8.0.1 Lab 5 - Deadband Compensation for Bidirectional Motion

8.0.1.1 - Purpose

To compensate for ’stiction’ in a bidirectional open loop controller.

8.0.1.2 - Background

Friction in motors will prevent them from turning when small voltages are applied. This effect is normally known as ’stiction’, a combination of the words static and friction. Friction is present in all motors, however this effect is greater in lower cost motors. When a motor is connected to a mechanical system the friction may increase greatly. In most cases we can reduce the cost of a new design by using less expensive motors and compensating for larger friction effects in software.

8.0.1.3 - Theory

In motors there are two types of friction that must be considered. The static friction, ’stiction’, will prevent initial motion. If the motor supplies enough torque to break free and start turning, the kinetic friction will provide a roughly constant friction torque.

Figure 8.1 shows an example of how this friction affects the relationship between the voltage applied to a motor, and the resulting velocity. At very low voltages the torque is too low to overcome the stiction. The region where the applied voltage has no effect is called the deadband. Once the applied voltage exceeds the deadband limits the motor will start to turn. While turning, the motor experiences kinetic friction, damping and other effects. If the voltage supplied to the motor is slowly decreased it will eventually ’stick’. However, the sticking force will be lower than the static friction force. This alternate path is shown in the figure as a heavy dashed line.

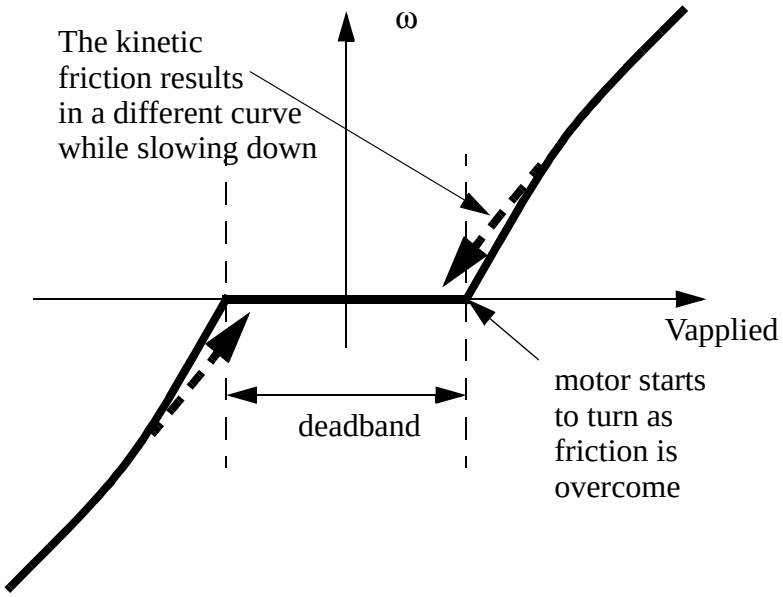


Figure 8.1 Motor deadband for a bidirectional motor

There are a number of methods for compensating for a non-linear deadband in a system. A simple method is shown in Figure 8.2. This method involves finding the voltages where the motor breaks free in the positive and negative directions. These are used in a program to change the wanted output voltage, to an adjusted output voltage.

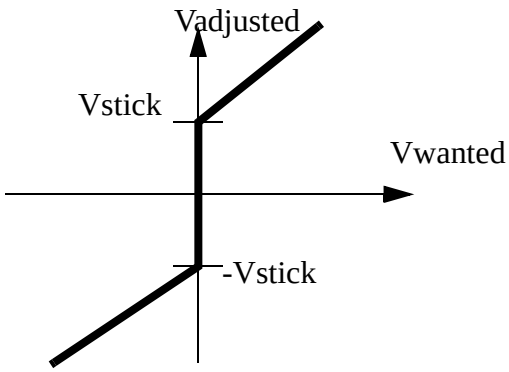
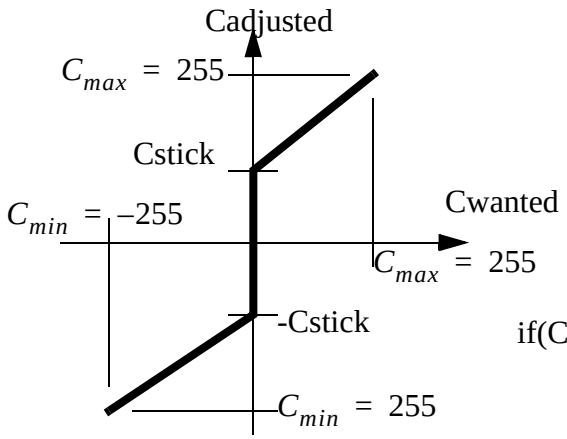


