# Software and data handling.

## Overview.

The following chapter introduces EX1 software design. This can be divided into two main parts: one regarding the central operating system (OS) and one regarding low-level control software. The central OS runs on the P-OBC and supports the following general functionality:

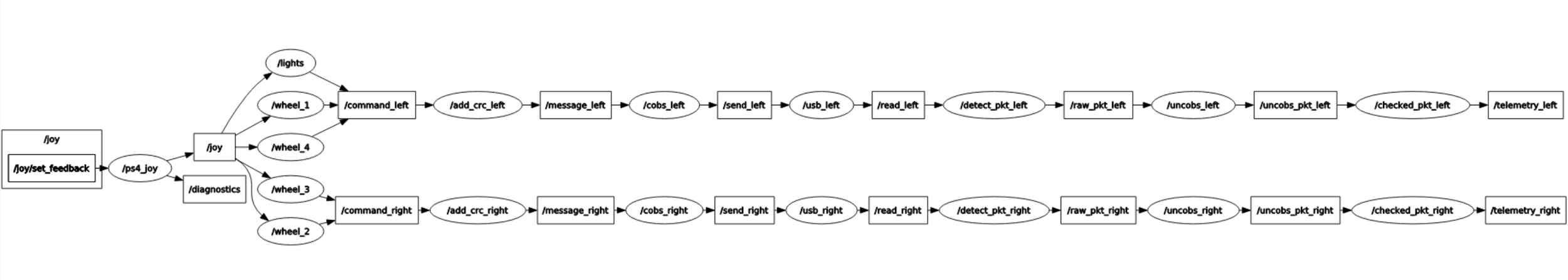
* Wireless communication with Ground Control Station (GCS) via a WiFi Network.
* Wireless communication with Teleoperation Joystick via Bluetooth 4.1.
* High-level control tasks, in particular those related to locomotion, i.e., driving forward/backwards and steering.
* Serial communication of commands with low-level control software.
* Telemetry data retrieval and storage.
* On/Off switch of low-visibility lights.

On the other hand, the low-level control software is in charge, as the name suggests, of handling low-level control tasks; thus, it runs primarily on the motor controllers of the ADE. Low-level software general functionality includes:

* Communication with central OS for the transmission of commands and telemetry data requests.
* Execution of low-level control tasks.
* Reading ADE sensors data.

## Operating system.

The central OS is based on the **Robot Operating System (ROS)** **platform**, in particular, the **Melodic Morenia distribution** over **Ubuntu 18.04**. The code is, for the most part, implemented in **C++**. All the code associated with the central OS can be found in the repository *h*[*ero\_ex1\_sw*](https://bitbucket.org/hero_team/hero_ex1_sw) at HERO team’s bitbucket workspace (<https://bitbucket.org/hero_team/>). The complete node graph can be seen in Fig. 6-1. The vast majority of the nodes and topics created are used as part of the internal communication protocol (refer to Section 7.2.). For details on the different topics, custom message types, and nodes used, one should refer directly to the actual files in *h*[*ero\_ex1\_sw*](https://bitbucket.org/hero_team/hero_ex1_sw) repository.



***Fig. 6-1.*** *ROS node graph of the central OS.*

## Low-level software.

As already mentioned, the low-level software is implemented in the motor controllers and it is, therefore, in charge of listening and executing orders received from the central OS running on the P-OBC, informing the central OS of the current state of all the ADE components at a frequency high enough to allow the central OS to be able to make automatic decisions on a timely manner. The language used for the low-level software implementation was **C**. All the related code can be found in the [*3\_ex1\_hibot\_sandbox*](https://bitbucket.org/hero_team/3_ex1_hibot_sandbox) repository.

The Integrated Development Environment (IDE) used for developing the low-level software was **Eclipse for C/C++[[1]](#footnote-0) with the associated GNU for ARM Embedded programming toolchain**. For more detailed information, the HiBot company made a tutorial on how to link Eclipse with the HiBot Titech M4 micro-controller[[2]](#footnote-1). All the low-level software code has been appropriately documented using the **Doxygen[[3]](#footnote-2) syntax** and the available Doxygen plugin for Eclipse. It results in a so-called embedded documentation that everyone can generate on demand in both PDF and HTML format.

