NATURAL BIONICS EU H2020 PROJECT

Electronic boards framework

Firmware, Hardware and Software Documentation

ISTITUTO ITALIANO DI TECNOLOGIA
SOFT ROBOTICS FOR HUMAN COOPERATION AND REHABILITATION

NATURAL BIONICS

Centro E. Piaggio
bioengineering and robotics research center

NMMI

Summary

Framework

This document is intended to explain the basics of hardware, firmware and software configuration and use of the NMMI logic board with PSoC5-based microcontroller.

Graphical schemes, design project including use case models, sequence diagrams, flow chart models, and other supporting requirement information are reported.

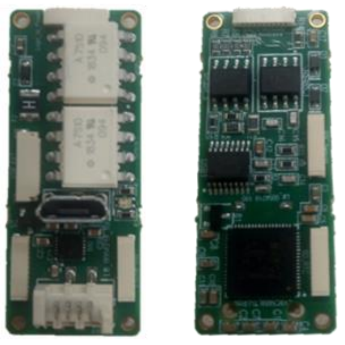


Figure 1: PSoC5 logic board

This document is divided into four main sections with different subsections respectively. First, a hardware description is presented with a brief description of the electrical scheme and the connections, then a general common firmware code flow is described, in order to highlight the main functionalities and how these interact. From this concept, three different firmwares can derive and are used: a *SoftHand Pro firmware*, a *Generic configuration* and a *Dummy configuration*. These will be explained together with the software used to communicate with the board and a related ROS node.

Hardware

Preliminary information and destination of use

The logic board is the main electronical board of *SoftHand Pro* device and Generic systems. It is usually used in conjunction with a power board in order to control devices motors, but it can be also connected to other electronic boards depending of the necessity of the configuration.

In this latter case, the other boards are directly powered by the logic board and work directly connected to it. This multi-layered programmable board manages logic signals and provide proper routing to interface the devices each other.

In the board there is programmable component: a PSoC5 by Cypress® microcontroller. For details on how this component works, please refer to its datasheet, LINK. Briefly, here are summarized the main characteristics of this controller.

- DEVICE NAME: CY8C5888LTI-LP097
- FAMILY: PSOC5LP 68-pin
- CPU Core: 32-bit Arm Cortex M3
- Max. operating frequency: 80 MHz
- EEPROM size: 2 KB
- FLASH size: 256 KB
- 38 GPIOs

For example, the logic board is used to control DC motor according with the data read by an external magnetic encoder, to read the signals from the EMG sensors and even to allow the system to communicate with a computer. It can read also analog sensors with ADC or IMU inertial units. It can be also used to merely as communication interface or to supply power to other boards in the system.

Electronic system

The logic board is a 41x16 mm electronic board, created and designed to control mainly the *SoftHand Pro* terminal device, but it can also be used in generic configurations. It can read external magnetic encoders, current sensors, input sensors and to control the motor, usually with one closed loop on the current position of the shaft of the motor, or with two closed loops; the first internal one, on the current position of the shaft of the motor and the second one on the current used up by the motor.

The board also allows to fix motor references also externally by a computer via an USB communication or by other boards via a RS485 communication along one of the two ERNI connectors on the bottom of the board. At the same time, using the RS485 communication the board can send to the other boards, which mount the same communication protocol, the values read from the sensors or to give them commands.

To control the motor, the board mounts a driver which receives the PWM signals generated by the micro-controller.

Electrical scheme

The micro-controller can read the values from magnetic encoders, EMG sensors, analog sensors and IMUs, can acquire digital measures of board voltage and motor absorbed current. It is also able to generate PWM signals to control motors and to manage the communication with a computer connected to the USB connector or other boards connected via the power/communication connectors.

The following figures show electrical schemes of the considered board. In detail, Figure 2 shows the connections to the microcontroller and the configuration of the on board IMU, while Figure 3 is about the communication part. This is composed by three components: an UART-USB converter and two RS485 transceivers. The UART-USB converter is used to allow the communication between the board and a computer. When the board receives a package data by the computer, it arrives to the micro-controller on the board and at the same time it is brought on the RS485 protocol and sent to the other boards, even, connected on the communication connectors on the bottom of the board. Vice versa if the micro-controller on the board send a package data it will be on the USB port to a computer (even connected) and to the other boards (even) connected to the communication connectors. In this manner the board is able to communicate with a computer and with other boards; to manage the communication between two or more board, each one has an its own ID, written on the EEPROM of the micro-controller.

Figure 4 shows how the board reads voltage generated by EMG sensors. In particular two optically isolated amplifiers are used (refer to manufacturer web-site for details - https://www.promelec.ru), this choice allows to convert EMG voltages to values which can be read by the analogue-to-digital converter inside the micro-controller, isolating at the same time from EMG voltage potentials.

Finally, Figure 5 reports all the connectors on the board, in particular there is an on board uUSB connector and a remotization of t through an external board, two encoder lines connector, the EMG connector, two connections for power boards, an expansion port and a ADC channels dedicated port. The list below summarizes the board connectors:

- Encoder-IMU 1 (Right Side)
- Encoder-IMU 2 (Left Side)
- SD-RTC module
- ADC port (for EMGs and analog sensors)
- Logic to Power (1st motor)
- Additional GPIO port (Logic to Power for 2nd motor)
- External USB support

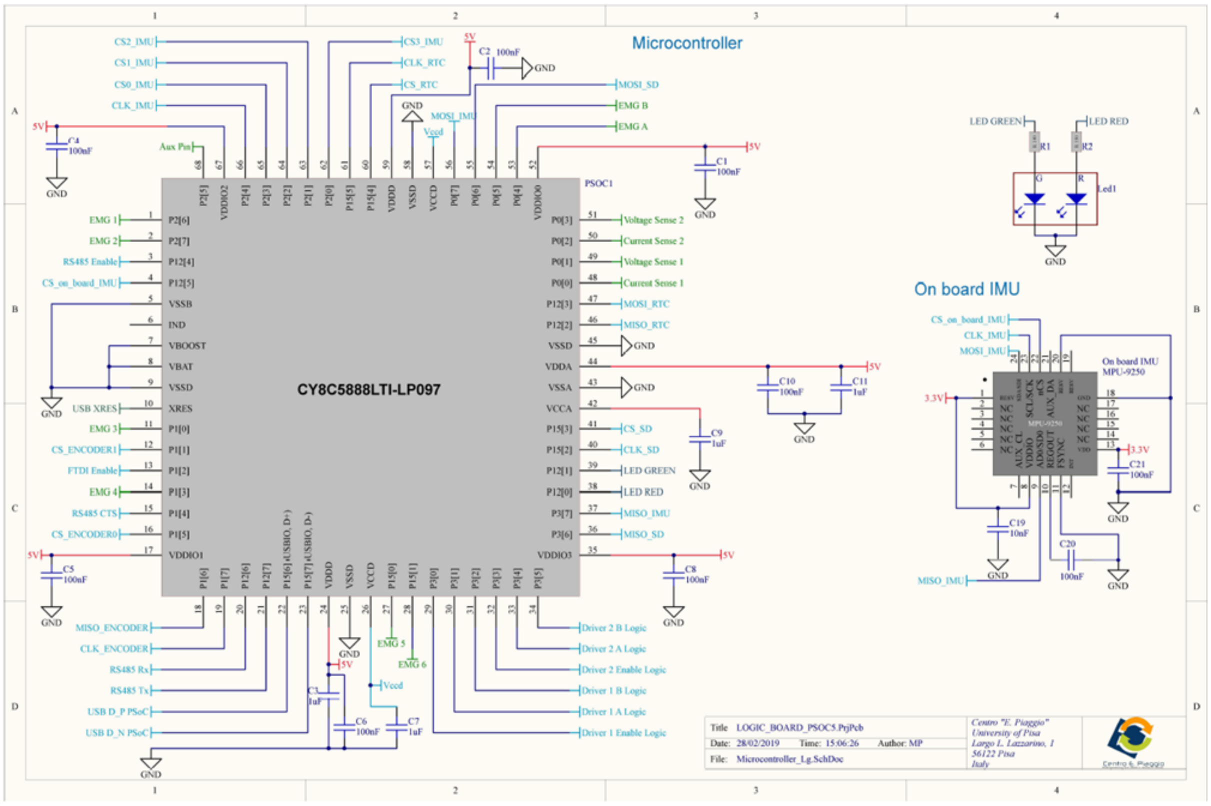

Microcontroller
CY8C5888LTI-LP097
PSOC1
On board IMU
On board IMU MPU-9250
Title LOGIC_BOARD_PSOC5.PrjPcb
Date: 28/02/2019 Time: 15:06:26 Author: MP
File: Microcontroller_Lg.SchDoc
Centro "E. Piaggio" University of Pisa Largo L. Lazzarino, 1 56122 Pisa Italy

Figure 2: Logic board: Micro-controller and on board IMU

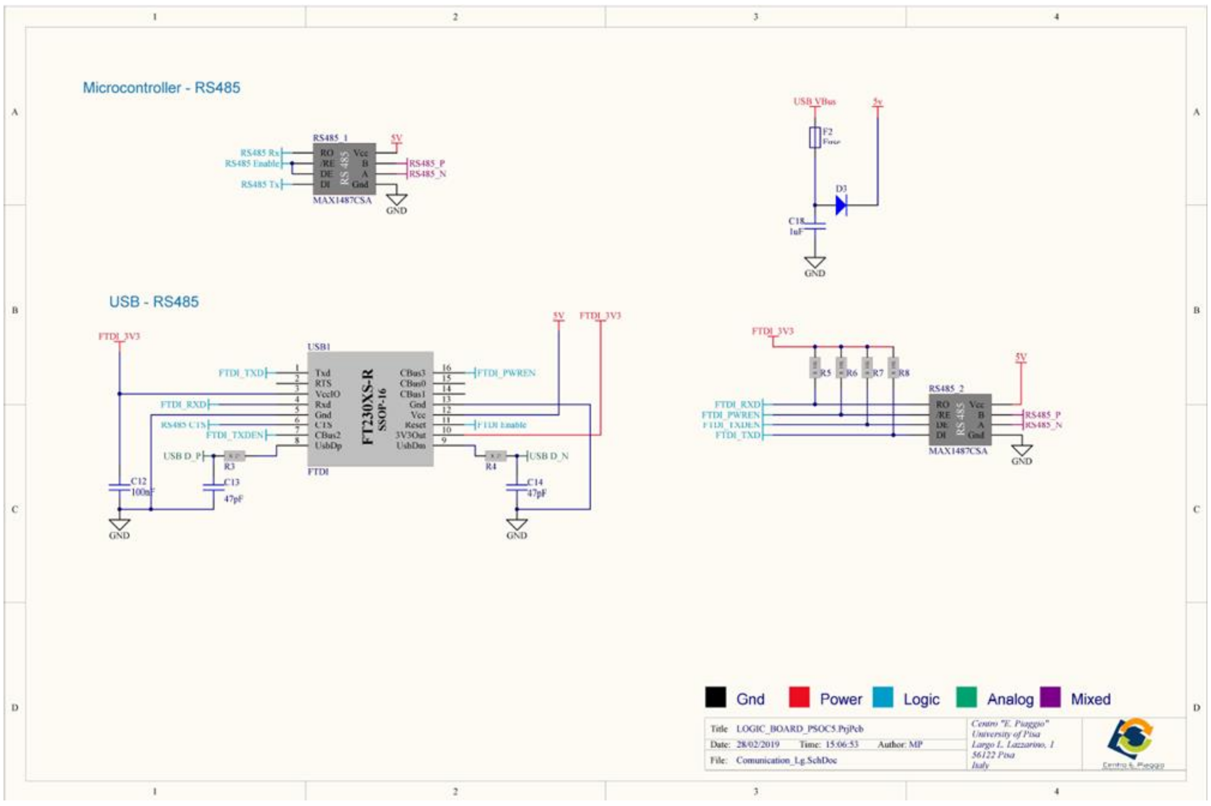


Figure 3: Logic board: Communication

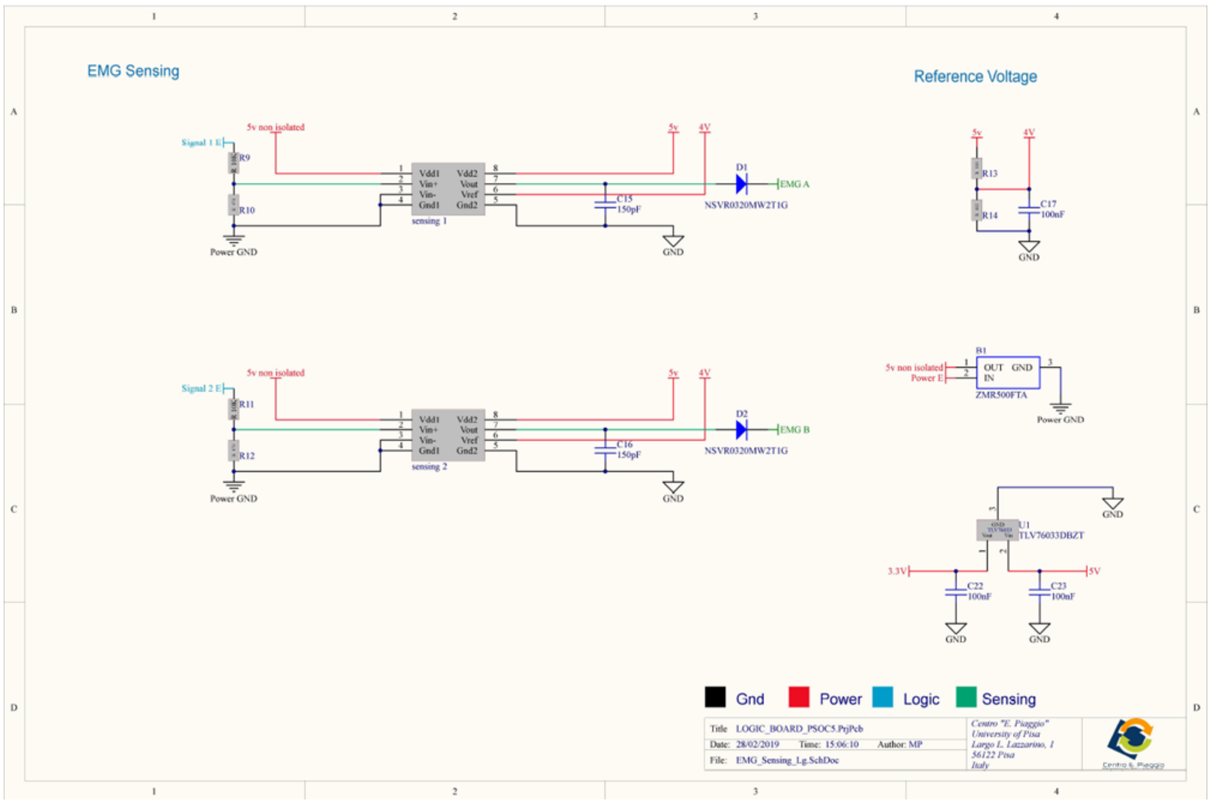


Figure 4: Logic board: EMG Sensing

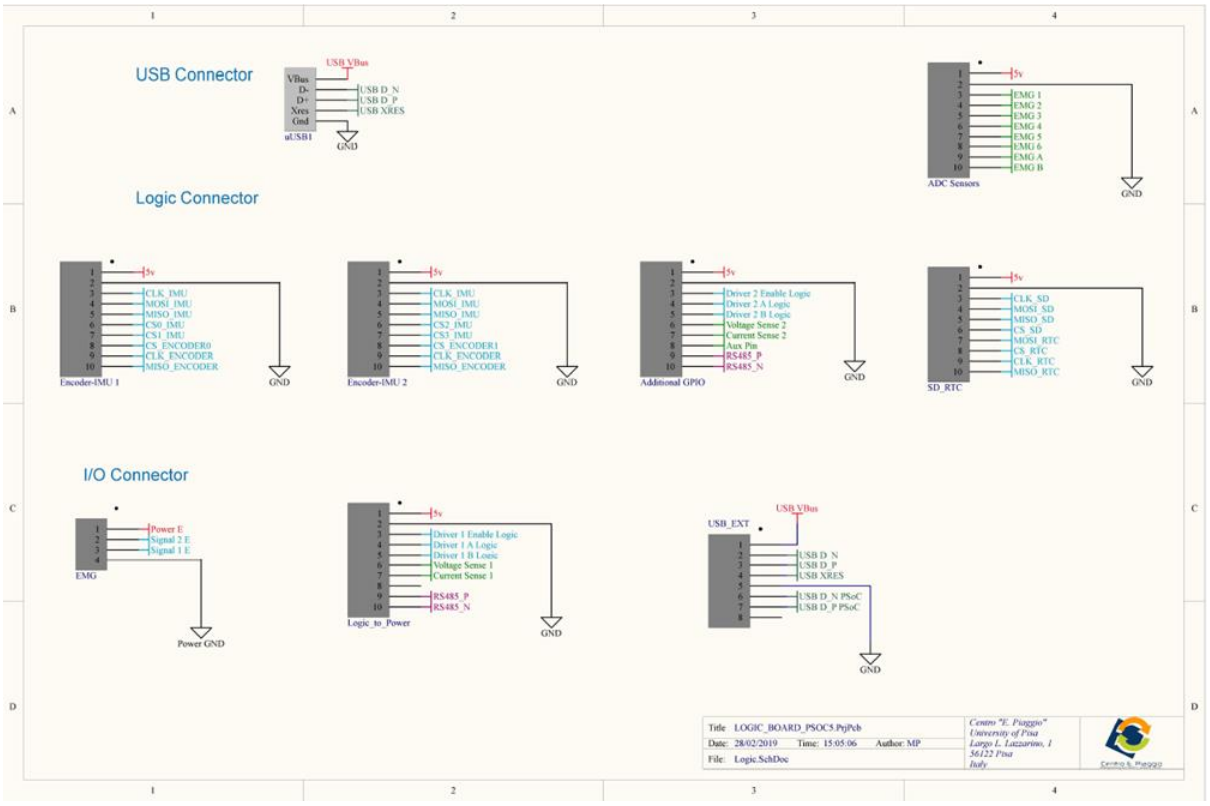


Figure 5: Logic board: Connectors

Board connections

This section simply reports how to make board connection when using one of the following configurations.

SoftHand Pro device

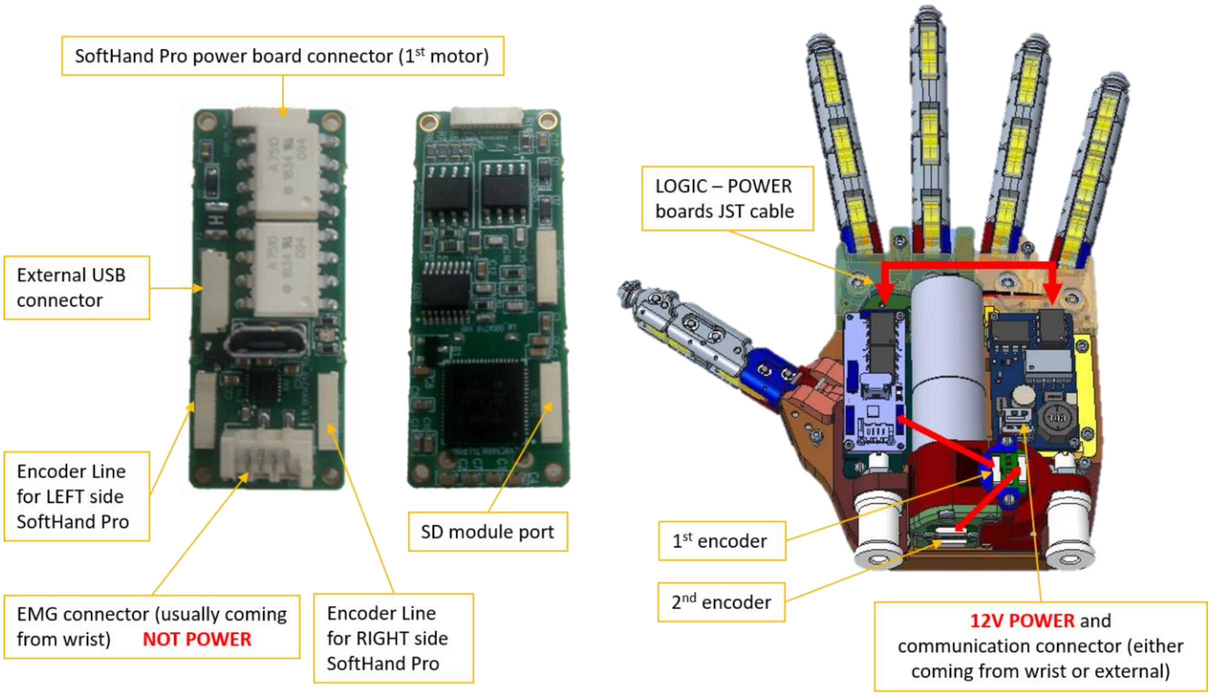

SoftHand Pro power board connector (1st motor)
External USB connector
Encoder Line for LEFT side SoftHand Pro
EMG connector (usually coming from wrist) NOT POWER
Encoder Line for RIGHT side SoftHand Pro
SD module port
LOGIC – POWER boards JST cable
1st encoder
2nd encoder
12V POWER and communication connector (either coming from wrist or external)

Figure 6: Logic board connections in SoftHand Pro configuration

Generic configuration

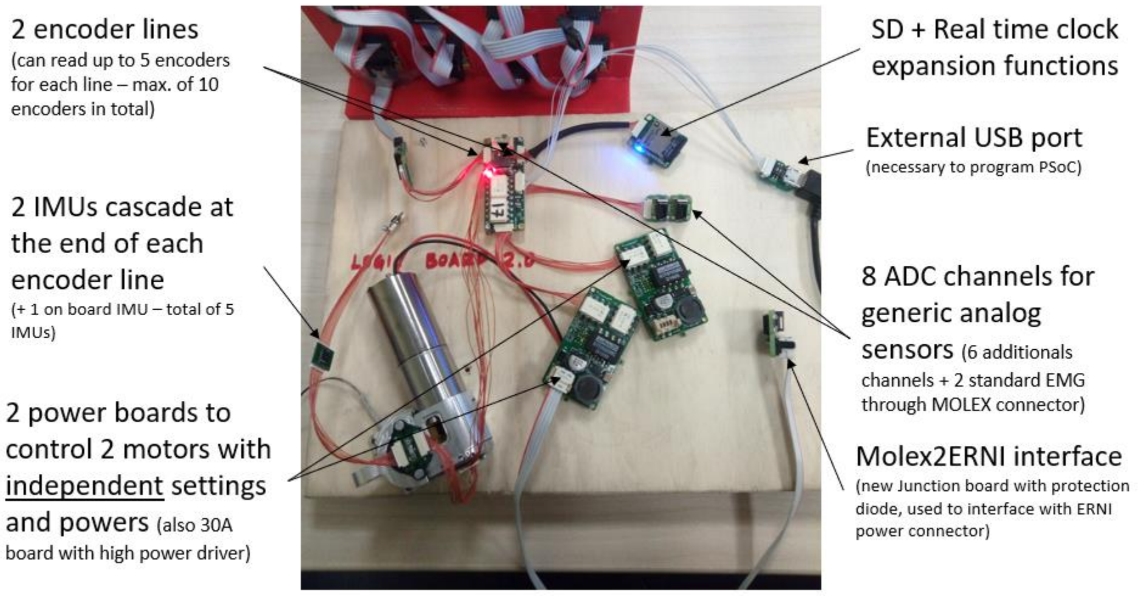

2 encoder lines
(can read up to 5 encoders for each line – max. of 10 encoders in total)
2 IMUs cascade at the end of each encoder line
(+ 1 on board IMU – total of 5 IMUs)
2 power boards to control 2 motors with independent settings and powers (also 30A board with high power driver)
SD + Real time clock expansion functions
External USB port
(necessary to program PSoC)
8 ADC channels for generic analog sensors (6 additionals channels + 2 standard EMG through MOLEX connector)
Molex2ERNI interface
(new Junction board with protection diode, used to interface with ERNI power connector)

Figure 7: Logic board connections in Generic configuration

Firmware

General firmware structure

All firmware is written in C language for Cypress® CY8C5888LTI-LP097 PSoC5 microcontroller. While the details will be described in a dedicated section, here a general structure is presented.

The PSoC is the main component of the logic electronic board and is the control unit of all our system devices. All the other electronic devices and boards are, at different levels, interfaced with such a microcontroller.

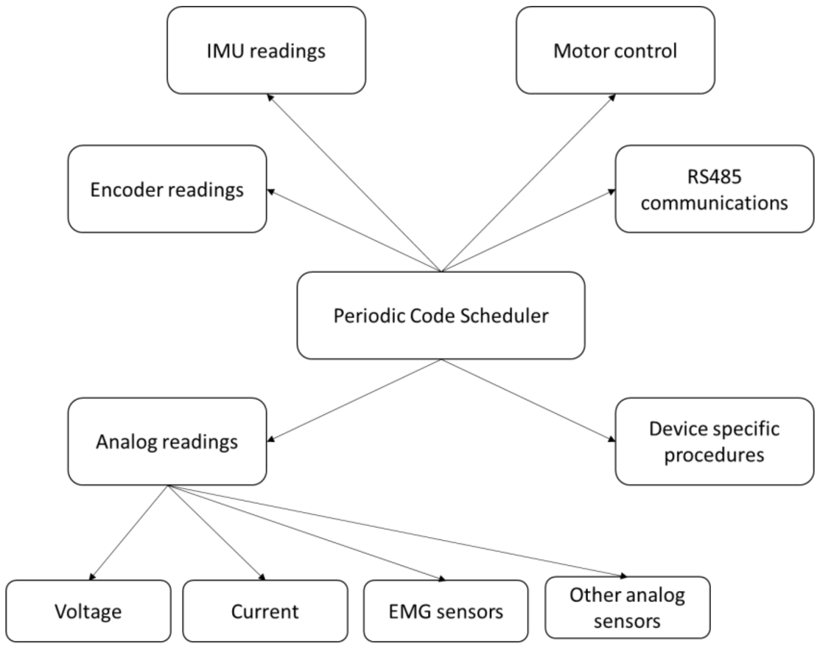


Figure 8: General firmware code flow

Figure 8 reports the main core of firmware: a function scheduler that executes periodically any of the tasks (Analog readings, Encoder readings, IMU readings, Motor control, RS485 communication, Other device specific functions).

At every scheduler cycle this sequence of actions is performed:

- digital reading of analog signals such as battery voltage, board current and electromyographic sensors (hereinafter called EMG sensors) or generic analog sensors (hereinafter called ADC sensors);
- digital reading of motor encoders to get current angular position of the motor pulley;
- digital reading of connected to board IMU and on board IMU (if enabled and configured);
- motor control action computation, through different types of controls and inputs;
- (if required) send or receive packets to/from other boards using serial RS485 communication protocol.

General communication framework

The firmware works as standalone, since the control of embedded device runs periodically. However, it is fundamental to have the possibility to access the firmware current state, get or set firmware parameters and in general, to exchange information with other devices or PCs.

The figure below shows an example of external communication framework: the firmware is equipped with a serial RS485 communication module that works as an interface between two firmwares configurations and/or between firmware and PC. These devices constitute a network and packets are exchanged between them at defined times and with a specific sequence and formatting.

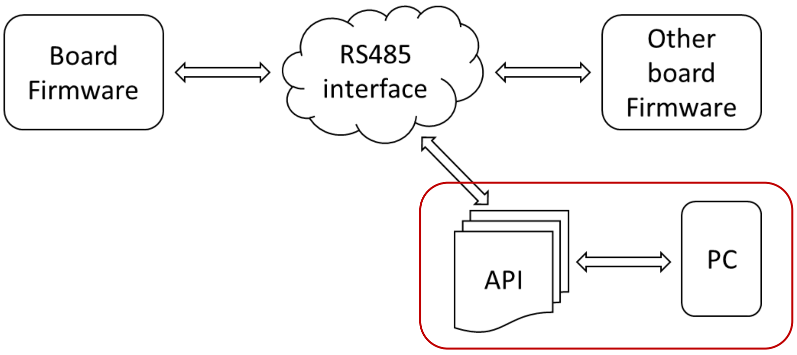


Figure 9: External communication framework

Users interface with firmware through red labelled elements in figure: custom software for PCs and Application Programming Interface (API) functions. For further information on APIs and communication packets, refer to section Software..

Firmware description

Firmware destination of use is intended for implementing safety measurements, controlling a DC motor connected to a power board, managing external sensors information and communicating with the final user.

Introduction of firmware lifecycle

Firmware usual main requirement is to control motor position in a reliable and efficient manner. For this purpose, a valid control loop based on measures acquired through a couple of encoders has been implemented. It is due to improve user's safety measurements to better monitor hand pulley angular position. Usually, first encoder is used to track motor pulley angular position and the second to count motor pulley turns. In this way, it is possible to reconstruct a reliable angular position to be used for control purposes. Position measurements are linked to motor position references through a standard Proportional-Integral-Derivative (hereinafter PID) controller. This aspect will be better analyzed in the following when motor control block will be explained.

General firmware code flow, described in Figure 8, has shown all the conceptual blocks that are implemented in the general firmware code. From this, three firmware configurations born in which not necessarily all these blocks are carried out:

- **SoftHand Pro firmware** for *SoftHand Pro* specific terminal device control (usually in prosthetic configuration);
- **Generic firmware** to control up to two generic motors and to read encoders, EMG and ADC sensors signals;
- **Dummy firmware** for both prosthetic and non-prosthetic configurations in cases an intermediate board is necessary; it usually acts as bridge connection within boards and devices (only RS485 communication block is enabled and used).

In the next subsections, each block will be described and explained through pseudocode spots. In addition, Doxygen-like documentation has been generated and will be reported in the appendix.

Files organization

Firmware is developed using PSoC Creator® 4.2 IDE software. This environment allows to organize the firmware in two parallel levels of interface: a lower level in which code can be directly written and a higher level in which graphical interface permits to organize code structure in a simpler manner and where all the main components such as timers, counters, registers can be easily configured and connected to Cypress® microcontroller pins.

PSoC Creator® 4.2 IDE use will be better explained in section Firmware development environment - PSoC Creator.

The code is organized in couples of header and source files:

- *globals.h* and *globals.c*, in which are defined macros and global variables definition;
- *commands.h*, in which are defined communication protocol commands;
- *device.h*, generic include file for Cypress PSoC firmwares;
- *interruptions.h* and *interruptions.c*: main firmware functions core. Analog readings, encoder readings and motor control code are located here;
- *command_processing.h* and *command_processing.c*, in which are reported all the definitions of the functions used to process the commands sent from the user interfaces (Simulink®, command line, GUI);
- *IMU_functions.h* and *IMU_functions.c*, in which specific functions to interface and configure IMU sensors are implemented;
- *SD_RTC_functions.h* and *SD_RTC_functions.c*, in which specific functions to interface and configure SD card expansion module and to use real time clock are implemented;
- *utils.h* and *utils.c*, in which are reported signal filters and other utility functions definition;
- other files specific for firmware development environment.

Every block shown in firmware code flow figures, comprehends code from different of the previous listed files. By this, following sub-section will describe in-depth how these blocks are made.

Periodic Code Scheduler

The scheduler is a function that executes periodically at a frequency of about 5 KHz (depending on the configuration). These are the steps sequence in pseudocode C-like format.

```
void function_scheduler() {

    Analog_Readings();

    if (interrupt_flag) {
      RS485_Communication();
    }

    Encoder_Main_Line_Reading();

    if (interrupt_flag) {
      RS485_Communication();
    }

    Motor_Control();

    if (interrupt_flag) {
      RS485_Communication();
    }

    Encoder_Secondary_Line_Reading();

    if (interrupt_flag) {
      RS485_Communication();
    }

    Second_Motor_Control();

    if (interrupt_flag) {
      RS485_Communication();
    }

    Read_IMUs();

    if (interrupt_flag) {
      RS485_Communication();
    }

    Other_minor_checks();

    if (interrupt_flag) {
      RS485_Communication();
    }

    Update_Firmware_State();
}
```

Periodic Function scheduler has, at least, five sub-modules:

- Analog Readings;
- Encoder Readings;
- IMU Readings;

- Motor Control;
- RS485 Communication.

Other_minor_checks() block and *Update_Firmware_State()* describe respectively checks on motor current absorption limitation and update state variables such as sensors measurements, references, SD , RTC and firmware parameters.

Between a module and another, a flag status is tested to verify if a communication request has been received on the RS485 bus.

The scheduler code and all the sub-modules are located in *interruptions.c* file. Further information can be founded in the Appendix.

Analog readings

Voltage sense, current sense, EMG signals and other analog sensors are directly connected to some of the analog pins on the microcontroller as in Figure 10. These values are read by the Delta-Sigma Analog-Digital converter like a potential difference that is proportional to the power board voltage, current, EMG activation and analog sensors output voltage respectively. In this section, the additional analog signals are labelled as other EMG input sensors (e.g. EMG1, EMG2, .., EMG6).

