ERTC: Motor Control

Motor control

- Our goal is to implement a controller within an embedded system; the control problem that we are going to study is a well known one: controlling the angular speed of a wheel, rotating under the action of a DC motor.
- In order to reach our goal we are going to implement a PI controller
- The reference signal will be angular speed of the wheel, measured in revolutions per minute [rpm] or in rad/s

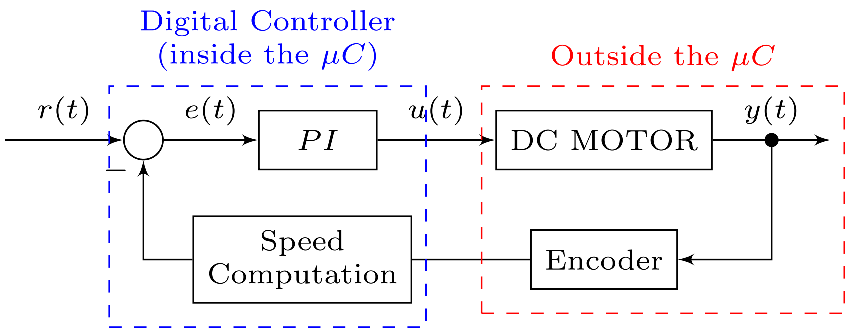
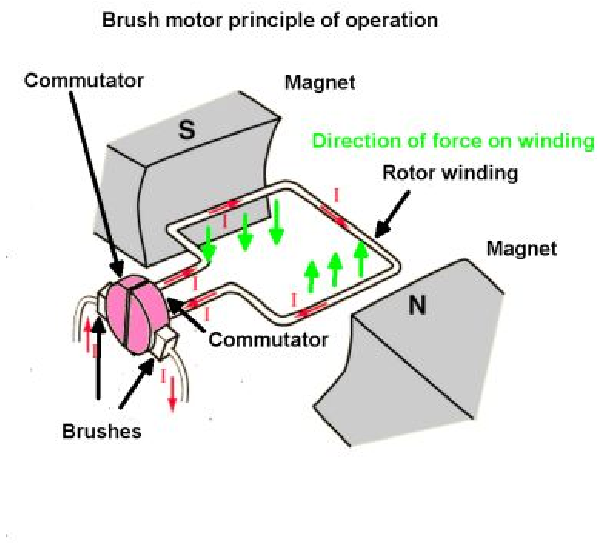


Figure 1: System model

Motor model



Brush motor principle of operation
Commutator
Magnet
S
Direction of force on winding
Rotor winding
Magnet
N
Commutator
Brushes

Figure 2: Brushed DC motor

- Stator: the part of the motor that does not move. Composed by the permanent magnet that generate a magnetic field. The field can be also generated by other stator windings, which in this case generate an electromagnetic field.
- Rotor: the part of the motor that rotates. Composed by a winding that generates a magnetic field when a current flows through it.
- Commutator: a mechanical switch that allows the rotor to rotate continuously. It is composed by a set of brushes that are in contact with the rotor winding.
- When the current flows through the rotor winding, it generates a magnetic field that interacts with the stator magnetic field. These two fields try to repeal each other. A torque is generated that makes the rotor rotate.
- The torque is proportional to the current flowing through the rotor winding. $\tau = k_t i$ where k_t is the torque constant.
- Called DC because the voltage/current is constant.
- We cannot use directly the GPIOs pins to drive the motor, because the low current and voltage of the GPIOs pins are not enough to drive the motor.

Equations of the motor (without load)

- Electrical: From Kirchhoff's Voltage Law (KVL) to the armature circuit

$$\begin{aligned} V_a(t) &= R_a i_a(t) + L_a \frac{\mathrm{d}i_a(t)}{\mathrm{d}t} + V_b(t) \\ &= R_a i_a(t) + L_a \frac{\mathrm{d}i_a(t)}{\mathrm{d}t} + K_b \omega(t) \end{aligned} \tag{1}$$

- Mechanical: From Newton's second law to the rotor

$$\begin{aligned} J\frac{\mathrm{d}\omega(t)}{\mathrm{d}t} &= T_m(t) - b\omega(t) \\ &= K_t i_a(t) - b\omega(t) \end{aligned} \tag{2}$$