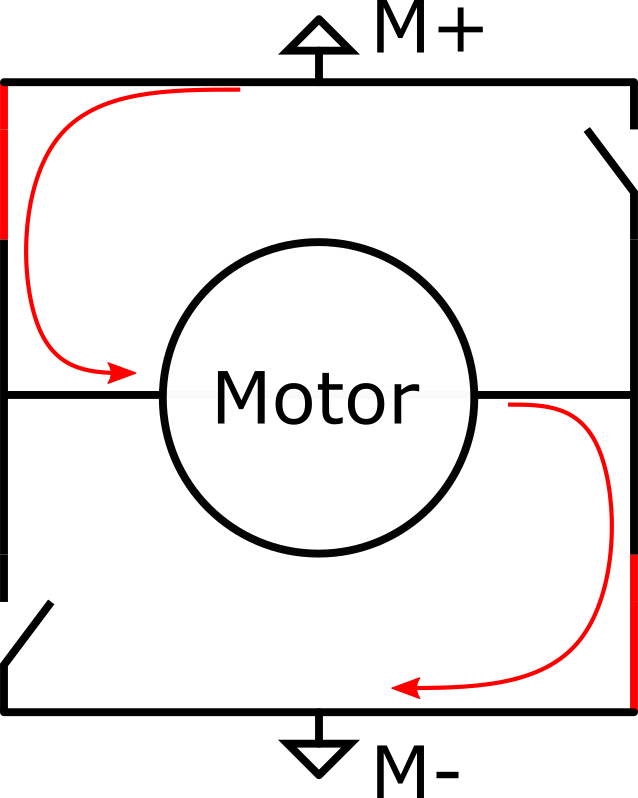
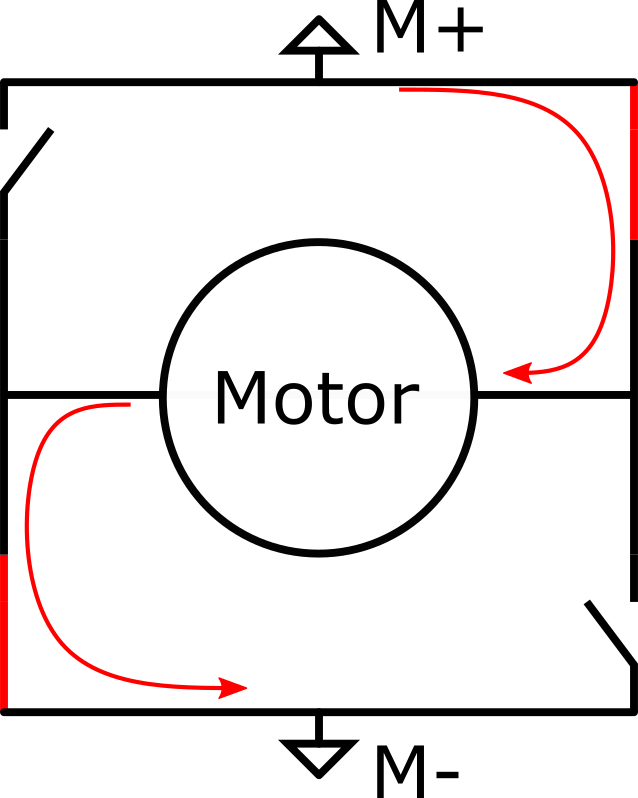
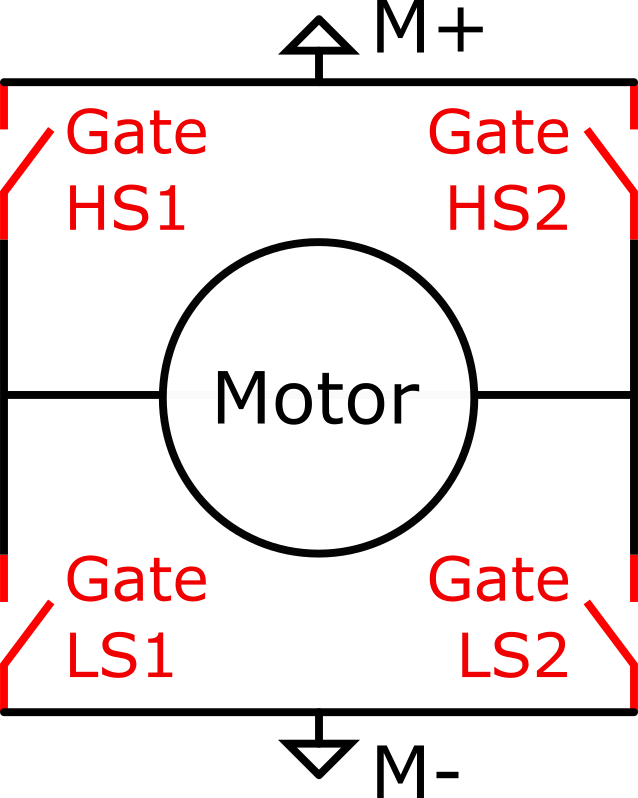
A more detailed functionality of the low-level software, which will be explained in the following sections, include:

* Communication of commands with central OS.
* Execution of commands. In particular, automatic speed (driving motors) or position (steering motors) control for each motor. This includes:
  + Double PWM application to each motor driver.
  + Reading output through encoder data.
  + Automatic adjustment based on closed feedback loop.
* Constant supervision and report to the central OS of the ADE state; this includes: data of the position and velocity of steering motors, velocity of driving motors, current drawn by each motor, and individual rocker angles.

*Note: the following subsections provide an in-depth description of how the low-level control tasks were implemented and the functionality and theoretical principles behind them. They are not considered absolutely necessary unless low-level improvements are intended. General users of the EX1 platform are encouraged to skip these sections*

### Motor control.

A motor driver consists primarily of an H-bridge—a circuit of 4 switches—allowing a motor to turn both clockwise (CW) and counterclockwise (CCW). Its principle of operation is shown on Fig. 6-2. As can be seen, the direction of rotation of the motor can be controlled based on the On/Off configuration of the H-bridge



1. *H-bridge gates naming convention. (b) CW configuration (c) CCW configuration.*

***Fig. 6-2.*** *Working principle of an H-bridge.*

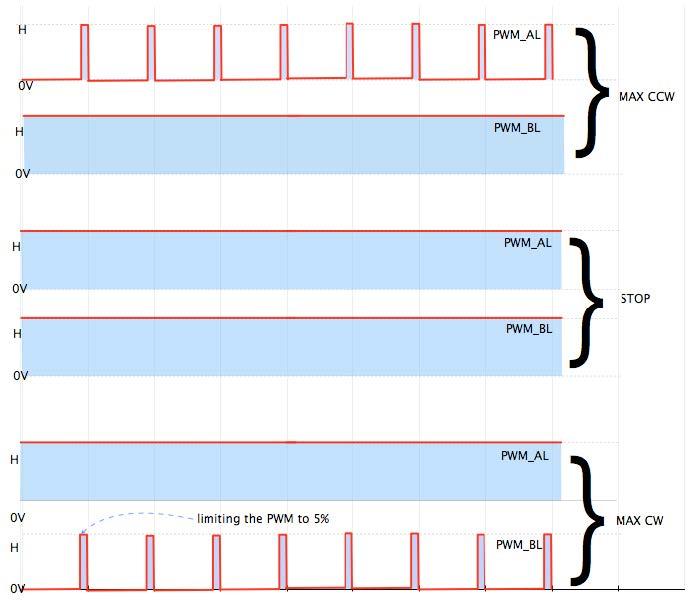
In order to control a motor through its motor driver in both CW and CCW directions, two PWM signals are required (PWM AL and PWM BL) as per the truth table displayed in Table 6-1.

