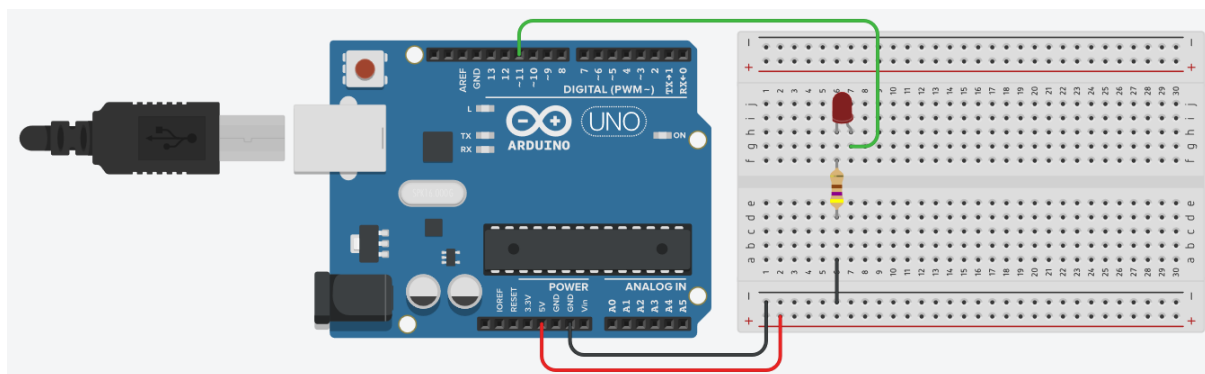


Figure 3. Test circuit for Exercise 1.

Procedure:

1. Construct the circuit for this exercise, as shown in Figure 3.
2. Download the `LED_Fade_Example.ino` sketch from the Blackboard folder for this practical.

3. Run the program and observe the effects on the LED of the different values being passed to the analogWrite function.

4.2 Exercise 2: Controlling a Standard Servomotor

The aim of this exercise is to demonstrate the control of a position control servo, in this case the SG90 servomotor. The SG90 is a very low power servo motor and can be driven directly from the Arduino board power supply.

Note: Servomotors can drain a lot of current, especially when changing direction. When multiple servos are connected, it is better to use an external power supply (battery or AC-DC Plug-in Power Supply) to avoid either damaging the Arduino board or to problems caused by the Arduino USB connection limiting the available current on the 5V power supply.

See the document “Powering A Servo Using the External Power Supply”, located in the Blackboard Folder, for more details on providing an external power supply to power larger servomotors or when controlling several smaller servomotors.

During this exercise, we will use the circuit shown in Figure 1, where:

- The red wire of the servo is the positive supply, and is connected to the 5V of the Arduino
- The brown wire is the negative supply and should be connected to the ground line.
- The orange wire is the control line and should be connected to a PWM enabled I/O pin on the Arduino board, in this example, pin 9.

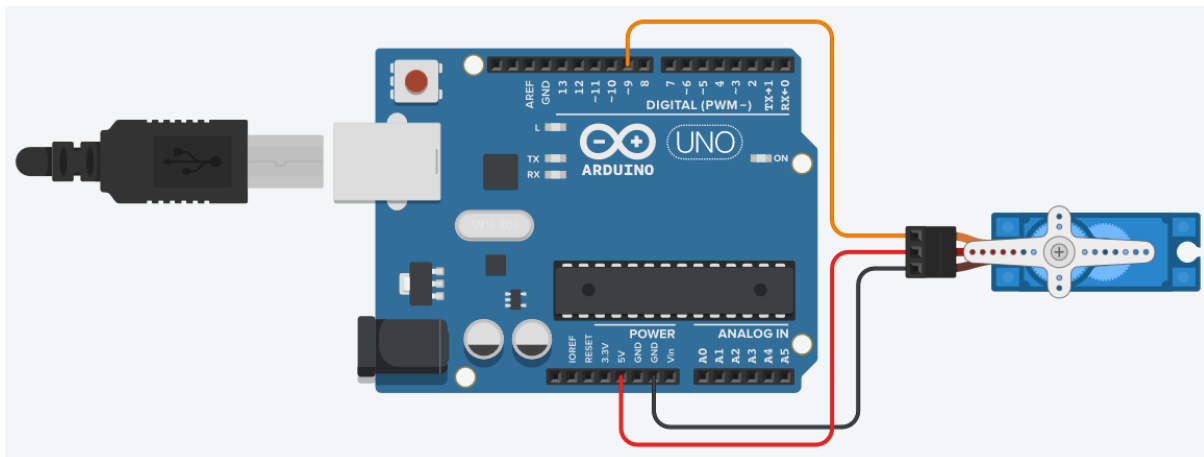


Figure 4. Circuit diagram for Exercise 2.