Using the Laplace transform on (1) and (2) and assuming $K = K_t = K_b$

$$G(s) = \frac{\Omega(s)}{V_a(s)} = \frac{K}{L_a J s^2 + (R_a J + L_a b)s + (R_a b + K^2)} \tag{3}$$

Table 1: DC Motor Variables and Parameters

Symbol	**Description**	**Units**
$V_a(t)$	Armature voltage (Input voltage)	V
$i_a(t)$	Armature current	A
R_a	Armature resistance	Ω
L_a	Armature inductance	H
$V_b(t)$	Back EMF (Electromotive Force)	V
K_b	Back EMF constant	V/(rad/s)
$\omega(t)$	Angular velocity of the rotor	rad/s
$T_m(t)$	Motor torque developed by the motor	Nm
K_t	Torque constant	Nm/A
J	Moment of inertia of rotor and load	kg·m^2
b	Viscous friction coefficient (damping)	Nm/(rad/s)

DRV8871

Since we want to use PWM as a control signal for the motor, we use a chip that can convert a PWM input into a proper output, both in terms of voltage and current, to drive the motor. The chosen chip is the DRV8871 by Texas Instruments.

- DRV8871 is a brushed DC motor driver.
- It can control a motor bidirectionally by implementing an H-bridge.
- Operating voltage: 6.5 – 45 [V].
- Peak current output: 3.6 [A].
- DRV8871 Datasheet: [link to datasheet].

DRV8871: Pinout

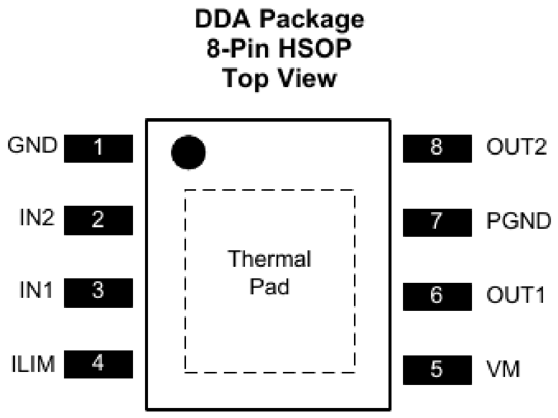


Figure 3: DRV8871 pinout

Below is a description of some important pins for our case. The description of the pins related to the power supply can be found in the datasheet.

Pin name	Pin number	Pin type	Description
IN1	3	Input	Input of the chip that allows controlling the output of the H-bridge. These pins receive the PWM signal from the microcontroller.
IN2	2	Input	-
OUT1	6	Output	-
OUT2	8	Output	H-bridge output that goes to the motor.

DRV8871: Bridge control

The table below describes how the input signals can be used to control the output.

IN1	IN2	OUT1	OUT2	Mode name
0	0	Hi-z	Hi-z	*Coast*
0	1	L	H	*Reverse*
1	0	H	L	*Forward*
1	1	L	L	*Brake*

An electrical circuit, called an H-bridge, is used to select the needed mode. The coast mode allows the motor to coast to a stop, which means that the motor is not driven and will move by inertia. On the other hand, the brake mode will stop the motor faster. When used to drive the motor with a PWM control signal with a duty cycle less than 100% and, for example, in forward mode, it is possible to alternate between forward and brake mode or between forward and coast mode.

