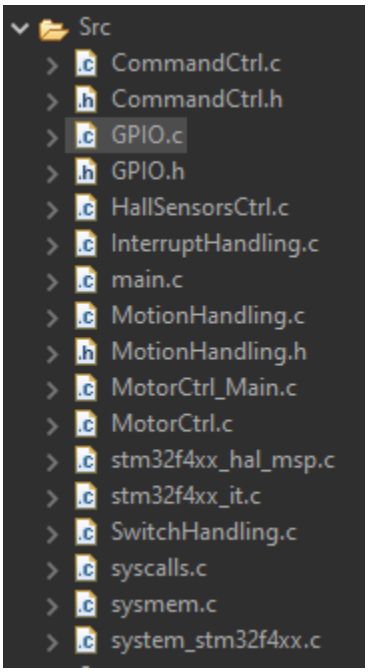
# MOTION CONTROL PROJECT DOCUMENTATION

The project structure is made of separate modules for different implemented functionalities, each one consisting of a C file and a header file.

**Obs:** Implemented files (the names of generated files when the project is configured in the IDE are in lowercase only) are the ones for which the names start with capital letter.

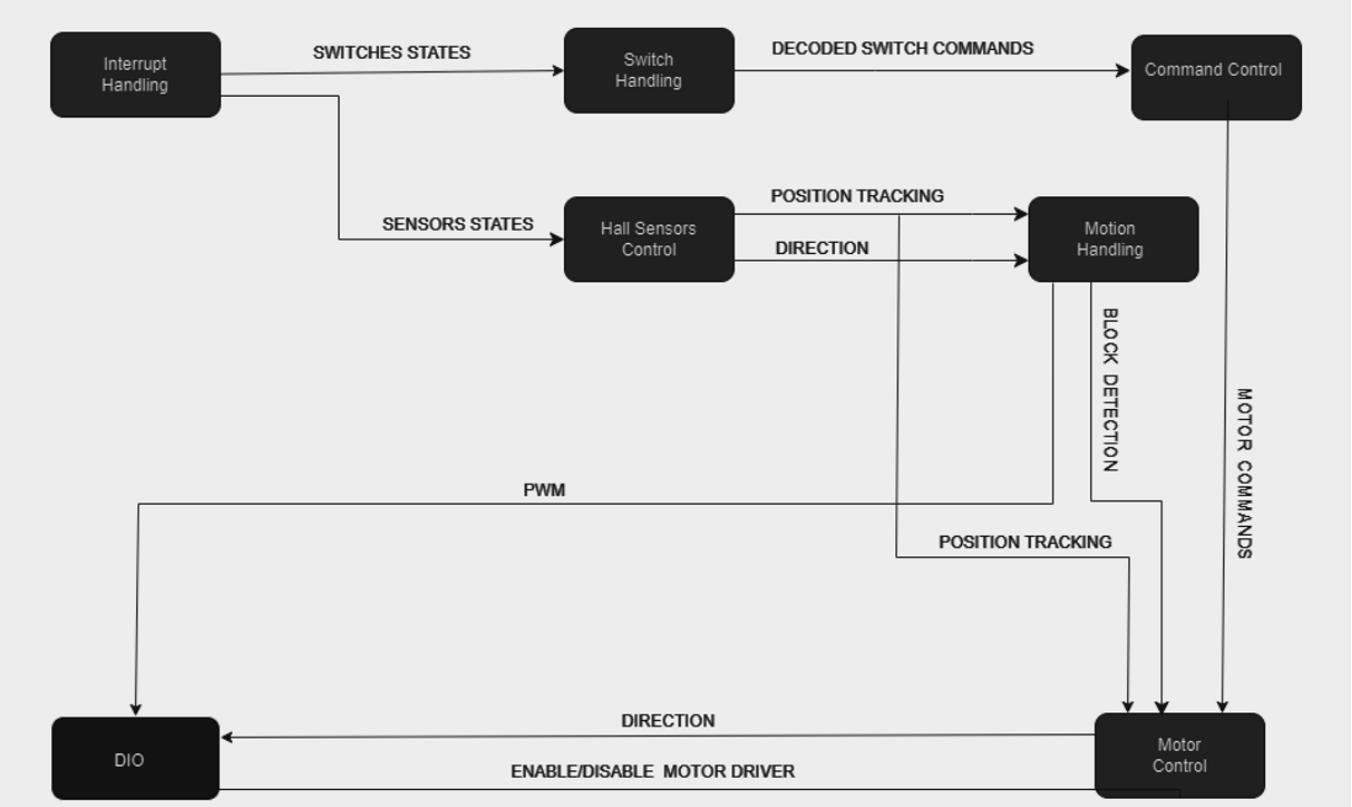
In the main.c file we have a cyclic task (while loop) where the main task of the project gets called.