Figure 8.2 Deadband approximation for a bidirectional motor

The equations to implement a deadband compensator are shown in Figure 8.3. These adjust the wanted voltage to exclude the deadband. In this case the equations also adjust the relationship to allow the full use of the PWM output scale.



if(Cwanted > 0)

$$C_{adjusted} = C_{stick} + \left(\frac{C_{wanted}}{C_{max}}\right)(C_{max} - C_{stick})$$

if(Cwanted < 0)

$$C_{adjusted} = -C_{stick} + \left(\frac{C_{wanted}}{C_{min}}\right)(C_{min} - C_{stick})$$

if(Cwanted = 0)

$$C_{adjusted} = 0$$

Figure 8.3 Deadband approximation for a bidirectional motor

The equations from the previous example are implemented in the subroutine given in Figure 8.4. This subroutine actually allows the positive and negative deadband limits to be different, as would be expected in an actual motor. The subroutine also verifies that the corrected values are within the limits of the PWM output. In the example the equations have been rearranged to prevent roundoff errors during the integer calculations. The moving variable must be set so that when the velocity of the system is approaximately zero, it will have a value of 1. The algorithm includes an option to apply a higher (static) torque when not moving (moving == 0) to 'break away'. Normally this would be used for a short period of time to allow the motor to start moving, and then be reduced (moving = 1) so that the kinetic friction torque is used. If a system has feedback the detection of motion can be used to change the coefficient used.

```
#define c_kinetic_pos  0x08 /* static friction */
#define c_kinetic_neg  0x08 /* make the value positive */
#define c_static_pos   0x0a /* kinetic friction */
#define c_static_neg   0x0a /* make the value positive */
#define c_max          255
#define c_min          255 /* make the value positive */
int     moving = 0;    /* assume it starts without moving */

int deadband(int c_wanted){ /* call this routine when updating */
        int     c_pos;
        int     c_neg;
        int     c_adjusted;

        if(moving == 1){
                c_pos = c_kinetic_pos;
                c_neg = c_kinetic_neg;
        } else {
                c_pos = c_static_pos;
                c_neg = c_static_neg;
        }
        if(c_wanted == 0){ /* turn off the output */
                c_adjusted = 0;
        } else if(c_wanted > 0){ /* a positive output */
                c_adjusted = c_pos +
                        (unsigned)(c_max - c_pos) * c_wanted / c_max;
                if(c_adjusted > c_max) c_adjusted = c_max;
        } else { /* the output must be negative */
                c_adjusted = -c_neg -
                        (unsigned)(c_min - c_neg) * -c_wanted / c_min;
                if(c_adjusted < -c_min) c_adjusted = -c_min;
        }

        return c_adjusted;
}
```

Figure 8.4 Deadband compensation for an ATMega32 controlled output

Limiting function are used when the requested output exceeds the available output range. An example is shown in Figure 8.5.

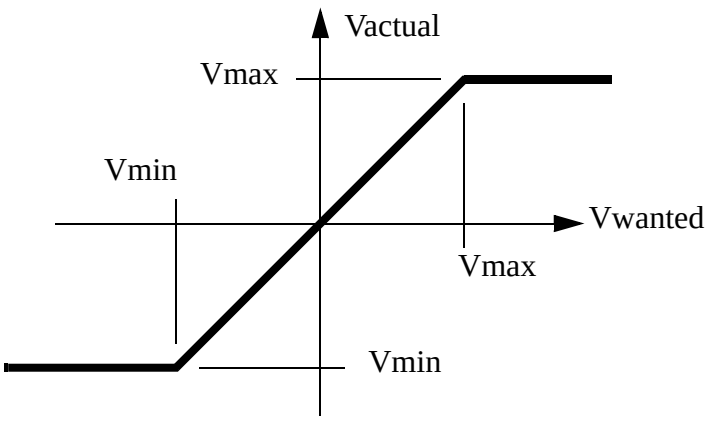


Figure 8.5 An output limiter

In previous labs we used a transistor to control motor movement in one direction.

