I3GFV
DC Motor Control

Gruppe 27
Simon Hjortgaard Jeppesen - 201910459
Mathias Birk Olsen - 202008722

20. September 2021

Contents

Introduction

This journal will document lab exercise 1: DC Motor control. Fist an experiment in controlling the speed of a DC Motor with a simple mosfet driver will be walked though, then how to control the direction of the same motor with an h-bridge. The third experiment will be about controlling the varies aspects of a stepper, speed, direction and drive-mode.

1 Experiment 1: PWM Control of a DC Motor

1.1 Introduction

This experiment will utilise a Mosfet IRLZ44 board to control the speed of a DC motor with a PWM signal Gennerated by a PSoC 5LD. First, it will describe how the PWM signal is generated on the PSoC and controlled with UART. Then we will outline how the hardware is connected and how the signals run through the setup. Lastly we will confirm the PWMs with an oscilloscope and measure the speed at various duty cycles and conclude upon when the greatest effect occurs.

1.2 PSoC project

The PSoC program is able to control the speed of the DC motor via a terminal inputs using UART. The UART uses an interrupt on the RX signal. The motor speed is controlled by a 8-bit PWM contorller with a top (period) of 100. This period was chosen to make scaling super easy (0-100%). The frequency is controlled with a clock signal.

1.2.1 UART

The UART is set up to make an interrupt when it receives a character from a terminal this character. The Interrupt checks for specific characters that has different actions in the program. The interface is shown in tab. 1.

1	Start motor
2	Stop motor
i	**I**ncrease motor speed
d	**D**ecrease motor speed

Tab. 1: *Interface characters for controlling the DC motor*

The interrupt only sets a flag according to the received character and the actual effect happens outside the interrupt. If no action is defined for a received character it is echoed back. Fig. 1 shows the actions based on received characters. When any action based on a flag is completed, it resets the corresponding flag.

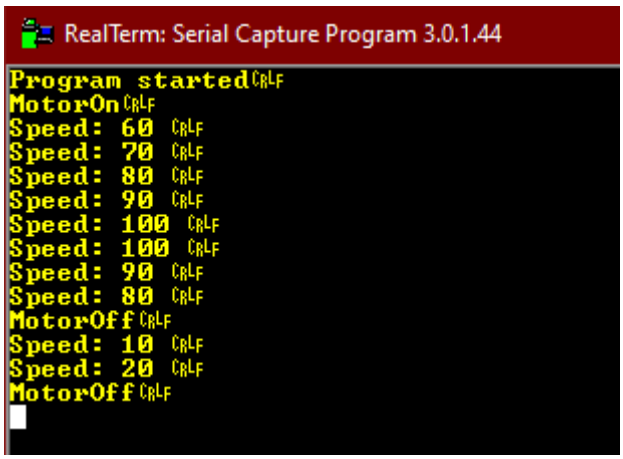


Fig. 1: *RealTemp Sreenshot*

1.2.2 Start/stop functionality

In this experiment the PWM signal is used as start/stop, meaning that when the stopMotor() function is called it sets the PWM duty cycle to 0% and likewise the startMotor() function sets the duty cycle to 50%.

1.2.3 Change speed functionality

To keep track of the speed, a global variable "int8_t speed" is defined in main.c.
To increase the motor speed the "speed" variable is increased by 10. This value is then written to the PWM compare value via the function "PWM_WriteCompare(speed);". The value is limited at 100, since the PWM top (period) is set to 100.
To decrease the motor speed almost the same happens. The variable is decreased by 10 and sent to the PWM compare value. It is verified that the value is limited at 0.

In code snippet 1 two of the actions to flags set by interrupts from the UART are shown and in code snippet 2 the setSpeed() function of motorControl.c is shown.

```
        if (bStopMotor)
        {
            stopMotor();
            UART_PutString("MotorOff\r\n");
            speed = 0;                      // Set speed variable to 0%
            bStopMotor = false;             // Reset flag
        }
        if (incSpeed)
        {
            speed += 10;                     // Increase current speed
   by 10%
            if (speed > 100)                 // Speed can't exceed 100%
            {
                speed = 100;
            }
            setSpeed(speed);
            snprintf(buffer, sizeof(buffer),
                "Speed: %d \r\n", speed);    // Print speed status
            UART_PutString(buffer);
            incSpeed = false;                // Reset flag
```

Code snippet: 1: Some of the looped functionality in main.c

```
// Set motor speed by by setting duty cycle as desired
// Limited to 0-100%
void setSpeed(int8_t speed)
{
    if (speed > 100)
    {
        speed = 100;
    }
    if (speed < 0)
    {
        speed = 0;
    }
    PWM_WriteCompare(speed);
    return;
```

Code snippet: 2: The setSpeed() function of motorControl.c

1.3 Hardware

Motor: 238-9737
Mosfet: RLZ44

The motor and the PSoC are connected to the mosfet board as seen in the diagram in figure 2 and the physical setup in figure 3.