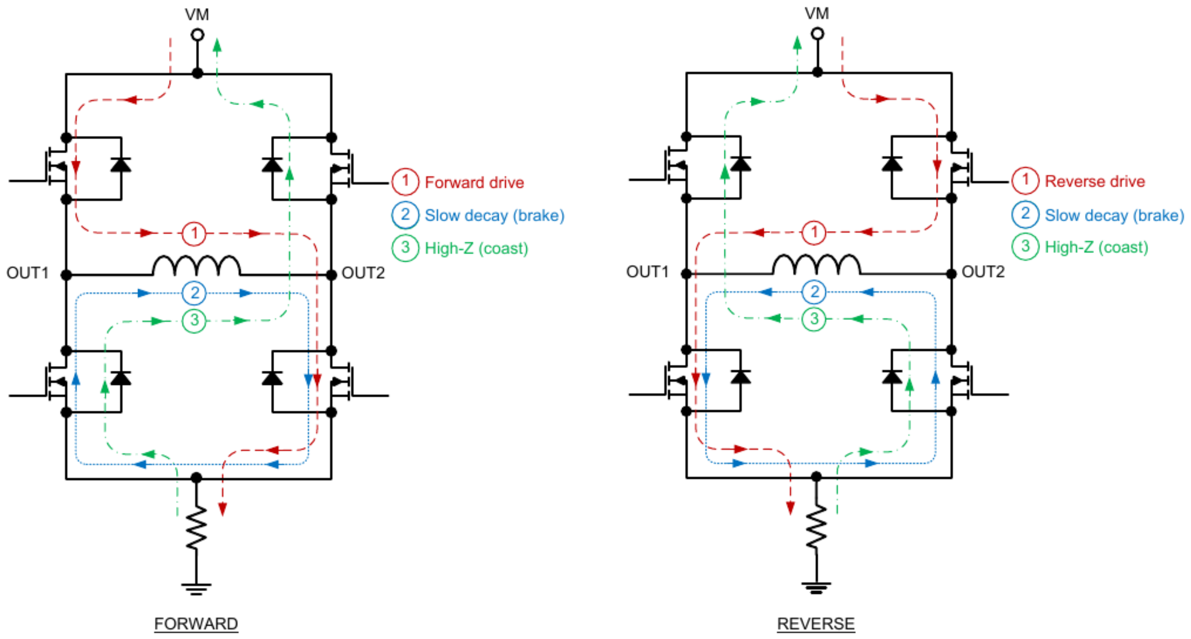


Figure 4: DRV8871 H-bridge

Driving the motor

Three modes are available to drive the motor:

- Mode 1: Using continuous signal on the inputs pins of the DRV8871
- Mode 2: Speed modulation using PWM alternating between forward and brake mode
- Mode 3: Speed modulation using PWM alternating between forward and coast mode

MODE 1

$$if \quad \left.\begin{array}{l} IN1 = 1 \\ IN2 = 0 \end{array}\right\} \text{Motor FWD, 100\% speed}$$

MODE 2 Forward and brake. Suggested by the datasheet.

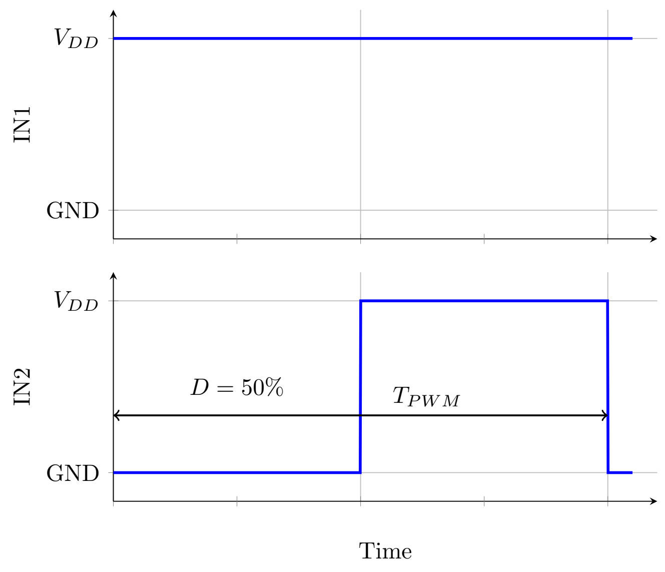


Figure 5: Forward and brake mode

$$Speed \propto 1 - D$$

MODE 3 Forward and coast.