**Project modules**

This project has defined specific functionalities related to movement for a DC motor such as:

- Position Tracking - by using the Hall sensors incorporated in the motor
- Motion Control - motion related events (e.g. Block Detection, Soft Start, Soft Stop, etc.)
- Diagnosis – short circuits on Hall sensors
- 2 switches that fully control the motor (each switch for each direction)




PROJECT FLOW DIAGRAM

## INTERRUPT HANDLING MODULE

When there is a button press or signal from Hall sensors (transitions of the signal), the main task stops being executed in order to execute the code implemented in the corresponding interrupts. When the program exits the interrupt code, it resumes to the main task that is called in a cyclic manner.

For using the buttons connected to the microcontroller, there are attached interrupts for each of them that are triggered when a button press occurs, that means when there is a digital high signal from the buttons and also when the button is not pressed anymore that results in a transition of the digital signal form high to low. This way we can control the DC motor by (variations of) button presses.

Also, for getting information from the 2 Hall sensors embedded on the motor, an interrupt is attached. With the info that we get in this way we know how to further control the motor knowing the current state of it.

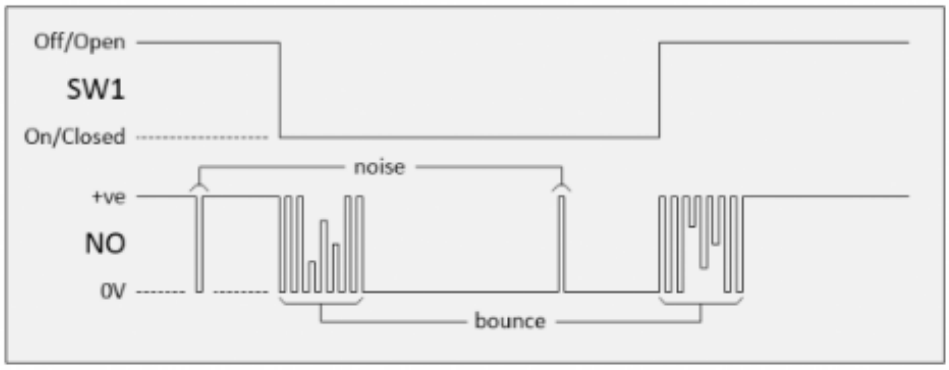
The input that we get from the interrupts is passed forward to other modules that implement different functionalities.

## SWITCH HANDLING MODULE

The 2 switches (each for each direction of the motor) work as follows: when pressed a short amount of time, the switch press acts as AUTO motion for the motor (motor moves even after button is released); on the other hand, when the button is pressed for a longer period of time (greater than 800 ms), the motor moves in MANUAL motion, so when the button is released, the motion stops (it moves as long as the button is pressed).

These movements take place between 2 limits set by using the feedback form Hall sensors (doing position tracking of the motor)

This module gets the state of the buttons by using the interrupts attached. In order to have an accurate information about these button states, the signal corresponding to the button states needs debouncing. In that way, we clear out the signal of any interferences (that we identify as spikes on the digital signal, similar to the picture below) that may occur.




**Button debouncing**

The logic for this debouncing is taking timestamps when there are transitions of the signal (LOW to HIGH / HIGH to LOW). By measuring the width of the signal using these timestamps we can consider it as a valid width that cannot be mistaken as noise (which should have a much smaller width compared to a valid signal that is an actual press of a button).

As we have auto and manual movements of the motor which are set on the button presses, we need to differentiate the AUTO and MANUAL button presses. For this, we use a similar logic to the debouncing one. The change is just on the valid time of the pulse, which in this case is bigger. What is lower than this time parameter is considered as AUTO button press and what is over it is MANUAL button press.

The output of the Switch Handling module is the state of the button press: AUTO or MANUAL. This information is then sent to Command Control module.