Note: connecting the wires directly into the servo connector will result in a weak mechanical connection. This connection should be sufficient for this exercise, but if you wish for a more robust connection, you can push the male header pins into the breadboard and use this as intermediate connection.

Procedure:

1. Connect the circuit shown in Figure 4
2. Download the Standard_Servo_Example.ino code from the Blackboard folder for this practical.

Looking at the code, you will see that the servo library had been included in the sketch. This external library contains several functions that allow the Arduino to control a variety of servo motors. The Arduino reference library webpage for the servo library is located here:

<https://www.arduino.cc/reference/en/libraries/servo/>

Before you can use the servo control commands in your code, you must first perform the following operations in your code, (look at the code to see how this has been done):

- In the global namespace area:
 - Include the servo library
 - Declare a servomotor class variable. This is used by the Arduino IDE to direct your commands to the correct servo motor.
- In the setup() function:
 - The servo motor class needs to be linked to an output pin. This is achieved using the attach() function.

In Standard_Servo_Example.ino we have declared a servo class variable as “servo1”, we have defined I/O pin 9 as the servo control pin, and we attached the servo1 class to pin 9, using the following command:

```
servo1.attach(pinServo1);
```

The servomotor is controlled using an encoded PWM signal. The on time of the PWM signal should be bounded between 1000µs and 2000µs, with a pulse repetition rate between 40 and 200Hz, i.e. a period between 5ms and 25ms.

The servo centre position, or ‘neutral’ position, of the servo is commanded using a 1500µs pulse width. Typically, the 90° clockwise position is commanded with a pulse width of approximately 500µs, and the 90° anticlockwise position is commanded with a pulse width of approximately 2500µs. These 90° position pulse widths are often slightly different for each servo model, so you may need to refine these timings for servos you have been provided.

The servo library has two functions for controlling the position of the servo:

- write(), which directly commands the servo angle.
- writeMicroseconds(), which commands a specifically timed pulse width for the servo output.

The write() function may not be accurate for the servomotors you have been provided, and may vary between different models. You should investigate the accuracy of these commands when using the write() and writeMicroseconds() for applications which require accurate output angles.

3. Look at the Standard_Servo_Example.ino code, and see how the servo library has been used, how a servo is declared and attached to the system.
4. Run the sketch and observe its operation
5. Write a program to map the rotational angle of the potentiometer into the rotational angle of the servo, such that:
 - a. The potentiometer measurement is mapped to its rotation angle, (approximately -150° to +150°, across the measurement range).
 - b. The centre point of the potentiometer, 0°, should command the neutral position of the servo.
 - c. The extreme clockwise rotation of the potentiometer, (approx. +150°), should command the servo to move to the extreme position in the clockwise direction, (approx. +90°).
 - d. The extreme anticlockwise rotation of the potentiometer, (approx. -150°), should command the servo to move to the extreme position in the anticlockwise direction, (approx. -90°).
 - e. Stream the potentiometer angle measurement and the servo command angle to serial monitor, in a neatly formatted message.
 - f. You should use the MG996 servo, powered from the external power supply, when demonstrating this experiment. (See “Powering A Servo Using the External Power Supply” as a guide for connecting the MG996 servo).

Note: This Exercise will need to be demonstrated in the final video assignment. See the Introduction to Arduino assignment brief on the Blackboard site for more details.

4.3 Exercise 3: Controlling a DC motor, using a driver board

The aim of this exercise is to control a DC motor from an Arduino, using a DC motor driver interface.

A DC motor cannot be powered directly from the pins of the Arduino board, because the current drawn by the DC motor is too large and could damage the Arduino by overloading the output pins. To overcome this problem, a power electronic driver circuit is required to interface the motor to the Arduino microcontroller board. During this exercise, you will be using a TB6612FNG motor driver breakout board, as shown in Figure 5, and an external power supply, to drive motor from control signals generated in the Arduino.