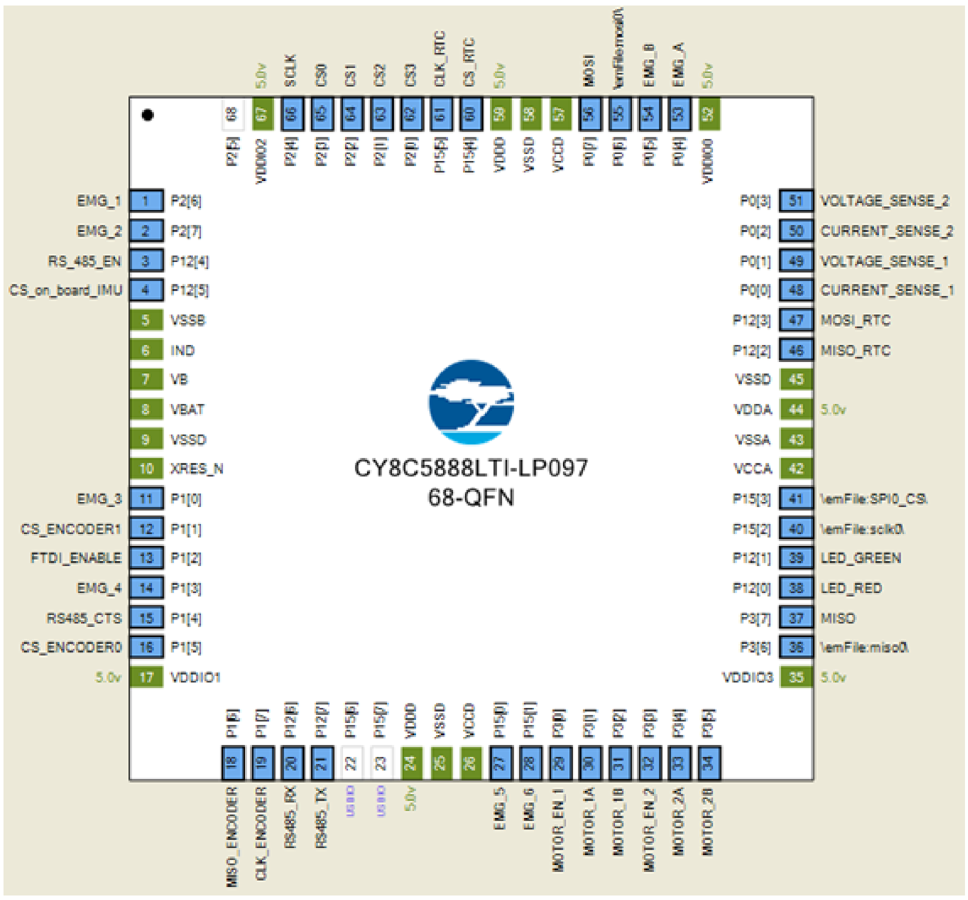


Figure 10: PSoC5 pinout

The Delta-Sigma Analog-Digital converter (hereinafter called ADC for ease of reading) - Figure 11 - reads voltages from 0 to 5 Volts and convert them in a digital value that belongs to the 0-4095 range, in a 12 bits representation. These quantities are processed and converted sequentially thanks to a synchronized counter that changes ADC input periodically. Converted values are then stored in memory and can be accessed by the firmware code through Direct Memory Access (DMA).

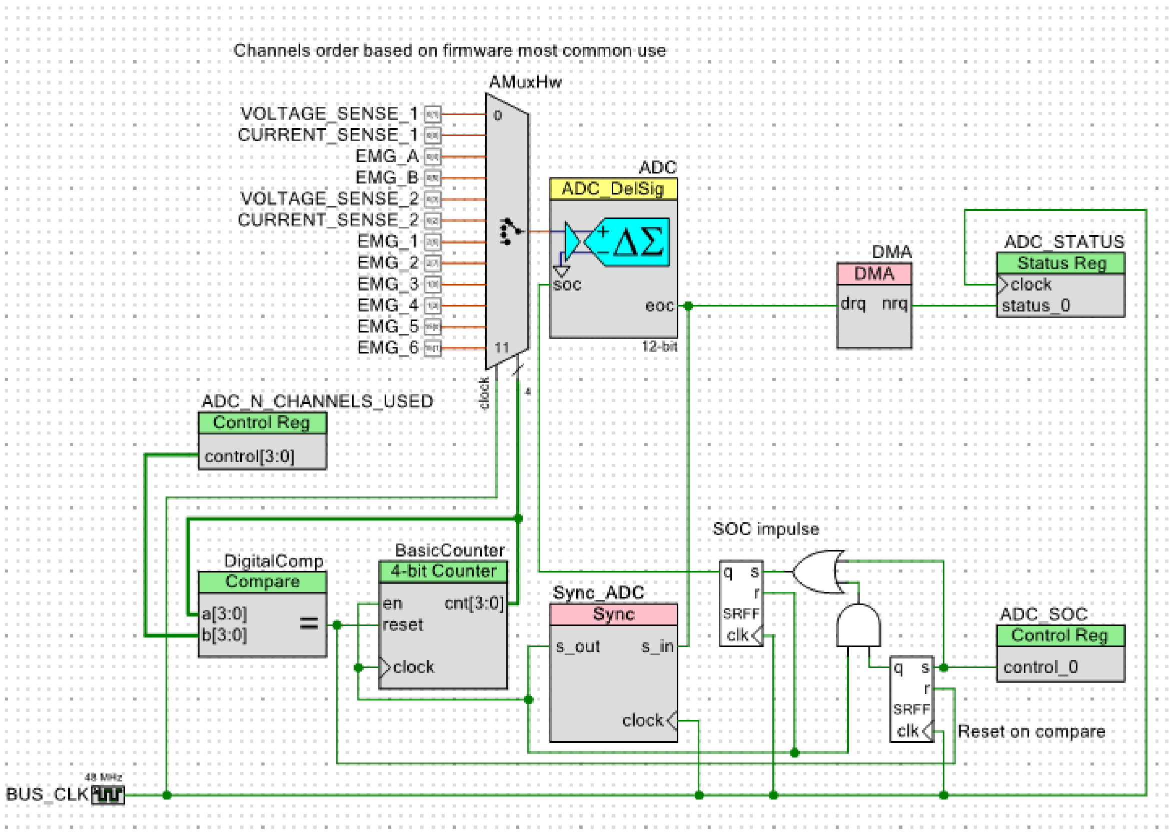


Figure 11: ADC configuration and connection scheme

Depending on the needed configuration, not all the analog converted quantities are always used, e.g. no one is useful for Dummy firmware, while *voltage_sense_2* and *current_sense_2* reading is useless in *SoftHand Pro* device firmware.

After the in-depth description of the previous cited signals:

- voltage signal;
- current signal;
- EMG signals.

Voltage reading

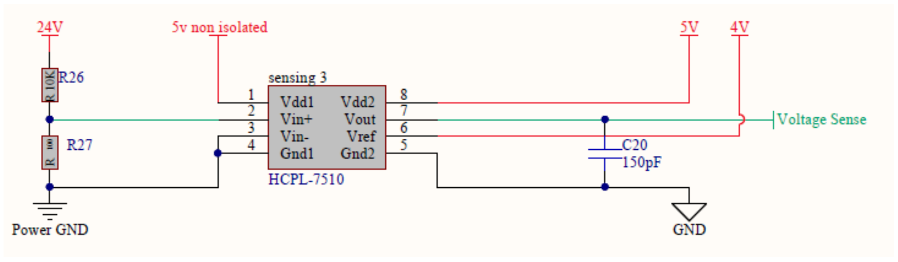


Figure 12: Voltage sense schematic

From the schematic reported in Figure 12, the ideal formula to obtain voltage measurements should be:

$$tension\ [mV] = \left((read\_value\_mV\ \ - offset) * 101\right)/gain$$

where

- $read\_value\_mV$ is equal to $counts\_read/0.819$ for operating conversion from counts to mV;
- $offset$ value is 2000, due to hardware amplifier bias in mV;
- the amplifier gain is equal to 8.086.

The real formulation is instead a trade-off in good performance and accuracy, since it has been taken into account measures are expressed in counts, so in the firmware the reader will find:

```
dev_tension = ((int32)(ADC_measure - 1638) * 1952) >> 7;
```

Current reading

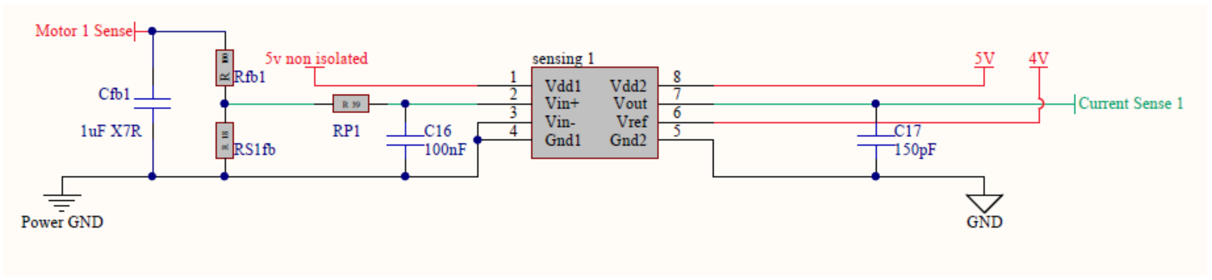


Figure 13: Current sense schematic

From the schematic reported in Figure 13, the ideal formula to obtain current measurements should be:

$$current\ [mA] = \left((read\_value\_mV\ \ - offset) * 375\right)/(gain * resistor)$$

where

- $read\_value\_mV$ is equal to $counts\_read/0.819$ for operating conversion from counts to mV;
- $offset$ value is 2000, due to hardware amplifier bias in mV;
- the amplifier gain is equal to 8.086;
- resistor voltage divider is equal to 18 KOhm.

The real formulation is instead a trade-off in good performance and accuracy, since it has been considered measures are expressed in counts, so in the firmware the reader will find:

```
curr = ((int32)(ADC_measure - 1638) * 25771) >> 13;
```

EMG reading

EMG reading sub-module has been implemented in firmware because EMG sensors are used in myoelectric prosthetic configurations. EMG analog quantities are directly sampled by the ADC. These values are first divided by a maximum value threshold before been used for terminal devices control. Thresholds are different from a User to another and then are tuned during a calibration procedure that occurs in case a terminal device control with EMG as inputs is needed. Details on this procedure are shown in Software section, while Figure 14 shows the electronics used for the acquisition.

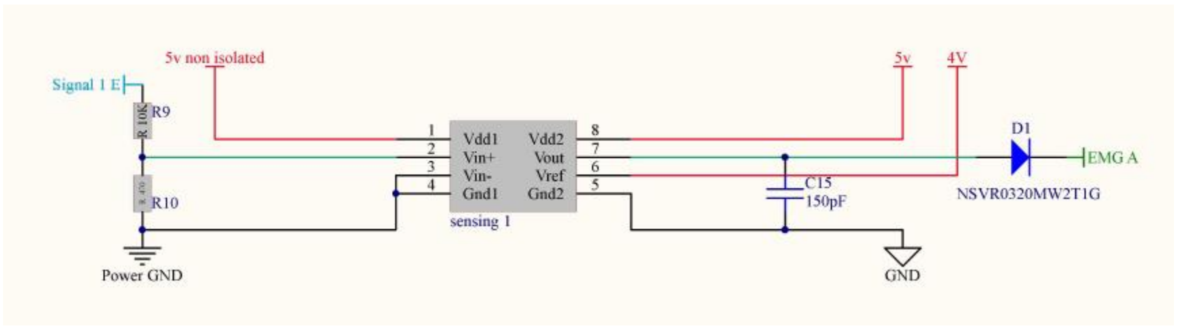


Figure 14: EMG sensing schematic

Encoder readings

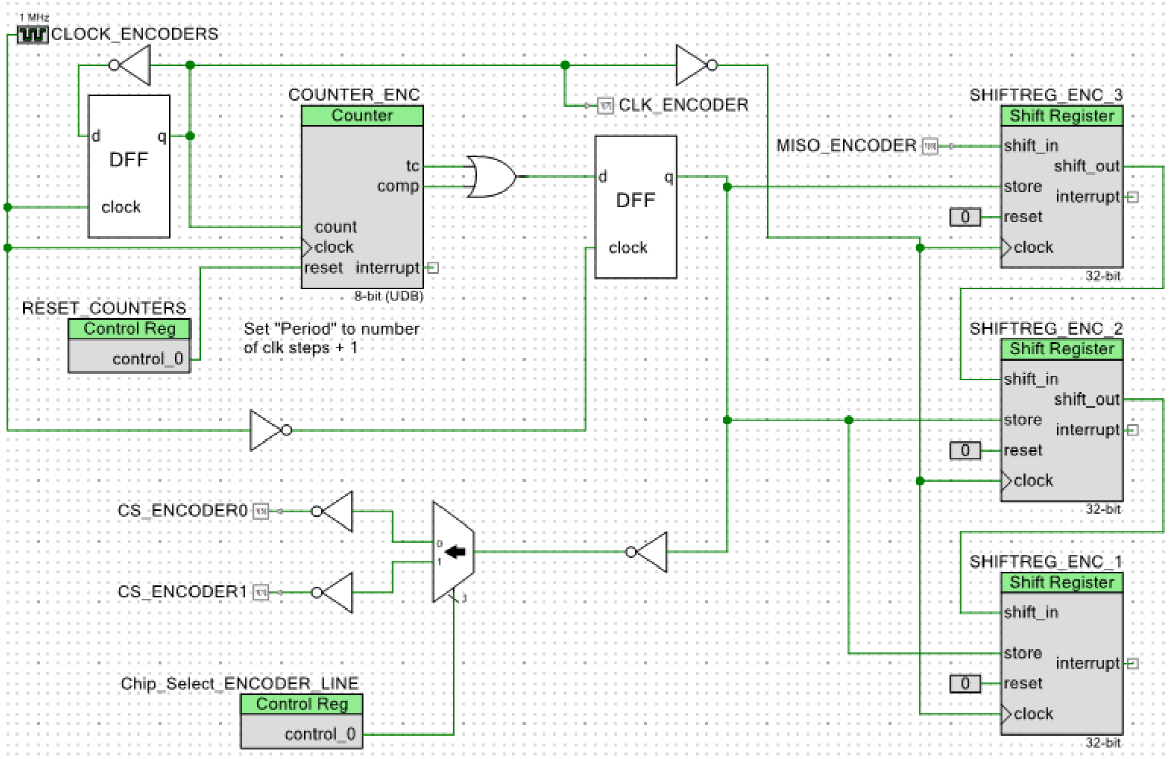


Figure 15: Encoder reading configuration and connection scheme

Figure 15 shows firmware structure needed for encoder reading procedure. Up to five encoders can be connected in daisy chain to each encoder line of the microcontroller.

The data is read using SSI protocol through a shift register that registers bits sequentially.

The encoder is an AS5045 12-bit contactless magnetic position sensor for accurate angular measurement over a full turn of 360°. It provides instant indication of the magnet's angular position with a resolution of 0.0879° = 4096 position per revolution. This signal data is decoded in 12 bits resolution with 6 additional control bits for a whole package of 18 bits.

Since firmware main requirement is to control motor position in a reliable and efficient manner, a valid control loop based on measures acquired through a couple of encoders has been implemented. E.g. for the *SoftHand Pro* device, the encoders are fixed on the bottom side of the terminal device, close to wrist interface (see Figure 6). Respectively, first encoder is used to track motor pulley angular position and the second to count motor pulley turns. In this way, it is possible to reconstruct a reliable angular position to be used for control purposes. This is done every time the board is powered on or resets, before enabling the terminal device control.

In the following, a pseudocode performing encoder reading is reported.

```
// Shift 1 right to erase Dummy bit of chain
data_encoder = ReadEncoderLine(i); //read all bytes coming from the line

if (good_reading) {
    value_encoder = (WRAP(data_encoder - ENCODER_OFFSET) << 4);
// reset last 6 control bits and wrap encoder data in the range [-2048,
2047], then shift to have 16 bits value

    // Take care of rotations
    aux = value_encoder - LAST_VALUE_ENCODER;

    // if we are in the right interval, take care of rotation
    // and update the variable only if the difference between
    // one measure and another is less than 1/4 of turn

    // Considering we are sampling at 1kHz, this means that our shaft needs
    // to go slower than 1/4 turn every ms -> 1 turn every 4ms
    // equal to 250 turn/s -> 15000 RPM

    if (aux > 49152)
        rot--;
    else {
        if (aux < -49152)
            rot++;
    }

    LAST_VALUE_ENCODER = value_encoder;

    value_encoder += rot << 16;

    if (multiplier != 1.0) {
        value_encoder *= multiplier;
    }

    measured_position = value_encoder;
}
```

In addition to sensor reading, also count of rotations and measurement multiplier must be considered before having a consistent motor pulley angular position data. Figure 16 reports how the sensor readings are mapped in 1 encoder turn.

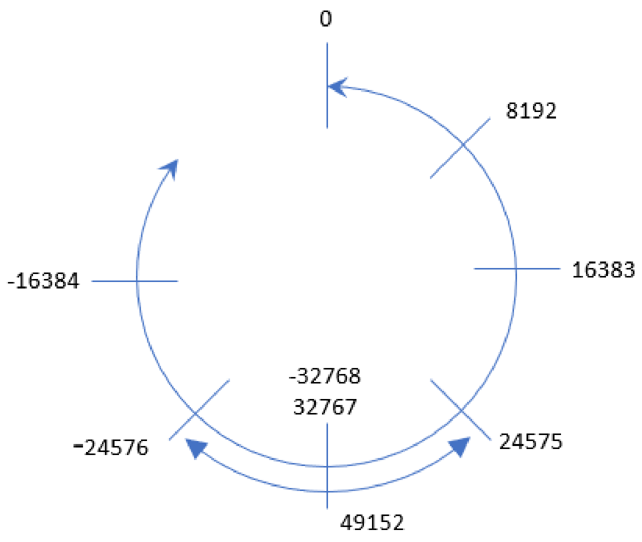


Figure 16: Encoder turn – sensor reading map

IMU readings

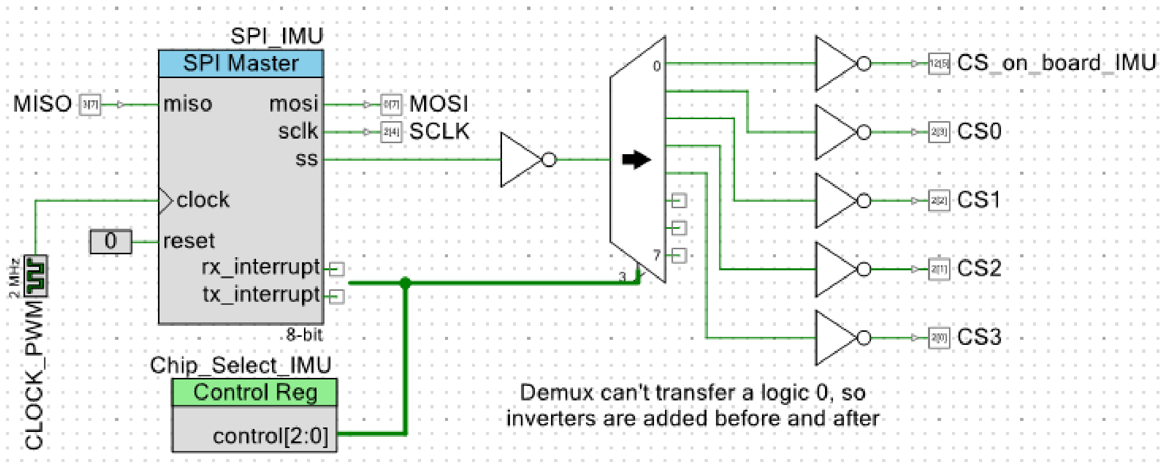


Figure 17: IMU reading configuration and connection scheme

Figure 17 shows firmware structure needed for IMU reading procedure. Up to five IMUs can be connected to this microcontroller. Each IMU module has a separate cable select pin (CS**). Through this pin an IMU module is selected and can send sensors information to the microcontroller via SPI communication protocol.
When the Cypress® PSoC microcontroller is powered, there is a firmware initialization phase in which a check is performed to make the firmware know how many IMU modules are connected and to configure them by setting sensors filters, scale factors and performing also sensor calibration. All these settings follow IMU MPU9250 datasheet [LINK].
Then, the periodic scheduler performs IMU readings procedure for each connected module according to its configuration as the following steps sequence in pseudocode C-like format shows.

```
void ReadAllIMUs(){

   for (k_imu = 0; k_imu < N_IMU_Connected; k_imu++){
      // Read k_imu connected IMU module
      ChipSelector(IMU_connected[k_imu]);
```

```
        if (read_Acc[n]) ReadAccelerometers(n);
        if (read_Gyro[n]) ReadGyroscopes(n);
        if (read_Mag[n]) ReadMagnetometers(n);
        if (read_Quat[n]) ReadQuaternion(n);
        if (read_Temp[n]) ReadTemperature(n);
    }
}
```

Motor control

DC motors used in our systems are controlled by a driver embedded in one of our power boardd. This driver is driven by a couple of digital pins that define motor enable and movement direction as shown in Figure 18. General firmware can control up to two motors, even if some devices have only one motor. *MOTOR_ A* and *MOTOR_ B* in Figure 18 come from the output of a logic AND between a Pulse Width Modulation (PWM) output and a direction register. In this way, once fixed a turn direction, the PWM passes in only one pin between *MOTOR_ A* and *MOTOR_ B* and it is driven high while the other is low and does not activate the driver.

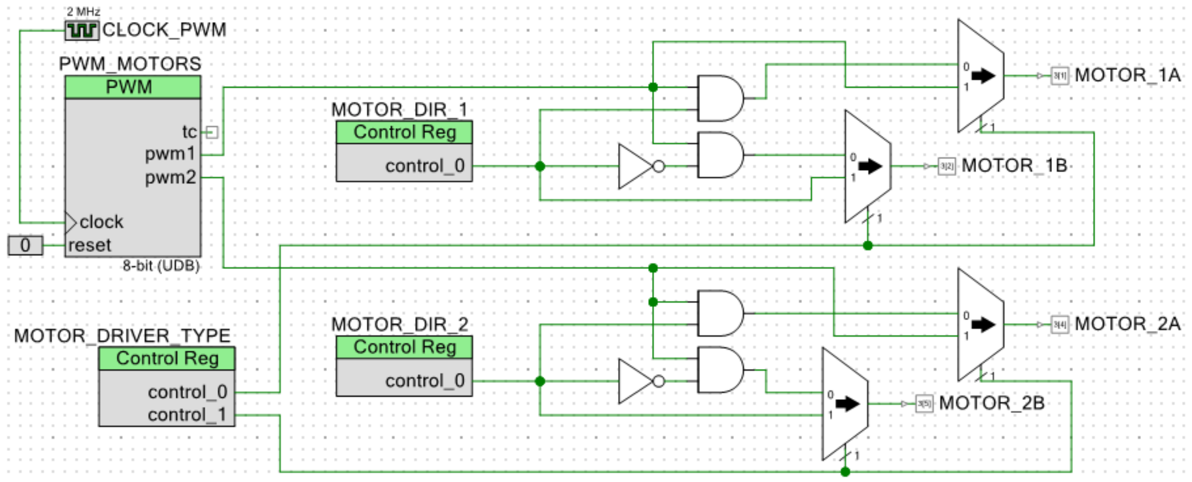


Figure 18: Motor control configuration and connection scheme

The firmware implements different types of controls, such as

- position control;
- direct PWM control;
- current control;
- current and position control

by using the inputs methods listed below:

- USB (references through external commands);
- HANDLE (a specified encoder drives the terminal device input);
- EMG proportional (use EMG measures to proportionally drive the position of the terminal device motor);
- EMG integral (use both EMG signals to drive terminal device motor position);
- EMG FCFS (use both EMG. The first reaching threshold wins, and its value defines terminal device position reference);

- EMG FCFS advanced (use both EMG. The first reaching threshold wins, and its value defines terminal device position reference, but waits for both EMG to lower under activation threshold).

In the following, an example of how a PWM input is generated is reported. The code is different from a control mode to another but the main scheme that is underneath can be summarized as an error computation between a reference and actual measure (position, current, etc..), a weighted contribution of this error of proportional, integral and derivative actions to map the error in a PWM input. Thus, motor direction must be computed, together with respect to limitations and PWM boundaries.

```
pos_error = reference_position - measured_position;

pos_error_sum += pos_error;

// anti-windup (for integral control)
if (pos_error_sum > ANTI_WINDUP)
    pos_error_sum = ANTI_WINDUP;
else {
    if (pos_error_sum < -ANTI_WINDUP)
        pos_error_sum = -ANTI_WINDUP;
}

// Proportional
if (k_p!= 0)
    pwm_input = k_p * pos_error;

// Integral
if (k_i!= 0)
    pwm_input += k_i * pos_error_sum;

// Derivative
if (k_d!= 0)
    pwm_input += k_d * (pos_error - prev_pos_err);

// Update measure
prev_pos_err = pos_error;

if (pwm_input > 0)
    motor_dir = 1;
else
    motor_dir = 0;

if (pwm_input > PWM_MAX_VALUE)
      pwm_input = PWM_MAX_VALUE;
if (pwm_input < -PWM_MAX_VALUE)
      pwm_input = -PWM_MAX_VALUE;

if (control_mode!= CONTROL_PWM)
   pwm_input = (((pwm_input << 10) / PWM_MAX_VALUE) * dev_pwm_limit) >> 10;

   pwm_sign = SIGN(pwm_input);

if (motor_dir)
   MOTOR_DIR = 1;
else
   MOTOR_DIR = 0;

PWM_MOTORS_WriteCompare(abs(pwm_input));
```

The board interfaces with both the default SoftHand driver (MC33887) power board and the high power VNH5019 driver power board and supports both back drivable and not-back drivable motors.

RS485 communication

As seen in the function scheduler, between every two tasks an *RS485_Communication()* procedure is called if necessary (detailed code in *interruptions.c* file). This equals a communication request, that is managed like the following:

```
// Processing RS-485 data frame:
//
// - WAIT_START:    Waits for beginning characters;
// - WAIT_ID:       Waits for ID;
// - WAIT_LENGTH:   Data length;
// - RECEIVE:       Receive all bytes;
// - UNLOAD:        Wait for another device end of transmission;
//

rx_data = UART_RS485_GetChar();

switch (state) {
    //-----     wait for frame start     -------------------------------
    case WAIT_START:

        // Check for header configuration package
        if (check_for_headers() == "::") {
            state       = WAIT_ID;
        }
        break;

    //-----     wait for id     ----------------------------------------
    case  WAIT_ID:

        // packet is for my ID or is broadcast
        if (rx_data == BOARD_ID)
            rx_data_type = FALSE;
        else                //packet is for others
            rx_data_type = TRUE;

        data_packet_length = 0;
        state = WAIT_LENGTH;
        break;

    //-----     wait for length     ------------------------------------
    case  WAIT_LENGTH:

        data_packet_length = rx_data;
        // check validity of pack length
        if (data_packet_length <= 1) {
            data_packet_length = 0;
            state = WAIT_START;
        } else if (data_packet_length > 128) {
            data_packet_length = 0;
            state = WAIT_START;
        } else {
            data_packet_index = 0;

            if(rx_data_type == FALSE)
```

```
                state = RECEIVE;          // packet for me or broadcast
            else
                state = UNLOAD;           // packet for others
        }
        break;

    //-----     receiving     ------------------------------------------
    case RECEIVE:

        data_packet_buffer[data_packet_index] = rx_data;
        data_packet_index++;

        // check end of transmission
        if (data_packet_index >= data_packet_length) {
            // verify if frame ID corresponded to the device ID
            if (rx_data_type == FALSE) {

                // copying data from buffer to global packet
                copy_data_from_buffer_to_packet();
                g_rx.length = data_packet_length;
                g_rx.ready  = 1;
                process_command();
            }

            data_packet_index  = 0;
            data_packet_length = 0;
            state              = WAIT_START;
        }
        break;

    //-----     other device is receiving     ---------------------------
    case UNLOAD:
        if (!(--data_packet_length)) {
            data_packet_index  = 0;
            data_packet_length = 0;
            RS485_CTS_Write(1);
            RS485_CTS_Write(0);
            state = WAIT_START;
        }
        break;
}
```

As shown in the previous reported pseudo-code, *RS485_Communication()* procedure, manages with RS485 packets. The receiving procedure is organized as a machine state and the passage from a state to another is accomplished only if the received packet is formatted as expected. It is important to notice that communication is broadcast and the packet is received or rejected by the firmware according to its ID, that is an embedded firmware parameter.

Dummy mode

Dummy mode is a specific firmware configuration. When in dummy mode, the firmware works as a remote communication bridge and no encoders or motors are directly connected to the board. The ID firmware parameter must be set equal to 250 in order to activate this functionality. Details on how this setting must be done, refer to Command-line tools section.

Commands

Commands mode can be used in not Dummy firmware configuration. Once a packet has been received, several commands can be performed. Some of the allowed commands are reported below. By the way, an exhaustive list of commands can be found in Appendix A section.

```
[…]

CMD_SET_ZEROS     = 1 //command for setting the encoders zero position.
CMD_GET_INFO      = 6 //ask for a string of information about.
CMD_GET_PARAM_LIST = 12 //command to get the parameters list or to set a
defined value chosen by the use.

CMD_HAND_CALIBRATE = 13 //starts a series of opening and closures of the
SoftHand Pro terminal device.

CMD_ACTIVATE      = 128 //command for activating/deactivating the device.
CMD_SET_INPUTS    = 130 //command for setting reference inputs.
CMD_GET_MEASUREMENTS = 132 //command for asking position measurements.
CMD_GET_CURRENTS  = 133 //command for asking current measurements.
CMD_GET_EMG       = 136 //command for asking EMG sensors measurements.

[…]
```

For all the three modes of communication, also firmware parameters can be read or set by communicating through RS485. These data are packed and sent to the receiver firmware or device in a single request. For detailed information on how to get or set parameters, please refer to

Software tools section.

Firmware configurations

As mentioned during previously description, three configurations are defined according to the needing:

- *SoftHand Pro* terminal device configuration;
- Generic use configuration;
- Dummy configuration.

SoftHand Pro terminal device configuration

This configuration implements all of the previously explained scheduler modules, since it is specific for *SoftHand Pro* terminal devices control with multiple input options, i.e. EMG sensors for prosthetic configurations or handle for non-prosthetic ones.

Generic use configuration

This firmware configuration implements up to two motors control, up to two lines encoder readings, 5 IMUs, 2 EMG and 8 ADC sensors, update of usage statistics and save them to SD card module.

Dummy configuration

Dummy configuration firmware scheduler implements only RS485 communication module. No encoders or motors are directly connected to the board, that works only as a bridge. This mode is usually used to remotely communicate with other boards of the system.

Software

Software destination of use is intended for monitoring safety measurements, allowing Users to communicate with the logic board, to manage and configure external sensors information and firmware parameters.

Introduction of software lifecycle

Software main requirement is to exchange information between firmware and the users, in a reliable and efficient manner. For this purpose, a series of tools has been developed for command line, GUI and third-party software uses.

In next subsections software environment will be described in two main subparts:

- Firmware development environment - PSoC Creator section: software environment used for writing firmware and programming Cypress® microcontrollers;

Application Programming Interface and

- Software tools section: the aiding function and API developed for communicating with firmware and the related software tools.

Also, Simulink® libraries will be described as an example of a third-party software interface (Simulink® library subsection).

Firmware development environment - PSoC Creator

For Windows® OS only

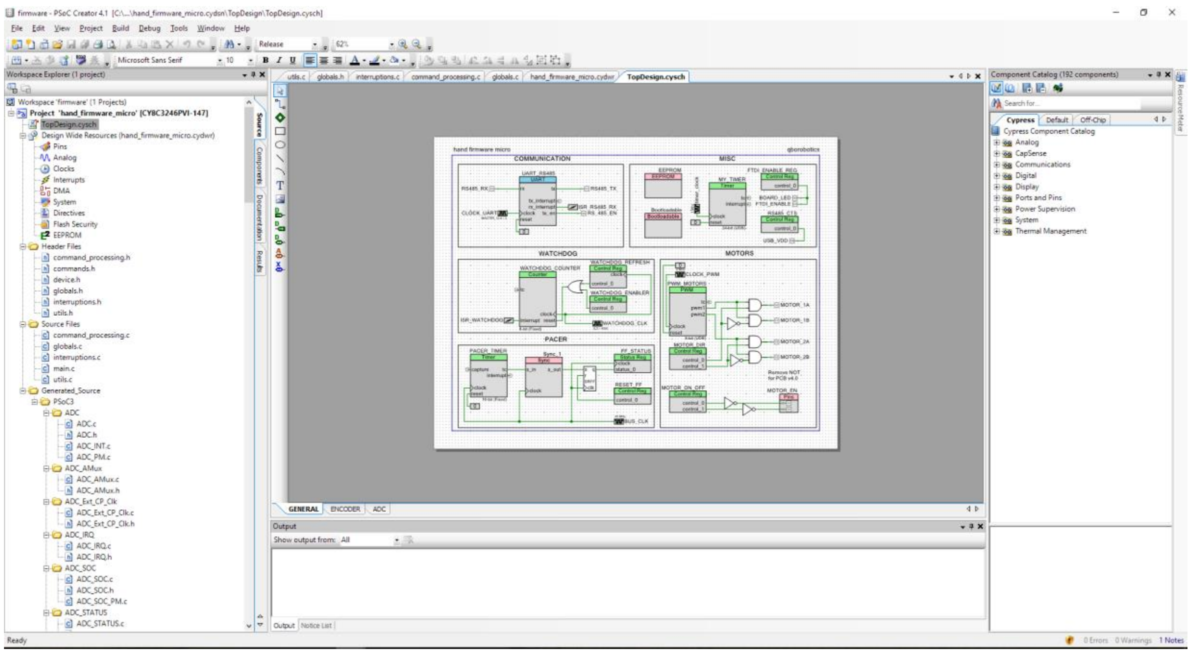

firmware - PSoC Creator 4.1 [C:\...\hand_firmware_micro.cydsn\TopDesign\TopDesign.cysch]
Workspace Explorer (1 project)
Project 'hand_firmware_micro' [CY8C3246PVI-147]
TopDesign.cysch
Release
Component Catalog (192 components)
COMMUNICATION
MISC
WATCHDOG
MOTORS
PACER
Output

Figure 19: PSoC Creator example of development environment

In order to change or update whatever firmware, Technician needs the PSoC Creator® software (available from Cypress® website) in addition of naturally the firmware (available in the following repository, https://github.com/NMMI/SoftHandPRO-and-Generic-FW-PSoC5).