## COMMAND CONTROL MODULE

In this module we define the motor commands for each direction (OPEN for one switch and CLOSE for the other):

- *AUTO OPEN* (short btn 1 press -> the motor moves in OPN direction after btn release)
- *MAN OPEN* (long btn 1 press -> the motor moves in OPN direction as long as the button is pressed)
- *AUTO CLOSE* (short btn 2 press -> the motor moves in CLS direction after btn release)
- *MAN CLOSE* (long btn 2 press -> the motor moves in OPN direction as long as the button is pressed)
- *STOP* (stops any current motor command)
- *OFF* (disables the motor after STOP command)

For defining these commands, we used a switch case, a case for each motor command.

These defined commands are actually transitions of the specific commands to other commands.

- *AUTO OPEN Command*
  The current motor command is AUTO OPEN. Further, depending on the next button command that is received, we analyze the conflicts that may appear between the current command and the next one and we set the motor command resulted as such.

  **Obs:** The button command and the motor command are not the same. Button command cannot put the motor in motion. The motor command is set after we analyze the state (command) of each button in order to manage any conflict between them or to continue the current motion of the motor unchanged.

As we currently have AUTO OPEN command, we first check if this is the first button press for AUTO OPEN. We check this because, for it to get to the manual command, there needs to be a transition from auto to manual. If it is not the first press of the button in auto, we set the motor to STOP command.

We analyze the next button state as follows:

- ➔ if the press is AUTO OPN, the motor is set to AUTO OPN
- ➔ if the press is MAN OPN, the motor does the transition from AUTO OPN to MAN OPN
  **Obs:** Also, here the flag for the first button press on AUTO OPN is set to FALSE because before MAN OPN, there was first AUTO OPN which is considered as the first AUTO press. The next AUTO press after this MAN OPN command will be considered as second AUTO press in which case the motor receives STOP command.
- ➔ if there is no press on the open button, the motor remains in AUTO OPN and the flag is set to FALSE as the first auto press just occurred

  After switch 1 (OPEN switch) is checked, the state of switch 2 must be checked as well. Anything other then OFF button command (no press on button 2) is considered as conflict between buttons and the motor receives STOP command.

- *MAN OPEN Command*
  The current motor command is MANUAL OPEN. Next, the state of each button is analyzed to set the next motor command.
  - ➔ if switch 1 is OFF (that means it's released), the motor command is OFF as the MANUAL command is running as long as the button is pressed.
  - ➔ if switch 2 is not OFF (that means it's pressed), it's considered as conflict between the buttons and the motor stops immediately as the STOP motor command is set.

- *AUTO CLOSE Command*
  The current motor command is AUTO CLOSE, which is set from switch 2. This is similar to AUTO OPEN command just that the logic is for switch 2 this time. We also have the first_auto_press_flag to differentiate between the first and the second auto button press.

- *MAN CLOSE Command*
  This command has the same logic as MAN OPEN, just for switch 2. If button 1 is pressed, the motor stops as it is considered as conflict. If button 2 is released, the motor receives OFF command (it immediately stops).

- *STOP Command*
  From STOP motor command, if neither of the buttons is pressed, the motor is disabled (OFF command).

- *OFF Command*

  From OFF motor command, in the first place, the transitions can only be AUTO.

  ➔ if button 1 is pressed (OPEN), the motor is set to AUTO OPEN

  ➔ if button 2 is pressed (CLOSE), the motor is set to AUTO CLOSE

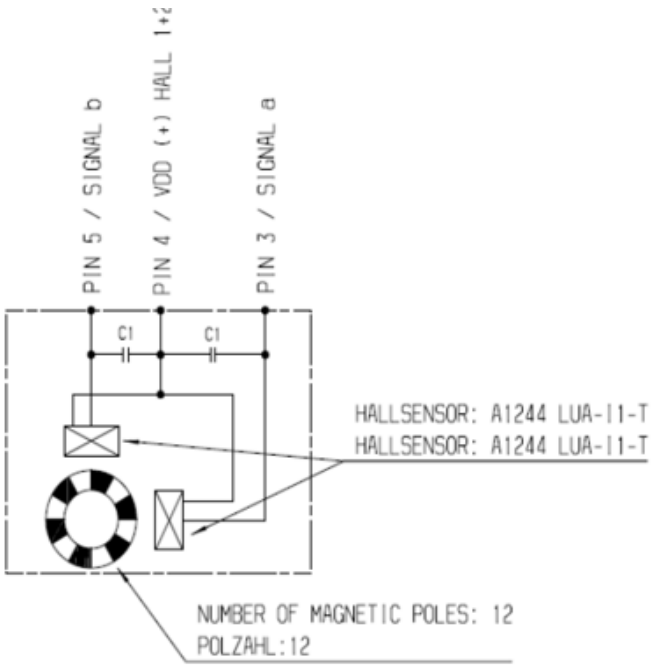
  **Obs:** The exception here is the use of a flag that signals the norming state of the motor. For now, it is good to know that when the flag is set to FALSE (motor is not normed), the only possible motor commands are MANUAL commands. So, instead of the transition from OFF to AUTO, there will only be OFF to MANUAL as long as the norming flag is FALSE. If the flag is TRUE, the motor commands are working as usual: **OFF -> AUTO -> MANUAL**