***Table 6-1****. Truth table of the H-bridge.*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| PWM AL | Gate LS1 | Gate HS1 | PWM BL | Gate LS2 | Gate HS2 |
| 0 | Open | Closed | 0 | Open | Closed |
| 1 | Closed | Open | 1 | Closed | Open |

In order to control a motor as wanted, one needs to follow the chronogram shown in Fig. 6-3. It should be noted that in order to change the speed of rotation, it is possible to change the duty cycle of the active PWM (i.e., change the time at low-state for the non-constant PWM signal).

*Note: It is also important to note that the motor driver used does not make use of a built-in charge pump. This means that the duty cycle of the input PWM signal has to be necessarily between 10% and 90%. Tests were conducted outside this range and the consequences have been spurious and jittering effects experienced by the motor. This should be regarded over the 5% to 95% working range described in the motor driver datasheet (see Appendix C - Datasheets).*



***Fig. 6-3.*** *Different command signals applied to the H-bridge and their effect.*

The PWM signals are directly generated by the built-in timers of the micro-controllers. The level of command is directly linked to the duty cycle of the active PWM. The PWM frequency has been chosen to be 32 kHz for the following reasons:

* A frequency above 20 kHz ensures a reduction of motor operating noise.
* The selected frequency needs to be low enough to comply with the driver capabilities, in this case 200 kHz based on the motor driver data sheet (see Appendix tk- Datasheets).
* A slower PWM frequency implies that more clock ticks exist during one PWM period and, consequently, the accuracy over the duty cycle (the command level) improves.
* Due to the motor being a second-order low-pass electro-mechanical filter, the higher the frequency, the smoother the output.

The *motor\_driver* package includes initialization functions and an apply-command function. This apply-command function changes the active PWM and its duty cycle according to the sign of its argument and its magnitude. Note that this apply-command function has nothing to do with automatic control; it only applies a raw command. Automatic control will be the subject of another section.

### Encoder reading.

Each motor comes with a built-in quadrature encoder; i.e., a relative angular positioning sensor. A quadrature encoder is capable of providing information on both displacement and direction of motion. This type of encoders are characterized by the number of ticks per revolution, which in turns dictates its level of accuracy. By counting the number of ticks on the output of the encoder, one can linearly compute the angle of displacement.

#### Position reading

The microcontrollers used on EX1 come with a built-in encoder interfacing hardware. This allows us to have an accurate counting mechanism without completely depleting the computational resources of the microcontroller. This interfacing hardware feeds a N-bits (32 its for driving motors, 16 bits for steering motos) counter that keeps track of the number of ticks given by the encoder.

In order to be able to record both CW and CCW rotations, the counter is initialized at *Co=2N-1*. The total angular displacement on the output shaft with respect to the initialization time of the counter is given by:

,

where is the total output travel in degrees, is the difference from reference point in the interface counter in number of ticks, is the resolution of the encoder in degrees/tick, and is the reduction factor between the motor and the output shaft (smaller than 1 in this convention).

One may notice that there is a problem for *2 N-1* . This is due to the counter not being able to count anymore; this problem is known as overflow or underflow and will be covered later on.

Assuming that over- or underflow has not happened yet, the angle of the output shaft with respect to the startup angle can be known at any moment in time, by simply reading the counter and using the formula previously mentioned.

#### Speed calculation

Unlike position reading, speed calculation requires dealing with time. Speed calculation requires a first order derivative at an approximately constant rate. This has two major effects. On the one side, the position is known with a quantization error, meaning that deriving over extremely short periods of time will result in very noise-prone and inaccurate results. On the other side, if the derivative time step is too long, the automatically controlled system becomes slow and unstable in its response.

In order to protect against noisy outputs due to short derivation periods, the speed computation function stores its last call timestamp and acts consequently (i.e, by giving the last speed value known if the last function call is below a time threshold, or otherwise too recent). To protect against excessively long derivation periods, the speed computation function is periodically called, regardless of whether it is needed or not, in order to keep the system up to date on the motor speed.

Taking the derivative of the position between 500𝜇s and 10ms has so far proven to be qualitatively and experimentally acceptable for our application. The speed is then given by

#### Handling over- and underflow

A preventive software action was implemented in order to deal with this problem. At a certain amount of ticks before over- or underflow occurs, an interrupt signal is emitted. The ticks counter is immediately reset to its middle position *2N-1* before incrementing (when underflown) or decrementing (when overflown) the counter.