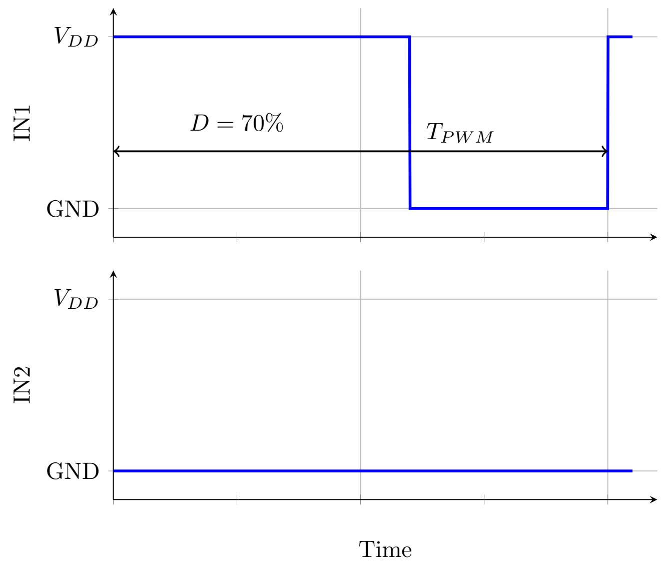


Figure 6: Forward and coast mode

$$Speed \propto D$$

PWM Settings

The voltages applied to the inputs should have at least 800 ns of pulse width to ensure detection. Typical devices require at least 400 ns. If the PWM frequency is 200 kHz, the usable duty cycle range is 16% to 84%.

Position Encoder

The position encoder is a sensor that transforms position information into an electrical signal. The encoder type we are interested in is:

- Rotary encoder: a type of encoder that is made to measure rotational positions.
- Incremental encoder: generates a pulse when the position changes.

A quadrature encoder employs two outputs A and B, which are called quadrature outputs, as they are 90 degrees out of phase. The direction of the motor depends on which phase's signal leads over the other.

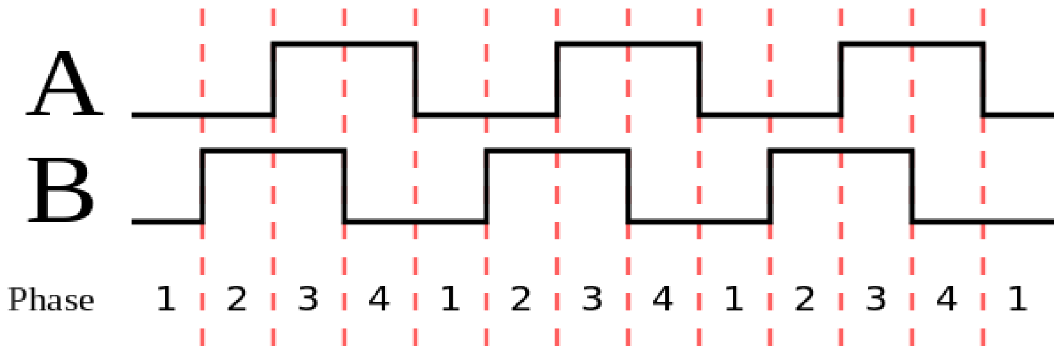


Figure 7: Signals generated by a quadrature encoder with B leading over A

A counter can be used to measure the number of pulses (a pulse can be counted when an edge happens).