This guide is tested on PSoC Creator 3.3® SP1, PSoC Creator 4.1® and PSoC Creator 4.2®.

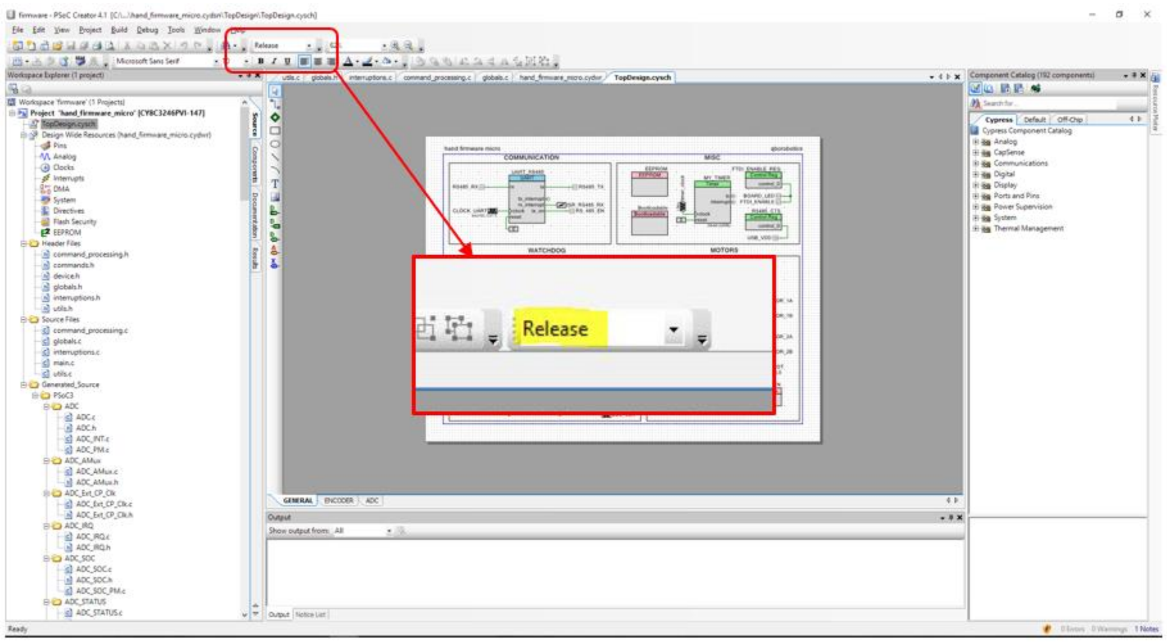

firmware - PSoC Creator 4.1 [C:\...\hand_firmware_micro.cydsn\TopDesign\TopDesign.cysch]
Release

Figure 20: detail on Build configuration combo box

First, after the download of software and firmware files, you should open firmware project on PSoC Creator® and compile the entire workspace it by clicking from the **Build** menu: **Build 'your project name'**. Pay attention to choose **Release** as active build configuration (Figure 20). The firmware should be compiled with no errors. In the workspace there are two firmwares: the one of SoftHand Pro device and that for Generic use.

There are two ways to load the firmware into the logic board microcontroller: by using the dedicated hardware programmer or through bootloader mode.

Load by using programmer

The dedicated hardware programmer is called MiniProg3 and should be connected to the micro-USB board connector located in the external adapter support. In fact, in this release of the board it is possible to program the board only by using the external JST 8 pin – micro USB external adapter and connecting the MiniProg3 to that USB connector.

1. Note that, first of all, the programming switch pins (refer to Figure 21) should be both on **ON** position;
2. PSoC Creator® software should be configured as settings shown in Figure 22;
3. select **Program** from the **Debug** menu to access the programming window;
4. click on **Port acquire** and then **OK**;
5. after programming, switch the programmer pins again to **OFF** position.

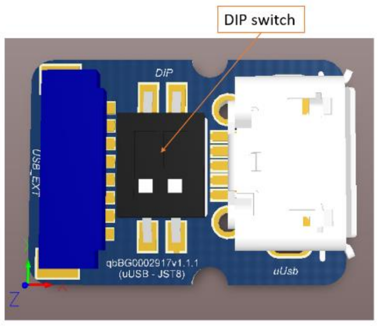
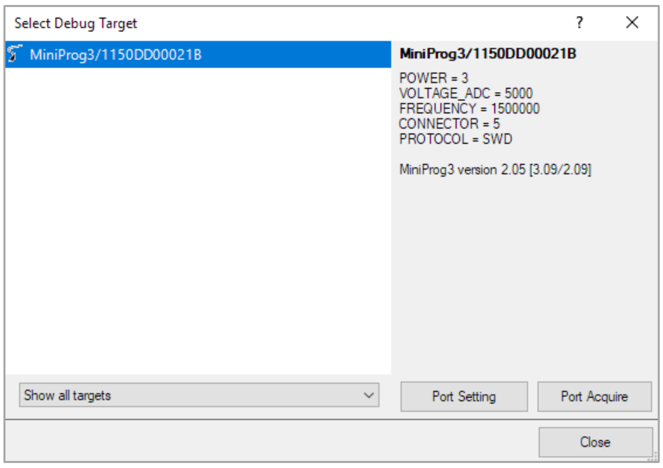


Figure 21: logic board: connections useful for programming


Select Debug Target

MiniProg3/1150DD00021B

MiniProg3/1150DD00021B
POWER = 3
VOLTAGE_ADC = 5000
FREQUENCY = 1500000
CONNECTOR = 5
PROTOCOL = SWD

MiniProg3 version 2.05 [3.09/2.09]

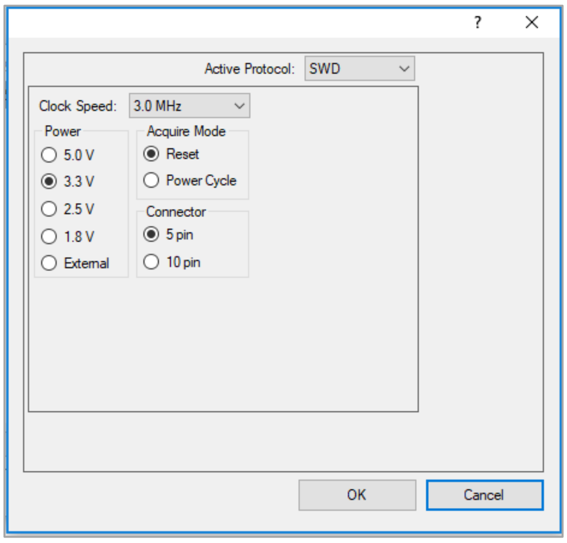

Active Protocol: SWD
Clock Speed: 3.0 MHz
Power: ( ) 5.0 V [✓ 3.3 V] ( ) 2.5 V ( ) 1.8 V ( ) External
Acquire Mode: [✓ Reset] ( ) Power Cycle
Connector: [✓ 5 pin] ( ) 10 pin

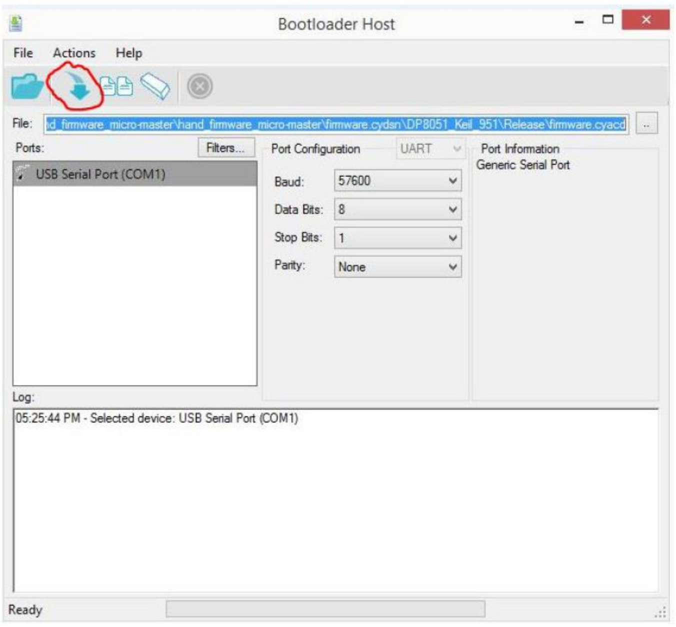
Figure 22: MiniProg3 configuration for PSoC5 microcontroller

Load by using bootloader

Second way to load the firmware into the microcontroller is by switching it in bootloader mode. To do this, you have two possibilities: by clicking the related flag in the GUI or by sending a specific command through API (see section Software tools). Then:

1. open **Bootloader Host** from **Tools** menu;
2. search ***.cyacd** file that should be uploaded on the board (it is usually in the folder *\CortexM3\ARM_GCC_541\Release*) and set the serial port parameters as the baud rate, the number of bytes, etc.. as shown in Figure 23;
3. click the **Upload** button to program the microcontroller.

Sometimes, Windows® operating system gives some communication problems when using the **Bootloader Host**. To avoid these problems, it is advisable to associate the board to the first COM ports, e.g. COM1, COM2 or COM3. Details on which this problem can be fixed, please refer to Software tools section.



File: id_firmware_micro-master\hand_firmware_micro-master\firmware.cydsn\DP8051_Keil_951\Release\firmware.cyacd

Ports: USB Serial Port (COM1)

Port Configuration: UART

Baud: 57600

Data Bits: 8

Stop Bits: 1

Parity: None

Port Information: Generic Serial Port

Log:

```
05:25:44 PM - Selected device: USB Serial Port (COM1)
```

Figure 23: Bootloader Host

Application Programming Interface

For Windows® OS, MacOS® and Unix

Application Programming Interface (API) functions have been developed in order to communicate between a computer (represented by Software tool) and the logic board firmware. APIs represent thus a low-level interface for the developed software tools.

In the following, chosen RS485 serial protocol is explained together with a generic implementation of such functions and a relative example of use. Further information can be read in the dedicated section (see Appendix B – Software detailed documentation).

RS485 serial protocol - firmware expects, from a generic master (either firmware or software), a package of at least 6 bytes. Those packages must be built as chars or integers. The headers of the packages are defined as two ':' symbols (also ASCII integer equivalent '58' can be used). The packages end with 1 checksum bit (hereinafter CHK).

Typical package structure

HEAD	HEAD	ID	LENGHT	COMMAND	DATA[1-N]	CHK

Following this structure, APIs open and close communication; get or set firmware parameters; send commands or retrieve information from the board. Examples of packages to send, to execute the relative functions or read measurements, are the following (under the hypothesis that the hand has ID equal to 10). These examples are referred to following actions:

- activate the motors on the device;
- set inputs to close or open the *SoftHand Pro* terminal device;
- encoder measurements;
- current measurements.

Activate Hand

:	:	10	3	128	1	CHK

If in the sixth field instead of “1” is sent “0”, the device deactivates its driver and turns off. Its good practice to deactivate all the devices once software work ends.

Set inputs to close or open SoftHand Pro system terminal device

:	:	10	6	130	IN0[1]	IN0[0]	IN1[1]	IN1[0]	CHK

The set inputs function needs 2 inputs in order to command two different motors, the first one using IN0, the second ont using IN1. E.g. for *SoftHand Pro* device the first input must be a value between 0 and 16000 (split in 2 bytes).
Checksum bit is calculated as an EXCLUSIVE OR of all the bytes that compose the package, starting after LENGHT field. In this example, the calculation would be:

CHK = 130 ^ IN0[1] ^ IN0[0] ^ IN1[1] ^ IN1[0]

Encoder measurements

:	:	10	2	132	132

Expected reply (12 bytes):

:	:	10	8	132	M0[1]	M0[0]	M1[1]	M1[0]	M2[1]	M2[0]	CHK

The only significant encoder values are the first two, built using the M0 LSB(M0[0]) and MSB(M0[1]) and M1 LSB(M1[0]) and MSB(M1[1]) respectively.

Current measurements

:	:	10	2	133	133

Expected reply (12 bytes):

:	:	10	6	133	C0[1]	C0[0]	C1[1]	C1[0]	CHK

The only significative encoder value is the first one, built using the C0 LSB(C0[0]) and MSB(C0[1]).

Every of these commands is linked to a specific function written in C++ language (see *qbmove_communications.cpp* file in Appendix B – Software detailed documentation section). As an example, the function used to activate terminal device motor is reported as pseudocode.

```
void commActivate(comm_settings *comm_settings_t, int id, char activate) {

    char data_out[BUFFER_SIZE];      // output data buffer

    data_out[0] = ':';
    data_out[1] = ':';
    data_out[2] = (unsigned char) id;
    data_out[3] = 1;
    data_out[4] = CMD_ACTIVATE;                  // command
    data_out[5] = activate ? 1 : 0;
    data_out[6] = checksum(data_out + 4, 2);     // checksum

    write(comm_settings_t->file_handle, data_out, 7);
}
```

Software tools

For Windows® OS, MacOS® and Unix

A Graphical User Interface (hereinafter GUI) software and a set of command-line tools have been developed to permit to the User to easily manage and set firmware parameters. In following sub-section an in-depth description of these instruments is reported.

Graphical User Interface

This a useful software tool to interface with NMMI devices.

The GUI is multi-platform and very intuitive, it allows to read data from the electronic boards located in NMMI devices and to store and get firmware parameters.

Software files can be downloaded at https://www.naturalmachinemotioninitiative.com/nmmi-gui.

Installation procedure

MacOS®

1. Download Software file and if not already installed, it is necessary to install the FTDI serial drivers for seeing the device (here intended as SoftHand board) as a serial port;
2. Open the App by double clicking add the developer as *trusted* (check this page);
3. [optional for having the application on User/Technician dock] **.dmg** package must be opened and moved to the dock or the **Application** folder.

Windows®

1. Download Software file and if not already installed, it is necessary to install the FTDI serial drivers for seeing the device (here intended as logic board) as a serial port;
2. if the serial port is not seen on the GUI main window, it is possible that the system has recognized COM port with a high COM number. The supported serial ports are from COM1 to COM29. To change port number:
    a. **Control Panel→Hardware and Sound→Device Manager** must be opened;
    b. select COM & LPT ports, drop down menu, right click on the COM port and select **Properties**;
    c. select **Port settings** tab and then click on **Advanced**;
    d. in **Advanced** menu, select from the drop-down menu a COM number between COM1 and COM29 and click on **OK**. The device should be recognized properly.

Unix

1. Download software file and give execution permissions to it. From a terminal, move to the folder with the App and run the command

```
chmod a+x NMMI\ GUI\ Linux.AppImage
```

2. Before using the GUI, it is mandatory to add the computer user to the **dialout** group. In order to do so, the User/Technician must put the following command on a terminal:

```
sudo adduser user_name dialout
```

where `user_name` is the username under which the GUI is used. Once this command is executed, it is necessary to log out and back in for the changes to take effect.

Main layout

Remember to power the board (or a chain of these) before using it

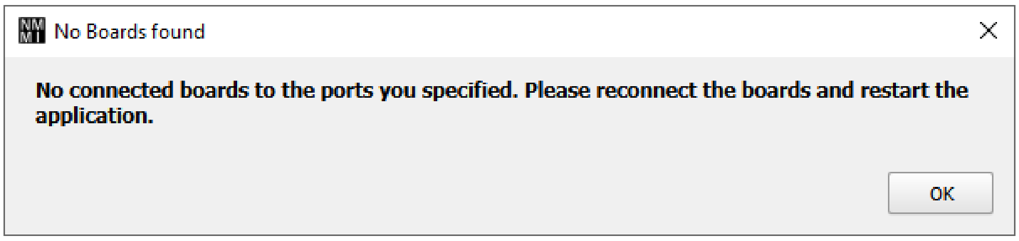
GUI application is structured into five tabs for improve User's safety and for better manage firmware functionalities. Some of these are accessible to the User and all of these are naturally accessible to the Technician:

- System Check: gives a snapshot of the current board configuration
- Check Movement: if the board is properly configured, here it is possible to activate the motors and move the device
- Live Data: useful tab to monitor motors positions and current and to record and save these data
- EMG Setup: tba dedicated to EMG input; here it is possible to configure and have a look on EMG signals and activations
- Parameters: it is divided in sub-tabs to allow the User/Technician to get/set all the boards parameters and to modify the current configuration

Notice that by default, only the first three tabs (*System Check, Check Movement, Live Data*) are enabled, to unlock the other tabs (*EMG Setup, Parameters*) please contact directly Manufacturer's staff.

To use the device with this application, the User/Technician must first connect the board to the computer and then double-click and open the GUI. If the board is properly connected, the serial port can be read in the bottom of the App window on the left and the device is connected.

If some problems occur while connecting to the board (the cable disconnects, the communication bus is busy and so on) an error popup window should appear like that in Figure 24.


No Boards found

No connected boards to the ports you specified. Please reconnect the boards and restart the application.

Figure 24: GUI connection error

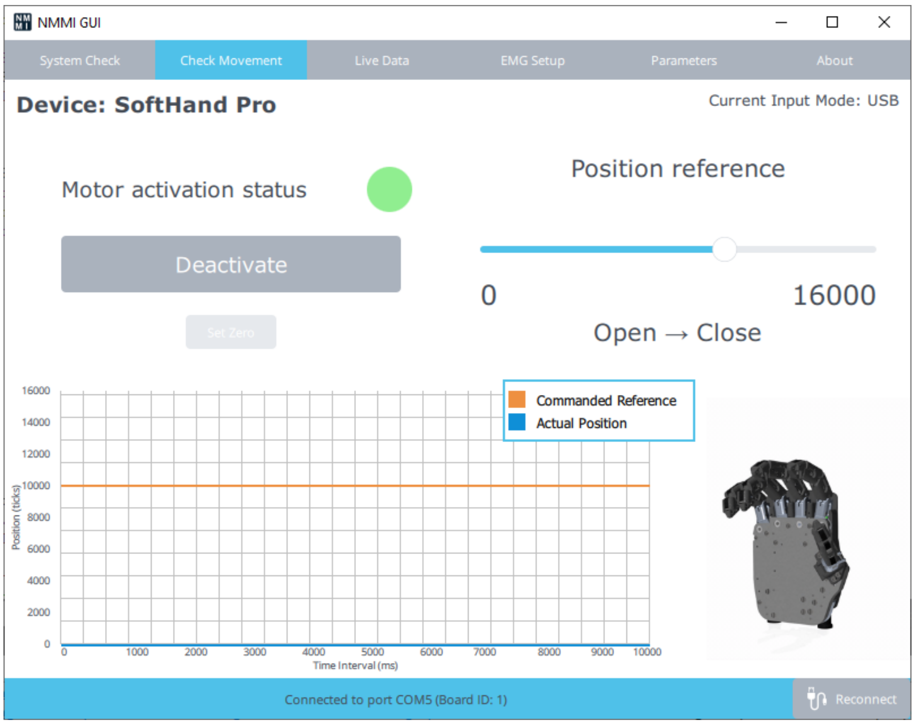


Figure 25: GUI main view

Command-line tools

Command line tools can be downloaded at https://github.com/NMMI/qbAPI/tree/centropiaggio and https://github.com/NMMI/qbadmin/tree/centropiaggio and can be used through operating systems command window.

There are two main levels of command-line tools: qbAPI functions represent the lower level and are the properly named APIs, while qbadmin, qbparam and nmmi_param tools represent the functions higher level. qbAPI functions open, close serial port communication, send and receive commands like the format decoded as in The board interfaces with both the default SoftHand driver (MC33887) power board and the high power VNH5019 driver power board and supports both back drivable and not-back drivable motors.

RS485 communication section. Each of these functions can be called at higher level using qbadmin, qbparam and nmmi_param software tools.

Other tools that can be found in the downloaded package are only demonstration ones and will be explained only in Appendix B – Software detailed documentation.

Notice that all the APIs are written in C++ language and are in qbmove_communications.cpp file, while all the commands are in commands.h file.

How to install qbadmin, qbparam, nmmi_param and nmmi_param_imu

1. At the following links, User can find all the command line tools useful to handle and communicate with the board: https://github.com/NMMI/qbadmin and https://github.com/NMMI/qbAPI . You need to download and install both, following the readme provided in the github pages.
2. To compile the source files, from a terminal move to qbAPI/src folder and type make. Then, to compile the tools, move to qbadmin/src folder and type make.
3. If everything is ok, you should see a folder tree like this:

  - qbadmin
  - bin_unix
  - conf_files
  - objs_unix
  - src

4. Whenever, in the readme you can find software installation instructions under Installation requirements and Compile the libraries section.
5. Once compiled previously specified files, the *qbadmin/bin* folder should contain multiple executable files, the most important of which are qbadmin, qbparam, nmmi_param and nmmi_param_imu.
6. Connect the board to the PC through the micro USB cable. Then, execute *qbadmin -t* to choose the serial COM port and e.g. *qbadmin -p* to see information of the connected device and to check if all is OK.

qbadmin tool

qbadmin provides several commands to interface with the firmware. In the following most of possible program executions are listed:

- *qbadmin -t*: it is used to list the connected board or multiples of these, connected to the computer and to establish a communication with them. When launched, the program lists connected boards and allow to select which to connect to, by pressing the number associated to this one.

e.g. qbadmin -t, then digit 1 when prompted

1. *qbadmin -p*: it returns a long status which contains all information about the board connected, as: ID, current position, actual motor current, PID parameters.
2. *qbadmin -a*: it activates the motor control loop.
3. *qbadmin -d*: it deactivates the motor control loop.
4. *qbadmin -g*: it starts a loop which continuously returns and outputs on screen read values of the encoder on the board. It returns three different values, even if the number of mounted encoder is minor. The terminal devices have two internal encoders, so the first two values will be the measurements of these ones while the third value is associated to the measurement of the handle encoder when the configuration expects it.
5. *qbadmin -s*: it sets the desired position for the encoder on the terminal device. It takes two values but the second one is ignored. Before giving a value to the terminal device, always control closure value or rather the maximum value which the device can support. A too high setting value can cause the terminal device serious problems.

e.g. qbadmin -s 10000,0

6. *qbadmin -z*: it is used to set the encoders position as zero. E.g. zero position, conventionally in *SoftHand Pro*, corresponds to the position in which each *SoftHand pro* terminal device finger is on the same plane of the *SoftHand pro* terminal device palm. If zero position correspond to a different fingers alignment (Figure 26), first of all, fingers must be moved in correct position by qbadmin -s command (DO NOT move them manually when DC motor is deactivated since *SoftHand Pro* device motor is not back-drivable). If *SoftHand Pro* terminal device is in a different position, this will be defined as the new zero position. When this command is launched, the routine starts to return the current value of the encoder, pressing CTRL+C two times to start the procedure and to set the position. *qbadmin -z* is also used when you switch the control input mode in handle, indeed in this case it is recommended to reset the encoder value to align it to the handle encoder measurement.

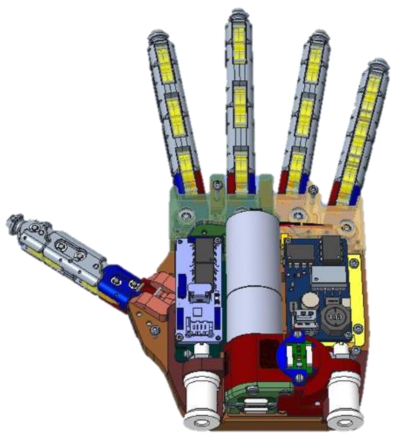


Figure 26: correct SoftHand Pro terminal device zero position

7. *qbadmin -q -v*: it starts a loop which continuously returns and outputs on screen the read values of the EMG sensors. File *emg_values.csv* is also produced in the same folder of *qbadmin* executable file and is filled with the same output as on screen.
8. *qbadmin -c*: it starts a loop which continuously returns and outputs on screen the read motor current. It returns two different values: the first shows actual motor absorbed current while the second shows relevant measure for different additional devices (e.g. haptic feedback). For details on this functionality, please refer to the haptic feedback relative documents.
9. *qbadmin -b*: it switches the board in bootloader mode. This operation mode is used only if is necessary to reprogram the board to change or to update the internal PSoC5 firmware. When this command is executed, the board led turn off and the board is ready to be programmed. See
10. Load by using bootloader section for further information about programming.
11. *qbadmin -v*: this option is used as an appendix to another command for verbose output. If launched alone, it will return an overview about all possible *qbadmin* commands.

```
e.g. qbadmin -s 1000,0 -v
```

12. *qbadmin -k*: it starts a long series of opening and closing of the *SoftHand Pro* terminal.

Notice that these commands are valid for all devices connected to the computer. If two boards are connected to the computer and it is necessary to command only one, board ID must be specified to the command. So, for example if two boards with ID respectively 61 and 4, to use only the first one with ID = 61 the effective command must be:

qbadmin 61 -command

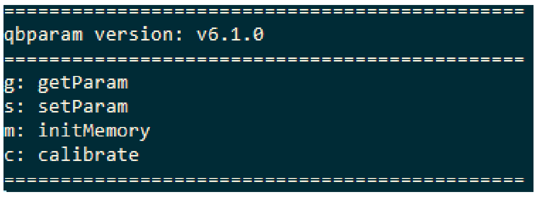
In this manner only the first board will receive the command.

13. *qbadmin -r*: (specific for *SoftHand Pro*) print information about cycle counters and usage of the *SoftHand Pro* device.
14. *qbadmin -S*: (specific for *SoftHand Pro*) get some files with list of current parameters and data present on the SD card (if configured and enabled).
15. *qbadmin -A*: (specific for Generic firmware) get the raw reading of ADC channels on the board as configured with *ADC channel[]* parameters with either *qbparam* or *nmmi_param*
16. *qbadmin -E*: (specific for Generic firmware) get the raw reading of connected encoders on the board as configured with *Read enc raw line* parameters with either *qbparam* or *nmmi_param*
17. *qbadmin -Q:* read all the connected IMUs if enabled and as configured with *nmmi_param_imu* tool (see section nmmi_param_imu tool)

qbparam and nmmi_param tool

qbparam and nmmi_param are parallel tools used to get or set firmware parameters that are stored in logic board memory. Once called, with the ID of the board to configure, you must follow displayed instructions as in Figure 27:

1. g: getParam. Get firmware parameters;
2. s: setParam. Set firmware parameters;
3. m: initMemory. Restore firmware parameters to default settings;
4. c: calibrate. Not used.

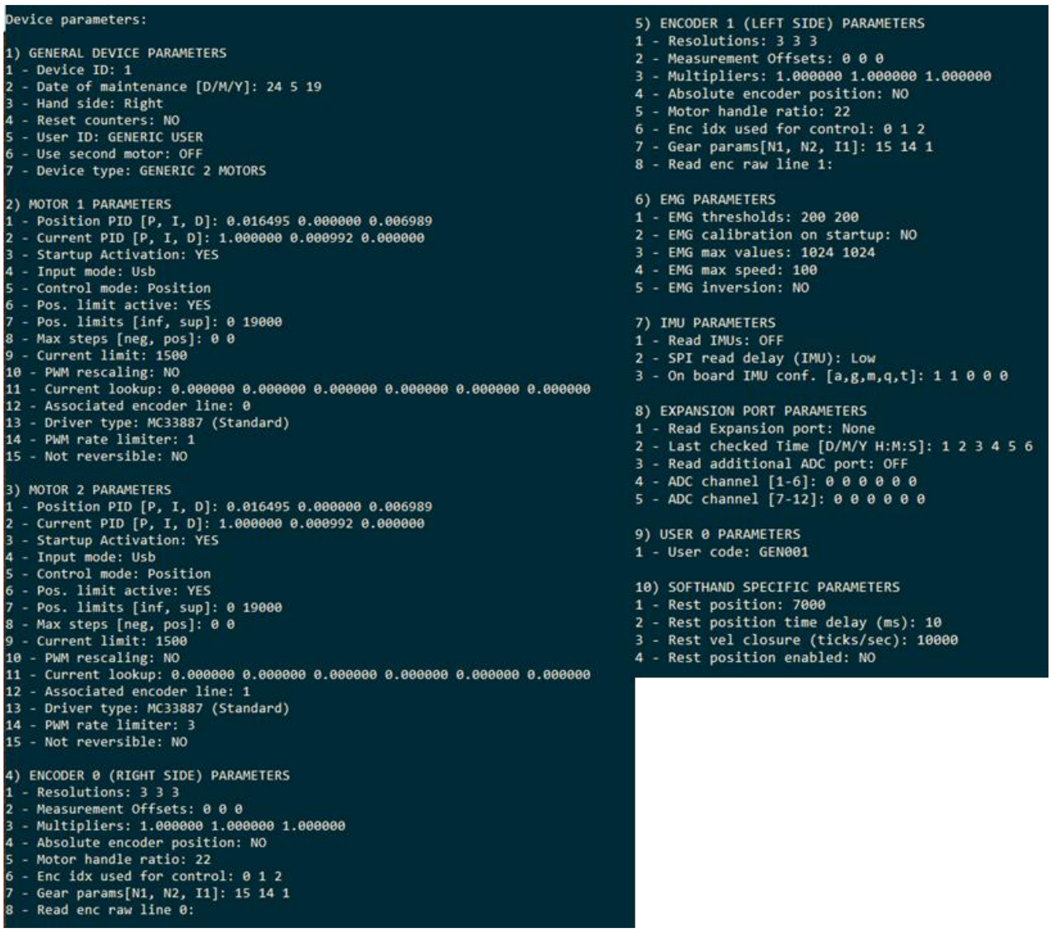


```
============================================
qbparam version: v6.1.0
============================================
g: getParam
s: setParam
m: initMemory
c: calibrate
============================================
```

Figure 27: qbparam main display

The main difference between the two tools is that *qbparam* parameters list view has been reorganized with new tool *nmmi_param* since the number of parameters has increased respect to previous firmware versions and new parameters and commands have been created.

While *qbparam* tool retrieves the whole list of parameters, *nmmi_param* tool works in the same as *qbparam* but it is organized in sections. With it it's possible also only to access specific sections without viewing all the parameters list (see Figure 28).



```
Device parameters:

1) GENERAL DEVICE PARAMETERS
1 - Device ID: 1
2 - Date of maintenance [D/M/Y]: 24 5 19
3 - Hand side: Right
4 - Reset counters: NO
5 - User ID: GENERIC USER
6 - Use second motor: OFF
7 - Device type: GENERIC 2 MOTORS

2) MOTOR 1 PARAMETERS
1 - Position PID [P, I, D]: 0.016495 0.000000 0.006989
2 - Current PID [P, I, D]: 1.000000 0.000992 0.000000
3 - Startup Activation: YES
4 - Input mode: Usb
5 - Control mode: Position
6 - Pos. limit active: YES
7 - Pos. limits [inf, sup]: 0 19000
8 - Max steps [neg, pos]: 0 0
9 - Current limit: 1500
10 - PWM rescaling: NO
11 - Current lookup: 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
12 - Associated encoder line: 0
13 - Driver type: MC33887 (Standard)
14 - PWM rate limiter: 1
15 - Not reversible: NO

3) MOTOR 2 PARAMETERS
1 - Position PID [P, I, D]: 0.016495 0.000000 0.006989
2 - Current PID [P, I, D]: 1.000000 0.000992 0.000000
3 - Startup Activation: YES
4 - Input mode: Usb
5 - Control mode: Position
6 - Pos. limit active: YES
7 - Pos. limits [inf, sup]: 0 19000
8 - Max steps [neg, pos]: 0 0
9 - Current limit: 1500
10 - PWM rescaling: NO
11 - Current lookup: 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
12 - Associated encoder line: 1
13 - Driver type: MC33887 (Standard)
14 - PWM rate limiter: 3
15 - Not reversible: NO

4) ENCODER 0 (RIGHT SIDE) PARAMETERS
1 - Resolutions: 3 3 3
2 - Measurement Offsets: 0 0 0
3 - Multipliers: 1.000000 1.000000 1.000000
4 - Absolute encoder position: NO
5 - Motor handle ratio: 22
6 - Enc idx used for control: 0 1 2
7 - Gear params[N1, N2, I1]: 15 14 1
8 - Read enc raw line 0:

5) ENCODER 1 (LEFT SIDE) PARAMETERS
1 - Resolutions: 3 3 3
2 - Measurement Offsets: 0 0 0
3 - Multipliers: 1.000000 1.000000 1.000000
4 - Absolute encoder position: NO
5 - Motor handle ratio: 22
6 - Enc idx used for control: 0 1 2
7 - Gear params[N1, N2, I1]: 15 14 1
8 - Read enc raw line 1:

6) EMG PARAMETERS
1 - EMG thresholds: 200 200
2 - EMG calibration on startup: NO
3 - EMG max values: 1024 1024
4 - EMG max speed: 100
5 - EMG inversion: NO

7) IMU PARAMETERS
1 - Read IMUs: OFF
2 - SPI read delay (IMU): Low
3 - On board IMU conf. [a,g,m,q,t]: 1 1 0 0 0

8) EXPANSION PORT PARAMETERS
1 - Read Expansion port: None
2 - Last checked Time [D/M/Y H:M:S]: 1 2 3 4 5 6
3 - Read additional ADC port: OFF
4 - ADC channel [1-6]: 0 0 0 0 0 0
5 - ADC channel [7-12]: 0 0 0 0 0 0

9) USER 0 PARAMETERS
1 - User code: GEN001

10) SOFTHAND SPECIFIC PARAMETERS
1 - Rest position: 7000
2 - Rest position time delay (ms): 10
3 - Rest vel closure (ticks/sec): 10000
4 - Rest position enabled: NO
```

Figure 28: nmmi_param example of whole list of parameters

This is the usage of *nmmi_param* tool with allowed sections:

nmmi_param.exe ID section_identifier

Parameters Section	**Identifier**
Device	'*dev*' or '*device*'
Motor	'*mot*' or '*motor*'
Encoder	'*enc*' or '*encoder*'
EMG	'*emg*'
IMU	'*imu*'
Expansion port	'*exp*' or '*expansion*'
User	'*usr*' or '*user*'
SoftHand	'*SH*' or '*softhand*'
All parameters	'*all*'

In case you are viewing more than one section and in order to set parameters by using *nmmi_param* you must follow the displayed instructions by indicating first the section number and then number of the parameter you have to change. The list of following parameters (equal for getParam and setParam options) are defined:

- *ID*: it is the ID of the board and it is used to identify it. If this parameter is set to 250, the board will be configured in dummy mode.
- *PID param*: when the terminal device is controlled in position, a PID controller is used to drive the motor. The PID parameters are respectively the values for proportional, integrative and derivative actions. It is not possible to change only one value, so if a value must be modified, the proportional for example, the other two values must be remembered and redefined.
- *PID param current*: this parameter is the analogue of *PID param* and it is used when the motor is controlled in current mode.
- *Startup Activation flag*: to improve User's safety, always boards starts with motor control switched off and to activate it is necessary to use the command `qbadmin -a`. However, it is possible to change this configuration and start up the terminal device with the motor control switched on.
- *Input mode*: it is possible to use a terminal device in six different ways. The first mode is using a computer to give to the motor reference position or current. In this case terminal device is managed using the *qbadmin* command or the GUI. In the second mode handle is introduced and works as a joystick. The other four modes are used to manage e.g. *SoftHand Pro* terminal device using EMG signals in prosthesis configurations.
- *Control mode*: it is possible to control a terminal device in four different ways. The most commonly used mode is the position mode; however, it is possible to use EMG signals or current mode, plus a fourth mode which uses both position and current.
- *Resolution*: this parameter is related to the angular resolution of the encoder. E.g. the opening or closing of *SoftHand Pro* terminal device is given by a single tendon which is rewinded around a pulley. The resolution corresponds to motor revolutions needed to close *SoftHand Pro* terminal device. It is recommended to not change these values.
- *Measurements offset*: it is used to set the zero position of the encoder board readings. E.g. zero position, conventionally in *SoftHand Pro* device, corresponds to the position in which each

SoftHand Pro terminal device finger is on the same plane of the *SoftHand Pro* terminal device palm - Figure 26.