The H-bridge circuit shown in Figure 8.6 can be used to drive a motor in two directions. As before, the transistors are used to switch current flow. In this case applying a voltage to the A1 and A2 terminals would allow voltage to flow left-to-right through the motor. Turning on inputs B1 and B2 would cause current to flow right-to-left through the motor. When the inputs are controlled using a PWM signal the effective motor voltage can be varied.

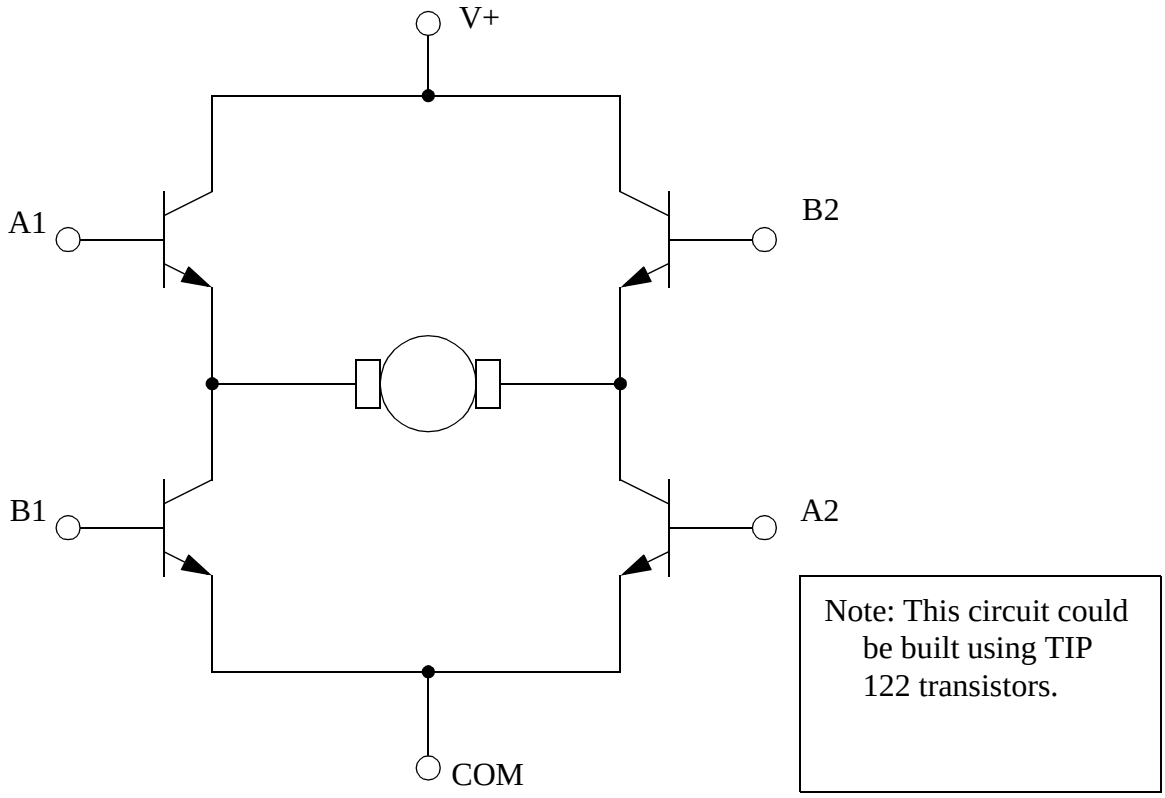
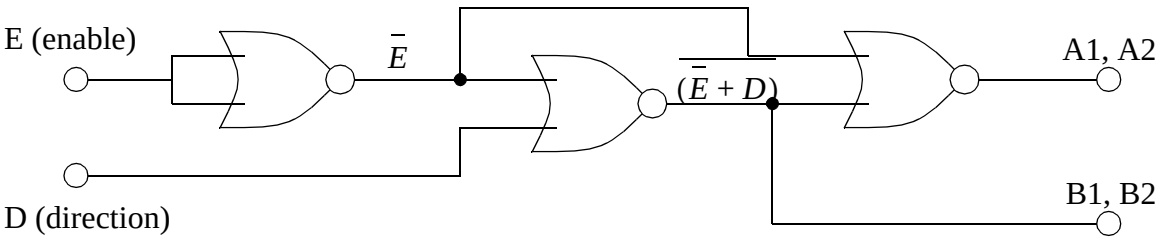