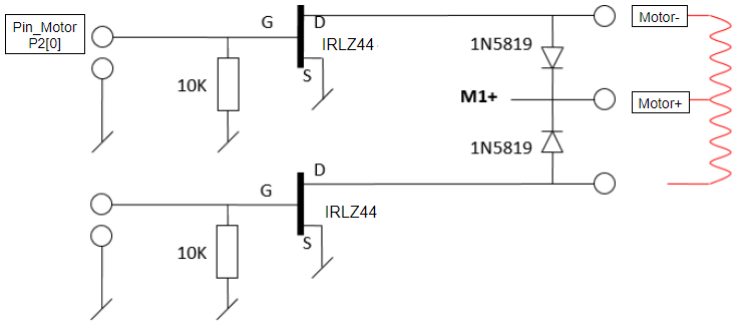


Fig. 2: *Hardware wiring*

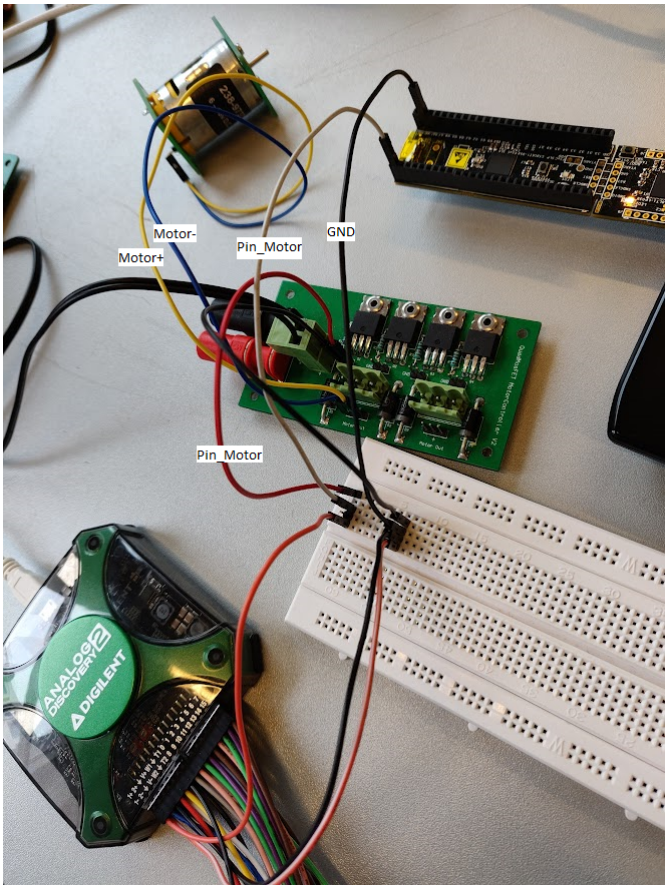


Fig. 3: *Hardware setup*

1.4 Measurements

We tried running the motor at different frequencies and with different duty cycles of the PWM. The frequencies we tried were 23.6kHz, 47kHz and 118kHz. The oscilloscope pictures with the frequency of 23.6kHz are shown in figures 4 to 6. These can be compared to the frequency at 47.3 kHz shown in figure 7 and 8 and to the frequency at 118 kHz in figure 9