- *Multipliers*: these values are used to define a relationship between the values read from the encoder and the values related to the desired positions.
- *Position limit flag*: in order to preserve the terminal device in the case the User gives a wrong value as desired to it, a special flag was inserted. If it is activated in case of wrong value, the program will not consider it.
- *Position limits*: these are two values, an **Inf** and a **Sup** which define the range of position values for the terminal device between fully opened and fully closed.
- *Current limit*: in case the motor needs more power, it increases its power consumption until the maximum value when it is in stall condition. It is possible to set a maximum value for the motor current consumption, so that motor life is preserved.
- *Emg Calibration flag*: setting this parameter to "YES" will make a calibration for Users' maximum values when the terminal device is powered.
- *Emg thresholds*: this parameter is used in prosthetic configurations. These parameters are used to close/open the terminal device.
- *Emg Max Value*: this parameter is used to manually set the Users' muscle activation maximum values. These parameters are used to correctly set the measurements' range to [0 - 1024] according to User's strength.
- *Emg Max Speed*: this parameter is used to speed up or slow down terminal device opening/closing when controlled with the EMGs.
- *Double encoder*: if this parameter is set to **true,** two encoders are mounted on the terminal device to retrieve motor position. Otherwise there is only one encoder mounted. E.g. for *SoftHand Pro* terminal devices this parameter must be set to **true**. Since this software is used also by other devices (e.g. haptic feedback), this setting must be specified.
- *Motor Handle ratio*: when the configuration needs the use of a handle, the terminal device is controlled in position with the reference given by the encoder mounted on the handle. Normally, the variation range of this encoder is about 900 ticks while the open-close range of e.g. *SoftHand Pro* terminal device is about 16000 ticks, so it is needed to use a multiplier to remap the values read from the handle encoder in right values for the terminal device. This multiplier depends by the terminal device and by the handle, normally this multiplier is set to 20.
- *Hand side*: set the side of the device (specific for *SoftHand Pro* device or feedback devices).
- *User ID*: can be chosen between Generic User and other known users. When changed, retrieve previously stored EMG parameters and User code
- *Enable IMUs*: enables on board and connected IMUs reading
- *Read Expansion port*: enables SD card saving (a board reset is done)
- *Last checked time*: parameter used to configure dare of internal RTC (if SD card module is present) [D/M/Y H:M:S]
- *User code*: 6 characters string to identify SD card internal folder where to save user usage data
- *Reset counters*: if set to true, it resets usage device statistics counters and then return to false. Automatically, the following usage counters update:
    1. *Total amount of time the board is powered on.*
    2. *Total amount of time the device stays in rest position. "Rest position" is a function that automatically drives e.g. the SoftHand Pro to a slightly closed, more natural, position if the hand is fully open and not activated for a certain amount of time. This function can be switched off, while the amount of waiting time and the rest position can be configured according to individual preferences. The "rest position" setting will be adjusted during the pre-intervention session. The firmware records the absolute time the SHP is in a "rest position" as well as the number of times per day.*
    3. *Amount of device movement (hystogram). The firmware records the amount of device movement and computes the movement data frequency distribution.*

4. *Grasp force. The amount of electrical current the e.g. the SoftHand Pro uses at maximum closure is proportional to the magnitude of grasp force. The firmware records the electrical current at e.g. the SoftHand Pro maximum closure and the frequency distribution of occurrences of maximum forces.*
5. *Amount of motion of the hand (total). The firmware records the amount of motor movement and computes the total integral over the full period of time.*
6. *Activation of each EMG channel. The firmware records the frequency with which each EMG channel is activated, i.e., how many times EMG amplitude crosses a given threshold.*

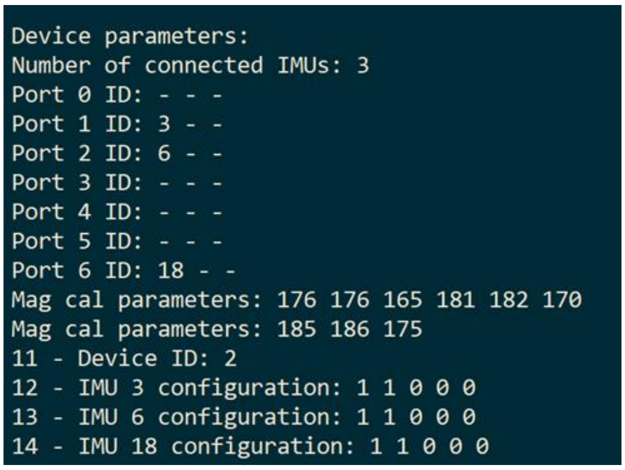
Notice that it is not recommended to change any of the previous values because doing so could damage the terminal device. Hence, be careful when modifying them.

Note also that some parameters change need a board reset that occurs automatically. An alert it is reported in case in its related menu.

nmmi_param_imu tool

This tool is used to get information and set parameters related to the IMUs connected to the board.

As the previously described tools you can get or set firmware parameters that are stored in logic board memory. Once called, with the ID of the board to configure, you must follow displayed instructions. Figure 29 shows an example of a board with three IMUs connected.



```
Device parameters:
Number of connected IMUs: 3
Port 0 ID: - - -
Port 1 ID: 3 - -
Port 2 ID: 6 - -
Port 3 ID: - - -
Port 4 ID: - - -
Port 5 ID: - - -
Port 6 ID: 18 - -
Mag cal parameters: 176 176 165 181 182 170
Mag cal parameters: 185 186 175
11 - Device ID: 2
12 - IMU 3 configuration: 1 1 0 0 0
13 - IMU 6 configuration: 1 1 0 0 0
14 - IMU 18 configuration: 1 1 0 0 0
```

Figure 29: nmmi_param_imu main display

With the tool you can change the board ID (as in *qbparam*) and if to enable (value 1) or not (value 0) the reading of each IMU sensors in the following order:

- Accelerometers
- Gyroscopes
- Magnetometers
- Quaternions
- Temperature

For example, the following configuration

IMU 3 configuration. 1 1 0 0 0

means you want to read accelerometers and gyroscopes from IMU 3, but not magnetometers, quaternions and temperatures.

Simulink® library

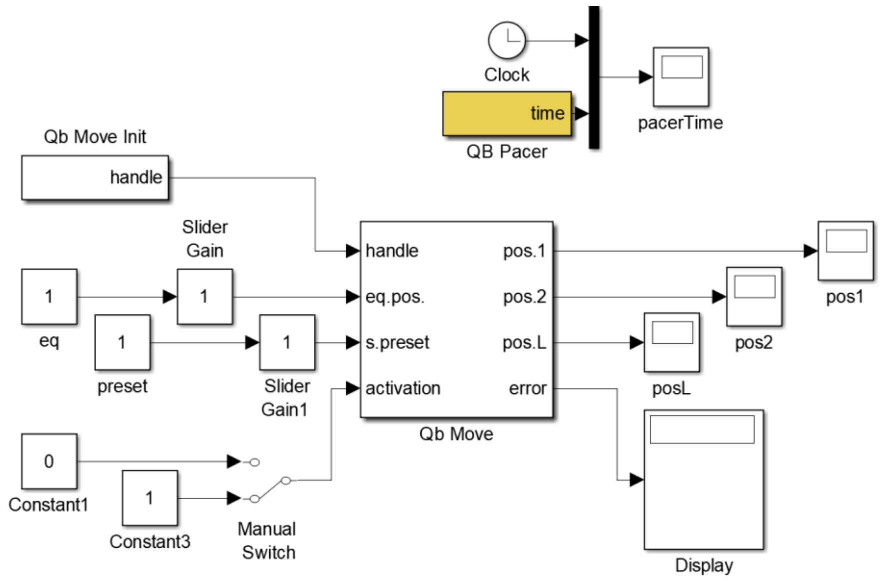


Figure 30: Simulink® scheme example that uses qbmove_simulink library

This tool allows to interface board firmware with Matlab®/Simulink® application and can be downloaded at https://github.com/NMMI/qbmove-simulink. This library is composed of a series of Simulink® blocks based on the APIs that implement communication functionalities.

How to install the library

Follow the instructions at the link https://github.com/NMMI/qbmove-simulink under Installation section for downloading, compiling and including the library in the Matlab® software.

Library description

It is called *qbmove_simulink* and is organized as multiple Simulink® blocks, each one implementing an API functionality. These blocks are listed and described below (refer to Figure 30):

- QB Pacer: mandatory block in the model. Otherwise, the simulation time will not match real time. A number can be set and represents the ratio between simulation time and the real time. Default value is 1.
- QB Move: interface between computer and the board. By default, 3 input ports and 4 outputs ports are defined. This means that the Simulink® block is set either to receive information from the encoder sensor and send new position reference to the SoftHand board. Double-clicking the Simulink® block, Function Block Parameters will open permitting the setting of the proper board ID and the Communication Direction. If **Tx** communication direction is set, only new reference input to the SoftHand board can be sent. If **Rx** communication direction is set, only position sensors value can be read. Either inputs and outputs are in degrees.
  To use this QB Move, QB Move Init block must be inserted in the model and connected to the first input handle. To use multiple boards, an array containing the IDs in the ID field must be filled and a multiplexer block can be used to send multiple inputs to the Simulink® block ports. ID = 0 must be set if board ID is unknown.
- QB Move Init: this Simulink® block is used to provide the right handle to the other Simulink® blocks and allows the communication between the computer and the SoftHand boards. The name port must be typed between single quotes as:

e.g. 'port_name'

- Windows® OS: boards are seen as Communication Ports (COM1, COM2, COM#...). If the board is not automatically recognized by Windows®, the procedure described in GUI section must be followed.
- UNIX or MacOS®: under Unix no special modification must be done. Under MacOS®, probably the FTDI driver must be installed. To retrieve the correct port name, *ls /dev/* must be typed on terminal window. Multiple interfaces can be found:
  - MacOS®: the interface is usually called "tty.usbserial-XX" where XX is the serial of the board.
  - UNIX: the interface is usually called "ttyUSB0" or similar.

e.g. '/dev/tty.usbserial-13' or '/dev/tty.USB0'

- Get IMU Readings (IMU): interface between computer and the board. There is 1 input port (handle) and as much outputs ports as connected IMU. This Simulink® block is used to receive information from the IMU sensors connected to the board, enabled and configured. Double-clicking the Simulink® block, Function Block Parameters will open permitting the setting of the proper board ID and the Number of connected IMUs. You can also choose the output unities of accelerometers (G or m/s^2) and gyroscopes (deg/s or rad/s).
  To use this block, QB Move Init block must be inserted in the model and connected to the first input handle. To use multiple boards, an array containing the IDs in the ID field must be filled and a multiplexer block can be used to send multiple inputs to the Simulink® block ports. ID = 0 must be set if board ID is unknown.
- Get IMU Readings (Sensors): interface between computer and the board. There is 1 input port (handle) and 5 outputs ports, one for each readable sensor. This Simulink® block is used to receive information from the IMU sensors connected to the board, enabled and configured. Double-clicking the Simulink® block, Function Block Parameters will open permitting the setting of the proper board ID and the Number of connected IMUs. You can also choose the output unities of accelerometers (G or m/s^2) and gyroscopes (deg/s or rad/s).
  To use this block, QB Move Init block must be inserted in the model and connected to the first input handle. To use multiple boards, an array containing the IDs in the ID field must be filled and a multiplexer block can be used to send multiple inputs to the Simulink® block ports. ID = 0 must be set if board ID is unknown.

Example of software usage

GUI

For GUI, first connect the board through a micro-USB-USB A cable to the computer, then double click on the downloaded application. If the board is properly connected, User/Technician will see the serial port on the bottom part of the window. Once the serial port is connected, all of the buttons are enabled Then, the GUI can be used as explained in GUI section.

qbadmin/qbparam/nmmi_param/nmmi_param_imu

To use these command-line tools, open a command window, move to the folder in which the softwares have been installed, then digit *qbadmin -t* first and then execute commands as described in Command-line tools section.

Example of Simulink libraries usage

A general use of the library is shown in the following:

1. From the Matlab® application, click on the **Simulink Library** icon or type *simulink* in the Matlab® Command Window.
2. Create a new model using **File→New→Model**.
3. In the new model go to: **Simulation** -> **Model Configuration Parameters**. Under **Solver** select as Type **Fixed-Step**, as Solver **ode1 (Euler)** or **discrete (no continuous states)** and as **Fixed-step size** type the *delta_t* in seconds. (e.g. if you want to retrieve positions and set new inputs every 5 milliseconds, type 0.005). Click **OK**. Every board needs at least 1 millisecond to read and set new positions, so the minimum step time allowed is 1 millisecond multiplied by the number of boards in the chain.

QB Pacer and Qb Move Library can be found under libraries in Simulink Library Browser. From here, drag and drop the desired blocks in the Simulink® model.

In the main *qbmove_simulink* folder there is an example called "*qbmove_example.slx*". This is a simple configuration which can be used to test a board and it is the same reported in Figure 30. On the left side of the figure, there are two slider gains which can be tweaked while the simulation is running. On the right side of the figure, there are three displays where the values in degree of the three encoders attached to the terminal device can be read. The output error can be used to see if there are errors in communication. It starts from zero and it is incremented by 1 every time a communication error occurs. In the bottom-left corner there is also the QB Pacer block used to ensure the correct synchronization between simulation and real time. If desired, it is possible to use the output of this block to trace the time of the simulation. In the scope two lines can be observed, one is the simulation clock and the other one is the real time pacer output. If they overlap, the sample time of the simulation is properly chosen, this means that the step size is big enough to let the communication finish between two consecutive steps. If the two lines diverge, it is advised to use a bigger step size.

In the configuration of Figure 30, the step size is set to 2 milliseconds and the computer should be able to run the simulation in real time. This means that every 2 milliseconds it is possible to send a new reference position to the board and read the current position. Furthermore, a current reading is performed to see the milliamperes absorbed by terminal device DC motor.

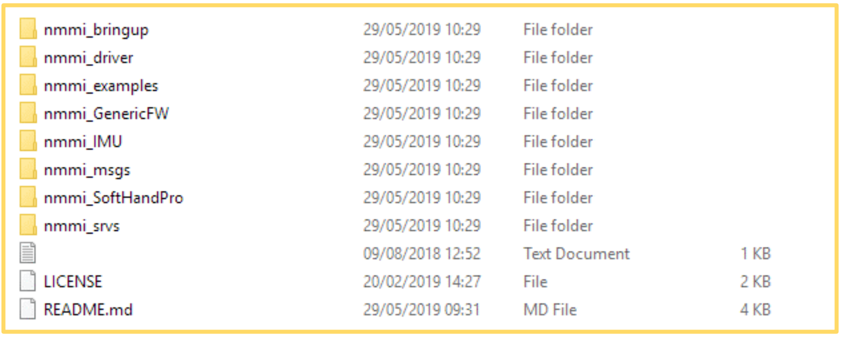
ROS packages

Preliminary considerations

This chapter will explain how it works and how to use ROS-based packages for SoftHand Pro, Generic and IMU devices, starting from the structure of QB Robotics® ROS packages for qbHand and qbMove devices. In fact, since the need of handling IMU and generic FW features, new nodes have been implemented:

- IMU, SoftHandPro and GenericFW hardware interfaces
- NMMI communication handler to expand the QB one with the additional features
- NMMI custom messages, services and bring-up files
- New states topic in addition to old ones

ROS packages destination of use is intended for monitoring safety measurements, allowing Users to communicate with the logic board, to retrieve sensors measurements and to command the devices in a safe and proper way.



nmmi_bringup	29/05/2019 10:29	File folder	
nmmi_driver	29/05/2019 10:29	File folder	
nmmi_examples	29/05/2019 10:29	File folder	
nmmi_GenericFW	29/05/2019 10:29	File folder	
nmmi_IMU	29/05/2019 10:29	File folder	
nmmi_msgs	29/05/2019 10:29	File folder	
nmmi_SoftHandPro	29/05/2019 10:29	File folder	
nmmi_srvs	29/05/2019 10:29	File folder	
	09/08/2018 12:52	Text Document	1 KB
LICENSE	20/02/2019 14:27	File	2 KB
README.md	29/05/2019 09:31	MD File	4 KB

Figure 31: list of generated files

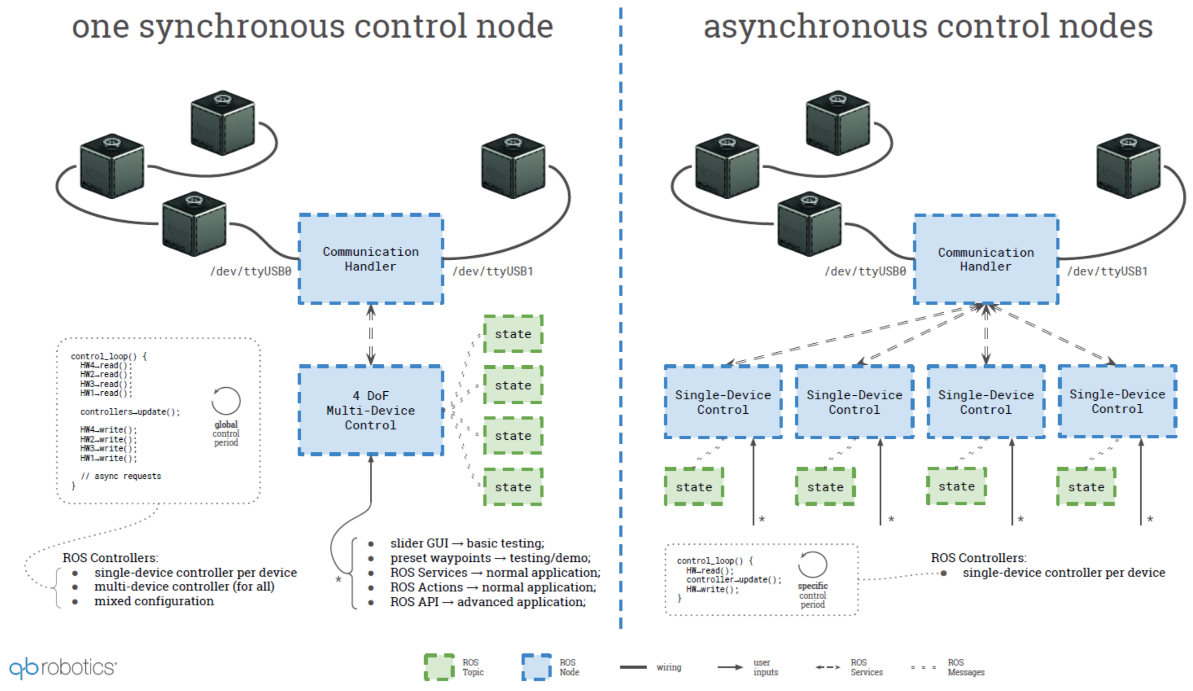
Figure 31 summarizes the folder organization for the new packages added to QB native ones. The whole documentation and installation guides for QB and NMMI packages can be found here:

- *ROS-base*: https://github.com/NMMI/ROS-base
- *ROS-qbmove*: https://github.com/NMMI/ROS-qbmove
- *ROS-SoftHand*: https://github.com/NMMI/ROS-SoftHand
- *ROS-examples*: https://github.com/NMMI/ROS-examples
- *ROS-NMMI*: https://github.com/NMMI/ROS-NMMI

Packages overview

This section shows packages interaction, messages and services exchanged between the developed ROS packages, but it is reported here only as a short recap. You can find more information about how the QB nodes work by reading the extended documentation at the just proposed repositories links.

As shown in Figure 32 there are two distinct configurations to control several QB devices connected to the system.


one synchronous control node
asynchronous control nodes
Communication Handler
/dev/ttyUSB0
/dev/ttyUSB1
4 DoF Multi-Device Control
state
global control period
ROS Controllers:
single-device controller per device
multi-device controller (for all)
mixed configuration
slider GUI → basic testing;
preset waypoints → testing/demo;
ROS Services → normal application;
ROS Actions → normal application;
ROS API → advanced application;
Single-Device Control
specific control period
ROS Controllers:
single-device controller per device
qbrobotics
ROS Topic
ROS Node
wiring
user inputs
ROS Services
ROS Messages

```
control_loop() {
  HW4->read();
  HW2->read();
  HW3->read();
  HW1->read();

  controllers->update();

  HW4->write();
  HW2->write();
  HW3->write();
  HW1->write();

  // async requests
}
```

```
control_loop() {
  HW->read();
  controller->update();
  HW->write();
}
```

Figure 32: QB node control types

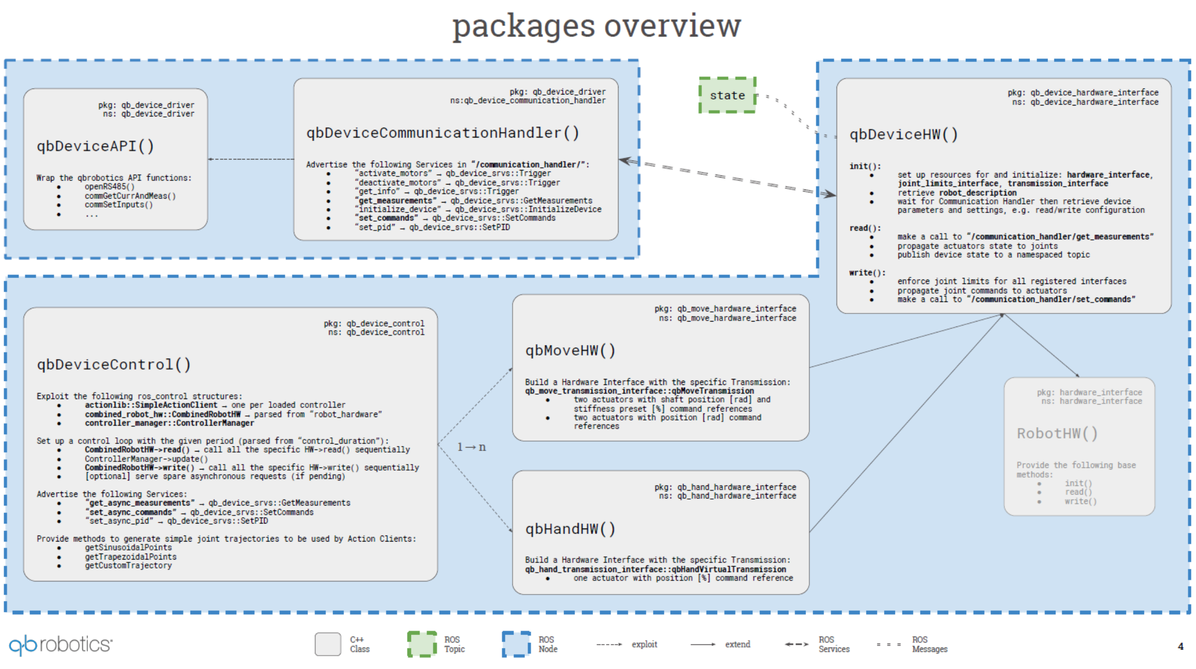
The first (and recommended) groups all the Hardware Interfaces together (thanks to the *combined_robot_hw*) and exploits them as a unique robot system. It is called "synchronous" just to point out that every sequence of reads and writes is always done in the same predefined order.

The second mode threats every device as an independent Hardware Interface with its dedicated ROS node which executes the control loop independently with the respect to the rest of the system, i.e. "asynchronously". Mixed configurations can be also achieved through a proper setup. In such a case we can think of synchronous sub-systems which execute asynchronously with the respect to each other. Note that in a single-device system the synchronous mode is a nonsense.

In both cases there is always one central Node which manages the shared resources for the serial communication (e.g. one or many USB ports) and which provides several ROS services to whom wants to interact with the connected devices. This Node is called Communication Handler and it is usually started in a separate terminal.

Please remember that in a multi-device configuration, each device connected to your system must have a unique ID.

To understand what is hiding under the hood, have a look at the C++ classes overview (see Figure 33) which sums up all the main concepts of our ROS packages.



pkg: qb_device_driver
ns: qb_device_driver

qbDeviceAPI()

Wrap the qbrobotics API functions:
- openRS485()
- commGetCurrAndMeas()
- commSetInputs()
- ...

pkg: qb_device_driver
ns:qb_device_communication_handler

qbDeviceCommunicationHandler()

Advertise the following Services in "**/communication_handler/**":
- "activate_motors" → qb_device_srvs::Trigger
- "deactivate_motors" → qb_device_srvs::Trigger
- "get_info" → qb_device_srvs::Trigger
- "**get_measurements**" → qb_device_srvs::GetMeasurements
- "initialize_device" → qb_device_srvs::InitializeDevice
- "**set_commands**" → qb_device_srvs::SetCommands
- "set_pid" → qb_device_srvs::SetPID

pkg: qb_device_hardware_interface
ns: qb_device_hardware_interface

qbDeviceHW()

init():
- set up resources for and initialize: **hardware_interface**, **joint_limits_interface**, **transmission_interface**
- retrieve **robot_description**
- wait for Communication Handler then retrieve device parameters and settings, e.g. read/write configuration

read():
- make a call to "**/communication_handler/get_measurements**"
- propagate actuators state to joints
- publish device state to a namespaced topic

write():
- enforce joint limits for all registered interfaces
- propagate joint commands to actuators
- make a call to "**/communication_handler/set_commands**"

pkg: qb_device_control
ns: qb_device_control

qbDeviceControl()

Exploit the following ros_control structures:
- **actionlib::SimpleActionClient** → one per loaded controller
- **combined_robot_hw::CombinedRobotHW** → parsed from "robot_hardware"
- **controller_manager::ControllerManager**

Set up a control loop with the given period (parsed from "control_duration"):
- **CombinedRobotHW->read**() → call all the specific HW->read() sequentially
- ControllerManager->update()
- **CombinedRobotHW->write**() → call all the specific HW->write() sequentially
- [optional] serve spare asynchronous requests (if pending)

Advertise the following Services:
- "**get_async_measurements**" → qb_device_srvs::GetMeasurements
- "**set_async_commands**" → qb_device_srvs::SetCommands
- "set_async_pid" → qb_device_srvs::SetPID

Provide methods to generate simple joint trajectories to be used by Action Clients:
- getSinusoidalPoints
- getTrapezoidalPoints
- getCustomTrajectory

pkg: qb_move_hardware_interface
ns: qb_move_hardware_interface

qbMoveHW()

Build a Hardware Interface with the specific Transmission:
qb_move_transmission_interface::qbMoveTransmission
- two actuators with shaft position [rad] and stiffness preset [%] command references
- two actuators with position [rad] command references

pkg: qb_hand_hardware_interface
ns: qb_hand_hardware_interface

qbHandHW()

Build a Hardware Interface with the specific Transmission:
qb_hand_transmission_interface::qbHandVirtualTransmission
- one actuator with position [%] command reference

pkg: hardware_interface
ns: hardware_interface

RobotHW()

Provide the following base methods:
- init()
- read()
- write()

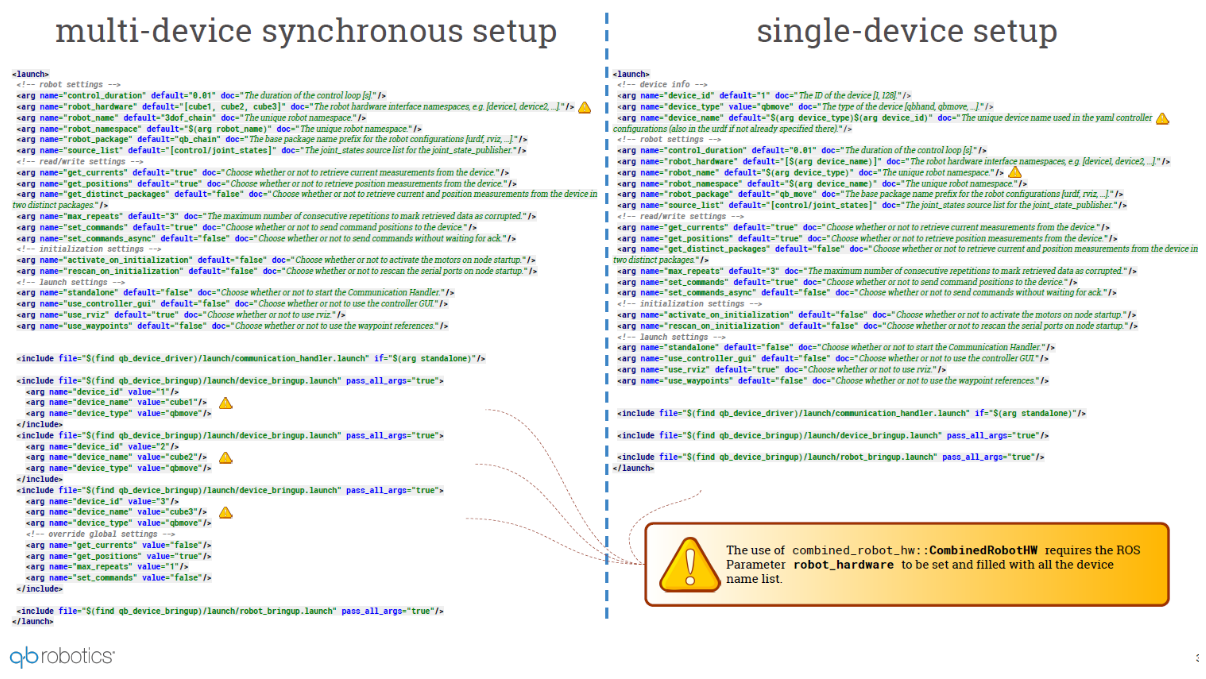
Figure 33: QB node packages overview

The control Node exploits the *ros_control* Controller Manager which loads and runs the device controllers. Each controller provides an Action Server that, together with the Hardware Interface structure, allows the user to send commands to the relative device and get its measurements.

From an API point of view, it is implemented an Action Client which matches the relative trajectory controller and provides a method to send Goals, i.e. command references, directly to the given device. Additionally, the Action Client is subscribed to a topic (**_controller/command*) that can be used to send reference commands from outside the code, e.g. asynchronously from the command line, or from a higher-level control Node, e.g. as a result of a planning algorithm.

It is recommended not to mix these two control modes: choose either to control the device directly from the code by extending our API or through this command Topic.