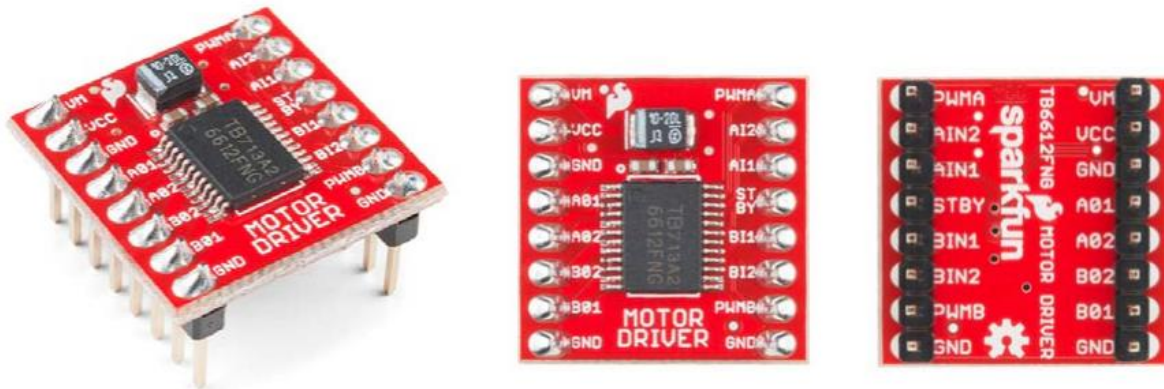


Figure 5. The TB6612FNG motor driver breakout board

The TB6612FNG driver IC chip is a dual brushed DC motor driver chip, capable of delivering 1.2A of average current to two independent motors. The TB6612FNG breakout board, shown in Figure 5 is connected to the Arduino using the circuit shown in Figure 6, where:

- The GND/0V connections for the Arduino, TB6612FNG and the external power supply should be linked together on the negative power rail of the breadboard.
- The standby pin, on the TB6612FNG driver board, should be connected to I/O pin 9 on the Arduino
- The AI1 and AI2 direction control pins on the TB6612FNG driver board should be connected to I/O pins 7 and 8, respectively, on the Arduino
- The PWMA input on the TB6612FNG driver board, should be connected to I/O pin 5 on the Arduino
- The V_m pin on the TB6612FNG driver board, is the motor power circuit supply, and must be connected to the +V from the external power supply – connecting this to the Arduino +5V may cause excess current draw from the Arduino, leading to brown out errors.
- The V_{cc} pin of the on the TB6612FNG driver board should be connected to a 5V power supply. This can either be the +5V external supply or the +5V pin on the Arduino. For convenience, we will use the external power supply +5V.
- The Pin out for the external DC power entry socket can be found in the “Using the TB6612FNG breakout board” document. Ensure the GND connection is connected to the common GND/0V rail on the breadboard, and the +5V connected to the +5V rail, as shown in Figure 6.

The connections, above, describe connecting the motor and Arduino to A channel of the TB6612FNG driver board. A second motor can be connected to channel B, for applications such as 2 wheeled robots, or anything else requiring 2 DC motors.

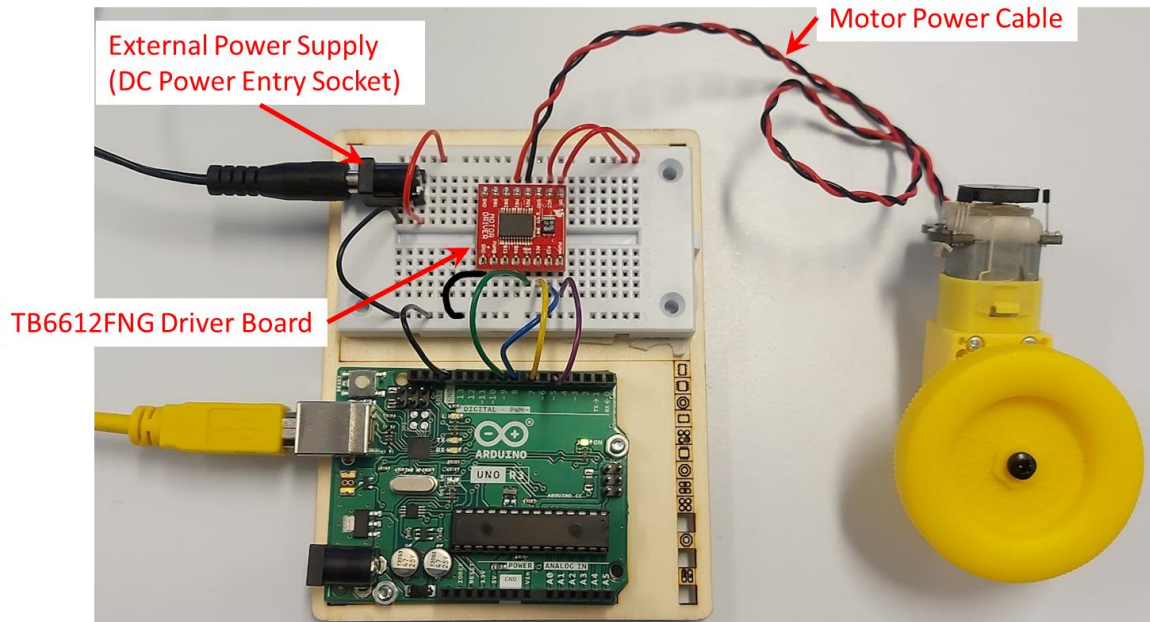


Figure 6. Circuit for connecting the DC motor power connections to the TB6612FNG driver board and Arduino

The document 'Using the TB6612FNG breakout board' is a brief introduction to using the TB6612FNG with an Arduino, and is available from the Blackboard folder for this activity. This document provides more details of the operation of the device, and links to online documentation for this device and general DC motor driver applications.