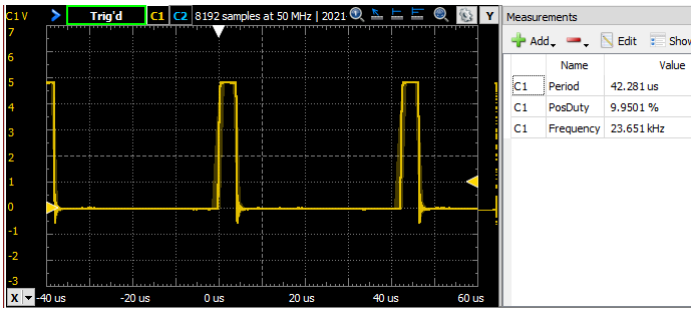


Fig. 4: *Oscilloscope view of the PWM running at 10% duty cycle, 23.6 kHz*

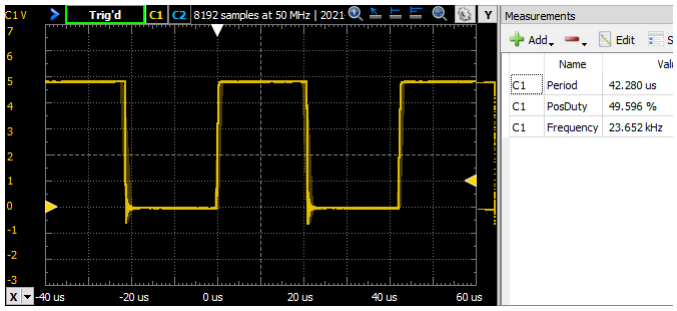


Fig. 5: *Oscilloscope view of the PWM running at 50% duty cycle, 23.6 kHz*

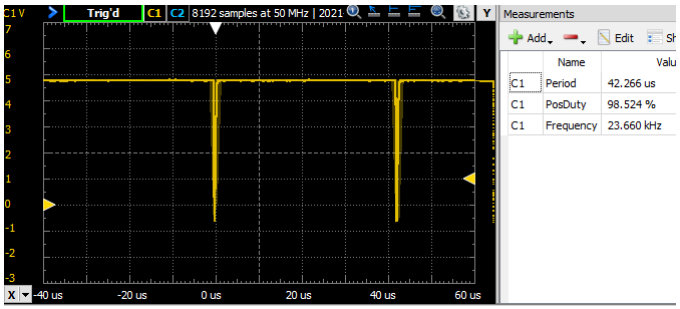


Fig. 6: *Oscilloscope view of the PWM running at 100% duty cycle, 23.6 kHz*

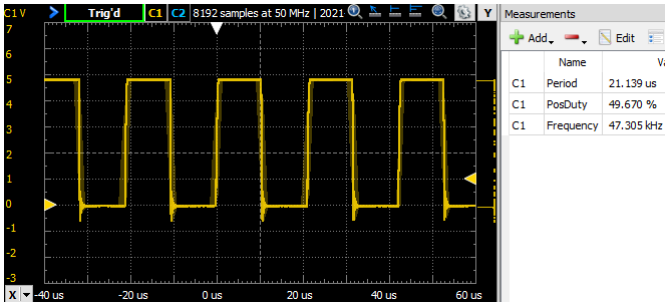


Fig. 7: *Oscilloscope view of the PWM running at 50% duty cycle, 47.3 kHz*

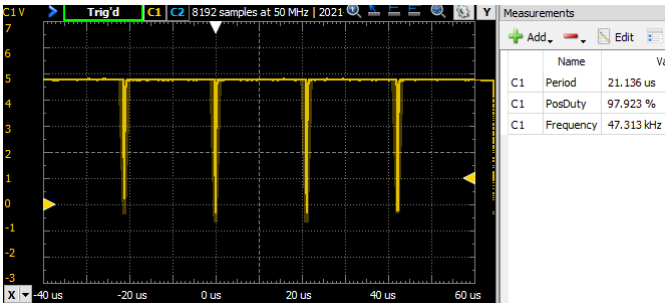


Fig. 8: *Oscilloscope view of the PWM running at 100% duty cycle, 47.3 kHz*

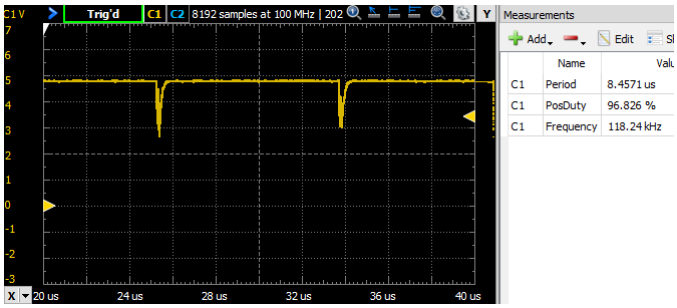


Fig. 9: *Oscilloscope view of the PWM running at 100% duty cycle, 118 kHz*

1.5 Discussion

It was discovered that a duty cuty cycle of a 100% was never fully reached, and with higher frequencies the distance to a 100% grew.
We were able to control the motor speed, and the frequencies we used (23.6 kHz - 118 kHz) didn't make a difference in this experiment. An argument could be made that a the top speed descended with higher frequencies. Because of the actual duty cycle on the pin. This could how ever not be observed.
In general, the higher duty cycle we used, the faster the motor ran - just as expected.

1.6 Conclusion

We were able to start, stop and change speeds of the DC motor using a PWM signal.