Regardless the control mode chosen for the given application, and apart form a customization of the API, the following launch file templates can be used respectively to control several devices or a single one (see Figure 34).


multi-device synchronous setup

```xml
<launch>
 <!-- robot settings -->
 <arg name="control_duration" default="0.01" doc="The duration of the control loop [s]."/>
 <arg name="robot_hardware" default="[cube1, cube2, cube3]" doc="The robot hardware interface namespaces, e.g. [device1, device2, ...]."/>
 <arg name="robot_name" default="3dof_chain" doc="The unique robot namespace."/>
 <arg name="robot_namespace" default="$(arg robot_name)" doc="The unique robot namespace."/>
 <arg name="robot_package" default="qb_chain" doc="The base package name prefix for the robot configurations [urdf, rviz, ...]."/>
 <arg name="source_list" default="[control/joint_states]" doc="The joint_states source list for the joint_state_publisher."/>
 <!-- read/write settings -->
 <arg name="get_currents" default="true" doc="Choose whether or not to retrieve current measurements from the device."/>
 <arg name="get_positions" default="true" doc="Choose whether or not to retrieve position measurements from the device."/>
 <arg name="get_distinct_packages" default="false" doc="Choose whether or not to retrieve current and position measurements from the device in two distinct packages."/>
 <arg name="max_repeats" default="3" doc="The maximum number of consecutive repetitions to mark retrieved data as corrupted."/>
 <arg name="set_commands" default="true" doc="Choose whether or not to send command positions to the device."/>
 <arg name="set_commands_async" default="false" doc="Choose whether or not to send commands without waiting for ack."/>
 <!-- initialization settings -->
 <arg name="activate_on_initialization" default="false" doc="Choose whether or not to activate the motors on node startup."/>
 <arg name="rescan_on_initialization" default="false" doc="Choose whether or not to rescan the serial ports on node startup."/>
 <!-- launch settings -->
 <arg name="standalone" default="false" doc="Choose whether or not to start the Communication Handler."/>
 <arg name="use_controller_gui" default="false" doc="Choose whether or not to use the controller GUI."/>
 <arg name="use_rviz" default="true" doc="Choose whether or not to use rviz."/>
 <arg name="use_waypoints" default="false" doc="Choose whether or not to use the waypoint references."/>


 <include file="$(find qb_device_driver)/launch/communication_handler.launch" if="$(arg standalone)"/>

 <include file="$(find qb_device_bringup)/launch/device_bringup.launch" pass_all_args="true">
   <arg name="device_id" value="1"/>
   <arg name="device_name" value="cube1"/>
   <arg name="device_type" value="qbmove"/>
 </include>
 <include file="$(find qb_device_bringup)/launch/device_bringup.launch" pass_all_args="true">
   <arg name="device_id" value="2"/>
   <arg name="device_name" value="cube2"/>
   <arg name="device_type" value="qbmove"/>
 </include>
 <include file="$(find qb_device_bringup)/launch/device_bringup.launch" pass_all_args="true">
   <arg name="device_id" value="3"/>
   <arg name="device_name" value="cube3"/>
   <arg name="device_type" value="qbmove"/>
   <!-- override global settings -->
   <arg name="get_currents" value="false"/>
   <arg name="get_positions" value="true"/>
   <arg name="max_repeats" value="1"/>
   <arg name="set_commands" value="false"/>
 </include>

 <include file="$(find qb_device_bringup)/launch/robot_bringup.launch" pass_all_args="true"/>
</launch>
```

single-device setup

```xml
<launch>
 <!-- device info -->
 <arg name="device_id" default="1" doc="The ID of the device [1, 128]."/>
 <arg name="device_type" value="qbmove" doc="The type of the device [qbhand, qbmove, ...]."/>
 <arg name="device_name" default="$(arg device_type)$(arg device_id)" doc="The unique device name used in the yaml controller configurations (also in the urdf if not already specified there)."/>
 <!-- robot settings -->
 <arg name="control_duration" default="0.01" doc="The duration of the control loop [s]."/>
 <arg name="robot_hardware" default="[$(arg device_name)]" doc="The robot hardware interface namespaces, e.g. [device1, device2, ...]."/>
 <arg name="robot_name" default="$(arg device_type)" doc="The unique robot namespace."/>
 <arg name="robot_namespace" default="$(arg device_name)" doc="The unique robot namespace."/>
 <arg name="robot_package" default="qb_move" doc="The base package name prefix for the robot configurations [urdf, rviz, ...]."/>
 <arg name="source_list" default="[control/joint_states]" doc="The joint_states source list for the joint_state_publisher."/>
 <!-- read/write settings -->
 <arg name="get_currents" default="true" doc="Choose whether or not to retrieve current measurements from the device."/>
 <arg name="get_positions" default="true" doc="Choose whether or not to retrieve position measurements from the device."/>
 <arg name="get_distinct_packages" default="false" doc="Choose whether or not to retrieve current and position measurements from the device in two distinct packages."/>
 <arg name="max_repeats" default="3" doc="The maximum number of consecutive repetitions to mark retrieved data as corrupted."/>
 <arg name="set_commands" default="true" doc="Choose whether or not to send command positions to the device."/>
 <arg name="set_commands_async" default="false" doc="Choose whether or not to send commands without waiting for ack."/>
 <!-- initialization settings -->
 <arg name="activate_on_initialization" default="false" doc="Choose whether or not to activate the motors on node startup."/>
 <arg name="rescan_on_initialization" default="false" doc="Choose whether or not to rescan the serial ports on node startup."/>
 <!-- launch settings -->
 <arg name="standalone" default="false" doc="Choose whether or not to start the Communication Handler."/>
 <arg name="use_controller_gui" default="false" doc="Choose whether or not to use the controller GUI."/>
 <arg name="use_rviz" default="true" doc="Choose whether or not to use rviz."/>
 <arg name="use_waypoints" default="false" doc="Choose whether or not to use the waypoint references."/>


 <include file="$(find qb_device_driver)/launch/communication_handler.launch" if="$(arg standalone)"/>

 <include file="$(find qb_device_bringup)/launch/device_bringup.launch" pass_all_args="true"/>

 <include file="$(find qb_device_bringup)/launch/robot_bringup.launch" pass_all_args="true"/>
</launch>
```

The use of `combined_robot_hw::CombinedRobotHW` requires the ROS Parameter `robot_hardware` to be set and filled with all the device name list.

Figure 34: QB node examples of launch files

In addition to packages shown in Figure 33, specific Hardware Interfaces and a new Communication Handler have been developed, as Figure 35 reports. The yellow highlighted packages derive from QB ones. In detail

- *IMUHW*: implements a Hardware Interface to manage IMU sensors connected to a board, enabled and properly configured. The H.I. inherits from *qbDeviceHW* and uses only the reading part, without configuring any actuator
- *SoftHandProHW*: implements a Hardware Interface to control a *SoftHand Pro* terminal device with PSoC5 logic board inside. The H.I. inherits from *qbDeviceHW* and uses the same variables and controls as a *qbHand* device
- *GenericFWHW*: implements a Hardware Interface to read multiple connected and configured encoders and to access measurements of analog sensors connected to the ADC of the board. In the future, this H.I. (who also inherits from *qbDeviceHW*) will be used to control two generic motors, but now, it is configured with no actuators as *IMUHW*
- *nmmiCommunicationHandler*: implements a ROS Communication Handler to be used in conjunction with NMMI Hardware Interfaces. It inherits from *qbDeviceCommunicationHandler* and extends its functionalities by adding new ROS services for reading IMU sensors, analog sensors and encoders
- *nmmiAPI*: implements the API-level for the *nmmiCommunicationHandler* additional functionalities

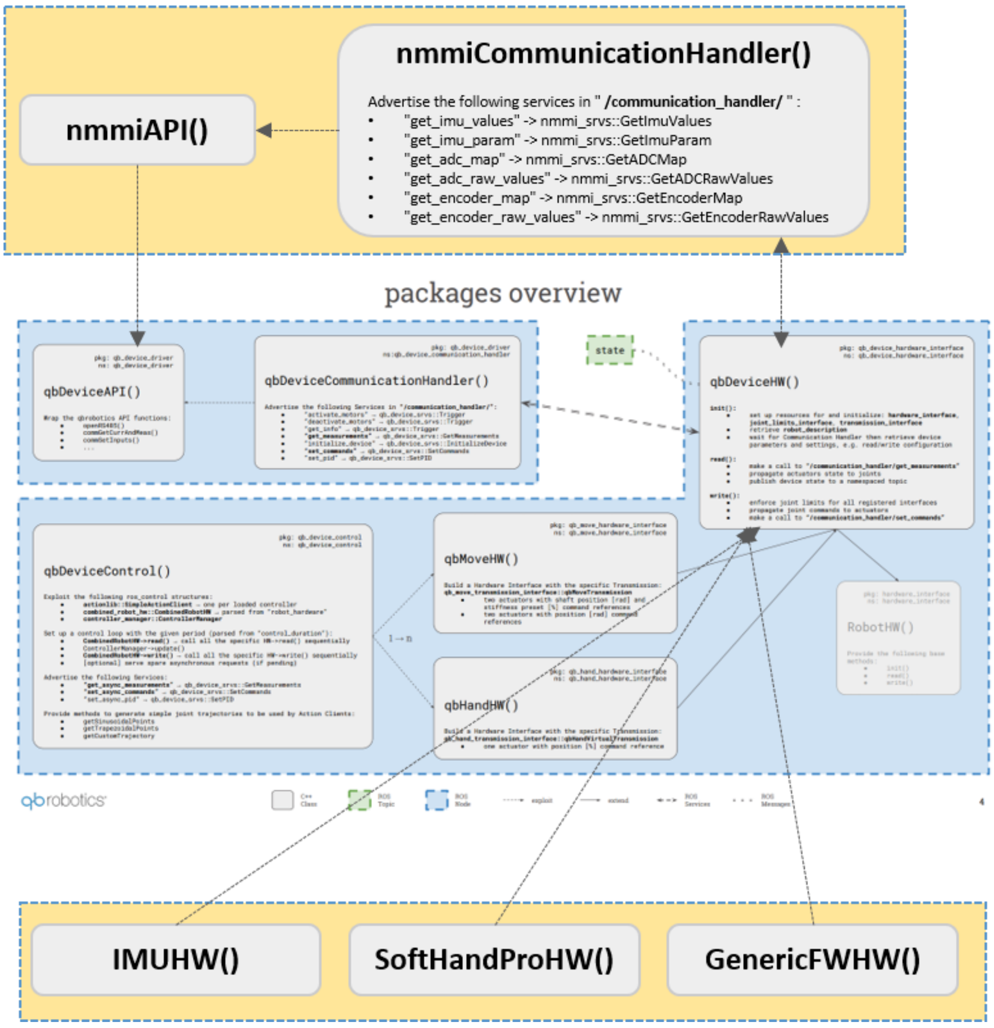

nmmiAPI()
nmmiCommunicationHandler()
Advertise the following services in " /communication_handler/ " :
"get_imu_values" -> nmmi_srvs::GetImuValues
"get_imu_param" -> nmmi_srvs::GetImuParam
"get_adc_map" -> nmmi_srvs::GetADCMap
"get_adc_raw_values" -> nmmi_srvs::GetADCRawValues
"get_encoder_map" -> nmmi_srvs::GetEncoderMap
"get_encoder_raw_values" -> nmmi_srvs::GetEncoderRawValues
packages overview
qbDeviceAPI()
qbDeviceCommunicationHandler()
state
qbDeviceHW()
qbDeviceControl()
qbMoveHW()
qbHandHW()
RobotHW()
1 → n
qbrobotics
IMUHW()
SoftHandProHW()
GenericFWHW()

Figure 35: NMMI packages extension of QB native ones

Using the node

This section briefly explains how to configure ROS in order to be used with the logic board. Independently of the device you are using, you should have *ROS-base* folder in your ROS workspace. Then you must also have:

- *ROS-qbmove*: if you will use a qbMove device from QB
- *ROS-SoftHand*: if you will use a qbHand device from QB
- *ROS-examples*: if you will use an example with one of the QB devices
- *ROS-NMMI*: if you will use SoftHand Pro device, you want to read one or more IMU or use Generic firmware features (read multiple encoders or ADC sampled analog sensors)

Once the necessary packages are in your workspace, run *catkin_make* to compile them.

Once compiled, prepare a launch file with all the necessary arguments (see Ready to use examples section). An exhaustive list is reported below:

- *device_id*: The ID of the device
- *device_type*: The type of the device [qbhand, qbmove, SoftHandPro, generic, ...]
- *control_duration*: The duration of the control loop [s]
- *robot_hardware*: The robot hardware interface names, e.g. [device1, device2, ...]
- *robot_name*: The unique robot name
- *robot_namespace*: The unique robot namespace
- *robot_package*: The base package name prefix for the robot configurations [urdf, rviz, ...], e.g. *softhandpro*
- *get_currents*: Choose whether or not to retrieve current measurements from the device
- *get_positions*: Choose whether or not to retrieve position measurements from the device
- *max_repeats*: The maximum number of consecutive repetitions to mark retrieved data as corrupted
- *set_commands*: Choose whether or not to send command positions to the device
- *set_commands_async*: Choose whether or not to send commands without waiting for ack
- *activate_on_initialization*: Choose whether or not to activate the motors on node startup
- *rescan_on_initialization*: Choose whether or not to rescan the serial ports on node startup
- *standalone*: Choose whether or not to start the Communication Handler
- *use_controller_gui*: Choose whether or not to use the controller GUI
- *use_rviz*: Choose whether or not to use rviz
- *get_imu_values*: Choose whether or not to retrieve IMU measurements from the device
- *compute_quaternions_node*: Choose whether or not to compute quaternions in ROS node
- *compute_angles*: Choose whether or not to compute roll-pitch -yaw angles from quaternions
- *get_adc_raw_values*: Choose whether or not to retrieve ADC raw measurements from the device
- *get_encoder_raw_values*: Choose whether or not to retrieve encoders raw measurements from the device

Then save the launch file and execute *roslaunch your_package your_launch_file.launch* to start ROS node. The node will start publishing all the configured states and measurements.

Accessible topics

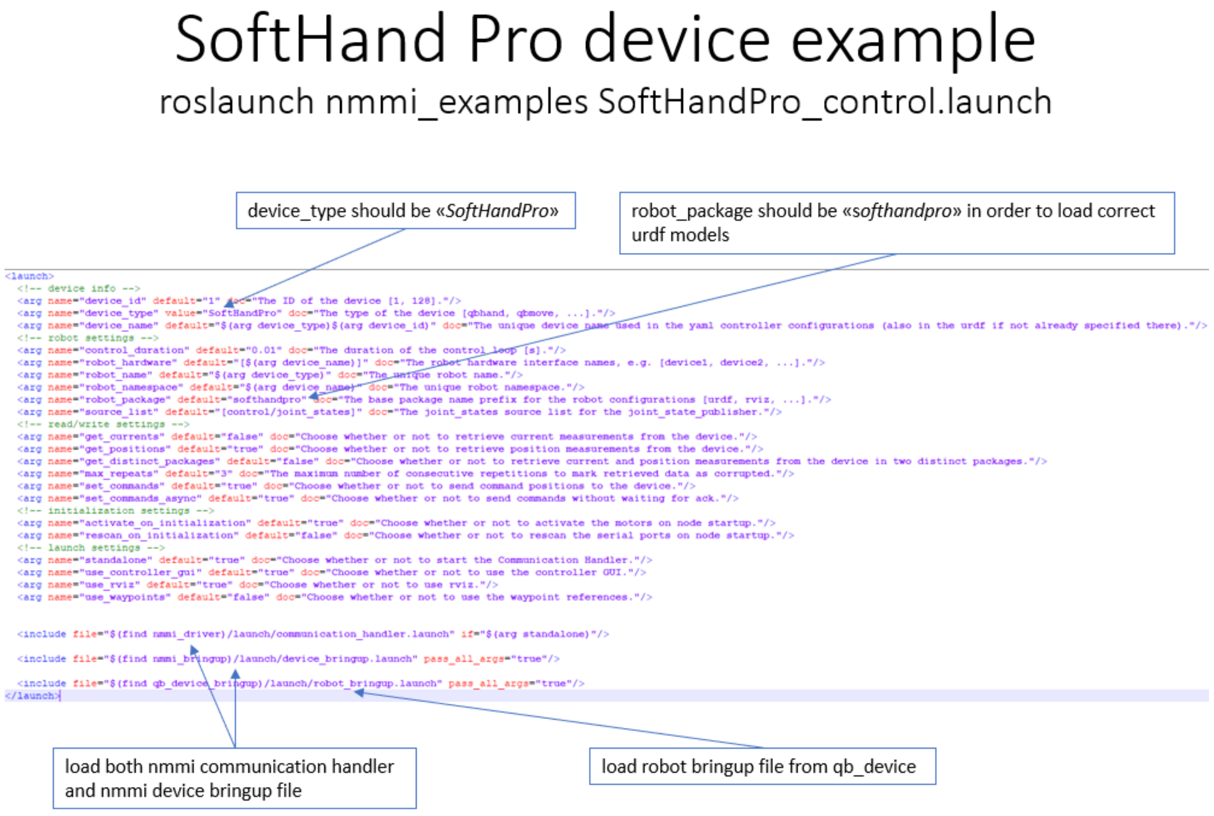
When running, in addition to QB device standard topics, the following topics should be accessible (depending on your firmware configuration):

- */ROBOTNAME/DEVICENAME/state*: state of all configured IMUs with related measures
- */ROBOTNAME/DEVICENAME/acc*: accelerometers reading on all configured IMUs (in g) */ROBOTNAME/DEVICENAME/gyro*: gyroscopes reading on all configured IMUs (in deg/s)
- */ROBOTNAME/DEVICENAME/mag*: magnetometers reading on all configured IMUs (in uT)
- */ROBOTNAME/DEVICENAME/quat*: quaternions reading on all configured IMUs, either read directly from the IMU or computed by the node
- */ROBOTNAME/DEVICENAME/temp*: temperatures reading on all configured IMUs (in °C)
- */ROBOTNAME/DEVICE_NAME/angles*: angles derived from quaternions on all configured IMUs (in g)
- */ROBOT_NAME/DEVICE_NAME/adc_state*: whole state of all ADC configured channels and related raw reading
- */ROBOT_NAME/DEVICE_NAME/adc*: ADC configured channels raw readings
- */ROBOT_NAME/DEVICE_NAME/encoders_state*: whole state of all encoders configured and related raw reading

- */ROBOT_NAME/DEVICE_NAME/encoders*: configured encoders raw readings

Ready to use examples

As listed below, *nmmi_examples* package contains a series of ready to use examples. From the command line type *roslaunch nmmi_examples file.launch* to use them.



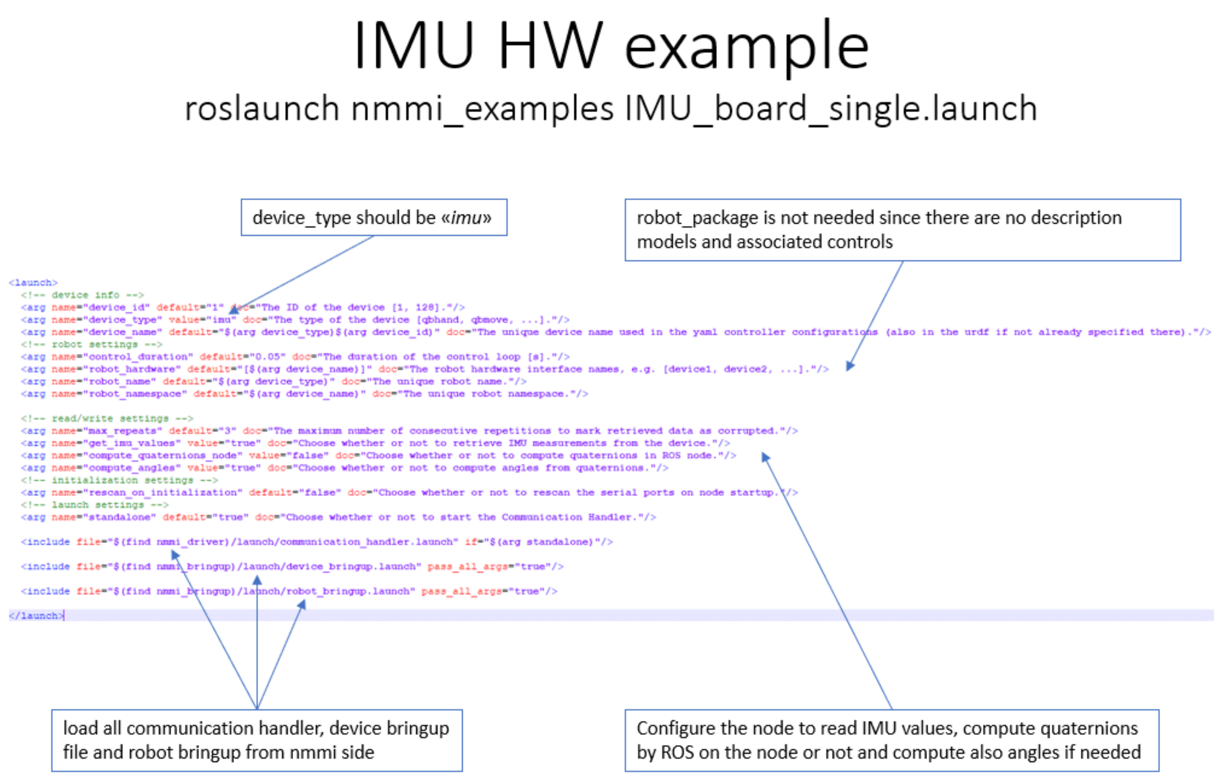
SoftHand Pro device example

roslaunch nmmi_examples SoftHandPro_control.launch

```
<launch>
  <!-- device info -->
  <arg name="device_id" default="1" doc="The ID of the device [1, 128]."/>
  <arg name="device_type" value="SoftHandPro" doc="The type of the device [qbhand, qbmove, ...]."/>
  <arg name="device_name" default="$(arg device_type)$(arg device_id)" doc="The unique device name used in the yaml controller configurations (also in the urdf if not already specified there)."/>
  <!-- robot settings -->
  <arg name="control_duration" default="0.01" doc="The duration of the control loop [s]."/>
  <arg name="robot_hardware" default="[$(arg device_name)]" doc="The robot hardware interface names, e.g. [device1, device2, ...]."/>
  <arg name="robot_name" default="$(arg device_type)" doc="The unique robot name."/>
  <arg name="robot_namespace" default="$(arg device_name)" doc="The unique robot namespace."/>
  <arg name="robot_package" default="softhandpro" doc="The base package name prefix for the robot configurations [urdf, rviz, ...]."/>
  <arg name="source_list" default="[control/joint_states]" doc="The joint_states source list for the joint_state_publisher."/>
  <!-- read/write settings -->
  <arg name="get_currents" default="false" doc="Choose whether or not to retrieve current measurements from the device."/>
  <arg name="get_positions" default="true" doc="Choose whether or not to retrieve position measurements from the device."/>
  <arg name="get_distinct_packages" default="false" doc="Choose whether or not to retrieve current and position measurements from the device in two distinct packages."/>
  <arg name="max_repeats" default="3" doc="The maximum number of consecutive repetitions to mark retrieved data as corrupted."/>
  <arg name="set_commands" default="true" doc="Choose whether or not to send command positions to the device."/>
  <arg name="set_commands_async" default="true" doc="Choose whether or not to send commands without waiting for ack."/>
  <!-- initialization settings -->
  <arg name="activate_on_initialization" default="true" doc="Choose whether or not to activate the motors on node startup."/>
  <arg name="rescan_on_initialization" default="false" doc="Choose whether or not to rescan the serial ports on node startup."/>
  <!-- launch settings -->
  <arg name="standalone" default="true" doc="Choose whether or not to start the Communication Handler."/>
  <arg name="use_controller_gui" default="true" doc="Choose whether or not to use the controller GUI."/>
  <arg name="use_rviz" default="true" doc="Choose whether or not to use rviz."/>
  <arg name="use_waypoints" default="false" doc="Choose whether or not to use the waypoint references."/>


  <include file="$(find nmmi_driver)/launch/communication_handler.launch" if="$(arg standalone)"/>

  <include file="$(find nmmi_bringup)/launch/device_bringup.launch" pass_all_args="true"/>

  <include file="$(find qb_device_bringup)/launch/robot_bringup.launch" pass_all_args="true"/>
</launch>
```

device_type should be «*SoftHandPro*»
robot_package should be «*softhandpro*» in order to load correct urdf models
load both nmmi communication handler and nmmi device bringup file
load robot bringup file from qb_device

Figure 36: SoftHand Pro device example launch file



IMU HW example

roslaunch nmmi_examples IMU_board_single.launch

device_type should be «*imu*»

robot_package is not needed since there are no description models and associated controls

```xml
<launch>
  <!-- device info -->
  <arg name="device_id" default="1" doc="The ID of the device [1, 128]."/>
  <arg name="device_type" value="imu" doc="The type of the device [qbhand, qbmove, ...]."/>
  <arg name="device_name" default="$(arg device_type)$(arg device_id)" doc="The unique device name used in the yaml controller configurations (also in the urdf if not already specified there)."/>
  <!-- robot settings -->
  <arg name="control_duration" default="0.05" doc="The duration of the control loop [s]."/>
  <arg name="robot_hardware" default="[$(arg device_name)]" doc="The robot hardware interface names, e.g. [device1, device2, ...]."/>
  <arg name="robot_name" default="$(arg device_type)" doc="The unique robot name."/>
  <arg name="robot_namespace" default="$(arg device_name)" doc="The unique robot namespace."/>

  <!-- read/write settings -->
  <arg name="max_repeats" default="3" doc="The maximum number of consecutive repetitions to mark retrieved data as corrupted."/>
  <arg name="get_imu_values" value="true" doc="Choose whether or not to retrieve IMU measurements from the device."/>
  <arg name="compute_quaternions_node" value="false" doc="Choose whether or not to compute quaternions in ROS node."/>
  <arg name="compute_angles" value="true" doc="Choose whether or not to compute angles from quaternions."/>
  <!-- initialization settings -->
  <arg name="rescan_on_initialization" default="false" doc="Choose whether or not to rescan the serial ports on node startup."/>
  <!-- launch settings -->
  <arg name="standalone" default="true" doc="Choose whether or not to start the Communication Handler."/>

  <include file="$(find nmmi_driver)/launch/communication_handler.launch" if="$(arg standalone)"/>

  <include file="$(find nmmi_bringup)/launch/device_bringup.launch" pass_all_args="true"/>

  <include file="$(find nmmi_bringup)/launch/robot_bringup.launch" pass_all_args="true"/>

</launch>
```

load all communication handler, device bringup file and robot bringup from nmmi side

Configure the node to read IMU values, compute quaternions by ROS on the node or not and compute also angles if needed

Figure 37: IMUs reading example launch file

- *IMU_board_single.launch*: configured for a board or device able to read IMU (see Figure 37)
- *2_IMU_boards_chain.launch*: configuration with 2 IMU boards with different IDs
- *SoftHandPro_control.launch*: configured for using with a SoftHand Pro NMMI device (see Figure 36)
- *generic_board_single.launch*: used together with generic FW to read raw ADC values and/or Encoders
- *SoftHandPro_IMU_chain.launch*: chain of a SoftHand Pro and a IMU board with different IDs
- *qbhand_IMU_chain.launch*: chain of a qbHand and a IMU board with different IDs
- *SoftHandPro_with_IMU_reading.launch*: configured to be used with a SoftHand Pro NMMI device with its own on board IMU reading active
- *SoftHandPro_with_generic_features.launch*: used together with generic FW but with device type parameter set to “SOFTHAND PRO”, so it’s a SoftHand Pro NMMI device but with generic FW features

Appendix A – Firmware detailed documentation

Document file is generated with Doxygen tool. The appendix is valid for both *SoftHand Pro* firmware and Generic firmware.

Appendix B – Software detailed documentation

Document file is generated with Doxygen tool. The appendix is divided into a first part with API documentation and a second part with Software tools documentation.

PSoC5 firmware

v. 1.0

Generated by Doxygen 1.8.13

Contents

Chapter 1

Firmware

This is the firmware of PSoC5 logic board.

Version

1.0

This is the firmware of PSoC5 logic board. Depending on the configuration, it can control up to two motors and read its encoders. Also can read and convert analog measurements connected to the PSoC microcontroller.

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 st_calib Struct Reference

Hand calibration structure.

```
#include <globals.h>
```

Data Fields

- uint8 **enabled**
- uint8 **direction**
- int16 **speed**
- int16 **repetitions**

4.1.1 Detailed Description

Hand calibration structure.

4.1.2 Field Documentation

4.1.2.1 direction

```
uint8 direction
```

Direction of motor winding.

4.1.2.2 enabled

```
uint8 enabled
```

Calibration enabling flag.

4.1.2.3 repetitions

```
int16 repetitions
```

Number of cycles of hand closing/opening.

4.1.2.4 speed

```
int16 speed
```

Speed of hand opening/closing.

The documentation for this struct was generated from the following file:

- **globals.h**

4.2 st_counters Struct Reference

Usage counters structure.

```
#include <globals.h>
```

Data Fields

- uint32 **emg_counter** [2]
- uint32 **position_hist** [10]
- uint32 **current_hist** [4]
- uint32 **rest_counter**
- uint32 **wire_disp**
- uint32 **total_time_on**
- uint32 **total_time_rest**

4.2.1 Detailed Description

Usage counters structure.

4.2.2 Field Documentation

4.2.2.1 current_hist

```
uint32 current_hist[4]
```

Current histogram - 4 zones.

4.2.2.2 emg_counter

```
uint32 emg_counter[2]
```

Counter for EMG activation - both channels.

4.2.2.3 position_hist

```
uint32 position_hist[10]
```

Positions histogram - 10 zones.

4.2.2.4 rest_counter

```
uint32 rest_counter
```

Counter for rest position occurrences.

4.2.2.5 total_time_on

```
uint32 total_time_on
```

Total time of system power (in seconds).

4.2.2.6 total_time_rest

```
uint32 total_time_rest
```

Total time of system while rest position is maintained.

4.2.2.7 wire_disp

```
uint32 wire_disp
```

Counter for total wire displacement measurement.

The documentation for this struct was generated from the following file:

- **globals.h**

4.3 st_data Struct Reference

Data sent/received structure.

```
#include <globals.h>
```

Data Fields

- uint8 **buffer** [128]
- int16 **length**
- int16 **ind**
- uint8 **ready**

4.3.1 Detailed Description

Data sent/received structure.

4.3.2 Field Documentation

4.3.2.1 buffer

```
uint8 buffer[128]
```

Data buffer [CMD | DATA | CHECKSUM].

4.3.2.2 ind

```
int16 ind
```

Data buffer index.

4.3.2.3 length

```
int16 length
```

Data buffer length.

4.3.2.4 ready

```
uint8 ready
```

Data buffer flag to see if the data is ready.

The documentation for this struct was generated from the following file:

- **globals.h**

4.4 st_device Struct Reference

Device related parameters structure.

```
#include <globals.h>
```

Data Fields

- uint8 **id**
- uint8 **hw_maint_date** [3]
- uint8 **stats_period_begin_date** [3]
- uint8 **right_left**
- uint8 **reset_counters**
- uint8 **use_2nd_motor_flag**
- uint8 **baud_rate**
- uint8 **user_id**
- uint8 **dev_type**
- uint8 **unused_bytes** [3]

4.4.1 Detailed Description

Device related parameters structure.

4.4.2 Field Documentation

4.4.2.1 baud_rate

```
uint8 baud_rate
```

Baud Rate setted.

4.4.2.2 dev_type

```
uint8 dev_type
```

Device type identificator.

4.4.2.3 hw_maint_date

```
uint8 hw_maint_date[3]
```

Date of last hardware maintenance.

4.4.2.4 id

```
uint8 id
```

Device id.

4.4.2.5 reset_counters

```
uint8 reset_counters
```

Reset counters flag.

4.4.2.6 right_left

```
uint8 right_left
```

Right/Left hand.

4.4.2.7 stats_period_begin_date

```
uint8 stats_period_begin_date[3]
```

Date of begin of usage statistics period.

4.4.2.8 unused_bytes

```
uint8 unused_bytes[3]
```

Unused bytes to fill row.

4.4.2.9 use_2nd_motor_flag

```
uint8 use_2nd_motor_flag
```

Use 2nd motor (2 powers).

4.4.2.10 user_id

```
uint8 user_id
```

User identificator (if usual user).

The documentation for this struct was generated from the following file:

- **globals.h**

4.5 st_eeprom Struct Reference

Memory variables.

```
#include <globals.h>
```

Collaboration diagram for st_eeprom:

Data Fields

- uint8 **flag**
- uint8 **unused_bytes** [15]
- struct **st_counters cnt**
- uint8 **unused_bytes1** [**EEPROM_BYTES_ROW** ∗4]
- struct **st_device dev**
- struct **st_motor motor** [**NUM_OF_MOTORS**]
- struct **st_encoder enc** [**N_ENCODER_LINE_MAX**]
- struct **st_emg emg**
- struct **st_imu imu**
- struct **st_expansion exp**
- struct **st_user user** [NUM_OF_USERS]
- struct **st_SH_spec SH**

4.5.1 Detailed Description

Memory variables.

4.5.2 Field Documentation

4.5.2.1 cnt

```
struct st_counters cnt
```

Statistics Counters.

4.5.2.2 dev

```
struct st_device dev
```

Device information.

4.5.2.3 emg

```
struct st_emg emg
```

EMG variables.

4.5.2.4 enc

```
struct st_encoder enc[ N_ENCODER_LINE_MAX]
```

Encoder variables (1 line).

4.5.2.5 exp

```
struct st_expansion exp
```

SD and ADC variables.

4.5.2.6 flag

```
uint8 flag
```

If checked the device has been configured.

4.5.2.7 imu

```
struct st_imu imu
```

IMU general variables.

4.5.2.8 motor

```
struct st_motor motor[ NUM_OF_MOTORS]
```

Motor variables.

4.5.2.9 SH

```
struct st_SH_spec SH
```

SoftHand specific variables.

4.5.2.10 unused_bytes

```
uint8 unused_bytes[15]
```

Leave bytes to align structures to memory rows.

4.5.2.11 unused_bytes1

```
uint8 unused_bytes1[ EEPROM_BYTES_ROW *4]
```

Leave for rows free for further uses.

4.5.2.12 user

```
struct st_user user[NUM_OF_USERS]
```

User variables.

The documentation for this struct was generated from the following file:

- **globals.h**

4.6 st_emg Struct Reference

EMG related parameters structure.

```
#include <globals.h>
```

Data Fields

- uint16 **emg_threshold** [**NUM_OF_INPUT_EMGS**]
- uint32 **emg_max_value** [**NUM_OF_INPUT_EMGS**]
- uint8 **emg_speed**
- uint8 **emg_calibration_flag**
- uint8 **switch_emg**
- uint8 **unused_bytes** [1]

4.6.1 Detailed Description

EMG related parameters structure.

4.6.2 Field Documentation

4.6.2.1 emg_calibration_flag

```
uint8 emg_calibration_flag
```

Enable emg calibration on startup.