Procedure:

1. Read the document: "Using the TB6612FNG breakout board"
2. Connect the circuit illustrated in Figure 6, and described above.
3. Download the TB6612FNG_Driver_Board_Demo.ino from the from the Blackboard folder for this practical.
4. Read the code, and inline comments. Run the code, and observe its operation.
5. Write a program that can control the speed of the DC motor using a potentiometer input.
 - The motor should rotate at maximum speed in a clockwise direction, when viewed from the shaft end, when the potentiometer is rotated to the extreme clockwise position.
 - The motor should rotate at maximum speed in an anticlockwise direction, when viewed from the shaft end, when the potentiometer is rotated to the extreme anticlockwise position.
 - When the potentiometer is in the centre position the motor shaft should be at zero speed, (stopped)
 - The speed of the motor should vary in a linear fashion between maximum speed, when the potentiometer is at its extreme rotation, and zero speed when the potentiometer is in the centre position.
 - You should display the following values on the serial monitor terminal, (with a similar text description to TB6612FNG_Driver_Board_Demo.ino):
 - i. ADC measurement value for the Potentiometer input
 - ii. PWM value, written to the analogWrite command
 - iii. Boolean values for AI1, AI2 and Standby

You can use/modify the TB6612FNG_Driver_Board_Demo.ino or other example sketches provided for this course to generate your solution for this exercise.

Note: This Exercise will need to be demonstrated in the final video assignment. See the Introduction to Arduino assignment brief on the Blackboard site for more details.

4.4 Exercise 4: Reading the output of a magnetic rotary encoder

Note: This is a formative exercise and will not be assessed, but you may find it useful to complete, to help you with your group project.

To control the rotation angle of the shaft of a motor, encoders are usually used. These devices can be absolute (there is a 1 to1 mapping between the angle and the output of the encoder) and incremental (the output consists of pulses that need to be counted to keep track of the rotation). Encoders also vary in sensing technologies, the most common being optical and magnetic. Throughout this exercise, we will be using the latter.

Starting from the circuit in the previous exercise, four additional connections are needed, as detailed in Figure 7.

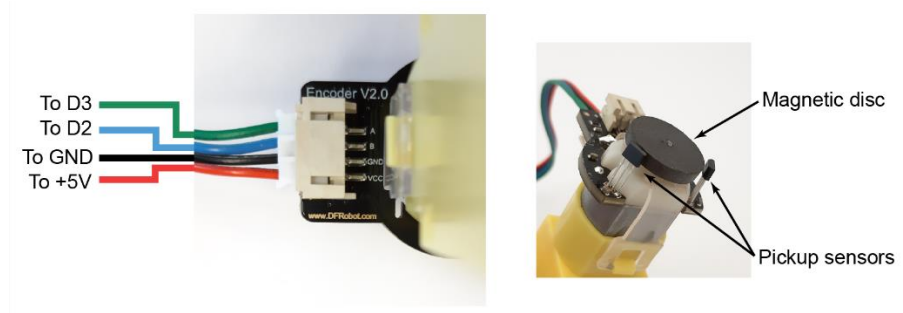


Figure 7. Connections between the encoder and the Arduino and components of the encoder.

With reference to Figure 7, the quadrature encoder consists of the black magnetic disc, connected to the shaft of the DC motor, the two magnetic pickup sensors and the additional circuitry to convert the output to square waves.

The encoder has two outputs, A and B, one for each magnetic pickup sensor. Each time a magnetic section of the disc passes by one of the pickups, the pickup output changes from 0 to 1 and back to 0, resulting in a square signal per output channel. The sensors are offset 90° along the circumference and, as a result, outputs A and B are 90 out of phase with respect to each other. This difference in phase is crucial in determining the direction of rotation of the disc.

The number of magnetic sections of the disc determines the number of “pulses per revolution”, which is a characteristic of each encoder. This is important in determining the minimum detectable angle of rotation and, therefore, the accuracy of the system overall. The encoders in the kit output 32 pulses per revolution of the disc; that means $360^\circ/32^\circ = 11.25^\circ$ per pulse.

