Motor control

ERTC

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 round per minute [rpm] or in $\frac{RAD}{s}$
- **Block scheme at the blackboard**

DC motor

- a DC motor is a *actuator* that converts a DC (Direct Current) signal into mechanical energy;
- there are two main type of DC motor implementations:
  - **Brushed** DC motor: this is the type that we are going to use in the rover
  - **Brush-less** DC motor

In the simplest case, A DC motor is made of:

- a *stator*, which can be implemented as a permanent magnet;
- a *rotor winding* or a *coil*, which can be simply a wire;
- a *commutator* is connected to the rotor winding and its purpose is to force the direction of the current flowing into the rotor to be always the same;
- *brushes* are used to supply current to the commutator and each brush is connected to one side of the power supplier;

The stator generate a magnetic field and the rotor winding, which is positioned within the field generated by the stator, when a current flow through it, is subjected to a magnetic force perpendicular to the direction of the stator field. The commutator has a ring-like shape and present gaps, so that each part of the commutator is connected to power source of opposite polarity, ensuring the current flows in the proper direction.

The main things that needs to be taken into account is that the *torque* τ that the motor can apply, is proportional to the current i_a flowing through the rotor; i_a is then related to the voltage used to drive the motor.

Inverting the power source polarity, invert the direction of the current, allowing the motor to rotate in both directions.

Real motors have multiple rotor and commutator, in order to make the rotation more regular; in fact, notice that when the rotor winding is perpendicular to the stator, the *torque* generated is near 0: having multiple rotors and commutators helps at solving this issue.

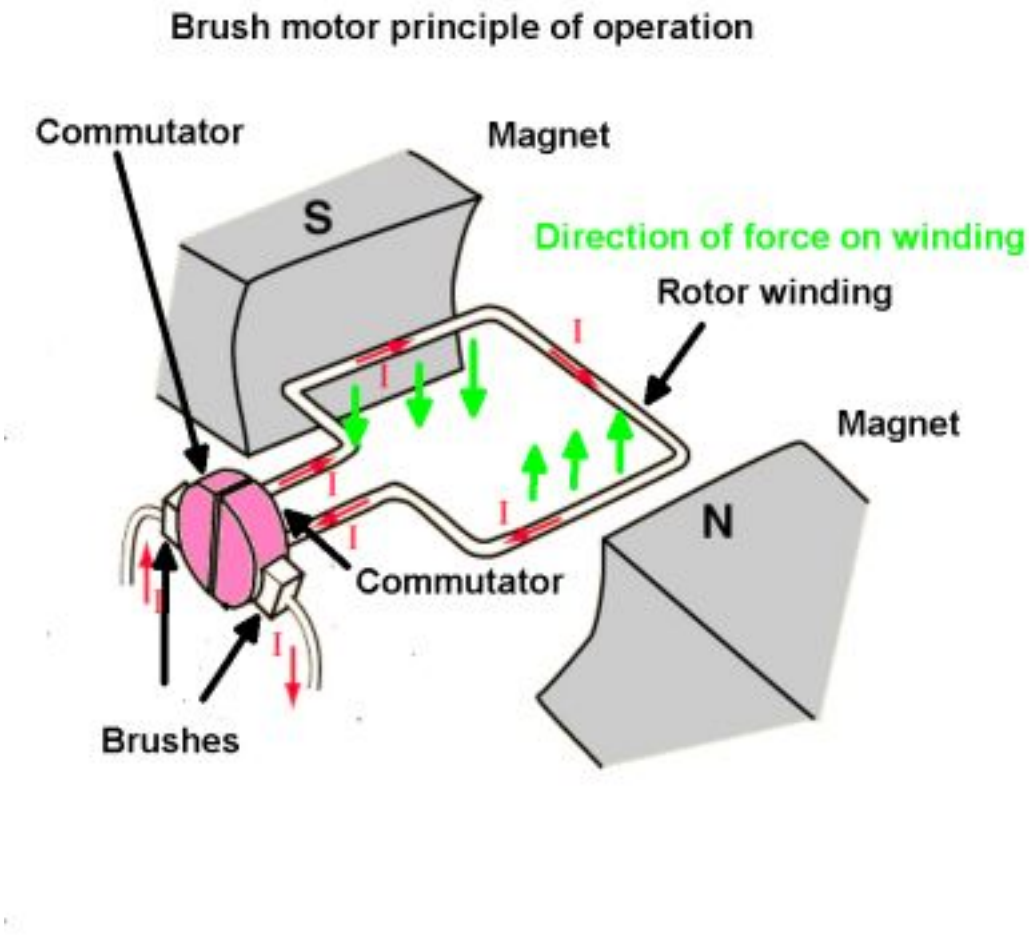

Figure 1: DC Motor simplified scheme

DRV8871

Since we want to use a 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 a DRV8871 by Texas Instruments.