The total angle travelled is then given by the following equation.

,

where is the algebraic number of over- or underflow (a negative number indicates underflow while a positive number indicates overflow) and is the number of ticks between the middle position and the number of ticks at the moment when the interrupt was sent in absolute value. As you may expect, the interrupt signal needs to be emitted before an over- or underflow occurs and not exactly at this moment ( *2 N-1).*

### Automatic motor control.

For both steering (position control) and driving (speed control), the **control scheme** is a **single input** (speed or position measured by the functions mentioned earlier) **single output** (active PWM selection and magnitude) **PI** (Proportional Integral) **loop**. Some extra functionalities were included, such as integral saturation.

Proportional action is taken to have "reactivity" in the system, and to reduce the time needed for the system to reach the set-point. Integral action is taken to cancel the error over time, making it tend towards zero as time increases. These two values—proportional and integral gains—have been tuned experimentally.

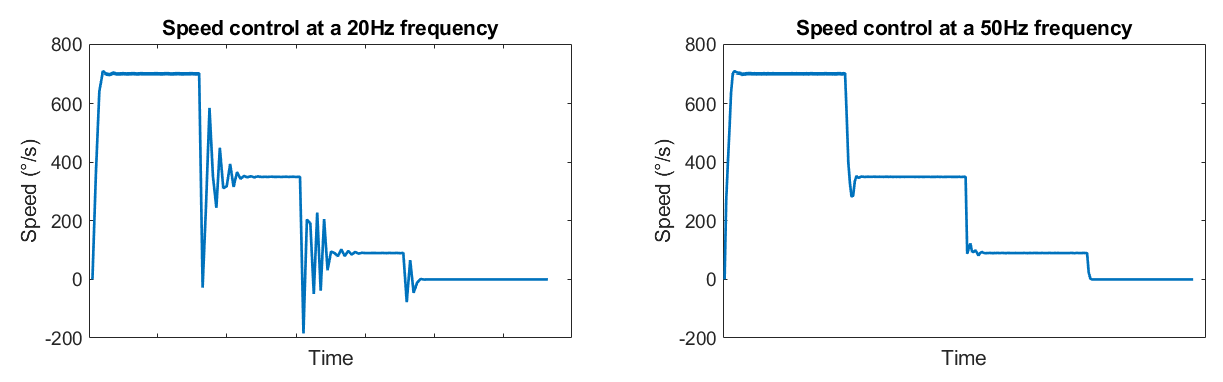
*Note: Indeed, automatic control is a science field of its own, and a lot more could have been done to characterise and model the responses and reactions of the system depending on the control algorithm, the strategy employed, the way to compute velocity, etc. Future students may consider looking deeper into this subject.*

A derivative term was not introduced, as derivative gains add a lot of complexity to the tuning process and the computation process.

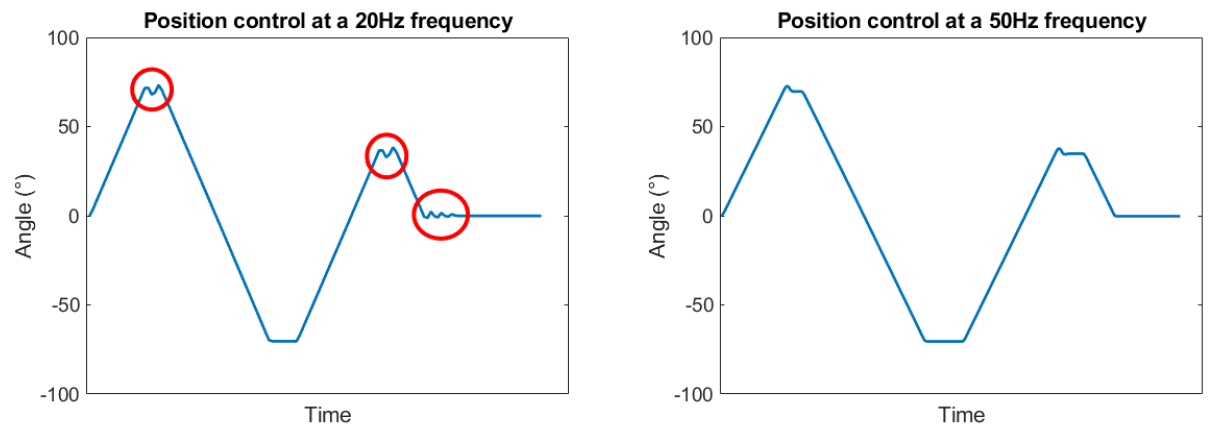
*Note: One should keep in mind that a third term to tune means another dimension in the space of available tunings and it takes more time to explore it. A derivative term also implies, for speed control, that the digital signal (the encoder output) had to be derived twice, resulting in even more noisy and inaccurate outcomes if special derivation techniques, which are often mathematically complex, are not implemented.*