Figure 8.6 H-bridge circuit for motor control

When controlling an H-bridge some extra logic circuitry is commonly used, such as that in Figure 8.7. The main purpose for this circuitry is to protect the H-bridge. For example, if A1 and B1 were both on at the same time, current would bypass the motor, on the left side. The current would be high, similar to a short circuit, eventually destroying the devices. The logic in the figure prevents this. In the logic the PWM output is connected to the E (enable) input. The direction is then used to determine which pair of transistors is turned on.



Proof

$$A = \overline{(\bar{E} + \overline{(\bar{E} + D)})} \qquad B = \overline{(\bar{E} + D)}$$

$$\therefore A = E \cdot (\bar{E} + D) \qquad \therefore B = E \cdot \bar{D}$$

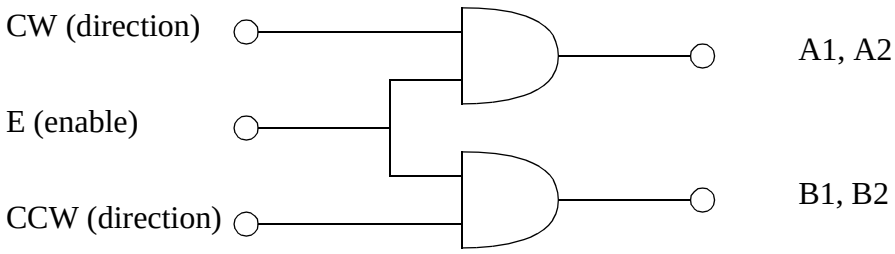
$$\therefore A = E \cdot \bar{E} + E \cdot D$$

$$\therefore A = E \cdot D$$

Note: This circuit could be built using an IC such as an 7402 quad-NOR gate.

Figure 8.7 H-bridge control logic

In this lab we will use an integrated circuit that already contains protection circuitry. For our purposes we can approximate the control logic in that chip with that in Figure 8.8. In this logic we will apply the PWM signal to the E (enable) input. We will then turn on the CW (clockwise) or CCW (counter clockwise) output to determine direction. Care is required to avoid turning both the CW and CCW outputs at the same time.



Boolean

$$A = E \cdot CW$$

$$B = E \cdot CCW$$

Figure 8.8 H-bridge control logic

As you will recall from previous labs the PWM output range is from 0 to 255. However for bidirectional motion we will have both positive and negative motor outputs. The program in Figure 8.9 is designed to compensate for this by checking the sign of the requested output value. Depending upon the sign the CW and CCW outputs will be switched, and the negative values made positive. This routine also checks to make sure that the values don't exceed the maximum limit of 255. This subroutine also changes the direction of rotation by reversing two of the output bits on port C.

```
void PWM_init(){ /* call this routine once when the program starts */
        DDRD |= (1 << PD5); /* set PWM outputs */
        DDRC |= (1 << PC0) | (1 << PC1);
                        /* set motor direction outputs on port C*/
        //using OCR1
        TCCR1A = _BV(COM1A1) | _BV(COM1B1) | _BV(WGM10);
                        // turn on both PWM outputs on counter 1
        TCCR1B = _BV(CS11) ; // set the internal clock to /8
}

void PWM_update(int value){ /* to update the PWM output */
        if(value > 255) value = 255;
        if(value < 0) value = 0;
        outint(value); outln(" = PWM");
        OCR1A = value; // duty cycle
}

int v_output(int v_adjusted){ /* call from the interrupt loop */
        int     RefSignal; // the value to be returned
        if(v_adjusted >= 0){
                /* set the direction bits to CW on, CCW off */
                PORTC = (PINC & 0xFC) | 0x02; /* bit 1 on, 0 off */
                if(v_adjusted > 255){ /* clip output over maximum */
                        RefSignal = 255;
                } else {
                        RefSignal = v_adjusted;
                }
        } else { /* need to reverse output sign */
                /* set the direction bits to CW off, CCW on */
                PORTC = (PINC & 0xFC) | 0x01; /* bit 0 on, 1 off */
                if(v_adjusted < -255){ /* clip output below minimum */
                        RefSignal = 255;
                } else {
                        RefSignal = -v_adjusted; /* flip sign */
                }
        }
        return RefSignal;
}
```