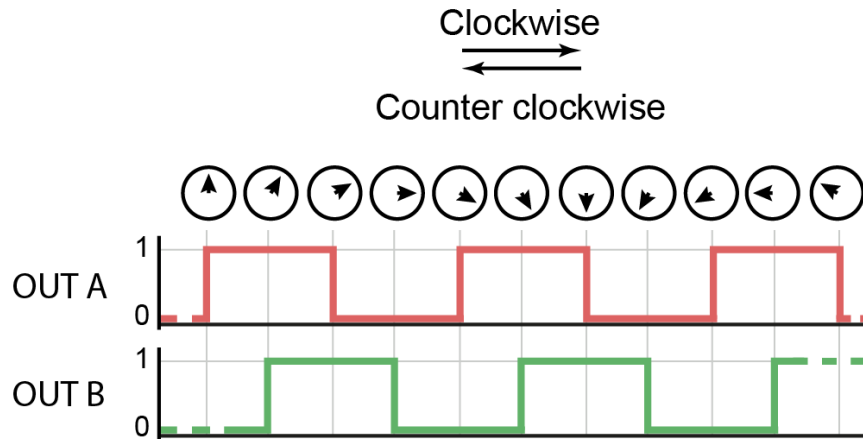


Figure 8. Example of encoder output

When acquiring the outputs of the encoder, a special method must be used when counting the pulses in order to correctly reconstruct the rotation. Each time outputs A or B move, the count is either incremented or decremented by 1. The state of output A compared to output B determines which of the two. Table 1 below summarises the technique. The first two columns of the table are derived from the signals in Figure 8, reading the figure first from left to right (clockwise rotation) and then right to left (anticlockwise rotation).

OUTPUT A	OUTPUT B	ROTATION	COUNT
0→1	0	CW	+1
1	0→1		+1
1→0	1		+1
0	1→0		+1
0→1	1	CCW	-1
1	1→0		-1
1→0	0		-1
0	0→1		-1

Table 1. Counting method for encoder outputs.

The rotation of the encoder can be fast, leading to high frequency pulses. Therefore, one cannot rely on reading the state of the pins the encoder is connected to in the loop function. If the microcontroller is busy executing other instructions when the pins change state, the pulse is not counted, resulting in a mismatch between the real angle and the software measurement. A much more reliable way of acquiring the outputs A and B is through interrupts. As the name suggests, an interrupt is a function whose execution can be tied to a specific change in conditions (software or hardware). When the conditions are met, the execution of the main loop is paused to process the interrupt service routine (the function associated to the interrupt). This means that whatever the microcontroller is busy doing, the execution is stopped to process the interrupt and no pulses are lost.

For a typical quadrature encoder, one needs two interrupts: one for each output.

```
1 #define PINA 3 //First output of the quadrature encoder connected to digital pin 3 of the arduino
2 #define PINB 2 //Second output of the quadrature encoder connected to digital pin 2 of the arduino
3
4 volatile long enc_count; //Total pulses from the encoder.
5
6 void setup() {
7   pinMode(PINA, INPUT); //Defining pin as digital input
8   pinMode(PINB, INPUT); //Defining pin as digital input
9   //Attaching interrupt to pin. When the pin changes state, the function channelA() gets executed
10  attachInterrupt(digitalPinToInterrupt(PINA), channelA, CHANGE);
11  //Attaching interrupt to pin. When the pin changes state, the function channelB() gets executed
12  attachInterrupt(digitalPinToInterrupt(PINB), channelB, CHANGE);
13
14  enc_count=0;
15 }
16
17 void loop() {
18
19 }
20
21 void channelA() {
22
23 }
24
25 void channelB() {
26
27 }
```

Figure 9. Basic code to set up hardware based interrupt service routines.

The snippet of code is defining pins D2 and D3 as digital inputs and assigning each one an interrupt service routine. When PINA changes (“CHANGE”) state, the function channelA is automatically executed. The same goes for PINB and channelB. The variable enc_count is updated inside these functions according to the information in Table 1.

Note: On the Arduino UNO, only pins D2 and D3 can be used with the attachInterrupt() function. On an Arduino Mega, pins D2, D3, D18, D19, D20, D21 are available (D20 and D21 cannot be used if I2C is used (*I2C is a communication protocol for some specific sensors; not among those in the Mechatronics kit*)).