  (P.S. The norming concept will be discussed later)

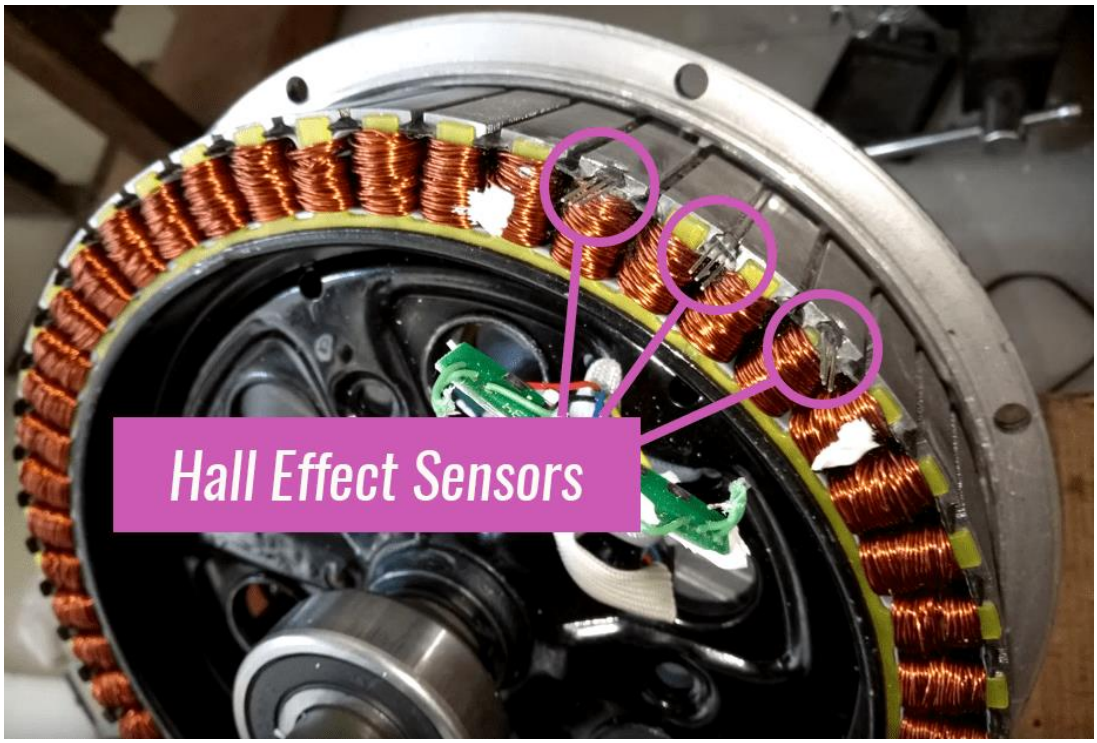
The output from Command Control module is then passed as input for Motor Control module.

## HALL SENSORS CONTROL MODULE

The 2 Hall sensors are incorporated into the DC motor as showed in the pictures below.



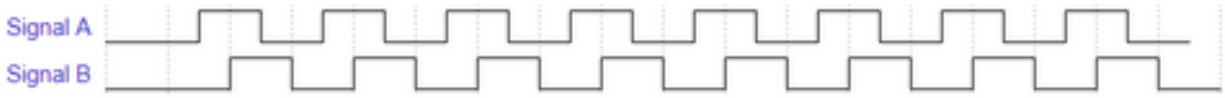

**The 2 Hall sensors in our DC motor**




**Hall sensors in real world motor for better visualization**

When the motor is in motion, the Hall sensors will generate a digital signal (with HIGH and LOW pulses) that is sent to the microcontroller as feedback from the motor.