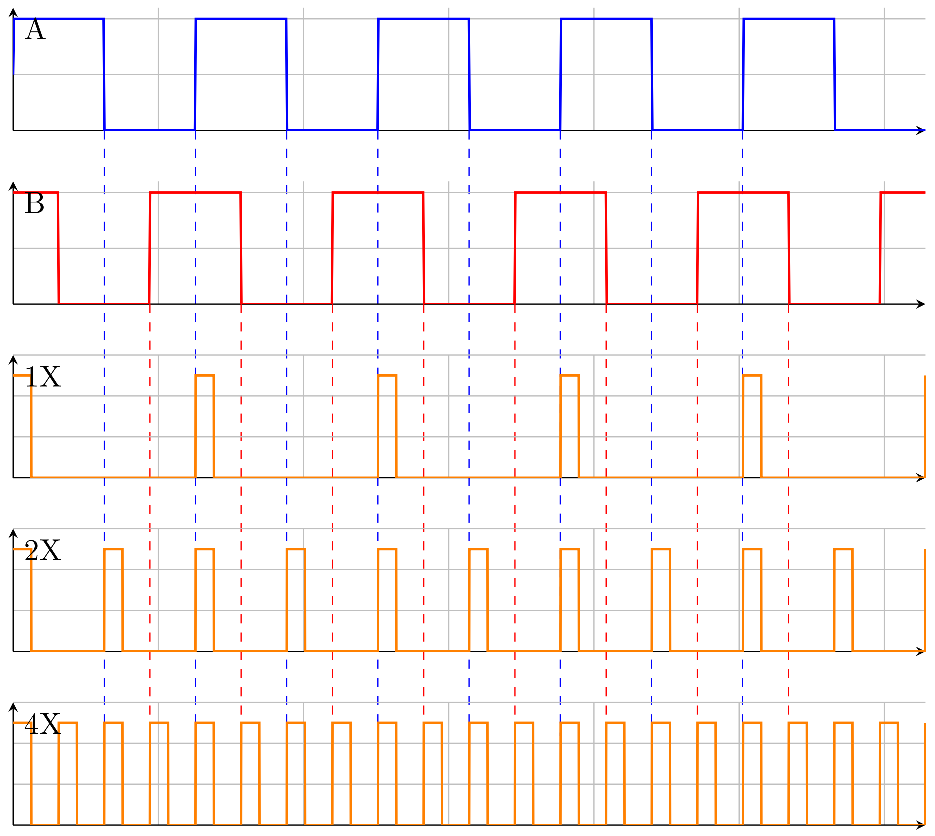


Figure 8: Counting modes

Implementation Details

Hardware Setup and Peripheral List

- Block scheme of the hardware implementing the motor control at the blackboard.
- TIM3 (16bit, general-purpose timer), channel 1 and channel 2: Encoder for motor 1.
- TIM4 (16bit, general-purpose timer), channel 1 and channel 2: Encoder for motor 2.
- TIM6 (16bit, basic timer): Provides the clock for the controller.
- TIM8 (16bit, advanced timer), channel 1 and channel 2: PWM for motor 1.
- TIM8 (16bit, advanced timer), channel 3 and channel 4: PWM for motor 2.

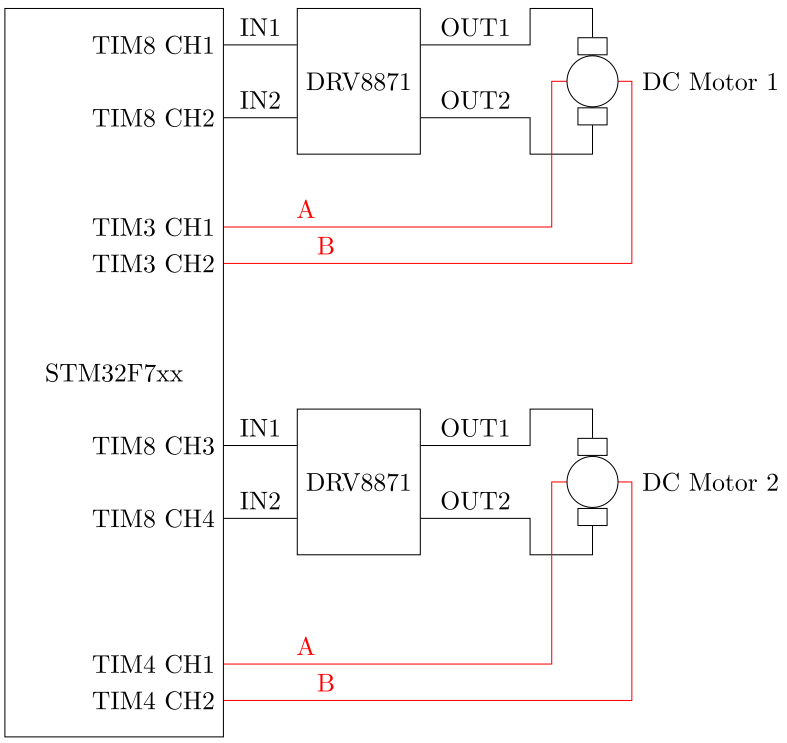


Figure 9: Connection scheme for the timers

Motor and Encoder

- The rover is equipped with a DC motor that comes along with an encoder.
- Electrical characteristics for the motors can be found in the datasheet.
- The module encompasses a gearbox with a 120:1 ratio, which means that for every round performed by the wheel, the motor rotates 120 times.