- DRV8871 is a brushed DC motor driver
- Can control a motor bidirectionally by implementing an H-bridge
- Operating voltage: $6.5 - 45$ [V]
- Peak current output: 3.6 [A]
- DRV8871 Datasheet

DRV8871: pinout

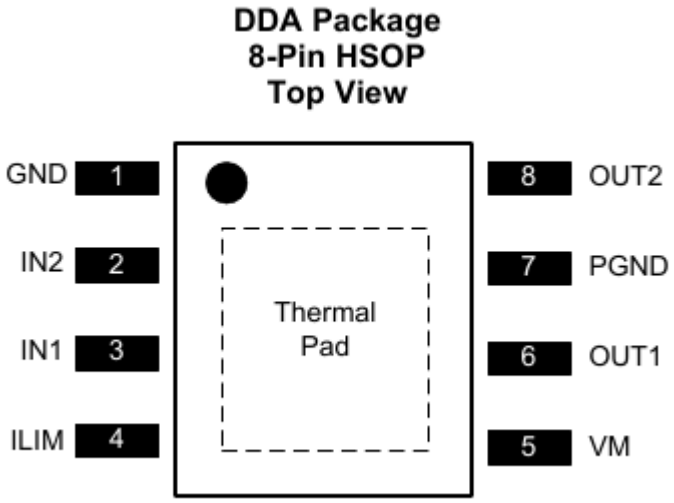

Figure 2: DRV8871 pinout

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

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

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 H-bridge, is used to select the needed mode; *coast* mode allow the motor to coast to a stop, which means that the motor is not driven and will move by inertia. On the other hand, *brake* mode will stop the motor faster.

When used to drive the motor with PWM control signal with a duty cycle $< 100\%$ and, for example, in

forward mode, it is possible to alternate between forward and brake mode or between forward and coast mode.

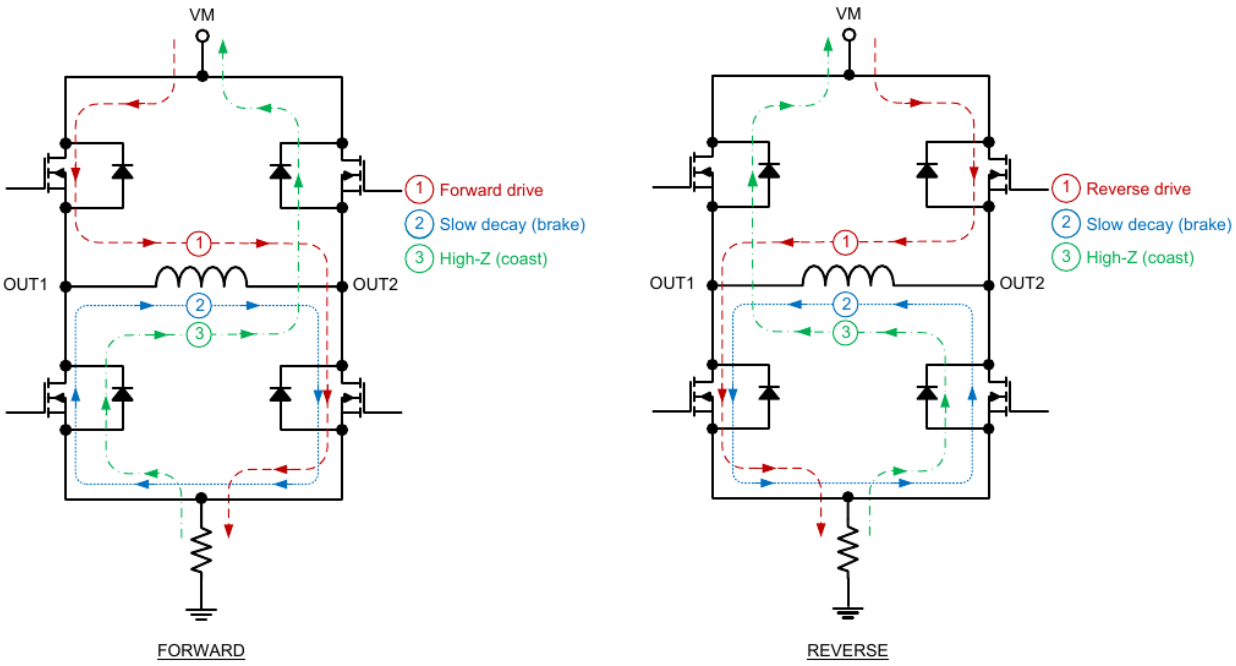

Figure 3: DRV8871 H-bridge

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 transform a 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: generate 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 lead over the other;