2 Experiment 2: DC Motor rotation direction

2.1 Introduction

This experiment will utilize an L298 H-Bridge board to change direction of the same DC motor used in experiment 1. Furthermore it still has to be able to change speeds using the PWM signal.

2.2 PSoC project

Much of experiment 1's PSoC project is reused, and simply expanded with extra functionality since the control part is the same.
In the TopDesign, two more pins are added: Pin_Motor_FWD and Pin_Motor_REV. These are used to control the H-Bridge.

2.2.1 UART

The interface from experiment 1 is expanded as shown in table 2.

1	Start motor
2	Stop motor
i	**I**ncrease motor speed
d	**D**ecrease motor speed
f	**F**orwards motor rotation
r	**R**everse motor rotation

Tab. 2: *Interface characters for controlling the DC motor*

2.2.2 Change direction functionality

The new functions in motorControl2.c are motorFwd() as seen in code snippet 3 and motorRev(). The two functions are nearly identical, only differing in which pin is written to. A delay is set between the two pin writes, to be sure the motor has stopped before it runs the opposite direction. This delay could have been much smaller in practice, but for a demonstration purpose it was set to 500 ms.

```
1 // Set motor direction to forward
2 void motorFwd()
3 {
4     Pin_Motor_REV_Write(0u);
5     CyDelay(500);
6     Pin_Motor_FWD_Write(1u);
7     return;
8 }
```

Code snippet: 3: motorFwd() function in motorControl2.c

2.3 Control sequence

The pins on the H-Bridge board are controlled as follows: when IN1 is set, the motor will run forward when energised. When IN2 is set, the motor will run backwards or reverse

when energised. The Enable pin controls the voltage sent to the motor, which makes it possible to control the speed with the before mentioned PWM signal.

2.4 Hardware

Motor: 238-9737
H-Bridge: L298

The motor and the PSoC are connected to the H-Bridge board as seen in the diagram in figure 10 and the physical setup in figure 11.