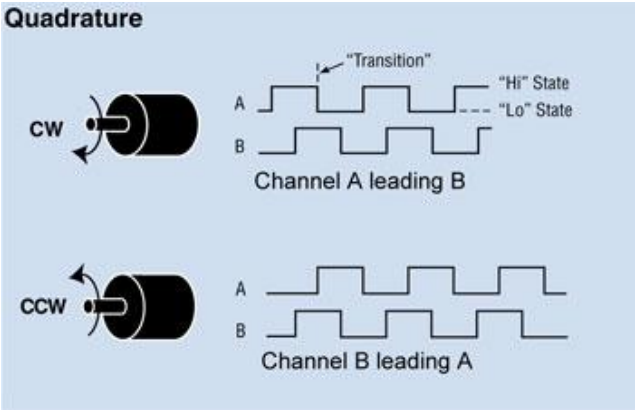
Because the sensors are placed side by side and not in the same place, so will their signals not be the same. The signal from the second sensor will have a small delay against the other.




This information will be used for:

- ➔ managing the position tracking of the motor = the number of pulses in one direction or another (the position will increment when motor is moving in OPEN direction and will decrement when the motor is moving in CLOSE direction)

➔ getting the direction of the motor




**Getting the direction of the motor using Hall sensors feedback**

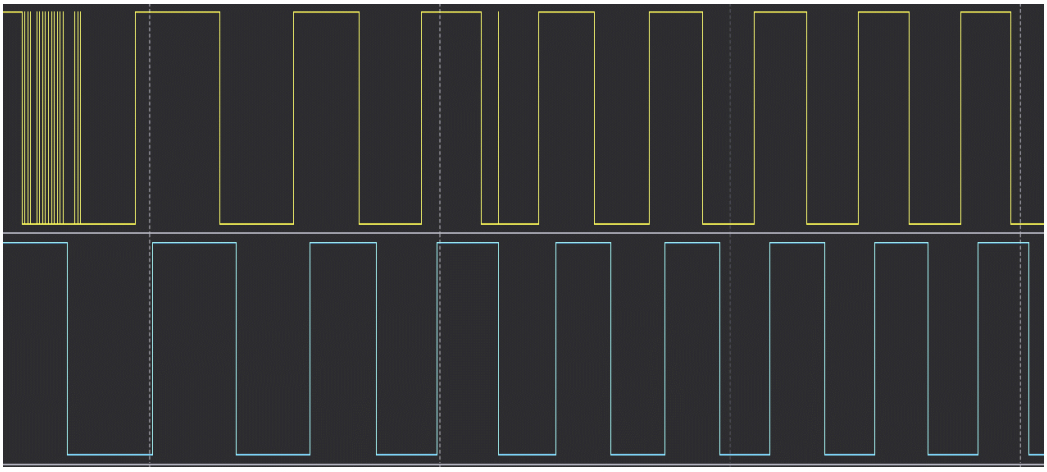
The directions of the motor are OPEN (clockwise) and CLOSE (counterclockwise). When the motor is rotating clockwise, we will get the first pulse from sensor A and, after a small delay, the pulse from sensor B as well.

When the direction is changed counterclockwise, the motor is rotating in the opposite direction, so now the first to generate a signal will be sensor B and after a small delay, sensor A.

- *Error filtering*

The signal that is received from the Hall sensors can also have multiple interferences that will look like spikes when visualizing the digital signal. This will not be useful for our application because we need the value of the motor position to be accurate and not to count, beside the valid pulses, the interferences as well.

Error filtering on the Hall signal will be needed for it to give an accurate and useful value of the number of pulses. This way, only the valid pulses will be used in implementing functionalities that use this information.



**The blue signal is the one filtered of any interferences that we see on the yellow signal**

This error filtering is done in a similar way to the debouncing of the buttons from earlier.

ErrorFiltering() is a function that is called in the main task. It uses the information from the attached interrupts of the Hall sensors. These interrupts will be triggered on the rising and the falling edges of the signal. The function will get this info in the form of boolean values: 1 or 0 (HIGH -> rising edge, LOW -> falling edge).