A few useful formulas:

Before attempting to control the rotation of the shaft of a DC motor, a few considerations are needed. In order to convert the number of pulses into revolutions of the encoder one can use the formula:

$$encoder_revs = \frac{total_pulses}{pulses_per_revolution}$$

If the angle is required instead:

$$encoder_angle = 360^\circ * \frac{total_pulses}{pulses_per_revolution}$$

Finally, it is very likely that a gearbox is present between the output shaft of the DC motor and the shaft to which something like a wheel can be connected to. This is done to reduce the rotational speed and amplify torque output from the DC motor. If a gear box is present, one can convert the revolutions of the encoder in rotation of the wheel by doing

$$wheel_angle = \frac{encoder_angle}{\omega}$$
$$wheel_revs = \frac{encoder_revs}{\omega}$$

Where ω is the reduction ratio of the gear box. If ω is not known, one can stick a piece of tape to the wheel and program the DC motor to slowly turn in one direction while counting the pulses from the encoder. After 5 full revolutions of the wheel, one can compute ω as:

$$\omega = \frac{5}{\text{encoder_revs}} = \frac{5 * 360^\circ}{\text{encoder_angle}}$$

Note: *It is better to let the wheel do more than one revolution as the resulting ω will be more precise. Here we suggested 5 revolutions, but feel free to do a few more and compare the results.*

4.4.1 Task:

Referring back to the circuits in Figure 6 and Figure 7 and using Table 1. Counting method for encoder outputs. and the code in Figure 10 as a guide, write a script that can rotate the wheel connected to the DC motor according the following cycle:

1. 45 clockwise
2. Delay 2s
3. 45 degrees counter clockwise (back to the starting position)
4. Delay 2s
5. 90 degrees counter clockwise
6. Delay 2s
7. 90 degrees clockwise
8. Delay 2s, then back to step 1

Note: To successfully control the rotation of the wheel, consider the number of pulses per revolution of the encoder as well as the reduction ratio of the gearbox connected to the DC motor.

The following snippet of code can be used as a template.

```
1 #define ENC_K ? //Number of pulses per revolution. Stated in the datasheet for the encoder and the assignment sheet
2 #define GEAR_RATIO ? //Ratio of the gearbox attached to the DC motor
3 #define PINA 3 //First output of the quadrature encoder connected to digital pin 3 of the arduino
4 #define PINB 2 //Second output of the quadrature encoder connected to digital pinsni 2 of the arduino
5
6 volatile long enc_count; //Total pulses from the encoder
7 volatile float enc_rev; //Revolutions of the encoder
8 volatile float wheel_angle; //Angle of the wheel in degrees
9
10 void setup() {
11     Serial.begin(9600);
12     pinMode(PINA, INPUT); //Defining pin as digital input
13     pinMode(PINB, INPUT); //Defining pin as digital input
14     //Attaching interrupt to pin. When the pin changes state, the function channelA() gets executed
15     attachInterrupt(digitalPinToInterrupt(PINA), channelA, CHANGE);
16     //Attaching interrupt to pin. When the pin changes state, the function channelB() gets executed
17     attachInterrupt(digitalPinToInterrupt(PINB), channelB, CHANGE);
18     enc_count=0;
19     enc_rev=0;
20 }
21
22 void loop() {
23     Serial.print("Enc count: ");
24     Serial.print(enc_count);
25     Serial.print("\tEnc revs: ");
26     Serial.print(enc_rev);
27     Serial.print("\tWheel angle: ");
28     Serial.print(wheel_angle);
29     Serial.println("");
30     delay(20);
31 }
32
33 void channelA(){
34     //Use if else statements to increase or decrease enc_count.
35     //Use the information in Table 1 to build the conditions (consider the rows where channel A is changing).
36     //Use digitalRead() to acquire the state of channels A and B.
37     if(){
38         if(){
39             enc_count++; //Encoder rotating one way e.g. clockwise
40         }
41         else{
42             enc_count--; //Encoder rotating the opposite way e.g. counterclockwise
43         }
44     }
45     else{
46         if(){
47             enc_count++; //Encoder rotating one way e.g. clockwise
48         }
49         else{
50             enc_count--; //Encoder rotating the opposite way e.g. counterclockwise
51         }
52     }
53     enc_rev = ; //Converting total pulses to number of revolutions. Use constant ENC_K for this.
54     wheel_angle = ; //Compute the angle of rotation of the wheel using the the revolutions of the encoder and GEAR_RATIO.
55 }
56
57 void channelB(){
58     //Use the function channelA as template.
59     //This time focus on the rows of the table where channel B is changing.
60 }
```

4.5 Exercise 5: (Advanced) Reading multiple encoders efficiently

Note: This is a formative exercise and will not be assessed, but you may find it useful to complete, to help you with your group project.

When controlling a mobile robot, it is usually necessary to read more than one encoder. Unfortunately, when using an Arduino UNO, the `attachInterrupt()` instruction can only be used with digital pins D2 and D3. Moving to a board with more IOs, such as the Mega, solves the problem to some degree. Still, when dealing with a larger number of encoders, a more general approach must be used.

A simple and efficient way of acquiring two or more encoders consists in using pin change interrupts. The pins on a microcontroller are grouped in ports. On typical Arduino boards each port consists of 8 pins. These can be found by looking at a pinout diagram for the board in use (see **Figure 10** for an example).