4.6.2.2 emg_max_value

```
uint32 emg_max_value[ NUM_OF_INPUT_EMGS]
```

Maximum value for EMG.

4.6.2.3 emg_speed

```
uint8 emg_speed
```

Maximum closure speed when using EMG.

4.6.2.4 emg_threshold

```
uint16 emg_threshold[ NUM_OF_INPUT_EMGS]
```

Minimum value for activation of EMG control.

4.6.2.5 switch_emg

```
uint8 switch_emg
```

EMG opening/closure switch.

4.6.2.6 unused_bytes

```
uint8 unused_bytes[1]
```

Unused bytes to fill row.

The documentation for this struct was generated from the following file:

- **globals.h**

4.7 st_emg_meas Struct Reference

Data Fields

- int32 **emg** [**NUM_OF_INPUT_EMGS**]
- int32 **add_emg** [**NUM_OF_ADDITIONAL_EMGS**]

4.7.1 Field Documentation

4.7.1.1 add_emg

```
int32 add_emg[ NUM_OF_ADDITIONAL_EMGS]
```

Additional EMG sensors values.

4.7.1.2 emg

```
int32 emg[ NUM_OF_INPUT_EMGS]
```

EMG sensors values.

The documentation for this struct was generated from the following file:

- **globals.h**

4.8 st_encoder Struct Reference

Encoder related parameters structure.

```
#include <globals.h>
```

Data Fields

- uint8 **Enc_raw_read_conf** [**N_ENCODERS_PER_LINE_MAX**]
- uint8 **res** [**NUM_OF_SENSORS**]
- int32 **m_off** [**NUM_OF_SENSORS**]
- float32 **m_mult** [**NUM_OF_SENSORS**]
- uint8 **double_encoder_on_off**
- uint8 **Enc_idx_use_for_control** [**NUM_OF_SENSORS**]
- int8 **motor_handle_ratio**
- int8 **gears_params** [3]
- uint8 **unused_bytes** [8]

4.8.1 Detailed Description

Encoder related parameters structure.

4.8.2 Field Documentation

4.8.2.1 double_encoder_on_off

```
uint8 double_encoder_on_off
```

Double encoder ON/OFF.

4.8.2.2 Enc_idx_use_for_control

```
uint8 Enc_idx_use_for_control[ NUM_OF_SENSORS]
```

Indices of encoder used for motor control.

4.8.2.3 Enc_raw_read_conf

```
uint8 Enc_raw_read_conf[ N_ENCODERS_PER_LINE_MAX]
```

Encoder configuration flags for raw reading.

4.8.2.4 gears_params

```
int8 gears_params[3]
```

Number of teeth of first and second gear and related invariant.

4.8.2.5 m_mult

```
float32 m_mult[ NUM_OF_SENSORS]
```

Measurement multiplier.

4.8.2.6 m_off

```
int32 m_off[ NUM_OF_SENSORS]
```

Measurement offset.

4.8.2.7 motor_handle_ratio

```
int8 motor_handle_ratio
```

Discrete multiplier for handle device.

4.8.2.8 res

```
uint8 res[ NUM_OF_SENSORS]
```

Angle resolution.

4.8.2.9 unused_bytes

```
uint8 unused_bytes[8]
```

Unused bytes to fill row.

The documentation for this struct was generated from the following file:

- **globals.h**

4.9 st_expansion Struct Reference

Expansion board related parameters structure.

```
#include <globals.h>
```

Data Fields

- uint8 **curr_time** [6]
- uint8 **read_exp_port_flag**
- uint8 **read_ADC_sensors_port_flag**
- uint8 **ADC_conf** [NUM_OF_ADC_CHANNELS_MAX]
- uint8 **unused_bytes** [12]

4.9.1 Detailed Description

Expansion board related parameters structure.

4.9.2 Field Documentation

4.9.2.1 ADC_conf

```
uint8 ADC_conf[NUM_OF_ADC_CHANNELS_MAX]
```

ADC configuration flags.

4.9.2.2 curr_time

```
uint8 curr_time[6]
```

Current time from RTC (DD/MM/YY HH:MM:SS).

4.9.2.3 read_ADC_sensors_port_flag

```
uint8 read_ADC_sensors_port_flag
```

Enable ADC sensors Port.

4.9.2.4 read_exp_port_flag

```
uint8 read_exp_port_flag
```

Enable Expansion Port.

4.9.2.5 unused_bytes

```
uint8 unused_bytes[12]
```

Unused bytes to fill row.

The documentation for this struct was generated from the following file:

- **globals.h**

4.10 st_filter Struct Reference

Filter structure.

```
#include <globals.h>
```

Data Fields

- int32 **old_value**
- int32 **gain**

4.10.1 Detailed Description

Filter structure.

4.10.2 Field Documentation

4.10.2.1 gain

```
int32 gain
```

New value filter weight.

4.10.2.2 old_value

```
int32 old_value
```

Old variable value.

The documentation for this struct was generated from the following file:

- **globals.h**

4.11 st_imu Struct Reference

IMU related parameters structure.

```
#include <globals.h>
```

Data Fields

- uint8 **read_imu_flag**
- uint8 **SPI_read_delay**
- uint8 **IMU_conf** [N_IMU_MAX][NUM_OF_IMU_DATA]
- uint8 **unused_bytes** [5]

4.11.1 Detailed Description

IMU related parameters structure.

4.11.2 Field Documentation

4.11.2.1 IMU_conf

```
uint8 IMU_conf[N_IMU_MAX][NUM_OF_IMU_DATA]
```

IMUs configuration flags.

4.11.2.2 read_imu_flag

```
uint8 read_imu_flag
```

Enable IMU reading feature.

4.11.2.3 SPI_read_delay

```
uint8 SPI_read_delay
```

Delay on SPI reading.

4.11.2.4 unused_bytes

```
uint8 unused_bytes[5]
```

Unused bytes to fill row.

The documentation for this struct was generated from the following file:

- **globals.h**

4.12 st_imu_data Struct Reference

IMU data structure.

```
#include <globals.h>
```

Data Fields

- uint8 **flags**
- int16 **accel_value** [3]
- int16 **gyro_value** [3]
- int16 **mag_value** [3]
- float **quat_value** [4]
- int16 **temp_value**

4.12.1 Detailed Description

IMU data structure.

The documentation for this struct was generated from the following file:

- **globals.h**

4.13 st_meas Struct Reference

Measurements structure.

```
#include <globals.h>
```

Data Fields

- int32 **pos** [**NUM_OF_SENSORS**]
- int32 **curr**
- int32 **estim_curr**
- int8 **rot** [**NUM_OF_SENSORS**]
- int32 **vel** [**NUM_OF_SENSORS**]
- int32 **acc** [**NUM_OF_SENSORS**]

4.13.1 Detailed Description

Measurements structure.

4.13.2 Field Documentation

4.13.2.1 acc

```
int32 acc[ NUM_OF_SENSORS]
```

Encoder rotational acceleration.

4.13.2.2 curr

```
int32 curr
```

Motor current.

4.13.2.3 estim_curr

```
int32 estim_curr
```

Current estimation.

4.13.2.4 pos

```
int32 pos[ NUM_OF_SENSORS]
```

Encoder sensor position.

4.13.2.5 rot

```
int8 rot[ NUM_OF_SENSORS]
```

Encoder sensor rotations.

4.13.2.6 vel

```
int32 vel[ NUM_OF_SENSORS]
```

Encoder rotational velocity.

The documentation for this struct was generated from the following file:

- **globals.h**

4.14 st_motor Struct Reference

Motor related parameters structure.

```
#include <globals.h>
```

Data Fields

- int32 **k_p**
- int32 **k_i**
- int32 **k_d**
- int32 **k_p_c**
- int32 **k_i_c**
- int32 **k_d_c**
- int32 **k_p_dl**
- int32 **k_i_dl**
- int32 **k_d_dl**
- int32 **k_p_c_dl**
- int32 **k_i_c_dl**
- int32 **k_d_c_dl**
- uint8 **activ**
- uint8 **activate_pwm_rescaling**
- uint8 **motor_driver_type**
- uint8 **pos_lim_flag**
- int32 **pos_lim_inf**
- int32 **pos_lim_sup**
- int32 **max_step_neg**
- int32 **max_step_pos**
- float **curr_lookup** [**LOOKUP_DIM**]
- int16 **current_limit**
- uint8 **input_mode**
- uint8 **control_mode**
- uint8 **encoder_line**
- uint8 **pwm_rate_limiter**
- uint8 **not_revers_motor_flag**
- uint8 **unused_bytes** [13]

4.14.1 Detailed Description

Motor related parameters structure.

4.14.2 Field Documentation

4.14.2.1 activ

```
uint8 activ
```

Startup activation.

4.14.2.2 activate_pwm_rescaling

```
uint8 activate_pwm_rescaling
```

Activation of PWM rescaling for 12V motor.

4.14.2.3 control_mode

```
uint8 control_mode
```

Motor Control mode.

4.14.2.4 curr_lookup

```
float curr_lookup[ LOOKUP_DIM]
```

Table of values to get estimated curr.

4.14.2.5 current_limit

```
int16 current_limit
```

Limit for absorbed current.

4.14.2.6 encoder_line

```
uint8 encoder_line
```

Encoder line associated to the motor control.

4.14.2.7 input_mode

```
uint8 input_mode
```

Motor Input mode.

4.14.2.8 k_d

```
int32 k_d
```

Position controller derivative constant.

4.14.2.9 k_d_c

```
int32 k_d_c
```

Current controller derivative constant.

4.14.2.10 k_d_c_dl

```
int32 k_d_c_dl
```

Double loop current controller deriv. constant.

4.14.2.11 k_d_dl

```
int32 k_d_dl
```

Double loop position controller deriv. constant.

4.14.2.12 k_i

```
int32 k_i
```

Position controller integrative constant.

4.14.2.13 k_i_c

```
int32 k_i_c
```

Current controller integrative constant.

4.14.2.14 k_i_c_dl

```
int32 k_i_c_dl
```

Double loop current controller integr. constant.

4.14.2.15 k_i_dl

```
int32 k_i_dl
```

Double loop position controller integr. constant.

4.14.2.16 k_p

```
int32 k_p
```

Position controller proportional constant.

4.14.2.17 k_p_c

```
int32 k_p_c
```

Current controller proportional constant.

4.14.2.18 k_p_c_dl

```
int32 k_p_c_dl
```

Double loop current controller prop. constant.

4.14.2.19 k_p_dl

```
int32 k_p_dl
```

Double loop position controller prop. constant.

4.14.2.20 max_step_neg

```
int32 max_step_neg
```

Maximum number of steps per cycle for negative positions.

4.14.2.21 max_step_pos

```
int32 max_step_pos
```

Maximum number of steps per cycle for positive positions.

4.14.2.22 motor_driver_type

```
uint8 motor_driver_type
```

Specify motor type.

4.14.2.23 not_revers_motor_flag

```
uint8 not_revers_motor_flag
```

Flag to know if the motor is reversible or not.

4.14.2.24 pos_lim_flag

```
uint8 pos_lim_flag
```

Position limit active/inactive.

4.14.2.25 pos_lim_inf

```
int32 pos_lim_inf
```

Inferior position limit for motor.

4.14.2.26 pos_lim_sup

```
int32 pos_lim_sup
```

Superior position limit for motor[0].

4.14.2.27 pwm_rate_limiter

```
uint8 pwm_rate_limiter
```

PWM rate limiter max asscoaited to the motor.

4.14.2.28 unused_bytes

```
uint8 unused_bytes[13]
```

Unused bytes to fill row.

The documentation for this struct was generated from the following file:

- **globals.h**

4.15 st_ref Struct Reference

Motor Reference structure.

```
#include <globals.h>
```

Data Fields

- int32 **pos**
- int32 **curr**
- int32 **pwm**
- uint8 **onoff**

4.15.1 Detailed Description

Motor Reference structure.

4.15.2 Field Documentation

4.15.2.1 curr

```
int32 curr
```

Motor current reference.

4.15.2.2 onoff

```
uint8 onoff
```

Motor drivers enable.

4.15.2.3 pos

```
int32 pos
```

Motor position reference.

4.15.2.4 pwm

```
int32 pwm
```

Motor direct pwm control.

The documentation for this struct was generated from the following file:

- **globals.h**

4.16 st_SH_spec Struct Reference

SoftHand specific related prameters structure.

```
#include <globals.h>
```

Data Fields

- int32 **rest_pos**
- int32 **rest_delay**
- int32 **rest_vel**
- uint8 **rest_position_flag**
- uint8 **unused_bytes** [3]

4.16.1 Detailed Description

SoftHand specific related prameters structure.

4.16.2 Field Documentation

4.16.2.1 rest_delay

```
int32 rest_delay
```

Hand rest position delay while in EMG mode.

4.16.2.2 rest_pos

```
int32 rest_pos
```

Hand rest position while in EMG mode.

4.16.2.3 rest_position_flag

```
uint8 rest_position_flag
```

Enable rest position feature.

4.16.2.4 rest_vel

```
int32 rest_vel
```

Hand velocity closure for rest position reaching.

4.16.2.5 unused_bytes

```
uint8 unused_bytes[3]
```

Unused bytes to fill row.

The documentation for this struct was generated from the following file:

- **globals.h**

4.17 st_user Struct Reference

User related parameters structure.

```
#include <globals.h>
```

Collaboration diagram for st_user:

Data Fields

- char **user_code_string** [8]
- struct **st_emg** **user_emg**
- uint8 **unused_bytes** [8]

4.17.1 Detailed Description

User related parameters structure.

4.17.2 Field Documentation

4.17.2.1 unused_bytes

```
uint8 unused_bytes[8]
```

Unused bytes to fill row.

4.17.2.2 user_code_string

```
char user_code_string[8]
```

User code string.

4.17.2.3 user_emg

```
struct  st_emg user_emg
```

st_emg (p. 15) structure to store user emg values.

The documentation for this struct was generated from the following file:

- **globals.h**

Chapter 5

File Documentation

5.1 command_processing.c File Reference

Command processing functions.

```
#include "command_processing.h"
```
Include dependency graph for command_processing.c:

5.2 command_processing.h File Reference

Received commands processing functions.

```
#include "globals.h"
#include "IMU_functions.h"
#include "Encoder_functions.h"
#include "SD_RTC_functions.h"
#include "interruptions.h"
#include "utils.h"
#include "commands.h"
#include <stdio.h>
```
Include dependency graph for command_processing.h: This graph shows which files directly or indirectly include this file:

Functions

Firmware information functions

- void **prepare_generic_info** (char ∗info_string)
- void **prepare_counter_info** (char ∗info_string)
- void **prepare_SD_info** (char ∗info_string)
- void **prepare_SD_param_info** (char ∗info_string)
- void **prepare_SD_legend** (char ∗info_string)
- void **IMU_reading_info** (char ∗info_string)
- void **infoSend** ()
- void **infoGet** (uint16 info_type)

Command receiving and sending functions

- void **commProcess** ()
- void **commWrite_old_id** (uint8 ∗packet_data, uint16 packet_lenght, uint8 old_id)
- void **commWrite** (uint8 ∗packet_data, uint16 packet_lenght)
- void **commWrite_to_cuff** (uint8 ∗packet_data, uint16 packet_lenght)

Memory management functions

- void **manage_param_list** (uint16 index)
- void **get_param_list** (uint8 ∗VAR_P[**NUM_OF_PARAMS**], uint8 TYPES[**NUM_OF_PARAMS**], uint8 NUM_ITEMS[**NUM_OF_PARAMS**], uint8 NUM_STRUCT[**NUM_OF_PARAMS**], uint8 ∗NUM_MENU, const char ∗PARAMS_STR[**NUM_OF_PARAMS**], uint8 CUSTOM_PARAM_SET[**NUM_OF_PARAMS**], const char ∗MENU_STR[**NUM_OF_PARAMS_MENU**])
- void **set_custom_param** (uint16 index)
- void **get_IMU_param_list** (uint16 index)
- void **setZeros** ()
- uint8 **memStore** (int displacement)
- void **memRecall** ()
- uint8 **memRestore** ()
- uint8 **memInit** ()
- void **memInit_SoftHandPro** ()

Utility functions

- uint8 **LCRChecksum** (uint8 ∗data_array, uint8 data_length)
- void **sendAcknowledgment** (uint8 value)

Command processing functions

- void **cmd_activate** ()
- void **cmd_set_inputs** ()
- void **cmd_get_measurements** ()
- void **cmd_get_curr_and_meas** ()
- void **cmd_get_velocities** ()
- void **cmd_get_accelerations** ()
- void **cmd_get_currents** ()
- void **cmd_get_currents_for_cuff** ()
- void **cmd_get_emg** ()
- void **cmd_get_activate** ()
- void **cmd_set_baudrate** ()
- void **cmd_get_inputs** ()
- void **cmd_store_params** ()
- void **cmd_ping** ()
- void **cmd_get_imu_readings** ()
- void **cmd_get_encoder_map** ()
- void **cmd_get_encoder_raw** ()
- void **cmd_get_ADC_map** ()
- void **cmd_get_ADC_raw** ()
- void **cmd_get_SD_files** ()

5.2.1 Detailed Description

Received commands processing functions.

Date

October 01, 2017

Author

Centro "E.Piaggio"

Copyright

This file contains all the definitions of the functions used to process the commands sent from the user interfaces (simulink, command line, GUI)

5.2.2 Function Documentation

5.2.2.1 cmd_activate()

```
void cmd_activate ( )
```

This function activates the board

5.2.2.2 cmd_get_accelerations()

```
void cmd_get_accelerations ( )
```

This function gets the encoders accelerations and puts them in the package to be sent.

5.2.2.3 cmd_get_activate()

```
void cmd_get_activate ( )
```

This function gets the board activation status and puts it in the package to be sent.

5.2.2.4 cmd_get_ADC_map()

```
void cmd_get_ADC_map ( )
```

This function gets ADC map

5.2.2.5 cmd_get_ADC_raw()

```
void cmd_get_ADC_raw ( )
```

This function gets Additional emg raw values

5.2.2.6 cmd_get_curr_and_meas()

```
void cmd_get_curr_and_meas ( )
```

This function gets the currents and encoders measurements and puts them in the package to be sent.

5.2.2.7 cmd_get_currents()

```
void cmd_get_currents ( )
```

This function gets the motor current and puts it in the package to be sent to the user.

5.2.2.8 cmd_get_currents_for_cuff()

```
void cmd_get_currents_for_cuff ( )
```

This function gets the motor current and puts it in the package to be sent to the Cuff device, using the **comm←Write_to_cuff** (p. 38) function.

5.2.2.9 cmd_get_emg()

```
void cmd_get_emg ( )
```

This function gets the electromyographic sensors measurements and puts them in the package to be sent.

5.2.2.10 cmd_get_encoder_map()

```
void cmd_get_encoder_map ( )
```

This function gets Encoder map

5.2.2.11 cmd_get_encoder_raw()

```
void cmd_get_encoder_raw ( )
```

This function gets Encoder raw values

5.2.2.12 cmd_get_imu_readings()

```
void cmd_get_imu_readings ( )
```

This function gets IMU readings

5.2.2.13 cmd_get_inputs()

```
void cmd_get_inputs ( )
```

This function gets the current motor reference inputs and puts them in the package to be sent.

5.2.2.14 cmd_get_measurements()

```
void cmd_get_measurements ( )
```

This function gets the encoders measurements and puts them in the package to be sent.

Bunch of functions used on request from UART communication

5.2.2.15 cmd_get_SD_files()

```
void cmd_get_SD_files ( )
```

This function gets both SD parameters and data files

5.2.2.16 cmd_get_velocities()

```
void cmd_get_velocities ( )
```

This function gets the encoders velocities and puts them in the package to be sent.

5.2.2.17 cmd_ping()

```
void cmd_ping ( )
```

This function is used to ping the device and see if is connected.

5.2.2.18 cmd_set_baudrate()

```
void cmd_set_baudrate ( )
```

This function sets the desired communication baudrate. It is possible to select a value equal to 460800 or 2000000.

5.2.2.19 cmd_set_inputs()

```
void cmd_set_inputs ( )
```

This function gets the inputs from the received package and sets them as motor reference.

5.2.2.20 cmd_store_params()

```
void cmd_store_params ( )
```

This function stores the parameters to the EEPROM memory

5.2.2.21 commProcess()

```
void commProcess ( )
```

This function unpacks the received package, depending on the command received.

5.2.2.22 commWrite()

```
void commWrite (
             uint8 * packet_data,
             uint16 packet_lenght )
```

This function writes on the serial port the package that needs to be sent to the user.

Parameters

packet_data	The array of data that must be written.
packet_lenght	The lenght of the data array.

5.2.2.23 commWrite_old_id()

```
void commWrite_old_id (
             uint8 * packet_data,
             uint16 packet_lenght,
             uint8 old_id )
```

This function writes on the serial port the package that needs to be sent to the user. Is used only when a new is set, to communicate back to the APIs that the new ID setting went fine or there was an error.

Parameters

packet_data	The array of data that must be written.
packet_lenght	The lenght of the data array.
old_id	The previous id of the board, before setting a new one.

5.2.2.24 commWrite_to_cuff()

```
void commWrite_to_cuff (
            uint8 * packet_data,
            uint16 packet_lenght )
```

This function writes on the serial port the package that needs to be sent to the Cuff device. It is used only when a specific device is connected to the hand. The Hand must have ID equal to the one of the Cuff plus one.

Parameters

packet_data	The array of data that must be written.
packet_lenght	The lenght of the data array.

5.2.2.25 get_IMU_param_list()

```
void get_IMU_param_list (
            uint16 index )
```

This function, depending on the **Firmware** (p. 1) received, gets the list of parameters with their values.

5.2.2.26 get_param_list()

```
void get_param_list (
            uint8 * VAR_P[NUM_OF_PARAMS],
            uint8 TYPES[NUM_OF_PARAMS],
            uint8 NUM_ITEMS[NUM_OF_PARAMS],
            uint8 NUM_STRUCT[NUM_OF_PARAMS],
            uint8 * NUM_MENU,
            const char * PARAMS_STR[NUM_OF_PARAMS],
            uint8 CUSTOM_PARAM_SET[NUM_OF_PARAMS],
            const char * MENU_STR[NUM_OF_PARAMS_MENU] )
```

This function, depending on the **Firmware** (p. 1) received, gets the list of parameters with their values.

5.2.2.27 IMU_reading_info()

```
void IMU_reading_info (
            char * info_string )
```

This function is used to prepare an information string about the IMU sensors last reading.

Parameters

info_string	An array of chars containing the requested information.

5.2.2.28 infoGet()

```
void infoGet (
            uint16 info_type )
```

This function sends the firmware information prepared with prepare_general_info or **prepare_counter_info** (p. 42) through the serial port to the user interface. Is used when the ID is specified.

Parameters

info_type	The type of the information needed.

5.2.2.29 infoSend()

```
void infoSend ( )
```

This function sends the firmware information prepared with infoPrepare through the serial port to the user interface. Is used when no ID is specified.

5.2.2.30 LCRChecksum()

```
uint8 LCRChecksum (
            uint8 * data_array,
            uint8 data_length )
```

This function calculates a checksum of the array to see if the received data is consistent.

Parameters

data_array	The array of data that must be checked.
data_lenght	Lenght of the data array that must be checked.

Returns

The calculated checksum for the relative data_array.

5.2.2.31 manage_param_list()

```
void manage_param_list (
            uint16 index )
```

This function, depending on the **Firmware** (p. 1) received, gets the list of parameters with their values and sends them to user or sets a parameter from all the parameters of the device.

Parameters

index	The index of the parameters to be setted. If 0 gets full parameters list.

5.2.2.32 memInit()

```
uint8 memInit ( )
```

This functions initializes the memory. It is used also to restore the the parameters to their default values.

Returns

A true value if the memory is correctly initialized, false otherwise.

5.2.2.33 memInit_SoftHandPro()

```
void memInit_SoftHandPro ( )
```

This functions initializes the memory. It is used also to restore the the parameters to their default values. Specific for SoftHand firmware

5.2.2.34 memRecall()

```
void memRecall ( )
```

This function loads user's settings from the EEPROM.

5.2.2.35 memRestore()

```
uint8 memRestore ( )
```

This function loads default settings from the EEPROM.

Returns

A true value if the memory is correctly restored, false otherwise.

5.2.2.36 memStore()

```
uint8 memStore (
             int displacement )
```

This function stores the setted parameters to the internal EEPROM memory. It is usually called, by the user, after a parameter is set.

Parameters

displacement	The address where the parameters will be written.

Returns

A true value if the memory is correctly stored, false otherwise.

5.2.2.37 prepare_counter_info()

```
void prepare_counter_info (
            char * info_string )
```

This function is used to prepare an information string about the cycles counter of the hand.

Parameters

info_string	An array of chars containing the requested information.

5.2.2.38 prepare_generic_info()

```
void prepare_generic_info (
            char * info_string )
```

This function is used to prepare a generic information string on the device parameters and measurements.

Parameters

info_string	An array of chars containing the requested information.

5.2.2.39 prepare_SD_info()

```
void prepare_SD_info (
            char * info_string )
```

This function is used to prepare an information string to be on a SD card

Parameters

info_string	An array of chars containing the requested information.

5.2.2.40 prepare_SD_legend()

```
void prepare_SD_legend (
            char * info_string )
```

This function is used to prepare an information string to be on a SD card

Parameters

info_string	An array of chars containing the requested information.

5.2.2.41 prepare_SD_param_info()

```
void prepare_SD_param_info (
            char * info_string )
```

This function is used to prepare an information string to be on a SD card

Parameters

info_string	An array of chars containing the requested information.

5.2.2.42 sendAcknowledgment()

```
void sendAcknowledgment (
            uint8 value )
```

This functions sends an acknowledgment to see if a command has been executed properly or not.

Parameters

value	An ACK_OK(1) or ACK_ERROR(0) value.

5.2.2.43 set_custom_param()

```
void set_custom_param (
            uint16 index )
```

This function, depending on the **Firmware** (p. 1) received, sets the specific parameters with their values and sends them to user or sets a parameter from all the parameters of the device.

Parameters

index	The index of the parameters to be setted.

5.2.2.44 setZeros()

```
void setZeros ( )
```

This function sets the encoders zero position.

5.3 commands.h File Reference

Definitions for SoftHand commands, parameters and packages.

This graph shows which files directly or indirectly include this file:

Macros

SoftHand Information Strings

- #define **INFO_ALL** 0
  Generic device information.
- #define **CYCLES_INFO** 1
  Cycles counter information.
- #define **GET_SD_PARAM** 2
  Read Firmware Parameters from SD file.
- #define **GET_SD_DATA** 3
  Read Usage Data from SD file.

SoftHand Commands

- #define **PARAM_BYTE_SLOT** 50
  Number of bytes reserved to a param information.
- #define **PARAM_MENU_SLOT** 150
  Number of bytes reserved to a param menu.
- enum **SH_command** {
  CMD_PING = 0, **CMD_SET_ZEROS** = 1, **CMD_STORE_PARAMS** = 3, **CMD_STORE_DEFAULT_PAR↩AMS** = 4,
  CMD_RESTORE_PARAMS = 5, **CMD_GET_INFO** = 6, **CMD_BOOTLOADER** = 9, **CMD_INIT_MEM** = 10,
  CMD_GET_PARAM_LIST = 12, **CMD_HAND_CALIBRATE** = 13, **CMD_ACTIVATE** = 128, **CMD_GET↩_ACTIVATE** = 129,
  CMD_SET_INPUTS = 130, **CMD_GET_INPUTS** = 131, **CMD_GET_MEASUREMENTS** = 132, **CMD_G↩ET_CURRENTS** = 133,
  CMD_GET_CURR_AND_MEAS = 134, **CMD_GET_EMG** = 136, **CMD_GET_VELOCITIES** = 137, **CMD↩_GET_ACCEL** = 139,
  CMD_GET_CURR_DIFF = 140, **CMD_SET_CUFF_INPUTS** = 142, **CMD_SET_BAUDRATE** = 144, **CM↩D_GET_IMU_READINGS** = 161,
  CMD_GET_IMU_PARAM = 162, **CMD_GET_ENCODER_CONF** = 163, **CMD_GET_ENCODER_RAW** = 164, **CMD_GET_ADC_CONF** = 165,
  CMD_GET_ADC_RAW = 166 }

- enum **SH_resolution** {
  RESOLUTION_360 = 0, **RESOLUTION_720** = 1, **RESOLUTION_1440** = 2, **RESOLUTION_2880** = 3,
  RESOLUTION_5760 = 4, **RESOLUTION_11520** = 5, **RESOLUTION_23040** = 6, **RESOLUTION_46080** = 7,
  RESOLUTION_92160 = 8 }
- enum **SH_input_mode** {
  INPUT_MODE_EXTERNAL = 0, **INPUT_MODE_ENCODER3** = 1, **INPUT_MODE_EMG_PROPORTION**←
  AL = 2, **INPUT_MODE_EMG_INTEGRAL** = 3,
  INPUT_MODE_EMG_FCFS = 4, **INPUT_MODE_EMG_FCFS_ADV** = 5 }
- enum **SH_control_mode** { **CONTROL_ANGLE** = 0, **CONTROL_PWM** = 1, **CONTROL_CURRENT** = 2,
  CURR_AND_POS_CONTROL = 3 }
- enum **motor_supply_type** { **MAXON_24V** = 0, **MAXON_12V** = 1 }
- enum **acknowledgment_values** { **ACK_ERROR** = 0, **ACK_OK** = 1 }
- enum **data_types** {
  TYPE_FLAG = 0, **TYPE_INT8** = 1, **TYPE_UINT8** = 2, **TYPE_INT16** = 3,
  TYPE_UINT16 = 4, **TYPE_INT32** = 5, **TYPE_UINT32** = 6, **TYPE_FLOAT** = 7,
  TYPE_DOUBLE = 8, **TYPE_STRING** = 9 }

5.3.1 Detailed Description

Definitions for SoftHand commands, parameters and packages.

Date

October 01, 2017

Author

Centro "E.Piaggio"

Copyright

(C) 2012-2016 qbrobotics. All rights reserved.
(C) 2017-2019 Centro "E.Piaggio". All rights reserved.

This file is included in the firmware, in its libraries and applications. It contains all definitions that are necessary for the contruction of communication packages.

It includes definitions for all of the device commands, parameters and also the size of answer packages.

5.3.2 Enumeration Type Documentation

5.3.2.1 SH_command

`enum SH_command`

Enumerator

CMD_PING	Asks for a ping message.
CMD_SET_ZEROS	Command for setting the encoders zero position.
CMD_STORE_PARAMS	Stores all parameters in memory and loads them.
CMD_STORE_DEFAULT_PARAMS	Store current parameters as factory parameters.
CMD_RESTORE_PARAMS	Restore default factory parameters.
CMD_GET_INFO	Asks for a string of information about.
CMD_BOOTLOADER	Sets the bootloader modality to update the firmware.
CMD_INIT_MEM	Initialize the memory with the defalut values.
CMD_GET_PARAM_LIST	Command to get the parameters list or to set a defined value chosen by the use.
CMD_HAND_CALIBRATE	Starts a series of opening and closures of the SoftHand.
CMD_ACTIVATE	Command for activating/deactivating the device.
CMD_GET_ACTIVATE	Command for getting device activation state.
CMD_SET_INPUTS	Command for setting reference inputs.
CMD_GET_INPUTS	Command for getting reference inputs.
CMD_GET_MEASUREMENTS	Command for asking device's position measurements.
CMD_GET_CURRENTS	Command for asking device's current measurements.
CMD_GET_CURR_AND_MEAS	Command for asking device's measurements and currents.
CMD_GET_EMG	Command for asking device's emg sensors measurements.
CMD_GET_VELOCITIES	Command for asking device's velocity measurements.
CMD_GET_ACCEL	Command for asking device's acceleration measurements.
CMD_GET_CURR_DIFF	Command for asking device's current difference between a measured one and an estimated one.
CMD_SET_CUFF_INPUTS	Command used to set Cuff device inputs .
CMD_SET_BAUDRATE	Command for setting baudrate of communication.

5.3.2.2 SH_control_mode

`enum SH_control_mode`

Enumerator

CONTROL_ANGLE	Classic position control.
CONTROL_PWM	Direct PWM value.
CONTROL_CURRENT	Current control.
CURR_AND_POS_CONTROL	Current and position control.

5.3.2.3 SH_input_mode

`enum SH_input_mode`

Enumerator

INPUT_MODE_EXTERNAL	References through external commands (default).
INPUT_MODE_ENCODER3	Encoder 3 drives all inputs.
INPUT_MODE_EMG_PROPORTIONAL	Use EMG measure to proportionally. drive the position of the motor.
INPUT_MODE_EMG_INTEGRAL	Use 2 EMG signals to drive motor position.
INPUT_MODE_EMG_FCFS	Use 2 EMG. First reaching threshold. wins and its value defines hand closure.
INPUT_MODE_EMG_FCFS_ADV	Use 2 EMG. First reaching threshold. wins and its value defines hand closure. Wait for both EMG to lower under threshold.

5.4 Encoder_functions.c File Reference

Implementation of SPI module functions.

```
#include "Encoder_functions.h"
```
Include dependency graph for Encoder_functions.c:

Functions

- void **Change_CS_EncoderLine** (int n)
- void **EncoderReset** ()
- void **InitEncoderGeneral** ()
- void **InitEncoderLine** (uint8 n)
- void **ReadEncoderLine** (int n_encoders, int n_line)

5.4.1 Detailed Description

Implementation of SPI module functions.

Date

February 13, 2019

Author

Centro "E.Piaggio"

Copyright

(C) 2012-2016 qbrobotics. All rights reserved.
(C) 2017-2019 Centro "E.Piaggio". All rights reserved.