Additionally, a set band has been put in place for position control, as readings from an encoder (which is discrete) and a given setpoint cannot exactly match. A control frequency sequence was also implemented. As our control is not a continuous control but a discrete one, instability is introduced by the control algorithm itself. The lower the frequency of operation, the more unstable the system becomes, but on the other hand, the higher the frequency of operation, the more computational resources are needed from the microcontroller.

Measurements were taken through UART (see Section 7.2. for details on the communication protocol implemented) for different frequencies of operation. One can see in Fig. 6-4 and Fig. 6-5 the effect of increasing the frequency of operation. Oscillations are easily observed on Fig. 6-4 and are highlighted on Fig. 6-5. Experimentally, **we choose to have a frequency of operation of 100Hz** as a good compromise. The control is good enough, and there is still plenty of CPU resources available for other tasks.



***Fig. 6-4****. Speed control with consecutive step commands for two different frequencies of operation.*



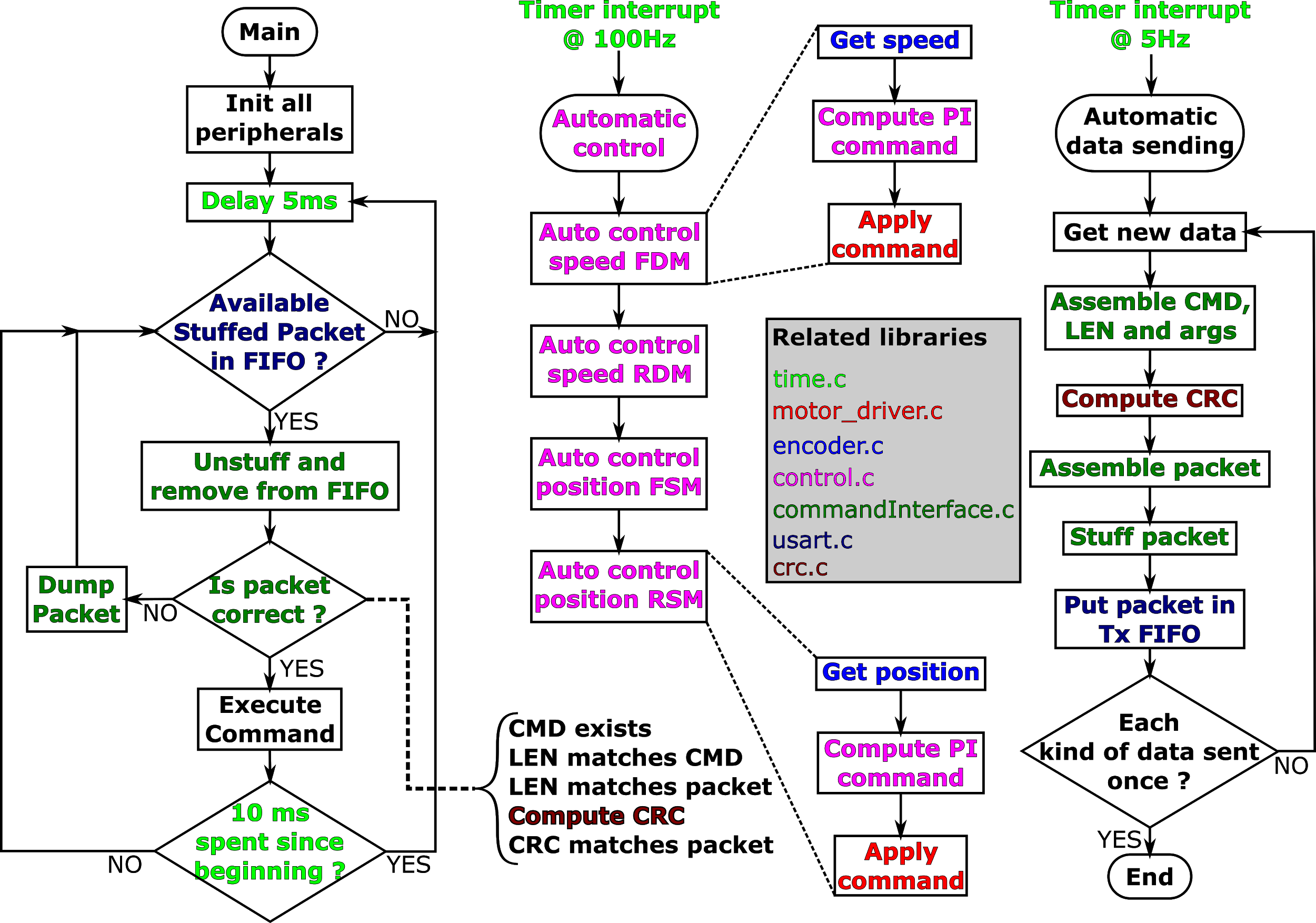
***Fig. 6-5****. Position control with consecutive step commands for two different frequencies of operation.*