Figure 8.9 Subroutines for PWM outputs and reversing the motor direction

The subroutines in Figure 8.10 deal with general IO updates, as well as a simple delay routine. It allows the deadband compensation to be disabled to simplify the identification of deadband limits.

```
int count = 0; // a count value to be output on port B

void IO_setup(){
        PWM_init();
}

int     db_correct = 0; /* deadband correction is off by default */

void IO_update(){// This routine will run once per interrupt for updates
        if(db_correct == 0){
                PWM_update(v_output(count));
        } else {
                PWM_update(v_output(deadband(count)));
        }
}

void delay(int ticks){ // ticks are approximately 1ms
        volatile int i, j;
        for(i = 0; i < ticks; i++){
                for(j = 0; j < 1000; j++){}
        }
}
```

Figure 8.10 The output update subroutines

The main program is shown in Figure 8.11. It includes numerous keystroke commands to allow the modification of settings.

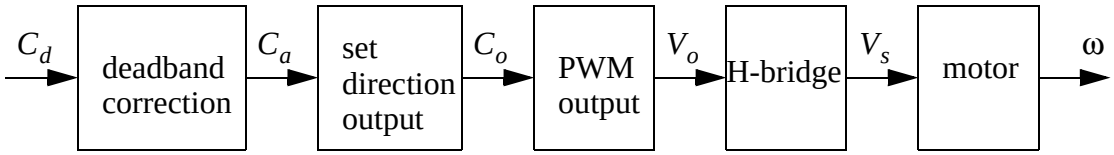
```
int main(){
	int	c;
	sio_init();
	IO_setup();
	outln("");
	outln("--Deadband compensation program");
	for(;;){
		while((c = input()) == -1){} // wait for a keypress
		if(c == '+'){ // increment the output on port B
			if(++count > 255) count = 255;
			if(count == 1){ // it has just started moving
				moving = 0;
				IO_update();
				delay(200);
				moving = 1;
			}
			IO_update();
			outint(count); outln(" = count, incremented");
		} else if(c == '-'){
			if(--count < -255) count = -255;
			if(count == -1){ // it has just started moving
				moving = 0;
				IO_update();
				delay(200);
				moving = 1;
			}
			IO_update();
			outint(count); outln(" = count, decremented");
		} else if(c == 'd'){
			if(db_correct == 0){
				db_correct = 1;
				outln("deadband correction ON");
			} else {
				db_correct = 0;
				outln("deadband correction OFF");
			}
			IO_update();
		} else if(c == 's'){
			count = 0;
			moving = 0;
			IO_update();
			outint(count); outln(" = count, motor stopped");
		} else if(c == 'h'){
			outln("    +: increment value");
			outln("    -: decrement value");
			outln("    d: enable or disable (default) deadband comp.");
			outln("    s: stop the motor");
			outln("    q: quit");
		} else if(c == 'q'){
			count = 0;
			moving = 0;
			IO_update();
			outint(count); outln(" = count, Quitting!");
			break;
		}
	}
	sio_cleanup();

	return 1;
}
```

Figure 8.11 The main test program

8.0.1.4 - Prelab

1. Develop a program for open loop velocity control using the subroutines given in the theory section. The program should compensate for the deadband of the motor, as described in the prelab. The main program should initialize the PWM routines and other outputs and allow keyboard inputs to change the setpoint. The routines that correct for the deadband stiction and update the PWM output are not part of an interrupt driven subroutine. This is because we are using feed-forward control and there is no feedback value available to update the system.



where,

C_d = the desired speed of the motor (int, -255 to 255)

C_a = the adjusted speed of the motor (int, -255 to 255)

C_o = the PWM output (unsigned char, 0 to 255)