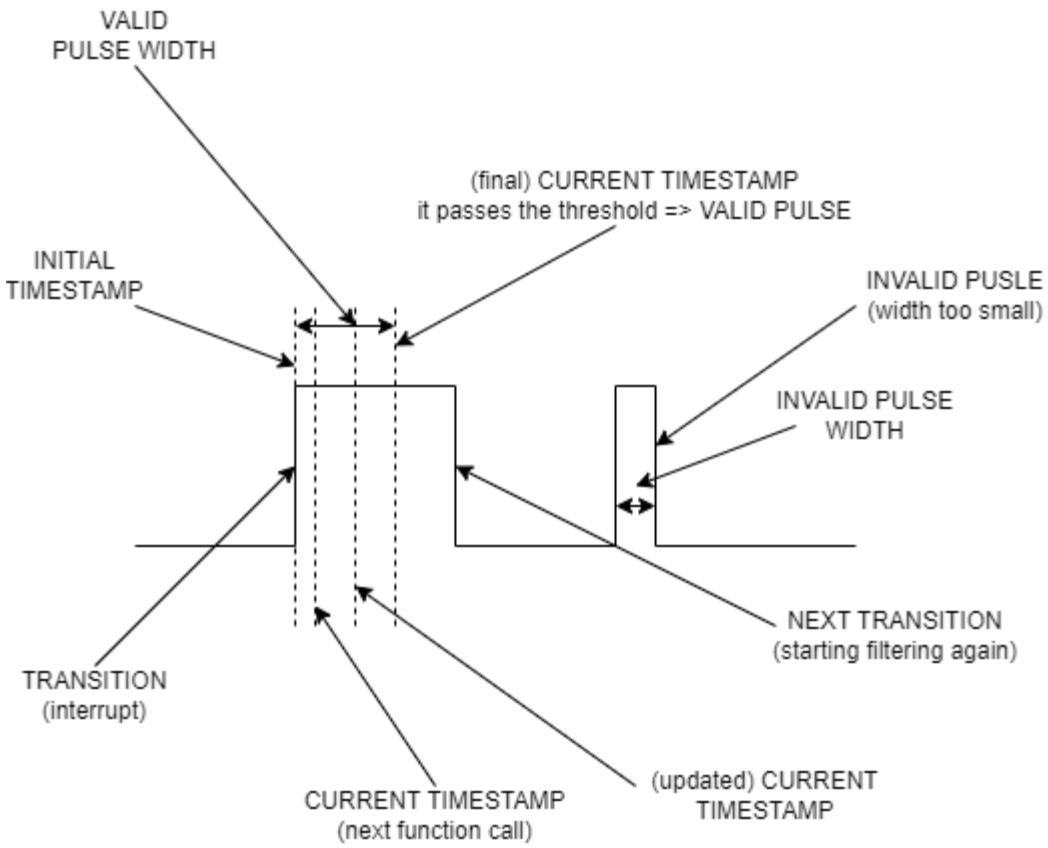
First, we check that there is a transition of the signal (HIGH -> LOW or LOW -> HIGH). That means that the previous state of the pulse must be checked as well to see if it is different than the current one.

**TRANSITION: previous = HIGH, current = LOW**

**previous = LOW, current = HIGH**

When we get a transition, in that moment a timestamp is set as the initial timestamp. After that we set the current timestamp that is updated at every function call until another transition of the signal occurs.

In other word, the transition occurs when an interrupt is triggered (falling/rising edge) when it is the moment to take the initial timestamp on signal change. In order to measure the width of the pulse we update repeatedly the current timestamp until the next interrupt trigger.




**How the error filtering works**

By setting a threshold parameter, if the width of the pulse is under this parameter value, the pulse is considered as an error and is not to be taken into consideration. On the other hand, if the pulse is over this threshold, it is considered as valid and can have further uses in the application.

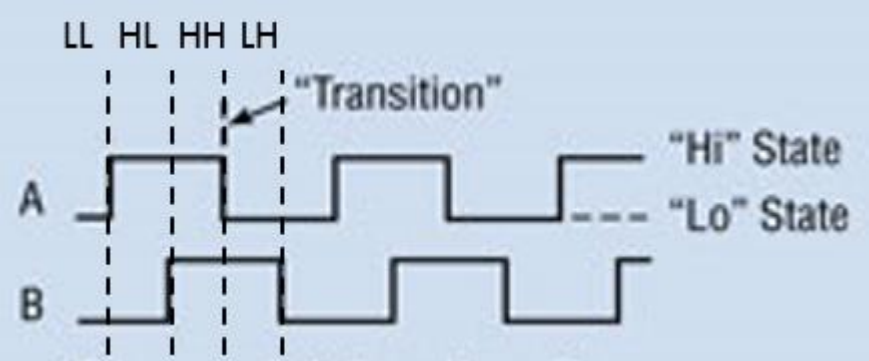
We do the filtering for each signal separately (signal A and signal B).

- *Position Tracking*

The signal filtered from any errors will function as input for position tracking.

This function will send an input to other functions, input that behaves as a reference of the motor movements. Each pulse that we get from Hall sensors when the motor is rotating is added to or subtracted from a variable that represents the position.