5.5 Encoder_functions.h File Reference

Definition of Encoder module functions.

```
#include <project.h>
#include "globals.h"
#include <stdlib.h>
#include <stdio.h>
```

Include dependency graph for Encoder_functions.h: This graph shows which files directly or indirectly include this file:

Functions

- void **EncoderReset** ()
- void **InitEncoderLine** (uint8 n)
- void **InitEncoderGeneral** ()
- void **ReadEncoderLine** (int n_encoders, int n_line)
- void **Change_CS_EncoderLine** (int n)

5.5.1 Detailed Description

Definition of Encoder module functions.

Date

February 13, 2019

Author

Centro "E.Piaggio"

Copyright

(C) 2012-2016 qbrobotics. All rights reserved.
(C) 2017-2019 Centro "E.Piaggio". All rights reserved.

5.6 FIRMWARE_CONFIGURATION.h File Reference

Definitions for SoftHand and Other Devices commands, parameters and packages.

This graph shows which files directly or indirectly include this file:

Macros

- #define **VERSION** "Generic firmware v. 1.7 (PSoC5)"
- #define **NUM_OF_DEV_PARAMS NUM_OF_PARAMS**
- #define **NUM_OF_DEV_PARAM_MENUS NUM_OF_PARAMS_MENU**
- #define **NUM_DEV_IMU** N_IMU_MAX

5.6.1 Detailed Description

Definitions for SoftHand and Other Devices commands, parameters and packages.

Date

January 30, 2019

Author

Centro "E.Piaggio"

Copyright

(C) 2019 Centro "E.Piaggio". All rights reserved.

This file is included in the firmware, in its libraries and applications. It contains all definitions that are necessary to discriminate the right firmware.

5.7 globals.c File Reference

Global variables.

```
#include "globals.h"
```
Include dependency graph for globals.c:

Variables

- struct **st_ref g_ref** [**NUM_OF_MOTORS**]
- struct **st_ref g_refNew** [**NUM_OF_MOTORS**]
- struct **st_ref g_refOld** [**NUM_OF_MOTORS**]
- struct **st_meas g_meas** [**N_ENCODER_LINE_MAX**]
- struct **st_meas g_measOld** [**N_ENCODER_LINE_MAX**]
- struct **st_emg_meas** g_emg_meas **g_emg_measOld**
- struct **st_data g_rx**
- struct **st_eeprom** g_mem **c_mem**
- struct **st_calib calib**
- struct **st_filter filt_v** [**NUM_OF_MOTORS**]
- struct **st_filter filt_curr_diff** [**NUM_OF_MOTORS**]
- struct **st_filter filt_i** [**NUM_OF_MOTORS**]
- struct **st_filter filt_vel** [**NUM_OF_SENSORS**]
- struct **st_filter filt_emg** [**NUM_OF_INPUT_EMGS**+ **NUM_OF_ADDITIONAL_EMGS**]
- uint16 **timer_value**
- uint16 **timer_value0**
- float **cycle_time**
- int32 **dev_tension** [**NUM_OF_MOTORS**]
- uint8 **dev_pwm_limit** [**NUM_OF_MOTORS**]
- uint8 **dev_pwm_sat** [**NUM_OF_MOTORS**] = {100,100}
- int32 **dev_tension_f** [**NUM_OF_MOTORS**]
- int32 **pow_tension** [**NUM_OF_MOTORS**]

- **counter_status** CYDATA **cycles_status** = **NONE**
- **emg_status** CYDATA **emg_1_status** = **RESET**
- **emg_status** CYDATA **emg_2_status** = **RESET**
- CYBIT **reset_last_value_flag**
- CYBIT **tension_valid**
- CYBIT **interrupt_flag** = FALSE
- CYBIT **cycles_interrupt_flag** = FALSE
- uint8 **maintenance_flag** = FALSE
- CYBIT **can_write** = TRUE
- uint8 **rest_enabled**
- uint8 **forced_open**
- uint8 **battery_low_SoC** = FALSE
- uint8 **change_ext_ref_flag** = FALSE
- CYBIT **reset_PSoC_flag** = FALSE
- int16 **ADC_buf** [NUM_OF_ADC_CHANNELS_MAX]
- uint8 **NUM_OF_ANALOG_INPUTS** = 4
- int8 **pwm_sign**
- uint32 **data_encoder_raw** [**N_ENCODERS_PER_LINE_MAX**]
- uint8 **N_Encoder_Line_Connected** [**N_ENCODER_LINE_MAX**]
- uint16 **Encoder_Value** [**N_ENCODER_LINE_MAX**][**N_ENCODERS_PER_LINE_MAX**]
- uint8 **Encoder_Check** [**N_ENCODER_LINE_MAX**][**N_ENCODERS_PER_LINE_MAX**]
- int32 **rest_pos_curr_ref**
- FS_FILE ∗ **pFile**
- char **sdFile** [100] = ""
- char **sdParam** [100] = ""
- uint8 **N_IMU_Connected**
- uint8 **IMU_connected** [N_IMU_MAX]
- int **imus_data_size**
- int **single_imu_size** [N_IMU_MAX]
- struct **st_imu_data** **g_imu** [N_IMU_MAX]
- struct **st_imu_data** **g_imuNew** [N_IMU_MAX]
- uint8 **Accel** [N_IMU_MAX][6]
- uint8 **Gyro** [N_IMU_MAX][6]
- uint8 **Mag** [N_IMU_MAX][6]
- uint8 **MagCal** [N_IMU_MAX][3]
- uint8 **Temp** [N_IMU_MAX][2]
- float **Quat** [N_IMU_MAX][4]

5.7.1 Detailed Description

Global variables.

Date

October 01, 2017

Author

Centro "E.Piaggio"

Copyright

(C) 2012-2016 qbrobotics. All rights reserved.
(C) 2017-2019 Centro "E.Piaggio". All rights reserved.

5.7.2 Variable Documentation

5.7.2.1 battery_low_SoC

```
uint8 battery_low_SoC = FALSE
```

Battery low State of Charge flag (re-open terminal device when active).

5.7.2.2 c_mem

```
struct  st_eeprom g_mem c_mem
```

Memory parameters.

5.7.2.3 calib

```
struct  st_calib calib
```

Calibration variables.

5.7.2.4 can_write

```
CYBIT can_write = TRUE
```

Write to EEPROM flag.

5.7.2.5 change_ext_ref_flag

```
uint8 change_ext_ref_flag = FALSE
```

This flag is set when an external reference command is received.

5.7.2.6 cycle_time

```
float cycle_time
```

Variable used to calculate how much time a cycle takes.

5.7.2.7 cycles_interrupt_flag

```
CYBIT cycles_interrupt_flag = FALSE
```

Cycles timer interrupt flag enabler.

5.7.2.8 cycles_status

```
counter_status CYDATA cycles_status = NONE
```

Cycles counter state machine status.

5.7.2.9 dev_pwm_limit

```
uint8 dev_pwm_limit[ NUM_OF_MOTORS]
```

Device pwm limit. It may change during execution.

5.7.2.10 dev_pwm_sat

```
uint8 dev_pwm_sat[ NUM_OF_MOTORS] = {100,100}
```

Device pwm saturation. By default the saturation value must not exceed 100.

5.7.2.11 dev_tension

```
int32 dev_tension[ NUM_OF_MOTORS]
```

Power supply tension.

5.7.2.12 dev_tension_f

```
int32 dev_tension_f[ NUM_OF_MOTORS]
```

Filtered power supply tension.

5.7.2.13 emg_1_status

```
emg_status CYDATA emg_1_status = RESET
```

First EMG sensor status.

5.7.2.14 emg_2_status

```
emg_status CYDATA emg_2_status = RESET
```

Second EMG sensor status.

5.7.2.15 filt_emg

```
struct st_filter filt_emg[ NUM_OF_INPUT_EMGS+ NUM_OF_ADDITIONAL_EMGS]
```

EMG filter variables.

5.7.2.16 filt_i

```
struct st_filter filt_i[ NUM_OF_MOTORS]
```

Voltage and current filter variables.

5.7.2.17 filt_vel

```
struct st_filter filt_vel[ NUM_OF_SENSORS]
```

Velocity filter variables.

5.7.2.18 forced_open

```
uint8 forced_open
```

Forced open flag (used in position with rest position control).

5.7.2.19 g_emg_measOld

```
struct st_emg_meas g_emg_meas g_emg_measOld
```

EMG Measurements.

5.7.2.20 g_measOld

```
struct st_meas g_measOld[ N_ENCODER_LINE_MAX]
```

Measurements.

5.7.2.21 g_refOld

```
struct st_ref g_refOld[ NUM_OF_MOTORS]
```

Reference variables.

5.7.2.22 g_rx

```
struct st_data g_rx
```

Incoming/Outcoming data.

5.7.2.23 interrupt_flag

```
CYBIT interrupt_flag = FALSE
```

Interrupt flag enabler.

5.7.2.24 maintenance_flag

```
uint8 maintenance_flag = FALSE
```

Maintenance flag.

5.7.2.25 NUM_OF_ANALOG_INPUTS

```
uint8 NUM_OF_ANALOG_INPUTS = 4
```

ADC measurements buffer.

5.7.2.26 pow_tension

```
int32 pow_tension[ NUM_OF_MOTORS]
```

Computed power supply tension.

5.7.2.27 pwm_sign

```
int8 pwm_sign
```

ADC currently configured channels. Sign of pwm driven. Used to obtain current sign.

5.7.2.28 reset_last_value_flag

```
CYBIT reset_last_value_flag
```

This flag is set when the encoders last values must be resetted.

5.7.2.29 reset_PSoC_flag

```
CYBIT reset_PSoC_flag = FALSE
```

This flag is set when a board fw reset is necessary.

5.7.2.30 rest_enabled

```
uint8 rest_enabled
```

Rest position flag.

5.7.2.31 rest_pos_curr_ref

```
int32 rest_pos_curr_ref
```

Rest position current reference.

5.7.2.32 tension_valid

```
CYBIT tension_valid
```

Tension validation bit.

5.7.2.33 timer_value

```
uint16 timer_value
```

End time of the firmware main loop.

5.7.2.34 timer_value0

```
uint16 timer_value0
```

Start time of the firmware main loop.

5.8 globals.h File Reference

Global definitions and macros are set in this file.

```
#include "FIRMWARE_CONFIGURATION.h"
#include "device.h"
#include "stdlib.h"
#include "string.h"
#include "stdio.h"
#include "math.h"
#include "commands.h"
#include "FS.h"
```

Include dependency graph for globals.h: This graph shows which files directly or indirectly include this file:

Data Structures

- struct **st_ref**

    Motor Reference structure.
- struct **st_meas**

    Measurements structure.
- struct **st_emg_meas**
- struct **st_data**

    Data sent/received structure.
- struct **st_counters**

    Usage counters structure.
- struct **st_device**

    Device related parameters structure.
- struct **st_motor**

    Motor related parameters structure.
- struct **st_encoder**

Encoder related parameters structure.
- struct **st_emg**

  EMG related parameters structure.
- struct **st_imu**

  IMU related parameters structure.
- struct **st_expansion**

  Expansion board related parameters structure.
- struct **st_user**

  User related parameters structure.
- struct **st_SH_spec**

  SoftHand specific related prameters structure.
- struct **st_eeprom**

  Memory variables.
- struct **st_imu_data**

  IMU data structure.
- struct **st_filter**

  Filter structure.
- struct **st_calib**

  Hand calibration structure.

Macros

- #define **NUM_OF_MOTORS** 2
- #define **NUM_OF_SENSORS** 3
- #define **NUM_OF_INPUT_EMGS** 2
- #define **NUM_OF_ADDITIONAL_EMGS** 6
- #define **NUM_OF_ADC_CHANNELS_MAX** (4+ **NUM_OF_INPUT_EMGS**+ **NUM_OF_ADDITIONAL_EM↩ GS**)
- #define **NUM_OF_PARAMS** 71
- #define **NUM_OF_PARAMS_MENU** 10
- #define **N_IMU_MAX** 5
- #define **NUM_OF_IMU_DATA** 5
- #define **N_ENCODER_LINE_MAX** 2
- #define **N_ENCODERS_PER_LINE_MAX** 5
- #define **N_ENCODERS NUM_OF_SENSORS**
- #define **CALIBRATION_DIV** 10
- #define **DIV_INIT_VALUE** 1
- #define **DMA_BYTES_PER_BURST** 2
- #define **DMA_REQUEST_PER_BURST** 1
- #define **DMA_SRC_BASE** (CYDEV_PERIPH_BASE)
- #define **DMA_DST_BASE** (CYDEV_SRAM_BASE)
- #define **WAIT_START** 0
- #define **WAIT_ID** 1
- #define **WAIT_LENGTH** 2
- #define **RECEIVE** 3
- #define **UNLOAD** 4
- #define **STATE_INACTIVE** 0
- #define **STATE_ACTIVE** 1
- #define **COUNTER_INC** 2
- #define **SPI_DELAY_LOW** 10
- #define **SPI_DELAY_HIGH** 100
- #define **EXP_NONE** 0

- #define **EXP_SD_RTC** 1
- #define **EXP_WIFI** 2
- #define **EXP_OTHER** 3
- #define **DRIVER_MC33887** 0
- #define **DRIVER_VNH5019** 1
- #define **RIGHT_HAND** 0
- #define **LEFT_HAND** 1
- #define **NUM_OF_USERS** 3
- #define **GENERIC_USER** 0
- #define **MARIA** 1
- #define **ROZA** 2
- #define **SOFTHAND_PRO** 0
- #define **GENERIC_2_MOTORS** 1
- #define **CUFF** 2
- #define **SH_N1** 35
- #define **SH_N2** 3
- #define **SH_I1** -1
- #define **ST_DEVICE** 0
- #define **ST_MOTOR** 10
- #define **ST_ENCODER** 20
- #define **ST_EMG** 30
- #define **ST_IMU** 40
- #define **ST_EXPANSION** 50
- #define **ST_USER** 60
- #define **ST_SH_SPEC** 70
- #define **FALSE** 0
- #define **TRUE** 1
- #define **DEFAULT_EEPROM_DISPLACEMENT** 50
- #define **EEPROM_BYTES_ROW** 16
- #define **EEPROM_COUNTERS_ROWS** 5
- #define **PWM_MAX_VALUE** 100
- #define **ANTI_WINDUP** 1000
- #define **DEFAULT_CURRENT_LIMIT** 1500
- #define **CURRENT_HYSTERESIS** 10
- #define **EMG_SAMPLE_TO_DISCARD** 500
- #define **SAMPLES_FOR_MEAN** 100
- #define **SAMPLES_FOR_EMG_MEAN** 1000
- #define **REST_POS_ERR_THR_GAIN** 10
- #define **POS_INTEGRAL_SAT_LIMIT** 50000000
- #define **CURR_INTEGRAL_SAT_LIMIT** 100000
- #define **PWM_RATE_LIMITER_MAX** 1
- #define **SAFE_STARTUP_MOTOR_READINGS** 8000
- #define **LOOKUP_DIM** 6
- #define **PREREVISION_CYCLES** 400000

Enumerations

- enum **emg_status** {
  NORMAL = 0, **RESET** = 1, **DISCARD** = 2, **SUM_AND_MEAN** = 3,
  WAIT = 4, **WAIT_EoC** = 5 }
- enum **counter_status** {
  PREPARE_DATA = 0, **WRITE_CYCLES** = 1, **WAIT_QUERY** = 2, **WRITE_END** = 3,
  NONE = 4 }

Variables

- struct **st_ref g_ref** [**NUM_OF_MOTORS**]
- struct **st_ref g_refNew** [**NUM_OF_MOTORS**]
- struct **st_ref g_refOld** [**NUM_OF_MOTORS**]
- struct **st_meas g_meas** [**N_ENCODER_LINE_MAX**]
- struct **st_meas g_measOld** [**N_ENCODER_LINE_MAX**]
- struct **st_emg_meas** g_emg_meas **g_emg_measOld**
- struct **st_data g_rx**
- struct **st_eeprom** g_mem **c_mem**
- struct **st_calib calib**
- struct **st_filter filt_v** [**NUM_OF_MOTORS**]
- struct **st_filter filt_curr_diff** [**NUM_OF_MOTORS**]
- struct **st_filter filt_i** [**NUM_OF_MOTORS**]
- struct **st_filter filt_vel** [**NUM_OF_SENSORS**]
- struct **st_filter filt_emg** [**NUM_OF_INPUT_EMGS**+ **NUM_OF_ADDITIONAL_EMGS**]
- uint16 **timer_value**
- uint16 **timer_value0**
- float **cycle_time**
- int32 **dev_tension** [**NUM_OF_MOTORS**]
- uint8 **dev_pwm_limit** [**NUM_OF_MOTORS**]
- uint8 **dev_pwm_sat** [**NUM_OF_MOTORS**]
- int32 **dev_tension_f** [**NUM_OF_MOTORS**]
- int32 **pow_tension** [**NUM_OF_MOTORS**]
- **counter_status** CYDATA **cycles_status**
- **emg_status** CYDATA **emg_1_status**
- **emg_status** CYDATA **emg_2_status**
- CYBIT **reset_last_value_flag**
- CYBIT **tension_valid**
- CYBIT **interrupt_flag**
- CYBIT **cycles_interrupt_flag**
- uint8 **maintenance_flag**
- CYBIT **can_write**
- uint8 **rest_enabled**
- uint8 **forced_open**
- uint8 **battery_low_SoC**
- uint8 **change_ext_ref_flag**
- CYBIT **reset_PSoC_flag**
- int16 **ADC_buf** [NUM_OF_ADC_CHANNELS_MAX]
- uint8 **NUM_OF_ANALOG_INPUTS**
- int8 **pwm_sign**
- uint32 **data_encoder_raw** [**N_ENCODERS_PER_LINE_MAX**]
- uint8 **N_Encoder_Line_Connected** [**N_ENCODER_LINE_MAX**]
- uint16 **Encoder_Value** [**N_ENCODER_LINE_MAX**][**N_ENCODERS_PER_LINE_MAX**]
- uint8 **Encoder_Check** [**N_ENCODER_LINE_MAX**][**N_ENCODERS_PER_LINE_MAX**]
- int32 **rest_pos_curr_ref**
- FS_FILE ∗ **pFile**
- char **sdFile** [100]
- char **sdParam** [100]
- uint8 **N_IMU_Connected**
- uint8 **IMU_connected** [N_IMU_MAX]
- int **imus_data_size**
- int **single_imu_size** [N_IMU_MAX]
- struct **st_imu_data g_imu** [N_IMU_MAX]
- struct **st_imu_data g_imuNew** [N_IMU_MAX]

- uint8 **Accel** [N_IMU_MAX][6]
- uint8 **Gyro** [N_IMU_MAX][6]
- uint8 **Mag** [N_IMU_MAX][6]
- uint8 **MagCal** [N_IMU_MAX][3]
- uint8 **Temp** [N_IMU_MAX][2]
- float **Quat** [N_IMU_MAX][4]

5.8.1 Detailed Description

Global definitions and macros are set in this file.

Date

February 01, 2018

Author

Centro "E.Piaggio"

Copyright

(C) 2012-2016 qbrobotics. All rights reserved.
(C) 2017-2019 Centro "E.Piaggio". All rights reserved.

5.8.2 Macro Definition Documentation

5.8.2.1 ANTI_WINDUP

```
#define ANTI_WINDUP 1000
```

Anti windup saturation.

5.8.2.2 CALIBRATION_DIV

```
#define CALIBRATION_DIV 10
```

Frequency divisor for hand calibration (100Hz).

5.8.2.3 COUNTER_INC

```
#define COUNTER_INC 2
```

Counter cycle increment.

5.8.2.4 CURR_INTEGRAL_SAT_LIMIT

```
#define CURR_INTEGRAL_SAT_LIMIT 100000
```

Anti windup on current control.

5.8.2.5 CURRENT_HYSTERESIS

```
#define CURRENT_HYSTERESIS 10
```

milliAmperes of hysteresis for current control.

5.8.2.6 DEFAULT_CURRENT_LIMIT

```
#define DEFAULT_CURRENT_LIMIT 1500
```

Default Current limit, 0 stands for unlimited.

5.8.2.7 DEFAULT_EEPROM_DISPLACEMENT

```
#define DEFAULT_EEPROM_DISPLACEMENT 50
```

Number of pages occupied by the EEPROM.

5.8.2.8 DIV_INIT_VALUE

```
#define DIV_INIT_VALUE 1
```

Initial value for hand counter calibration.

5.8.2.9 EEPROM_BYTES_ROW

```
#define EEPROM_BYTES_ROW 16
```

EEPROM number of bytes per row.

5.8.2.10 EEPROM_COUNTERS_ROWS

```
#define EEPROM_COUNTERS_ROWS 5
```

EEPROM number of rows dedicated to store counters.

5.8.2.11 EMG_SAMPLE_TO_DISCARD

```
#define EMG_SAMPLE_TO_DISCARD 500
```

Number of sample to discard before calibration.

5.8.2.12 LOOKUP_DIM

```
#define LOOKUP_DIM 6
```

Dimension of the current lookup table.

5.8.2.13 N_ENCODER_LINE_MAX

```
#define N_ENCODER_LINE_MAX 2
```

Max number of CS lines which can contain encoders.

5.8.2.14 N_ENCODERS_PER_LINE_MAX

```
#define N_ENCODERS_PER_LINE_MAX 5
```

Max number of encoders per line.

5.8.2.15 NUM_OF_ADDITIONAL_EMGS

```
#define NUM_OF_ADDITIONAL_EMGS 6
```

Number of additional emg channels.

5.8.2.16 NUM_OF_INPUT_EMGS

```
#define NUM_OF_INPUT_EMGS 2
```

Number of emg channels.

5.8.2.17 NUM_OF_MOTORS

```
#define NUM_OF_MOTORS 2
```

Number of motors.

5.8.2.18 NUM_OF_PARAMS

```
#define NUM_OF_PARAMS 71
```

Number of parameters saved in the EEPROM.

5.8.2.19 NUM_OF_PARAMS_MENU

```
#define NUM_OF_PARAMS_MENU 10
```

Number of parameters menu.

5.8.2.20 NUM_OF_SENSORS

```
#define NUM_OF_SENSORS 3
```

Number of encoders.

5.8.2.21 POS_INTEGRAL_SAT_LIMIT

```
#define POS_INTEGRAL_SAT_LIMIT 50000000
```

Anti windup on position control.

5.8.2.22 PREREVISION_CYCLES

```
#define PREREVISION_CYCLES 400000
```

Number of SoftHand Pro cycles before maintenance.

5.8.2.23 PWM_MAX_VALUE

```
#define PWM_MAX_VALUE 100
```

Maximum value of the PWM signal.

5.8.2.24 RECEIVE

```
#define RECEIVE 3
```

Package data receiving status.

5.8.2.25 REST_POS_ERR_THR_GAIN

```
#define REST_POS_ERR_THR_GAIN 10
```

Gain related to stop condition threshold in rest position routine.

5.8.2.26 SAFE_STARTUP_MOTOR_READINGS

```
#define SAFE_STARTUP_MOTOR_READINGS 8000
```

Number of encoder readings after position reconstruction before activating motor.

5.8.2.27 SAMPLES_FOR_EMG_MEAN

```
#define SAMPLES_FOR_EMG_MEAN 1000
```

Number of samples used to mean emg values.

5.8.2.28 SAMPLES_FOR_MEAN

```
#define SAMPLES_FOR_MEAN 100
```

Number of samples used to mean current values.

5.8.2.29 SH_I1

```
#define SH_I1 -1
```

First gear invariant value in SoftHandPro device.

5.8.2.30 SH_N1

```
#define SH_N1 35
```

Number of teeth of the first encoder gear in SoftHandPro device.

5.8.2.31 SH_N2

```
#define SH_N2 3
```

Number of teeth of the second encoder gear in SoftHandPro device.

5.8.2.32 STATE_ACTIVE

```
#define STATE_ACTIVE 1
```

Closed SoftHand position / EMG Active.

5.8.2.33 STATE_INACTIVE

```
#define STATE_INACTIVE 0
```

Open SoftHand position / EMG Inactive.

5.8.2.34 UNLOAD

```
#define UNLOAD 4
```

Package data flush status.

5.8.2.35 WAIT_ID

```
#define WAIT_ID 1
```

Package ID waiting status.

5.8.2.36 WAIT_LENGTH

```
#define WAIT_LENGTH 2
```

Package lenght waiting status.

5.8.2.37 WAIT_START

```
#define WAIT_START 0
```

Package start waiting status.

5.8.3 Enumeration Type Documentation

5.8.3.1 counter_status

```
enum counter_status
```

Enumerator

PREPARE_DATA	Prepare data to be written on EEPROM.
WRITE_CYCLES	Cycles writing on EEPROM is enabled and control is passed to query.
WAIT_QUERY	Wait until EEPROM_Query() has finished writing on EEPROM and then disable cycles writing.
WRITE_END	End of EEPROM writing.
NONE	Cycles writing on EEPROM is disabled.

5.8.3.2 emg_status

```
enum emg_status
```

Enumerator

NORMAL	Normal execution.
RESET	Reset analog measurements.
DISCARD	Discard first samples to obtain a correct value.
SUM_AND_MEAN	Sum and mean a definite value of samples.
WAIT	The second emg waits until the first emg has a valid value.
WAIT_EoC	The second emg waits for end of calibration.

5.8.4 Variable Documentation

5.8.4.1 battery_low_SoC

```
uint8 battery_low_SoC
```

Battery low State of Charge flag (re-open terminal device when active).

5.8.4.2 c_mem

```
struct  st_eeprom g_mem c_mem
```

Memory parameters.

5.8.4.3 calib

```
struct  st_calib calib
```

Calibration variables.

5.8.4.4 can_write

```
CYBIT can_write
```

Write to EEPROM flag.

5.8.4.5 change_ext_ref_flag

```
uint8 change_ext_ref_flag
```

This flag is set when an external reference command is received.

5.8.4.6 cycle_time

```
float cycle_time
```

Variable used to calculate how much time a cycle takes.

5.8.4.7 cycles_interrupt_flag

```
CYBIT cycles_interrupt_flag
```

Cycles timer interrupt flag enabler.

5.8.4.8 cycles_status

```
counter_status CYDATA cycles_status
```

Cycles counter state machine status.

5.8.4.9 dev_pwm_limit

```
uint8 dev_pwm_limit[ NUM_OF_MOTORS]
```

Device pwm limit. It may change during execution.

5.8.4.10 dev_pwm_sat

```
uint8 dev_pwm_sat[ NUM_OF_MOTORS]
```

Device pwm saturation.

Device pwm saturation. By default the saturation value must not exceed 100.

5.8.4.11 dev_tension

```
int32 dev_tension[ NUM_OF_MOTORS]
```

Power supply tension.

5.8.4.12 dev_tension_f

```
int32 dev_tension_f[ NUM_OF_MOTORS]
```

Filtered power supply tension.

5.8.4.13 emg_1_status

```
emg_status CYDATA emg_1_status
```

First EMG sensor status.

5.8.4.14 emg_2_status

```
emg_status CYDATA emg_2_status
```

Second EMG sensor status.

5.8.4.15 filt_emg

```
struct st_filter filt_emg[ NUM_OF_INPUT_EMGS+ NUM_OF_ADDITIONAL_EMGS]
```

EMG filter variables.

5.8.4.16 filt_i

```
struct st_filter filt_i[ NUM_OF_MOTORS]
```

Voltage and current filter variables.

5.8.4.17 filt_vel

```
struct st_filter filt_vel[ NUM_OF_SENSORS]
```

Velocity filter variables.

5.8.4.18 forced_open

```
uint8 forced_open
```

Forced open flag (used in position with rest position control).

5.8.4.19 g_emg_measOld

```
struct st_emg_meas g_emg_meas g_emg_measOld
```

EMG Measurements.

5.8.4.20 g_measOld

```
struct st_meas g_measOld[ N_ENCODER_LINE_MAX]
```

Measurements.

5.8.4.21 g_refOld

```
struct st_ref g_refOld[ NUM_OF_MOTORS]
```

Reference variables.

5.8.4.22 g_rx

```
struct st_data g_rx
```

Incoming/Outcoming data.

5.8.4.23 interrupt_flag

```
CYBIT interrupt_flag
```

Interrupt flag enabler.

5.8.4.24 maintenance_flag

```
uint8 maintenance_flag
```

Maintenance flag.

5.8.4.25 NUM_OF_ANALOG_INPUTS

```
uint8 NUM_OF_ANALOG_INPUTS
```

ADC measurements buffer (sizeof buffer equal to maximum number of ADC channels).

ADC measurements buffer.

5.8.4.26 pow_tension

```
int32 pow_tension[ NUM_OF_MOTORS]
```

Computed power supply tension.

5.8.4.27 pwm_sign

```
int8 pwm_sign
```

ADC currently configured channels. Sign of pwm driven. Used to obtain current sign.

5.8.4.28 reset_last_value_flag

```
CYBIT reset_last_value_flag
```

This flag is set when the encoders last values must be resetted.

5.8.4.29 reset_PSoC_flag

```
CYBIT reset_PSoC_flag
```

This flag is set when a board fw reset is necessary.

5.8.4.30 rest_enabled

```
uint8 rest_enabled
```

Rest position flag.

5.8.4.31 rest_pos_curr_ref

```
int32 rest_pos_curr_ref
```

Rest position current reference.

5.8.4.32 tension_valid

```
CYBIT tension_valid
```

Tension validation bit.

5.8.4.33 timer_value

```
uint16 timer_value
```

End time of the firmware main loop.

5.8.4.34 timer_value0

```
uint16 timer_value0
```

Start time of the firmware main loop.

5.9 IMU_functions.c File Reference

Implementation of IMU module functions.

```
#include "IMU_functions.h"
```

Include dependency graph for IMU_functions.c:

Functions

- void **ImusReset** ()
- void **InitIMU** ()
- void **InitIMUMagCal** ()
- void **ChipSelectorIMU** (int n)
- void **InitIMUgeneral** ()
- void **ReadIMU** (int n)
- void **ReadAcc** (int n)
- void **ReadGyro** (int n)
- void **ReadMag** (int n)
- void **ReadMagCal** (int n)
- void **ReadQuat** (int n)
- void **ReadAllIMUs** ()
- void **ReadTemp** (int n)
- void **WriteControlRegisterIMU** (uint8 address, uint8 dta)
- uint8 **ReadControlRegisterIMU** (uint8 address)
- void **SPI_delay** ()

Variables

- uint8 **Accel** [N_IMU_MAX][6]
- uint8 **Gyro** [N_IMU_MAX][6]
- uint8 **Mag** [N_IMU_MAX][6]
- uint8 **MagCal** [N_IMU_MAX][3]

5.9.1 Detailed Description

Implementation of IMU module functions.

Date

February 01, 2018

Author

Centro "E.Piaggio"

Copyright

(C) 2012-2016 qbrobotics. All rights reserved.
(C) 2017-2019 Centro "E.Piaggio". All rights reserved.

5.10 IMU_functions.h File Reference

Definition of IMU module functions.

```
#include <project.h>
#include "globals.h"
#include <stdlib.h>
#include <stdio.h>
#include "utils.h"
#include <SPI_IMU.h>
```

Include dependency graph for IMU_functions.h: This graph shows which files directly or indirectly include this file:

Macros

- #define **MPU9250_RCR** 0x80
- #define **MPU9250_WCR** 0x00
- #define **MPU9250_CONFIG** 0x1A
- #define **MPU9250_GYRO_CONFIG** 0x1B
- #define **MPU9250_ACCEL_CONFIG** 0x1C
- #define **MPU9250_ACCEL_CONFIG2** 0x1D
- #define **MPU9250_ACCEL_XOUT_H** 0x3B
- #define **MPU9250_ACCEL_XOUT_L** 0x3C
- #define **MPU9250_ACCEL_YOUT_H** 0x3D
- #define **MPU9250_ACCEL_YOUT_L** 0x3E
- #define **MPU9250_ACCEL_ZOUT_H** 0x3F
- #define **MPU9250_ACCEL_ZOUT_L** 0x40
- #define **MPU9250_TEMP_OUT_H** 0x41
- #define **MPU9250_TEMP_OUT_L** 0x42
- #define **MPU9250_GYRO_XOUT_H** 0x43
- #define **MPU9250_GYRO_XOUT_L** 0x44
- #define **MPU9250_GYRO_YOUT_H** 0x45
- #define **MPU9250_GYRO_YOUT_L** 0x46
- #define **MPU9250_GYRO_ZOUT_H** 0x47
- #define **MPU9250_GYRO_ZOUT_L** 0x48
- #define **MPU9250_USER_CTRL** 0x6A
- #define **MPU9250_PWR_MGMT_1** 0x6B
- #define **MPU9250_WHO_AM_I** 0x75
- #define **MPU9250_FIFO_EN** 0x23
- #define **MPU9250_I2C_MST_CTRL** 0x24
- #define **MPU9250_I2C_SLV0_ADDR** 0x25
- #define **MPU9250_I2C_SLV0_REG** 0x26
- #define **MPU9250_I2C_SLV0_CTRL** 0x27
- #define **MPU9250_I2C_SLV1_ADDR** 0x28
- #define **MPU9250_I2C_SLV1_REG** 0x29
- #define **MPU9250_I2C_SLV1_CTRL** 0x2A
- #define **MPU9250_EXT_SENS_DATA_00** 0x49
- #define **MPU9250_EXT_SENS_DATA_01** 0x4A
- #define **MPU9250_EXT_SENS_DATA_02** 0x4B
- #define **MPU9250_EXT_SENS_DATA_03** 0x4C
- #define **MPU9250_EXT_SENS_DATA_04** 0x4D
- #define **MPU9250_EXT_SENS_DATA_05** 0x4E
- #define **MPU9250_EXT_SENS_DATA_06** 0x4F
- #define **MPU9250_EXT_SENS_DATA_07** 0x50
- #define **MPU9250_I2C_SLV0_D0** 0x63
- #define **MPU9250_I2C_SLV1_D0** 0x64
- #define **MPU9250_I2C_MST_DELAY_CTRL** 0x67
- #define **AK8936_ADDRESS** 0x0C
- #define **AK8936_WIA** 0x00
- #define **AK8936_INFO** 0x01
- #define **AK8936_ST1** 0x02
- #define **AK8936_XOUT_L** 0x03
- #define **AK8936_XOUT_H** 0x04
- #define **AK8936_YOUT_L** 0x05
- #define **AK8936_YOUT_H** 0x06
- #define **AK8936_ZOUT_L** 0x07
- #define **AK8936_ZOUT_H** 0x08
- #define **AK8936_ST2** 0x09

- #define **AK8936_CNTL** 0x0A
- #define **AK8963_CNTL2** 0x0B
- #define **AK8936_ASTC** 0x0C
- #define **AK8936_I2CDIS** 0x0F
- #define **ACC_SF_2G** 0x00
- #define **ACC_SF_4G** 0x08
- #define **ACC_SF_8G** 0x10
- #define **ACC_SF_16G** 0x18
- #define **GYRO_SF_250** 0x00
- #define **GYRO_SF_500** 0x80
- #define **GYRO_SF_2000** 0x18
- #define **G_TO_MS2** 9.79
- #define **DEG_TO_RAD** (3.14159265359 / 180.0)
- #define **LP_ACC_FREQ_460** 0x00
- #define **LP_ACC_FREQ_184** 0x01
- #define **LP_ACC_FREQ_92** 0x02
- #define **LP_ACC_FREQ_41** 0x03
- #define **LP_ACC_FREQ_20** 0x04
- #define **LP_ACC_FREQ_10** 0x05
- #define **LP_ACC_FREQ_5** 0x06
- #define **TICK2GYRO** 0.000133158
- #define **TICK2ACC** 0.000061037
- #define **BETA** 100000.0
- #define **GYRO_THR** 0.2618

Functions

- void **ImusReset** ()
- void **InitIMU** ()
- void **InitIMUMagCal** ()
- void **InitIMUgeneral** ()
- void **ReadAcc** (int n)
- void **ReadGyro** (int n)
- void **ReadMag** (int n)
- void **ReadMagCal** (int n)
- void **ReadQuat** (int n)
- void **ReadTemp** (int n)
- void **ReadIMU** (int n)
- void **ReadAllIMUs** ()
- uint8 **ReadControlRegisterIMU** (uint8 address)
- void **WriteControlRegisterIMU** (uint8 address, uint8 dta)
- void **ChipSelectorIMU** (int n)
- void **SPI_delay** ()

5.10.1 Detailed Description

Definition of IMU module functions.