V_o = the effective PWM output voltage (0V to 5V)

V_s = the effective motor supply voltage (0V to supply voltage)

ω = the motor speed

Figure 8.12 The open-loop controller

2. Develop a plan for measuring the deadband limits for multiple motors.

8.0.1.5 - Equipment

computer with the winavr compiler and megaload
ATMega32 board including an L293D Push-pull four channel driver
3 motors
1 multimeter
strobe tachometer
external power supply

8.0.1.6 - Experimental

1. Set up the motor controller as shown in Figure 8.13, the pins for the device are shown in Figure 8.13, Figure 8.14 and Figure 8.15. The L293D actually con-

tains four drivers, or the equivalent of two H-bridges. In this case we will use two of these that can supply a peak current of 600mA.

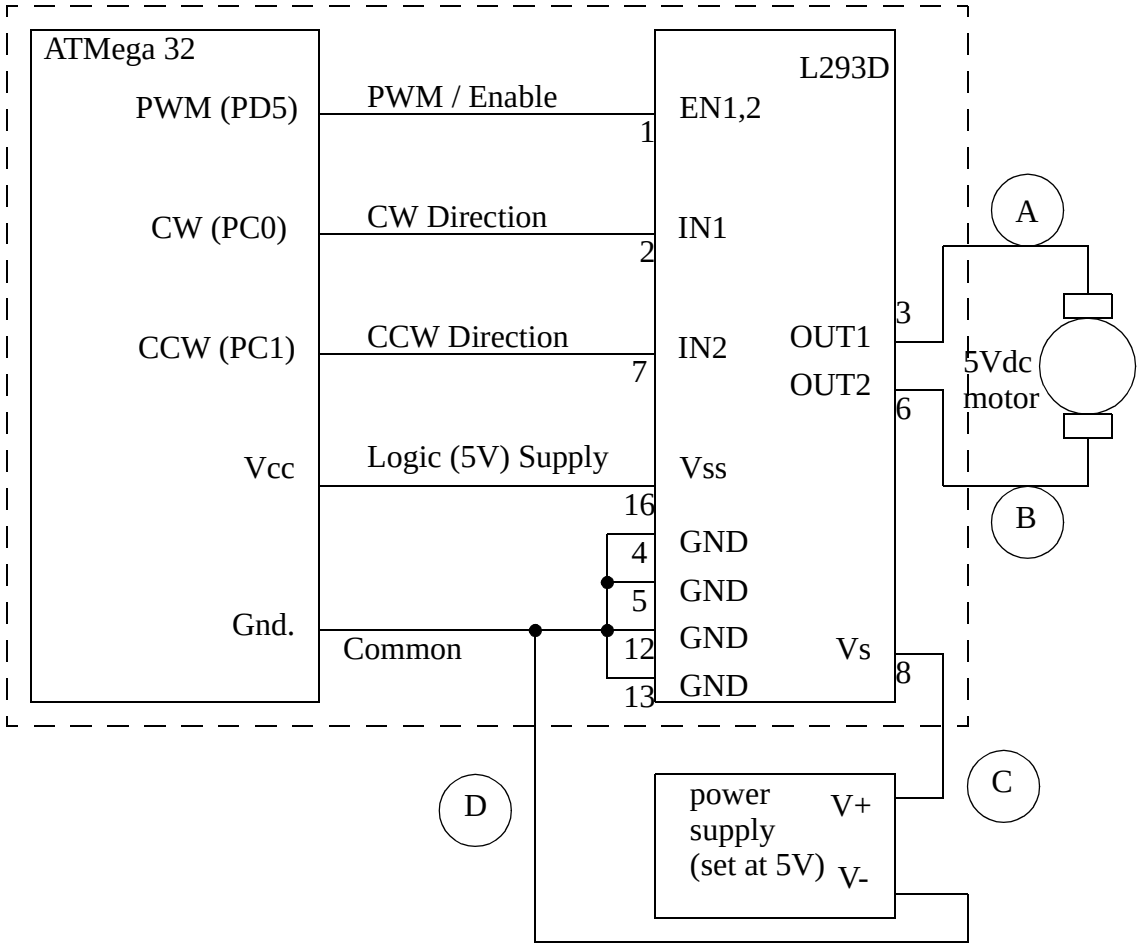


Figure 8.13 Using an ATMega32 motor controller with a L293Ddriver

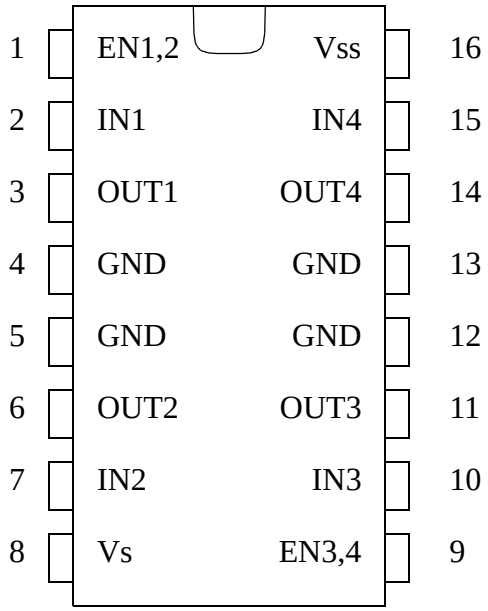