Position tracking needs to be done on half pulses for better accuracy. (see the picture below)




In the implemented function, using a switch case, each transition is analyzed to check in what direction the motor is rotating.

When we check the state of half pulses for both signals, we need to also check the previous states so that the position and the direction of the motor can be correctly determined. For example, if the current states (the current half pulses) for both the sensors are LOW, we check the previous states which are: LOW (sensor A), HIGH (sensor B) for clockwise direction and HIGH (sensor A), LOW (sensor B) for counterclockwise direction.

So, we have a switch with all these possible combinations of current states as cases of this switch and inside each case we have a switch with all the combinations of previous states which then can manage the position and the direction of the motor.

The position and the direction determined in this way can then be used for further motor command.

## MOTION HANDLING MODULE

In this module, there are functions responsible for managing the occurrence of a block and also for the ramp up and ramp down of speed when the motor gets a command.

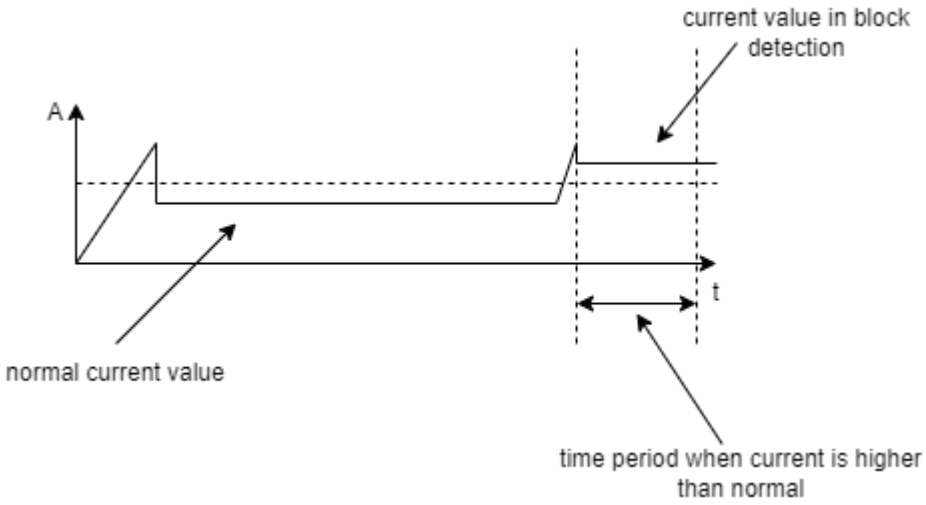
In the Block Detection function, we detect any obstacle that may appear when the motor is in motion. This detection is done for 2 cases: once when the Hall sensors work correctly giving feedback to the microcontroller in form of digital signal and once when there is some type of error with the sensors and we don’t get the feedback as usual.

*When the Hall sensors work correctly:*

In order to measure the time period starting from the last triggered interrupt on the Hall sensors, we set an initial timestamp on every interrupt that occurs. We update the current timestamp at every function call and when the difference between these 2 timestamps is bigger than a set threshold and also when the current that we get is the current that the motor normally gets when it's rotating, we consider that we have a block detection.

*When we have error on Hall sensors:*

In this case we can't rely on the sensor signal so we make use of the current values which, in block detection is higher than normal value for a longer period of time.




**Variations of current on normal motion and on block detection over time**

When we want to detect a block that may appear during motor motion, we first need to check if the Hall sensors work correctly. If they have a short circuit error, we can't detect the block based on the signals that we normally get from the sensors.

We have the condition that signals if motor is running or not. If, over a set period of time, there are no pulses received, the error flag is set to true and the motor is also denormed because the current position based on the number of pulses may have been compromised. This error flag is reset whenever an interrupt is triggered (that means whenever we get a rising or a falling edge of a pulse). So, if that period of time passes without the flag getting reset, it is considered a sensor error and we need to do the block detection based on the current feedback only.

Besides the block detection concept, in Motion Handling module is implemented the ramp up and ramp down of the speed when a given motor command is executed.

This means that the PWM that is sent to the motor driver in order to give the motor speed is set to increase gradually over a short period of time at the beginning of motion until it reaches maximum speed.

In a similar way works the ramp down but this time the PWM is decreased gradually that in the end reaches the minimum.