### Analog reading.

Analog reading is used to read the current going through the driving motors. Current is an analog value given by the current reading device built in the motor driver. Analog reading is also used to measure the angle of each rocker, using a potentiometer between the main chassis and the rocker shaft. In this case, an analog value lies between 0 and 3.3V with a 12bits Analog to Digital Converter (ADC) (i.e. it outputs values between 0 and 4096).

After experimental testing it was noticed that a constant input had a variation of approximately 6 bits in magnitude, which means we can assume a deviation of on the result (this will be taken into account in the post-processing phase). The post-processing phase includes filtering in order to smooth out the ADC variation and the PWM control inherent current variations (motor turning on and off a fast rate). Remember that the latter has been already partially cancelled by the hardware filter described in the previous section.

An overview of how the higher layer of the low-level control software works is shown in Fig. 6-6. To completely understand this diagram one should be familiar with the internal communication protocol described in Section 7.2.



***Fig. 6-6.*** *Overview of the high-level functions implemented in the low-level software*.

## On-board data handling system.

All the sensor data is received by and stored in the following directory of the P-OBC.

*/home/rovertx2/HERO\_EX1/src/ex1\_startup/record/*

Data is stored in three different rosbags[[4]](#footnote-3), namely:

* ***telemetry\_right\_.bag***: with information pertaining to all proprioceptive sensors allocated in the right rocker.
* ***telemetry\_left\_.bag***: with information pertaining to all proprioceptive sensors allocated in the left rocker.
* ***imu\_data\_.bag***: with information of accelerations, orientation, and internal temperature provided by the IMU.

*Note: Telemetry data is not stored by default. One should pass a specific argument when running the startup launch command as specified in Section 8.2.2 in order for the rosbags to be initialized.*

1. https://www.eclipse.org/downloads/packages/release/2019-12/r/eclipse-ide-cc-developers [↑](#footnote-ref-0)
2. https://www.hibot.co.jp/uploads/ecommerce/products-files/9f8dd17aa0aaef4688a6a9e874bb1d4a.pdf [↑](#footnote-ref-1)
3. https://github.com/theolind/mahm3lib/wiki/Integrating-Doxygen-with-Eclipse [↑](#footnote-ref-2)
4. A rosbag or bag is a file format in ROS for storing ROS message data. These bags are often created by subscribing to one or more ROS topics, and storing the received message data in an efficient file structure [↑](#footnote-ref-3)