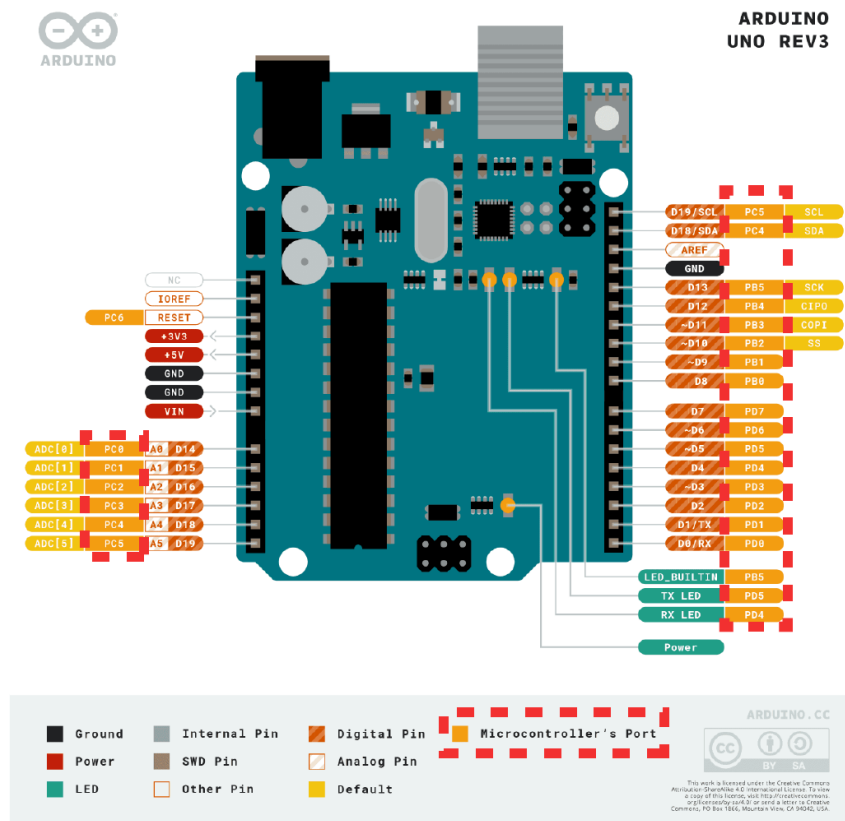


Figure 10. Pinout of the Arduino UNO board.

As visible in **Figure 10**, digital pins D0 to D7 are on the same port, PORT D. In the code, one can ask the microcontroller to monitor specific pins on a port and execute an interrupt service routine when one of these pins changes state. Within the interrupt service routine, one then needs to analyse which pin caused the execution of the interrupt and react accordingly.

The advantage of using this more advanced approach is that any pin on the microcontroller can be used to read encoder outputs, dramatically increasing the number of encoders that can be used in a system.

Let's imagine 2 encoders are connected to an Arduino UNO on pins D3, D4, D5, and D6. None of these pins could be used with the `attachInterrupt()` function. As shown in Figure 10, all this pins are part of PORT D of the microcontroller. The first step is to activate pin change interrupts on PORT D and then mask all pins, but the one connected to the encoders' outputs. The masking operation tells the microcontroller which pins it needs to listen to. This operation is needed to avoid the interrupt being triggered by changes in pins that are not connected to the encoders.

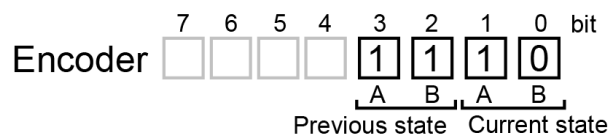
```

1 volatile long enc1_count, enc2_count;
2
3 void setup() {
4   PCICR |= 0b00000100; //Activate PORTD pin change interrupts
5   PCMSK2 |= 0b01111000; //Masking everything on PORT D but PB3 (digital pin 3) to PB6 (digital pin 6)
6 }
7
8 void loop() {
9
10 }
11
12 ISR(PCINT2_vect)
13 {
14
15 }

```

The interrupts are now active on all 4 pins. When one of the pins changes its state, the function `ISR(PCINT2_vect)` is automatically executed. ISR stands for “Interrupt Service Routine”, while `PCINT2_vect` is the interrupt request handler for PORT D.