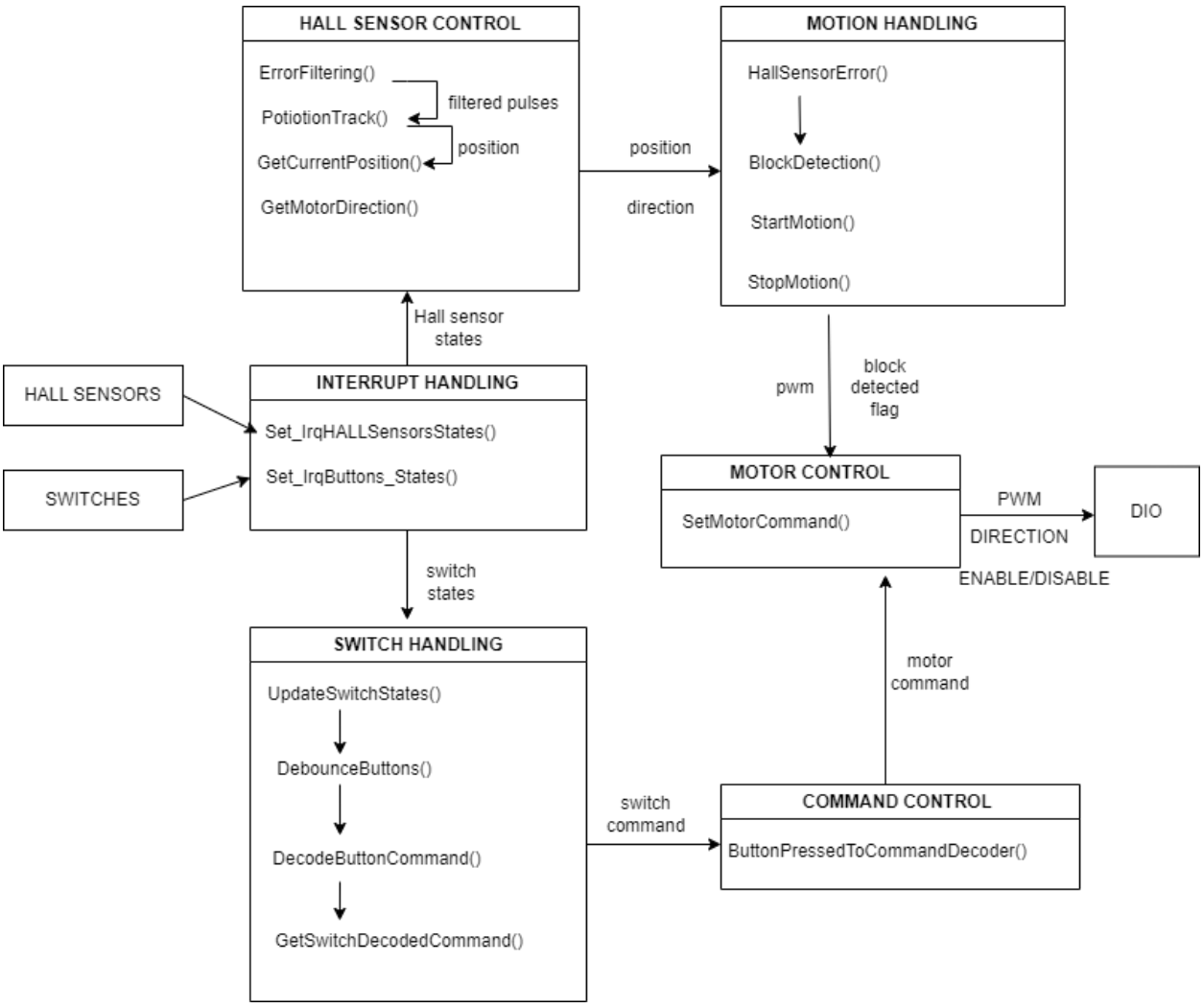
## MOTOR CONTROL MODULE

In this module we specify what each motor command means for the motor driver. That means we need to give input for the enable, the PWM and the direction pins on the driver.

For each command, before giving this information we need to check if there is a block detected at the moment. In this case, the motor will get a reverse motion command and after that it can receive any other usual command.

The reverse command (one for each direction) is given only when there is block detection in order for the obstacle to not get damaged. This reverse motion consists of rotating the motor in the opposite direction for a short amount of time and it then stops.

After this block_detected flag is checked, the usual command parameters (enable, direction, speed) can be set. The only observation is that it needs to perform between 2 set limits (close and open limits). When the position gets too close to or over either of these 2 limits, the motor receives the stop command regardless of the current command (so that it can’t force these limits).




**Project Modules Diagram**