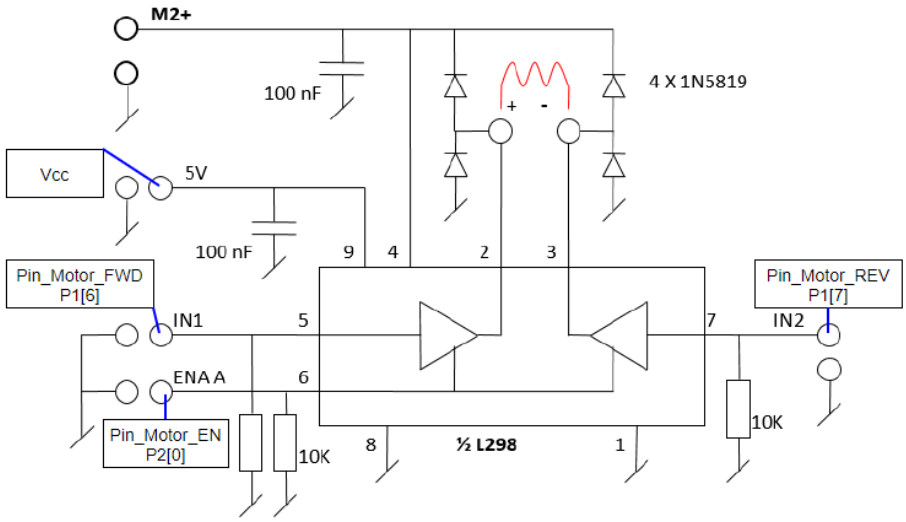


Fig. 10: *Hardware wiring*

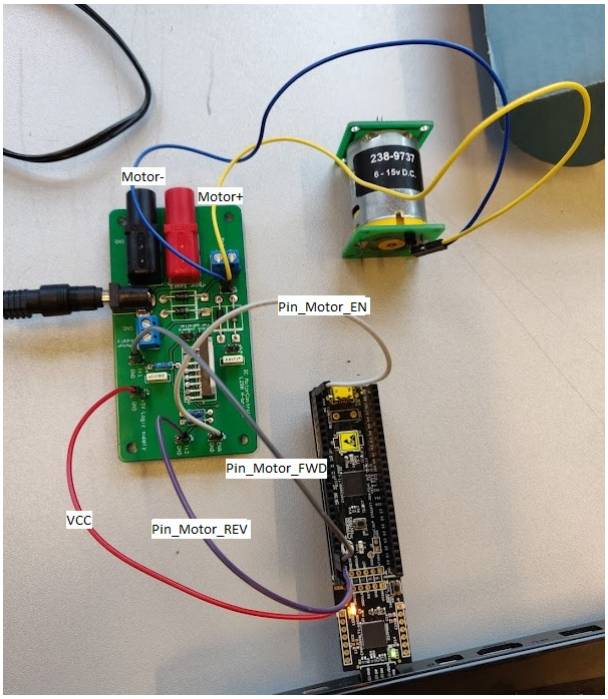


Fig. 11: *Hardware setup*

2.5 Observations and discussion

The same PWM frequency of 23.6 kHz from experiment 1 was used to control the speed of the motor.
The motor was able to change direction based on the command inputs of the terminal.
The motor speed was acting odd as the motor didn't actually run until the PWM duty cycle was over 50%. After a lot of experimentation the cause was found to be the frequency. When it was changed to around 500 Hz the motor ran just fine. Either the motor or the H-bridge had limitations in switching that was exceeded at the high frequencies. The exact cause was not found.

2.6 Conclusion

We were able to change the direction and motor speed through an H-Bridge as long as the frequency was correctly set.

3 Experiment 3: Stepper motor

3.1 Introduction

The objectives of this exercise is to control the various aspects of a stepper motor. The areas to be controlled are

1. on/off
2. speed
3. direction
4. drive mode
5. precise rotations from current rotations

The stepper motor can be controlled in 3 different ways: the wave-drive, full step and half step. This will explained later in section 3.2. Then the hardware and wiring will be discussed before the implementation of the controlling PSoC project will be walked though.

3.2 Drive modes of the stepper motor

There are 3 different drive-modes of a stepper motor, and understanding them is necessary for the implementations in the controlling software.

3.2.1 Wave-Drive

The wave-drive functions by activating one of the stators at a time, turning the rotor, one full step at a time. This drive mode has the same accuracy as the full-step, but half the torque.

	A(WHT)	B(BLU)	A\(RED)	B\(YEL)
Step 1	1	0	0	0
Step 2	0	1	0	0
Step 3	0	0	1	0
Step 4	0	0	0	1

Tab. 3: *step table of half-step*

in table 3 it can be seen in what sequence the stators need to be activated to make a single turn.

3.2.2 Full-Step

Full-step uses two stators at a time to make the rotor move one full step at a time. Here the rotor will be locked in between the stators, opposed to facing them, as it does in wave drive.

In table 4 it is seen how two sets of stators are activated at a time, to move one step.

	A(WHT)	B(BLU)	A\(RED)	B\(YEL)
Step 1	1	1	0	0
Step 2	0	1	1	0
Step 3	0	0	1	1
Step 4	1	0	0	1

Tab. 4: *Step table of Full-Step*

3.2.3 Half-Step

Half-Step utilises the ability to activate one stator and two stators at a time, to double the resolution of steps the motor can take. Here the motor alternates between activating one stator, then the next in line, waiting to turn off the original and thereby making half a step.

	A(WHT)	B(BLU)	A\(RED)	B\(YEL)
Step 1	1	0	0	0
Step 2	1	1	0	0
Step 3	0	1	0	0
Step 4	0	1	1	0
Step 5	0	0	1	0
step 6	0	0	1	1
step 7	0	0	0	1
step 8	1	0	0	1

Tab. 5: *steptable of half-step*

This is better illustrated in table 5, where it is also seen that the amount of steps to make a full turn has doubled.

3.3 Hardware

The stepper motor used in this experiment is of the type SP2575M0206-A this is a 6-lead stepper motor, who's wiring can be seen on figure 12.

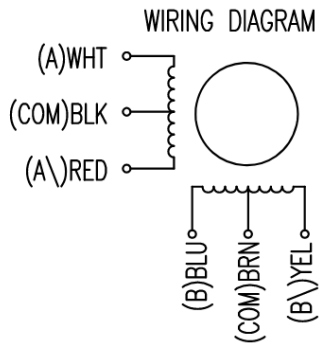


Fig. 12: *wiring diagram of SP2575M0206-A stepper motor (from datasheet)*

The stepper motor is then wired up to a board with 4 mosfet transistors used to control the power supply to each of the stators. This can be seen on figure Figure 13. The 4 mosfets are connected to the pins `P2_3..P2_5` as seen on the same figure.