Encoder

The selected timer needs to be configured in encoder mode. The encoder provides 8 pulses per round in 1X. This means that the number of pulses per round (ppr) from the wheel–side is 960. Assuming to use encoder mode 2X we have 16 pulses on the motor–side and 1920 on the wheel–side; this number is doubled when considering encoder mode 4X. Notice that this mode halves the quantization error when compared with the 2X mode.

Mode	**ppr motor–side**	**ppr wheel-side**
1X	8	960
2X	16	1920
4X	32	3840

To prevent mistakes in counting direction caused by the noise on the encoder lines, it is suggested to use the ”Input Filter” which can be configured by STM32CubeIDE, selecting the desired timer. Experimentally, we saw that a value of 15 for this parameter works best. The input filter samples the signals provided by the encoder at a configurable frequency f_T; the logical level of the signal is decided whenever a number n (in our case equal to 15) of consecutive samples have all the same logical level.

Another relevant parameter is the counter period which can be set up to be equal to the number of pulses counted in one round (1920 or 3840) or can be configured to the maximum allowed level (65535).

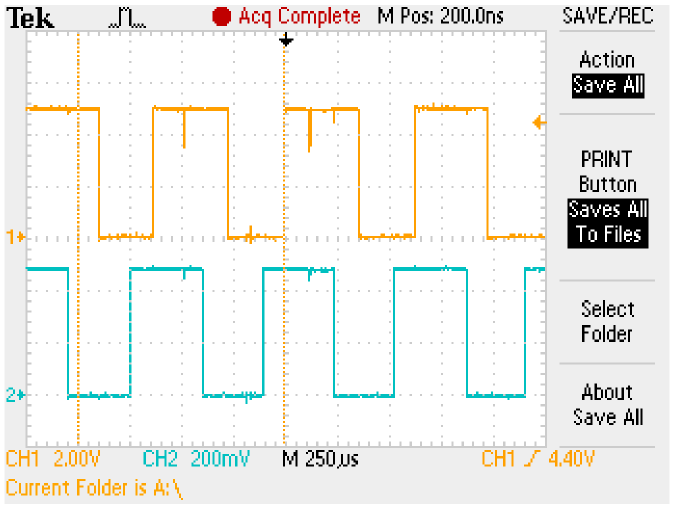


Figure 10: Spike on the encoder lines

In the first case,

```c
#define TIM3_ARR_VALUE 3840 // assuming we are using 4X encoder mode


void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    .
    .
    .
    uint32_t TIM3_CurrentCount;
    int32_t TIM3_DiffCount;
    static uint32_t TIM3_PreviousCount = 0;
    TIM3_CurrentCount = __HAL_TIM_GET_COUNTER(&htim3);
    /* evaluate increment of TIM3 counter from previous count */
    if (__HAL_TIM_IS_TIM_COUNTING_DOWN(&htim3))
    {
        /* check for counter underflow */
        if (TIM3_CurrentCount <= TIM3_PreviousCount)
            TIM3_DiffCount = TIM3_CurrentCount - TIM3_PreviousCount;
        else
            TIM3_DiffCount = -((TIM3_ARR_VALUE+1) - TIM3_CurrentCount) - TIM3_PreviousCount;
    }
    else
    {
        /* check for counter overflow */
        if (TIM3_CurrentCount >= TIM3_PreviousCount)
            TIM3_DiffCount = TIM3_CurrentCount - TIM3_PreviousCount;
        else
            TIM3_DiffCount = ((TIM3_ARR_VALUE+1) - TIM3_PreviousCount) + TIM3_CurrentCount;
    }
    .
    .
    .
    TIM3_PreviousCount = TIM3_CurrentCount;
    .
    .
    .
}
```

In the second case, it is not necessary to check for an overflow as long as only one overflow has occurred between two consecutive readings of the counter. The two's complement should guarantee that the difference $T_{Current} - T_{Previous}$ provides the correct result.