In order to efficiently keep track of the rotation of each encoder, one can use a lookup table technique. Each encoder is associated with 4 bits in a byte variable:



Using this encoding, one can create the following table:

Prev. state	Current state	Resulting Code	Rotation	Count
00	00	(base 2) 0000 = 0 (base 10)	NO	0
00	01	0001 = 1	CCW	-1
00	10	0010 = 2	CW	1
00	11	0011 = 3	NA	0
01	00	0100 = 4	CW	1
01	01	0101 = 5	NO	0
01	10	0110 = 6	NA	0
01	11	0111 = 7	CCW	-1
10	00	1000 = 8	CCW	-1
10	01	1001 = 9	NA	0
10	10	1010 = 10	NO	0
10	11	1011 = 11	CW	1
11	00	1100 = 12	NA	0
11	01	1101 = 13	CW	1
11	10	1110 = 14	CCW	-1
11	11	1111 = 15	NO	0

Table 2. Lookup tables for the encoder state

NO = No rotation (current state = previous state)

NA = Not applicable (Impossible condition. Only one bit at a time can change)

Note: Analyse Table 2 and you'll see that the number of rows that have a +1 or -1 is 8, exactly the same as the number of rows of Table 1. Try also comparing the rows between the two tables, to better understand how Table 2 was derived.

It is important to keep all entries, even the impossible ones, so that one can create the following array:

```
static const int8_t lookup_table[] = {0,-1,1,0,1,0,0,-1,-1,0,0,1,0,1,-1,0};
```

Each code in the table can be converted to a decimal number, from 0 to 15, and used as index for the array. With all of this in place, one can write a script to manage two encoders.

```
1 volatile long enc1_count, enc2_count;
2
3 void setup() {
4   PCICR |= 0b00000100; //Activate PORTD pin change interrupts
5   PCMSK2 |= 0b01111000; //Masking everything on PORT D but PB3 (digital pin 3) to PB6 (digital pin 6)
6   Serial.begin(115200); //Starts the serial communication
7 }
8
9 void loop() {
10  Serial.print("Enc1 count: ");
11  Serial.print(enc1_count);
12  Serial.print("\tEnc2 count: ");
13  Serial.println(enc2_count);
14  delay(20);
15 }
16
17 ISR(PCINT2_vect)
18 {
19   static const int8_t lookup_table[] = {0,-1,1,0,1,0,0,-1,-1,0,0,1,0,1,-1,0};
20   static uint8_t enc1_val = 0; //Byte that stores current and previous state of the encoder outputs for encoder 1.
21   static uint8_t enc2_val = 0; //Byte that stores current and previous state of the encoder outputs for encoder 2.
22
23   enc1_val = enc1_val << 2; //Stores the previous state of the encoder.
24   enc1_val = enc1_val | ((PIND & 0b00011000)>>3); //Masks all bits of PORTD apart from those of the encoder and saves the current state of the encoder.
25   enc1_count += lookup_table[enc1_val & 0b00001111]; //Masks the HIGH portion of the enc_val byte and performs the lookup operation.
26
27   enc2_val = enc2_val << 2; //Stores the previous state of the encoder.
28   enc2_val = enc2_val | ((PIND & 0b01100000)>>5); //Masks all bits of PORTD apart from those of the encoder and saves the current state of the encoder.
29   enc2_count += lookup_table[enc2_val & 0b00001111]; //Masks the HIGH portion of the enc_val byte and performs the lookup operation.
30 }
```

Note: Bitshift operations are used to manipulate variables at a bit level. The right bit-shift operator (\gg) is used to shift bits to the right (e.g. $y=x\gg 3$ means take x , shift the bits 3 positions to the right and save the results in y). The left bit-shift operator (\ll) operates in a similar way, but shifts bits to the left.

Note: After computing the encoder count, one still needs to convert it to revolutions of the encoder, or revolutions of motor shaft, using suitable constants.

4.5.1 Task:

Using the technique you just learnt, modify the script from the previous exercise to achieve the same functionality, but with the new approach.

Note: Try changing the pins the encoder is connected to and modify the script accordingly.

5 Acknowledgements

This document was written by Ben Taylor, with contributions from a number of authors: Dana Damian, Shuhei Miyashita, Kilian Marie, and Marco Pontin.

Introduction to Arduino:

Post-Laboratory Assignment

There is a video assignment for the three Introduction to Arduino laboratory exercises. The exercises you are required to demonstrate in the video are:

- Digital I/O - Exercise 3: LED Pattern
- Digital I/O - Exercise 4: Switch Debounce
- Analogue Sensors - Exercise 2: Calibrate Potentiometer Angle
- Analogue Sensors - Exercise 3: Measure the Output of the Sharp IR Distance Sensor
- Analogue Sensors - Exercise 5: Use the 1-D Lookup table to calibrate the IR sensor
- PWM Control of Actuators - Exercise 2: Controlling a Standard Servomotor
- PWM Control of Actuators - Exercise 3: Controlling a DC motor, using a driver board

See the Introduction to Arduino assignment brief on the Blackboard site for more details.