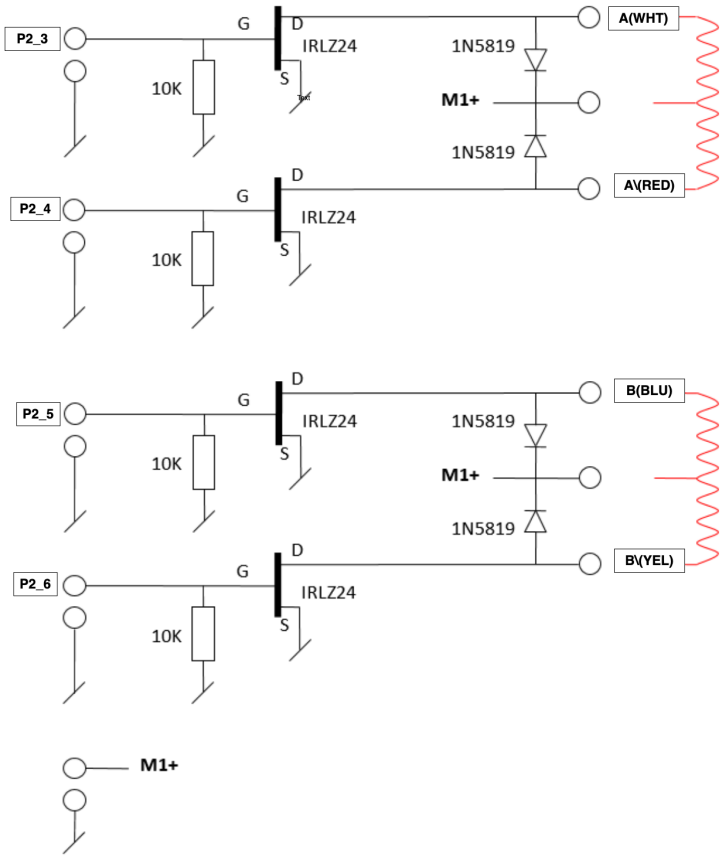


Fig. 13: *connection from PSoC to Mosfet board to Stepper Motor(modified from exercise)*

The connections and setup then allows for a straight forward wiring as seen here on Figure 14

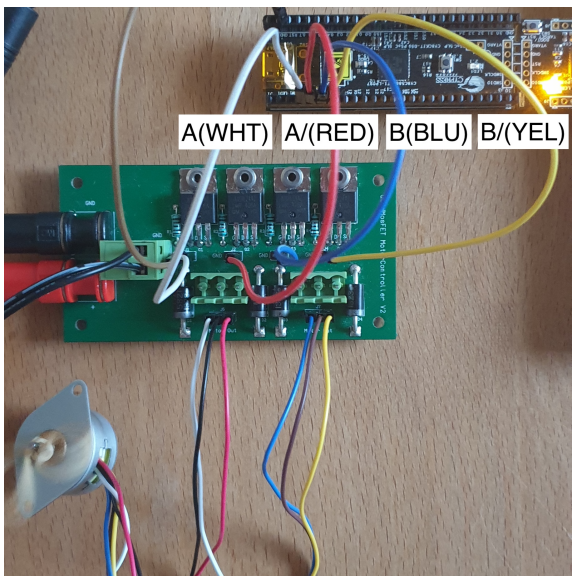


Fig. 14: *The wiring of the stepper motor from PSoC to Mosfet to motor*

3.4 PSoC Project

The PSoC program should control the stepper motor, and needs to have various functionalities:

- driving the motor
- shifting modes
- shifting direction
- stopping and starting the motor
- speed control
- precise rotations

3.4.1 Interface

Control of the stepper motor is down with a simple UART interrupt catching characters - the interface is mapped as follows:

w	set mode to **w**ave drive
f	set mode to **f**ull-step
h	set mode to **h**alf-step
s	**s**top driving
b	**s**tart(**b**egynd) driving
u	forward or **u**p
d	backwards or **d**own
1	1 rotation
-	decrese speed
+	increase speed

Tab. 6: *Interface characters for controlling the stepper motor*

3.4.2 Driving the stepper motor

The implementation of driving of the motor, shifting modes and changing directions, fall somewhat within the same box. Namely a state machine, whose state is made up of 3 parameters:

The direction, the drive mode and the current step.

- the direction can either be 0 or 1.
- the drive-mode takes either 0 = wave-drive, 1 = full-step or 2 = half step.
- each mode has 4 or 8 steps containing the pin status of each pin.

To allow for easy organisation of the state, wave-drive and full step is extended to having 8 steps. This means that there is $2 \times 3 \times 8 = 48$ primary states.

This can be organised into a four dimensional array, illustrated on Figure 15. The outermost layer of the array determines the directions. The next layer holds the modes and then each mode contains 8 rows of pin sequences.