PWM

Configuration for the timer generating the PWM signal (TIM8):

- Clock source: APB1-Timer_clocks at 96 MHz.
- Prescaler (PSC): 959 $\rightarrow f_T = \frac{96 \cdot 10^6}{959+1} = \frac{96 \cdot 10^6}{96 \cdot 10} = 10^5 Hz = 100kHz$
- Counter period (AutoReload Register): 399 $\rightarrow f_{PWM} = \frac{f_T}{399+1} = \frac{10^5}{400} = 250Hz \rightarrow T_{PWM} = 4 \cdot 10^{-3}$ s

To command a motor (motor 1 in this case):

```c
int32_t duty;
/* calculate duty properly */
if (duty >= 0) { // rotate forward
    /* alternate between forward and coast */
    __HAL_TIM_SET_COMPARE(&htim8, TIM_CHANNEL_1, (uint32_t)duty);
    __HAL_TIM_SET_COMPARE(&htim8, TIM_CHANNEL_2, 0);
    /* alternate between forward and brake, TIM8_ARR_VALUE is a define */
    /* __HAL_TIM_SET_COMPARE(&htim8, TIM_CHANNEL_1, (uint32_t)TIM8_ARR_VALUE);
        * __HAL_TIM_SET_COMPARE(&htim8, TIM_CHANNEL_2, (uint32_t)(TIM8_ARR_VALUE - duty));
        */
} else { // rotate backward
    __HAL_TIM_SET_COMPARE(&htim8, TIM_CHANNEL_1, 0);
    __HAL_TIM_SET_COMPARE(&htim8, TIM_CHANNEL_2, (uint32_t)-duty);
}
```

We are assuming a linear relationship between the duty cycle and the voltage applied to the motor, which we suppose to be in the range of $[0, V_{\text{PowerSupply}}]$. $V_{\text{PowerSupply}}$, depending on the power supply source, can be equal to $V_{\text{BAT}} \approx 8V$ or $V_{\text{WallPowerSupply}} = 12V$. In the second case, it is better to limit the duty cycle to about 50%, in order to avoid providing the motor with excessive over-voltage.

Basic software organization

When the controller is implemented in a computer, the analog inputs are read and the outputs are set with a certain sampling period. This is a drawback compared to the analog implementations, since the sampling introduces dead-time in the control loop. When a digital computer is used to implement a control law, the ideal sequence of operation is the following:

1. Wait for clock interrupt: the interrupt is raised, in our case by the timer TIM6, which needs to be properly configured to fire regularly at the desired sampling rate
2. Read the encoder value and compute the angular speed
3. Compute the error with respect to the reference signal and compute the control signal

Notice that the time required to measure, convert, compute the control signal and provide the proper output from the controller, needs to be less than the sampling period. The control algorithm can be implemented directly inside the timer callback.

```c
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
.
.
.
}
```

So, a general idea of how the software should be structured can be the following:

```c
#define TS 0.01 /* Sampling time. The callback needs to be executed with every TS seconds, in this
     example 0.01 seconds */
#define VBATT 8.0 /* Battery voltage, assuming it's constant */

/* macros to convert from voltage to duty cycle and vice-versa, assuming linear relationship
    between the two */
#define V2DUTY ((float)(TIM8_ARR_VALUE + 1) / VBATT)
#define DUTY2V ((float)VBATT / (TIM8_ARR_VALUE + 1))

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == TIM6) {
        /* 1. read the counter value from the encoder
            2. compute the difference between the current value and the old value
            3. compute the motor speed, in [rpm] for example
            4. compute the tracking error
            5. compute the proportional term
            6. compute the integral term (simplest way is to use forward Euler method)
            u_int = u_int + Ki * TS * err
            7. calculate the PI signal and set the pwm of the motor properly */
    }
}

void main(void)
{
    while (1) {
        /* low priority actions */
    }
}
```

Motor transfer function (approximated)

$$P(s) = \frac{\mu}{1 + Ts} \cdot e^{-\tau s} \quad \frac{rad/s}{V}$$

with:

- $\mu = 3.552$
- $T = 0.217$
- $\tau = 0.2$

Initial rise time guess: 0.05s to 0.1s.