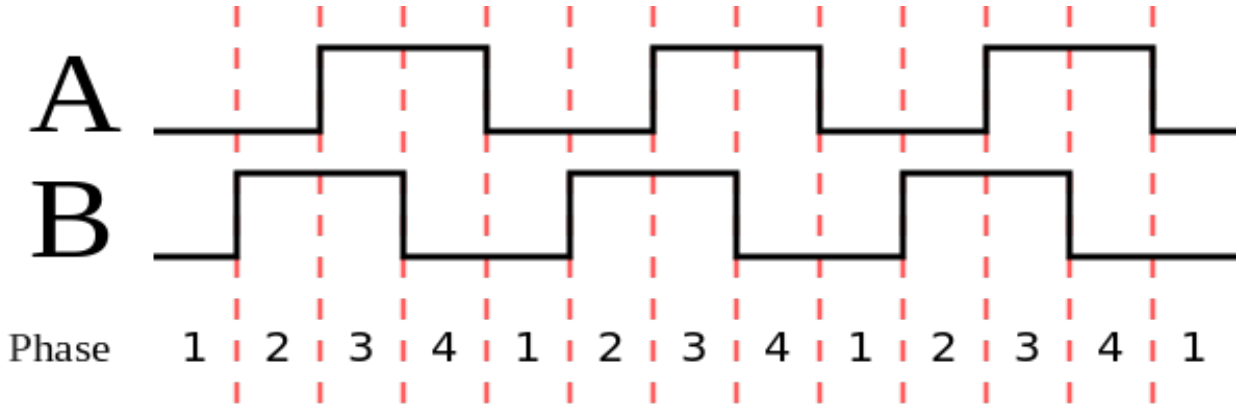

Figure 4: Signals generated by a quadrature encoder with B leading over A

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

Implementation details

Hardware setup and peripheral list

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

Motor and Encoder

- The rover is equipped with a DC motor which come along with an encoder;
- Electrical characteristic for the motors can be found in the datasheet;
- The module encompass a *gearbox* with a 120:1 ratio, which means that for every round performed by the wheel, the motor rotate 120 times;

Encoder

The selected timer needs to be configured in *encoder mode*.

The encoder provides 16 pulses per round. This means that the number of pulses per round from the wheel-side is 1920, assuming to use encoder mode 2X; this number is doubled when considering encoder mode 4X. Notice that this mode halve the quantization error, when compared with the 2X mode.

To prevent mistake 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 wanted 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.

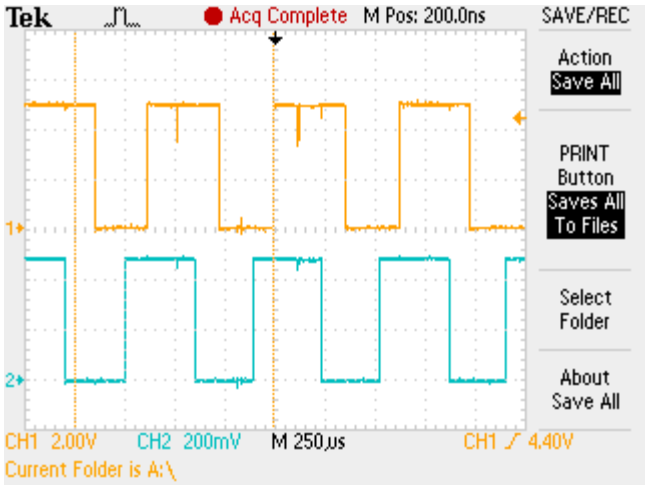

Figure 5: Spike on the encoder lines

Another relevant parameter is the counter period which can be setup 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).

In the first case,

```
#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 been 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): $959 \Rightarrow f_T = 96 * 10^6/96 * 10 = 10^5[Hz]$
- Counter period (AutoReload Register): $399 \Rightarrow T_{PWM} = 4 * 10^2 * 10^{-5} = 4 * 10^{-3}[s]$

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

```
uint32_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, 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_{PowerSupply}]. $V_{PowerSupply}$, depending on the power supply source, can be equal to $V_{BAT} \approx 8[V]$ or to $V_{WallPowerSupply} = 12[V]$. In this second case it is better to limit the duty cycle to about 50%, in order to avoid to provide the motor an 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.

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

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

```
/* the callback needs to be executed with every TS seconds, in this example 0.01 seconds
 * => fs = 100 [hz]
 */
#define TS        0.01

/* Battery voltage, assuming it' s constant */
#define VBATT     8.0

/* 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 */
  }
}
```