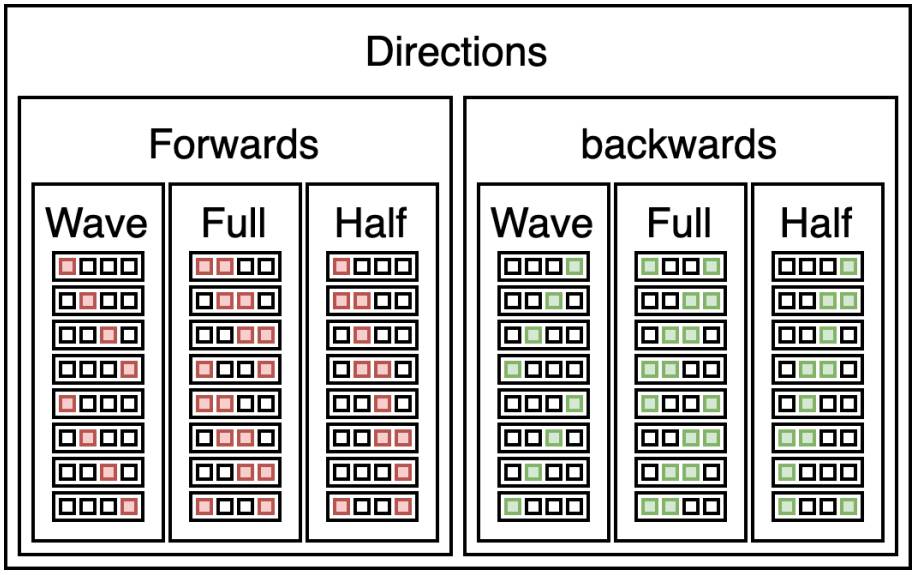


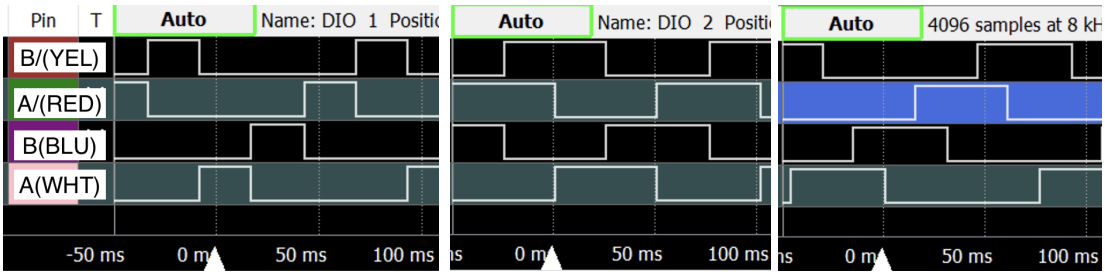
Fig. 15: *A visualisation of the array containing the pin sequnces for each step*

Then updating the state becomes rather simple, a function can be called that takes the stepper from one step to the next in alignment with the current direction and mode. This allows for seamless transitions between the states.

```
void nextStep(uint8_t direction, uint8_t mode, uint8_t step)
{
    // the pin state is loaded from wished direction, mode and the next
    step
    Pin_A_Write(squence[direction][mode][step][0]);
    Pin_B_Write(squence[direction][mode][step][1]);
    Pin_A1_Write(squence[direction][mode][step][2]);
    Pin_B1_Write(squence[direction][mode][step][3]);
}
```

Code snippet: 4: Impmentation of next step function

It can be confirmed that the drive modes shift by analysing the logic shifting on the 4 pins that control the motor. This can be seen on figure Figure 16.



(a) *Wavedrive* ***(b)*** *Full-step* ***(c)*** *Half-step(fast period)*

Fig. 16: *Logic output of sequences for driving the motor in the three different modes*

Figure 16 illustrates how each pin behaves by the varying overlapping. In Wavedrive each pin turns on and off for each step. In Full-step each pin is on for 2 steps in a row. In Half-step each pin is on for 3 steps in a row.