Figure 8.14 L293D pinout diagram (top view)

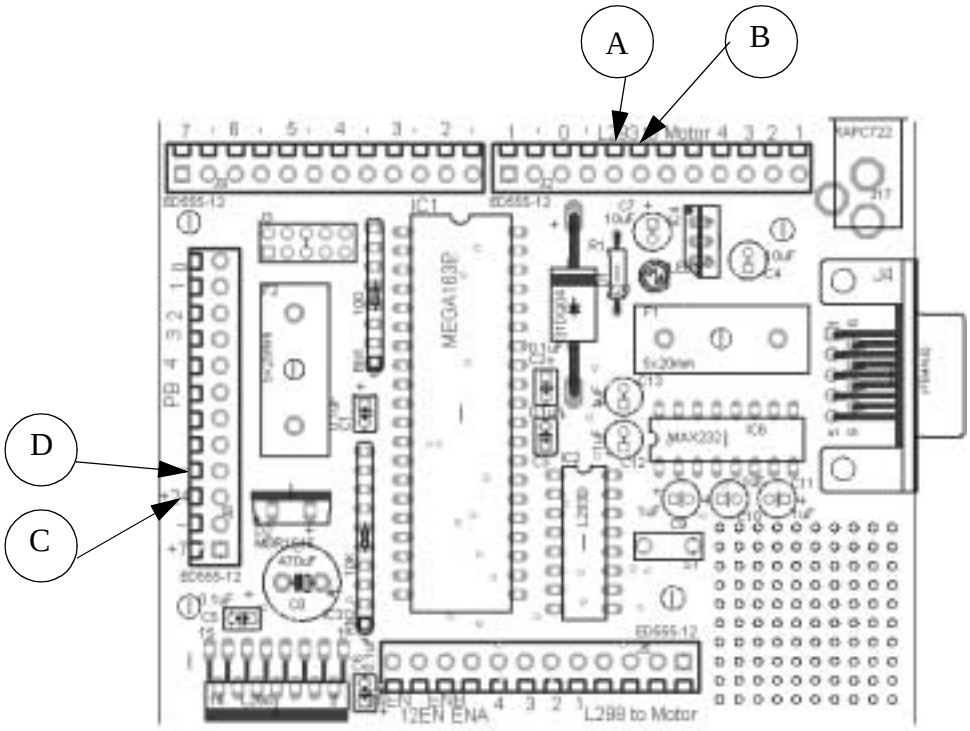


Figure 8.15 PCB connections

2. Enter the program developed for the prelab and test the motor controller.
3. Run the program with the deadband correction disabled. Vary the voltages for

the motor to fall between -255 and 255 and record the desired motor speed, the output voltage and the actual motor speed in a table. Note: that the negative values are probably shown as 2's compliment negative value and must be converted to positive values.

4. Use the data collected in the previous step to determine the deadband limits, and enable the deadband compensation in the program.

5. Repeat the readings taken in step 3 with the deadband values enabled.

6. Compare the results with and without deadband compensation on a graph.

7. Repeat the process for at least two other types of DC motors.