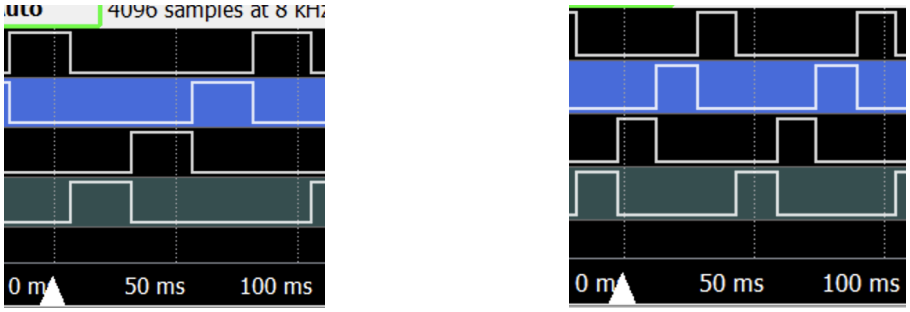
3.4.3 Speed Control

Because the general driving of the motor is done by timer interrupts, the speed can be controlled by changing the period between interrupts. This is done very simply by writing to the register holding the period for the timer.

```
uint8_t speedUp(uint8_t oldSpeed)
{
    if (oldSpeed <= 5) //If we are at or below the  fastest, return orignal speed
    {
        UART_PutString("already at fastest speed\r\n");
        return oldSpeed;
    }

    oldSpeed--; // make new faster speed

    Timer_WritePeriod(oldSpeed); // make speed faster

    return Timer_ReadPeriod();//returns the acctual speed from periode registre
}
```

Code snippet: 5: Implementation of speed up funktion

The result can be seen here on Figure 17, where the left side illustrates a slower speed and the right side illustrates a faster speed



(a) *Slower* ***(b)*** *Faster*

Fig. 17: *Speed control of stepper motor*

The maximum speed of the motor depends on the motor itself and the controller used to control it. in this exercise, the control has limits in updating the pin status before the next interrupt happen. This can be specified by analysing the instruction time it takes to write out to the pins and making sure the interrupt happens just after that. Then the time between interrupts and the step angle can be used to calculate the RPM. This is however beyond the scope of this exercise.

The speed parameter will be defined as a modifier. Because it does not directly affect each state, but rather how fast the transition is between them, when the interrupts are enabled.

3.4.4 Starting and stopping

The combination of the state machine and the timer interrupt driving, makes it very easy to stop and start the motor. Disable the interrupt. The current state is then just static until

something new happens.

New states of the state machine is then added: on and off. Bringing it up to 96 states.

3.4.5 Full rotation

The stepper motor used in this exercise has a step angle of 7.5 degrees. For wave-drive and full drive, that means $\frac{360^o}{7.5^o} = 48$ steps per full rotation for full step and wave drive. It will be twice that for half step, 96 steps.

This can again utilise the state machine. Because as soon as the interrupts are disabled, the motor locks on the current step. From here, its just a matter of running a loop until the desired number of full steps have been reached.

```
uint8_t turnSteps(uint8_t dir, uint8_t mode, uint8_t step, uint8_t
    number)
{

     // makes counter int and makes the local step state nStep.
    uint8_t i, nStep = step;
    // checks if mode is half, if yes take duble steps.
    uint8_t numberOfSteps = ((mode == 2) ? (number *= 2) : number);

    for(i = 0; i < numberOfSteps; i++)
    {
        // makes sure we count from the state we came from, and between
    0 and 7.
        nStep = (step+(i+1))%8;
        nextStep(dir, mode, nStep);
    }

    return nStep; //return new state.

}
```

Code snippet: 6: Implementation of turning a set number of full-steps

The call of the function then makes sure the motor is standing still.

```
        if (turn)
        {
             //if the motor is running, it wont take number of steps
            if (motorRunning)
            {
                UART_PutString("motor running, cant turn 90 when its
    already turning\r\n");
            }
            else
            {
                // turns a number of steps and saves the step its stops
    at to.
                step = turnSteps(direction, mode, step, numberOfSteps);
            }
             turn = false;
```

Code snippet: 7: Calling of turnStep from main loop

The use of the precise movement, does not follow the exercise instructions perfectly. Where the wish was that one key rotated counter clockwise, and another key rotated clockwise. This feature, with many others of the same type, could easily be implemented with the interface of the implementation. Just by passing the which mode and direction the rotation should be taking in to the function. A more scaleable solution has been created. Free of charge.

Code snippet: 7 is also a good demonstration of the main loop in general - The UART powered control has set a flag high, indicating something has changed. In this case turn. Then the state is updated accordingly to that procedure and the flag is lowered again.

3.5 Discussion of DC and Stepper motors

The DC motor is a brilliant dumb drive-train, for anything that needs a continuous rotating torque. The stepper motor turns one small step at a time, worsening the ability to drive something smooth. This does, how ever, allow for precise movement, which is nearly impossible with a regular DC motor.

3.6 Conclusion

In conclusion, all aspects of the stepper motor control was implemented by the means of different states and modifiers. This structure allowed for easy control over what position the stepper motor was in, and what direction and mode the next steps should be made with.

One aspect differs from the exercise: The ability to rotate one full rotation clockwise and counter clockwise. The implementation of precision rotation, does allow for easy expansion of the this functionality, or any other precise movement, if the desire was there.