Date

February 01, 2018

Author

Centro "E.Piaggio"

Copyright

5.11 interruptions.c File Reference

Interruption handling and firmware core functions.

```
#include "interruptions.h"
```
Include dependency graph for interruptions.c:

Functions

- **CY_ISR** (ISR_RS485_RX_ExInterrupt)
- **CY_ISR** (ISR_CYCLES_Handler)
- void **interrupt_manager** ()
- void **function_scheduler** (void)
- void **motor_control_SH** ()
- void **motor_control_generic** (uint8 idx)
- void **encoder_reading_SPI** (uint8 n_line, uint8 assoc_motor)
- void **analog_read_end** ()
- void **overcurrent_control** ()
- void **pwm_limit_search** (uint8 mot_idx)
- void **cycles_counter_update** ()
- void **save_cycles_eeprom** ()

Variables

- static const uint8 **pwm_preload_values** [29]

5.11.1 Detailed Description

Interruption handling and firmware core functions.

Date

October 01, 2017

Author

Centro "E.Piaggio"

Copyright

5.11.2 Function Documentation

5.11.2.1 analog_read_end()

```
void analog_read_end ( )
```

This function executes and terminates the analog readings.

5.11.2.2 cycles_counter_update()

```
void cycles_counter_update ( )
```

This function increases the cycles counters variables, depending on SoftHand position and the current absorbed by the motor.

5.11.2.3 encoder_reading_SPI()

```
void encoder_reading_SPI (
             uint8 n_line,
             uint8 assoc_motor )
```

This functions reads the value from all the connected encoders.

5.11.2.4 function_scheduler()

```
void function_scheduler (
             void  )
```

This function schedules the other functions in an order that optimizes the controller usage.

5.11.2.5 interrupt_manager()

```
void interrupt_manager ( )
```

This function is called in predefined moments during firmware execution in order to unpack the received package.

5.11.2.6 motor_control_generic()

```
void motor_control_generic (
             uint8 index )
```

This function controls the motor direction and velocity, depending on the input and control modality set.

5.11.2.7 motor_control_SH()

```
void motor_control_SH ( )
```

This function controls the motor direction and velocity, depending on the input and control modality set.

5.11.2.8 overcurrent_control()

```
void overcurrent_control ( )
```

This function increases or decreases the pwm maximum value, depending on the current absorbed by the motor.

5.11.2.9 pwm_limit_search()

```
void pwm_limit_search (
             uint8 mot_idx )
```

This function scales the pwm value of the motor, depending on the power supply voltage, in order to not make the motor wind too fast.

5.11.2.10 save_cycles_eeprom()

```
void save_cycles_eeprom ( )
```

This function saves cycles counters variables into EEPROM memory.

5.11.3 Variable Documentation

5.11.3.1 pwm_preload_values

```
const uint8 pwm_preload_values[29]  [static]
```

Initial value:

```
= {100,
                                         83,
                                         78,
                                         76,
                                         74,
                                         72,
                                         70,
                                         68,
                                         67,
                                         65,
                                         64,
                                         63,
                                         62,
                                         61,
                                         60,
                                         59,
                                         58,
                                         57,
                                         56,
                                         56,
                                         55,
                                         54,
                                         54,
                                         53,
                                         52,
                                         52,
                                         52,
                                         51,
                                         51}
```

5.12 interruptions.h File Reference

Interruptions header file.

```
#include "device.h"
#include "command_processing.h"
#include "IMU_functions.h"
#include "Encoder_functions.h"
#include "SD_RTC_functions.h"
#include "globals.h"
#include "utils.h"
```

Include dependency graph for interruptions.h: This graph shows which files directly or indirectly include this file:

Functions

- void **motor_control_generic** (uint8 index)
- void **save_cycles_eeprom** ()

Interruptions

- **CY_ISR_PROTO** (ISR_RS485_RX_ExInterrupt)
- **CY_ISR_PROTO** (ISR_CYCLES_Handler)

General function scheduler

- void **function_scheduler** (void)

Encoder reading SPI function

- void **encoder_reading_SPI** (uint8 n_line, uint8 assoc_motor)

Motor control function

- void **motor_control_SH** ()

Analog readings

- void **analog_read_end** ()

Interrupt manager

- void **interrupt_manager** ()

Utility functions

- void **pwm_limit_search** (uint8 mot_idx)
- void **overcurrent_control** ()
- void **cycles_counter_update** ()

5.12.1 Detailed Description

Interruptions header file.

Date

October 01, 2017

Author

Centro "E.Piaggio"

Copyright

(C) 2012-2016 qbrobotics. All rights reserved.
(C) 2017-2019 Centro "E.Piaggio". All rights reserved.

5.12.2 Function Documentation

5.12.2.1 analog_read_end()

```
void analog_read_end ( )
```

This function executes and terminates the analog readings.

5.12.2.2 CY_ISR_PROTO() [1/2]

```
CY_ISR_PROTO (
            ISR_RS485_RX_ExInterrupt  )
```

This interruption sets a flag to let the firmware know that a communication interruption is pending and needs to be handled. The interruption will be handled in predefined moments during the firmware execution. When this interruption is handled, it unpacks the package received on the RS485 communication bus.

5.12.2.3 CY_ISR_PROTO() [2/2]

```
CY_ISR_PROTO (
            ISR_CYCLES_Handler  )
```

This interruption sets a flag to let the firmware know that a cycles timer interruption is pending and needs to be handled. The interrpution will be handled in predefined moments during the firmware execution. When this interruption is handled, it updates cycles counters.

5.12.2.4 cycles_counter_update()

```
void cycles_counter_update ( )
```

This function increases the cycles counters variables, depending on SoftHand position and the current absorbed by the motor.

5.12.2.5 encoder_reading_SPI()

```
void encoder_reading_SPI (
             uint8 n_line,
             uint8 assoc_motor )
```

This functions reads the value from all the connected encoders.

5.12.2.6 function_scheduler()

```
void function_scheduler (
             void  )
```

This function schedules the other functions in an order that optimizes the controller usage.

5.12.2.7 interrupt_manager()

```
void interrupt_manager ( )
```

This function is called in predefined moments during firmware execution in order to unpack the received package.

5.12.2.8 motor_control_generic()

```
void motor_control_generic (
             uint8 index )
```

This function controls the motor direction and velocity, depending on the input and control modality set.

5.12.2.9 motor_control_SH()

```
void motor_control_SH ( )
```

This function controls the motor direction and velocity, depending on the input and control modality set.

5.12.2.10 overcurrent_control()

```
void overcurrent_control ( )
```

This function increases or decreases the pwm maximum value, depending on the current absorbed by the motor.

5.12.2.11 pwm_limit_search()

```
void pwm_limit_search (
            uint8 mot_idx )
```

This function scales the pwm value of the motor, depending on the power supply voltage, in order to not make the motor wind too fast.

5.12.2.12 save_cycles_eeprom()

```
void save_cycles_eeprom ( )
```

This function saves cycles counters variables into EEPROM memory.

5.13 main.c File Reference

Firmware main file.

```
#include "device.h"
#include "globals.h"
#include "interruptions.h"
#include "command_processing.h"
#include "IMU_functions.h"
#include "Encoder_functions.h"
#include "SD_RTC_functions.h"
#include "utils.h"
#include "project.h"
#include "FS.h"
```

Include dependency graph for main.c:

Functions

- int **main** ()

5.13.1 Detailed Description

Firmware main file.

Date

May 31, 2019

Author

Centro "E.Piaggio"

Copyright

(C) 2019 Centro "E.Piaggio". All rights reserved.

5.14 SD_RTC_functions.c File Reference

Implementation of SD and RTC module functions.

```
#include "SD_RTC_functions.h"
```

Include dependency graph for SD_RTC_functions.c:

Functions

- void **DS1302_write** (uint8 address, uint8 data_wr)
- void **DS1302_writeByte** (uint8 data_wr)
- uint8 **DS1302_read** (uint8 address)
- uint8 **DS1302_readByte** ()
- void **shiftOut_RTC** (uint8 val)
- void **store_RTC_current_time** ()
- void **set_RTC_time** ()
- void **InitSD_FS** ()
- void **Write_SD_Param_file** ()
- void **Read_SD_Param** (char ∗info_param, int n_p)
- void **Read_SD_Data** (char ∗info_data, int n_d)

5.14.1 Detailed Description

Implementation of SD and RTC module functions.

Date

February 13, 2019

Author

Centro "E.Piaggio"

Copyright

(C) 2012-2016 qbrobotics. All rights reserved.
(C) 2017-2019 Centro "E.Piaggio". All rights reserved.

5.15 SD_RTC_functions.h File Reference

Definition of SD and RTC module functions.

```
#include <project.h>
#include "globals.h"
#include <stdlib.h>
#include <stdio.h>
#include "command_processing.h"
```

Include dependency graph for SD_RTC_functions.h: This graph shows which files directly or indirectly include this file:

Macros

- #define **DS1302_SECONDS_WR** 0x80
- #define **DS1302_SECONDS_RD** 0x81
- #define **DS1302_MINUTES_WR** 0x82
- #define **DS1302_MINUTES_RD** 0x83
- #define **DS1302_HOUR_WR** 0x84
- #define **DS1302_HOUR_RD** 0x85
- #define **DS1302_DATE_WR** 0x86
- #define **DS1302_DATE_RD** 0x87
- #define **DS1302_MONTH_WR** 0x88
- #define **DS1302_MONTH_RD** 0x89
- #define **DS1302_YEAR_WR** 0x8C
- #define **DS1302_YEAR_RD** 0x8D

Functions

- void **DS1302_write** (uint8 address, uint8 data_wr)
- void **DS1302_writeByte** (uint8 data_wr)
- uint8 **DS1302_read** (uint8 address)
- uint8 **DS1302_readByte** ()
- void **shiftOut_RTC** (uint8 val)
- void **store_RTC_current_time** ()
- void **set_RTC_time** ()
- void **InitSD_FS** ()
- void **Write_SD_Param_file** ()
- void **Read_SD_Param** (char ∗, int)
- void **Read_SD_Data** (char ∗, int)

5.15.1 Detailed Description

Definition of SD and RTC module functions.

Date

February 13, 2019

Author

Centro "E.Piaggio"

Copyright

(C) 2012-2016 qbrobotics. All rights reserved.
(C) 2017-2019 Centro "E.Piaggio". All rights reserved.

5.16 utils.c File Reference

Definition of utility functions.

`#include "utils.h"`
Include dependency graph for utils.c:

Macros

- #define **M** 65536

  Number of encoder ticks per turn.

Functions

- int32 **curr_estim** (uint8 idx, int32 pos, int32 vel, int32 ref)
- int32 **filter** (int32 new_value, struct **st_filter** ∗f)
- uint32 **my_mod** (int32 val, int32 divisor)
- void **calibration** (void)
- int **calc_turns_fcn_SH** (const int32 pos1, const int32 pos2, const int N1, const int N2, const int I1)
- int **calc_turns_fcn** (const int32 pos1, const int32 pos2, const int N1, const int N2, const int I1)
- void **check_rest_position** (void)
- void **LED_control** (uint8 mode)
- void **battery_management** ()
- void **ADC_Set_N_Channels** ()
- void **enable_motor** (uint8 idx, uint8 val)
- void **reset_counters** ()
- float **invSqrt** (float x)
- void **v3_normalize** (float v3_in[3])
- void **v4_normalize** (float v4_in[4])

5.16.1 Detailed Description

Definition of utility functions.

Date

October 01, 2017

Author

Centro "E.Piaggio"

Copyright

(C) 2012-2016 qbrobotics. All rights reserved.
(C) 2017-2019 Centro "E.Piaggio". All rights reserved.

5.16.2 Function Documentation

5.16.2.1 calc_turns_fcn()

```
int calc_turns_fcn (
            const int32 pos1,
            const int32 pos2,
            const int N1,
            const int N2,
            const int I1 )
```

This function is used at startup to reconstruct the correct turn of the shaft connected to the motor. Generic. It need two encoders to work.

Parameters

pos1	First encoder position
pos2	Second encoder position

Returns

Returns the number of turns of motor pulley at startup

5.16.2.2 calc_turns_fcn_SH()

```
int calc_turns_fcn_SH (
            const int32 pos1,
            const int32 pos2,
            const int N1,
            const int N2,
            const int I1 )
```

This function is used at startup to reconstruct the correct turn of the shaft connected to the motor. Only for SoftHand 3.0. It need two encoders to work.

Parameters

pos1	First encoder position
pos2	Second encoder position

Returns

Returns the number of turns of motor pulley at startup

5.16.2.3 calibration()

```
void calibration ( )
```

This function counts a series of hand opening and closing used to execute a calibration of the device.

5.16.2.4 check_rest_position()

```
void check_rest_position ( )
```

This function checks for rest position and, in case, gives a position reference to SoftHand.

5.16.2.5 curr_estim()

```
int32 curr_estim (
            uint8 idx,
            int32 pos,
            int32 vel,
            int32 acc )
```

Function used to obtain current estimation through current lookup table.

Parameters

idx	Index of motor.
pos	Position of the encoder in ticks.
vel	Speed of the encoder.
accel	Acceleration of the encoder

Returns

Returns an estimation of the motor current, depending on its position, velocity and acceleration.

5.16.2.6 filter()

```
int32 filter (
            int32 value,
            struct st_filter * f )
```

Generic low pass filter. The weighted average between the old value and the new one is executed.

Parameters

value	New value of the filter.
f	Pointer to specific struct of type **st_filter** (p. 20).

Returns

Returns the filtered current value

5.16.2.7 LED_control()

```
void LED_control (
            uint8 mode )
```

This function switches between different LEDs condition depending on battery level of charge or needed maintenance.

5.16.2.8 my_mod()

```
uint32 my_mod (
            int32 val,
            int32 divisor )
```

This function computes the module function, returning positive values regardless of wheter the value passed is negative

Parameters

val	The value of which the module needs to be calculated
divisor	The divisor according to which the module is calculated

5.16.2.9 reset_counters()

```
void reset_counters ( )
```

This function reset statistics counters

5.17 utils.h File Reference

Utility functions declaration.

```
#include "globals.h"
#include <math.h>
```

Include dependency graph for utils.h: This graph shows which files directly or indirectly include this file:

Macros

- #define **ZMAX** 5
- #define **ZERO_TOL** 100
- #define **REFSPEED** 20
- #define **SIGN**(A) (((A) >=0) ? (1) : (-1))

Functions

Filters

- int32 **filter** (int32 value, struct **st_filter** ∗f)

Estimating current and difference

- int32 **curr_estim** (uint8 idx, int32 pos, int32 vel, int32 acc)

Utility functions

- uint32 **my_mod** (int32 val, int32 divisor)
- int **calc_turns_fcn_SH** (const int32 pos1, const int32 pos2, const int N1, const int N2, const int I1)
- int **calc_turns_fcn** (const int32 pos1, const int32 pos2, const int N1, const int N2, const int I1)
- void **calibration** ()
- void **check_rest_position** ()
- void **LED_control** (uint8 mode)
- void **battery_management** ()
- void **ADC_Set_N_Channels** ()
- void **enable_motor** (uint8 idx, uint8 val)
- void **reset_counters** ()
- float **invSqrt** (float x)
- void **v3_normalize** (float v3_in[3])
- void **v4_normalize** (float v4_in[4])

5.17.1 Detailed Description

Utility functions declaration.

Date

October 01, 2017

Author

Centro "E.Piaggio"

Copyright

(C) 2012-2016 qbrobotics. All rights reserved.
(C) 2017-2019 Centro "E.Piaggio". All rights reserved.

5.17.2 Macro Definition Documentation

5.17.2.1 REFSPEED

```
#define REFSPEED 20
```

Constant depending on PID values.

5.17.2.2 SIGN

```
#define SIGN(
              A ) (((A) >=0) ? (1) : (-1))
```

Sign calculation function.

5.17.2.3 ZERO_TOL

```
#define ZERO_TOL 100
```

Deadband used to put to zero the virtual position due to the fact that the friction model has errors when the position is near to zero.

5.17.2.4 ZMAX

```
#define ZMAX 5
```

Constant useful for current estimation procedure.

5.17.3 Function Documentation

5.17.3.1 calc_turns_fcn()

```
int calc_turns_fcn (
            const int32 pos1,
            const int32 pos2,
            const int N1,
            const int N2,
            const int I1 )
```

This function is used at startup to reconstruct the correct turn of the shaft connected to the motor. Generic. It need two encoders to work.

Parameters

pos1	First encoder position
pos2	Second encoder position

Returns

Returns the number of turns of motor pulley at startup

5.17.3.2 calc_turns_fcn_SH()

```
int calc_turns_fcn_SH (
            const int32 pos1,
            const int32 pos2,
            const int N1,
            const int N2,
            const int I1 )
```

This function is used at startup to reconstruct the correct turn of the shaft connected to the motor. Only for SoftHand 3.0. It need two encoders to work.

Parameters

pos1	First encoder position
pos2	Second encoder position

Returns

Returns the number of turns of motor pulley at startup

5.17.3.3 calibration()

```
void calibration ( )
```

This function counts a series of hand opening and closing used to execute a calibration of the device.

5.17.3.4 check_rest_position()

```
void check_rest_position ( )
```

This function checks for rest position and, in case, gives a position reference to SoftHand.

5.17.3.5 curr_estim()

```
int32 curr_estim (
             uint8 idx,
             int32 pos,
             int32 vel,
             int32 acc )
```

Function used to obtain current estimation through current lookup table.

Parameters

idx	Index of motor.
pos	Position of the encoder in ticks.
vel	Speed of the encoder.
accel	Acceleration of the encoder

Returns

Returns an estimation of the motor current, depending on its position, velocity and acceleration.

5.17.3.6 filter()

```
int32 filter (
            int32 value,
            struct  st_filter * f )
```

Generic low pass filter. The weighted average between the old value and the new one is executed.

Parameters

value	New value of the filter.
f	Pointer to specific struct of type **st_filter** (p. 20).

Returns

Returns the filtered current value

5.17.3.7 LED_control()

```
void LED_control (
            uint8 mode )
```

This function switches between different LEDs condition depending on battery level of charge or needed maintenance.

5.17.3.8 my_mod()

```
uint32 my_mod (
            int32 val,
            int32 divisor )
```

This function computes the module function, returning positive values regardless of wheter the value passed is negative

Parameters

val	The value of which the module needs to be calculated
divisor	The divisor according to which the module is calculated

5.17.3.9 reset_counters()

```
void reset_counters ( )
```

This function reset statistics counters

Index

Software documentation - API

Generated by Doxygen 1.8.13

Contents

Chapter 1

qbAPI Libraries

Those functions allows to use the board through a serial port

Version

7.0.0

Author

Mattia Poggiani

Date

January 25th, 2019

This is a set of functions that allows to use the board via a serial port.

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 comm_settings Struct Reference

Data Fields

- HANDLE **file_handle**

The documentation for this struct was generated from the following file:

- **qbmove_communications.h**

Chapter 5

File Documentation

5.1 commands.h File Reference

Definitions for board commands, parameters and packages.

This graph shows which files directly or indirectly include this file:

5.2 cp_commands.h File Reference

Definitions for additional commands, parameters and packages.

```
#include "commands.h"
```
Include dependency graph for cp_commands.h: This graph shows which files directly or indirectly include this file:

Macros

- #define **API_VERSION** "v7.0.0 Centro Piaggio"
- #define **NUM_OF_ADDITIONAL_EMGS** 6
- #define **NUM_OF_INPUT_EMGS** 2

Enumerations

Additional Commands

- enum **additional_command** {
  CMD_GET_IMU_READINGS = 161, **CMD_GET_IMU_PARAM** = 162, **CMD_GET_ENCODER_CONF** = 163, **CMD_GET_ENCODER_RAW** = 164,
  CMD_GET_ADC_CONF = 165, **CMD_GET_ADC_RAW** = 166 }

5.2.1 Detailed Description

Definitions for additional commands, parameters and packages.

This file is included in the board firmware, in its libraries and applications. It contains all definitions that are necessary for the contruction of communication packages.

It includes definitions for all of the device commands, parameters and also the size of answer packages.

5.2.2 Macro Definition Documentation

5.2.2.1 NUM_OF_ADDITIONAL_EMGS

```
#define NUM_OF_ADDITIONAL_EMGS 6
```

Number of additional emg channels.

5.3 cp_communications.cpp File Reference

Library of functions for serial port communication with a board in addition to standard qbmove_communications.

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <ctype.h>
#include <time.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <termios.h>
#include <sys/ioctl.h>
#include <dirent.h>
#include <sys/time.h>
#include <stdlib.h>
#include <linux/serial.h>
#include "cp_communications.h"
#include "cp_commands.h"
```

Include dependency graph for cp_communications.cpp:

Macros

- #define **BUFFER_SIZE** 500
  Size of buffers that store communication packets.

5.3.1 Detailed Description

Library of functions for serial port communication with a board in addition to standard qbmove_communications.

Check the **cp_communications.h** (p. 10) file for a complete description of the public functions implemented in **cp_communications.cpp** (p. 10).

5.4 cp_communications.h File Reference

Library of functions for SERIAL PORT communication additional to standard qbmove_communications. Function Prototypes.

```
#include "qbmove_communications.h"
```

Include dependency graph for cp_communications.h: This graph shows which files directly or indirectly include this file:

Functions

- int **commGetImuReadings** (**comm_settings** ∗comm_settings_t, int id, uint8_t ∗imu_table, uint8_t ∗imus←_magcal, int n_imu, float ∗imu_values)
- int **commGetIMUParamList** (**comm_settings** ∗comm_settings_t, int id, unsigned short index, void ∗values, unsigned short value_size, unsigned short num_of_values, uint8_t ∗buffer)
- int **commGetEncoderConf** (**comm_settings** ∗comm_settings_t, int id, uint8_t ∗num_enc_line, uint8_←t ∗num_enc_per_line, uint8_t ∗enc_map)
- int **commGetEncoderRawValues** (**comm_settings** ∗comm_settings_t, int id, uint8_t enc_total, uint16_t enc_val[ ])
- int **commGetADCConf** (**comm_settings** ∗comm_settings_t, int id, uint8_t ∗num_ch, uint8_t ∗adc_map)
- int **commGetADCRawValues** (**comm_settings** ∗comm_settings_t, int id, uint8_t num_channels, short int adc[3])

5.4.1 Detailed Description

Library of functions for SERIAL PORT communication additional to standard qbmove_communications. Function Prototypes.

This library contains all necessary functions for communicating with a board when using a USB to RS485 connector that provides a Virtual COM interface.

5.4.2 Function Documentation

5.4.2.1 commGetADCConf()

```
int commGetADCConf (
              comm_settings * comm_settings_t,
             int id,
             uint8_t * num_ch,
             uint8_t * adc_map )
```

This function gets ADC map from board connected to the serial port.

Parameters

comm_← settings_t	A ***comm_settings*** *(*p. *7)* structure containing info about the communication settings.
id	The device's id number.

Returns

num_ch Number of channels.
adc_map Vector of adc map.

5.4.2.2 commGetADCRawValues()

```
int commGetADCRawValues (
             comm_settings * comm_settings_t,
             int id,
             uint8_t num_channels,
             short int adc[3] )
```

This function gets position measurements from a board connected to the serial port.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
num_channels	Number of channels.
adc	ADC raw values.

Returns

Returns 0 if communication was ok, -1 otherwise.

Example

```
comm_settings   comm_settings_t;
int             device_id = 65;
short int       adc[3];

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");

if(!commGetADCRaw(&comm_settings_t, DEVICE_ID, adc))
    printf("ADC raw: %d\t%d\t%d\n",adc[0], adc[1], adc[2]);
else
    puts("Couldn't retrieve measurements.");

closeRS485(&comm_settings_t);
```

5.4.2.3 commGetEncoderConf()

```
int commGetEncoderConf (
             comm_settings * comm_settings_t,
             int id,
             uint8_t * num_enc_line,
             uint8_t * num_enc_per_line,
             uint8_t * enc_map )
```

This function gets encoder map from board connected to the serial port.

Parameters

comm_settings↩ _t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
num_enc_line	Number of encoder lines.
num_enc_per_line	Number of encoder per line.

Returns

encoder_map Vector of encoder map.

5.4.2.4 commGetEncoderRawValues()

```
int commGetEncoderRawValues (
              comm_settings * comm_settings_t,
             int id,
             uint8_t enc_total,
             uint16_t enc_val[] )
```

This function gets encoder raw readings from board connected to the serial port.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(*p. *7)* structure containing info about the communication settings.
id	The device's id number.

Returns

encoder_val Vector of encoder map.

5.4.2.5 commGetIMUParamList()

```
int commGetIMUParamList (
              comm_settings * comm_settings_t,
             int id,
             unsigned short index,
             void * values,
             unsigned short value_size,
             unsigned short num_of_values,
             uint8_t * buffer )
```

This function gets all the parameters that are stored in the board memory and sets one of them if requested

Parameters

comm_↩ settings_t	A ***comm_settings*** *(*p. *7)* structure containing info about the communication settings.
id	The device's id number.
index	The index relative to the parameter to be get.
values	An array with the parameter values.
value_size	The byte size of the parameter to be get
num_of_values	The size of the array of the parameter to be get
buffer	The array where the parameters' values and descriptions are saved

Example

```
comm_settings   comm_settings_t;
int             device_id = 65;
unsigned char   aux_string[2000];
int             index = 0;
int             value_size = 0;
int             num_of_values = 0;

// Get parameters
commGetIMUParamList(&comm_settings_t, device_id, index, NULL, value_size, num_of_values, aux_string);
string_unpacking_and_printing(aux_string);

// Set parameters

float           val[5];
val[0] = 1;
val[1] = 1;
val[2] = 0;
val[3] = 0;
val[4] = 0;
index = 2;
value_size = 6;
num_of_values = 5;
commGetIMUParamList(&comm_settings_t, device_id, index, pid, value_size, num_of_values, NULL);
```

5.4.2.6 commGetImuReadings()

```
int commGetImuReadings (
              comm_settings * comm_settings_t,
              int id,
              uint8_t * imu_table,
              uint8_t * imus_magcal,
              int n_imu,
              float * imu_values )
```

This function gets IMU readings from IMU board connected to the serial port.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
imu_table	IMU table configuration.
imus_magcal	IMU magnetometer calibratino parameters vector.
n_imu	Number of connected IMUs.

Returns

imu_values Vector of imu readings.

5.5 qbmove_communications.cpp File Reference

Library of functions for serial port communication with a board.

```
#include <stdio.h>
#include <string.h>
```

```
#include <stdint.h>
#include <ctype.h>
#include <time.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <termios.h>
#include <sys/ioctl.h>
#include <dirent.h>
#include <sys/time.h>
#include <stdlib.h>
#include <linux/serial.h>
#include "qbmove_communications.h"
#include "commands.h"
```
Include dependency graph for qbmove_communications.cpp:

Macros

- #define **BUFFER_SIZE** 500

  Size of buffers that store communication packets.

5.5.1 Detailed Description

Library of functions for serial port communication with a board.

Date

May 03, 2018

Author

Centro "E.Piaggio"

Copyright

(C) 2012-2016 qbrobotics. All rights reserved.
(C) 2017-2018 Centro "E.Piaggio". All rights reserved.

Check the **qbmove_communications.h** (p. 15) file for a complete description of the public functions implemented in **qbmove_communications.cpp** (p. 14).

5.6 qbmove_communications.h File Reference

Library of functions for SERIAL PORT communication with a board. Function Prototypes.

```
#include <termios.h>
#include "commands.h"
#include <stdint.h>
```
Include dependency graph for qbmove_communications.h: This graph shows which files directly or indirectly include this file:

Data Structures

- struct **comm_settings**

Macros

- #define **HANDLE** int
- #define **INVALID_HANDLE_VALUE** -1
- #define **BAUD_RATE_T_2000000** 0
- #define **BAUD_RATE_T_460800** 1
- #define **MAX_WATCHDOG_TIME** 500
- #define **READ_TIMEOUT** 4000

Typedefs

- typedef struct **comm_settings comm_settings**

Functions

Virtual COM (RS485) functions

- int **RS485listPorts** (char list_of_ports[60][255])
- void **openRS485** (**comm_settings** ∗comm_settings_t, const char ∗port_s, int BAUD_RATE=B2000000)
- void **closeRS485** (**comm_settings** ∗comm_settings_t)
- int **RS485read** (**comm_settings** ∗comm_settings_t, int id, char ∗package)
- int **RS485ListDevices** (**comm_settings** ∗comm_settings_t, char list_of_ids[255])
- void **RS485GetInfo** (**comm_settings** ∗comm_settings_t, char ∗buffer)

qbAPI Commands

- int **commPing** (**comm_settings** ∗comm_settings_t, int id)
- void **commActivate** (**comm_settings** ∗comm_settings_t, int id, char activate)
- void **commSetBaudRate** (**comm_settings** ∗comm_settings_t, int id, short int baudrate)
- void **commSetWatchDog** (**comm_settings** ∗comm_settings_t, int id, short int wdt)
- void **commSetInputs** (**comm_settings** ∗comm_settings_t, int id, short int inputs[ ])
- void **commSetPosStiff** (**comm_settings** ∗comm_settings_t, int id, short int inputs[ ])
- int **commGetInputs** (**comm_settings** ∗comm_settings_t, int id, short int inputs[2])
- int **commGetMeasurements** (**comm_settings** ∗comm_settings_t, int id, short int measurements[3])
- int **commGetCounters** (**comm_settings** ∗comm_settings_t, int id, short unsigned int counters[20])
- int **commGetCurrents** (**comm_settings** ∗comm_settings_t, int id, short int currents[2])
- int **commGetCurrAndMeas** (**comm_settings** ∗comm_settings_t, int id, short int ∗values)
- int **commGetEmg** (**comm_settings** ∗comm_settings_t, int id, short int emg[2])
- int **commGetVelocities** (**comm_settings** ∗comm_settings_t, int id, short int measurements[ ])
- int **commGetAccelerations** (**comm_settings** ∗comm_settings_t, int id, short int measurements[ ])
- int **commGetActivate** (**comm_settings** ∗comm_settings_t, int id, char ∗activate)
- int **commGetInfo** (**comm_settings** ∗comm_settings_t, int id, short int info_type, char ∗info)
- int **commBootloader** (**comm_settings** ∗comm_settings_t, int id)
- int **commCalibrate** (**comm_settings** ∗comm_settings_t, int id)
- int **commHandCalibrate** (**comm_settings** ∗comm_settings_t, int id, short int speed, short int repetitions)

qbAPI Parameters

- int **commSetZeros** (**comm_settings** ∗comm_settings_t, int id, void ∗values, unsigned short num_of_↩ values)

- int **commGetParamList** (**comm_settings** ∗comm_settings_t, int id, unsigned short index, void ∗values, unsigned short value_size, unsigned short num_of_values, uint8_t ∗buffer)
- int **commStoreParams** (**comm_settings** ∗comm_settings_t, int id)
- int **commStoreDefaultParams** (**comm_settings** ∗comm_settings_t, int id)
- int **commRestoreParams** (**comm_settings** ∗comm_settings_t, int id)
- int **commInitMem** (**comm_settings** ∗comm_settings_t, int id)

General Functions

- long **timevaldiff** (struct timeval ∗starttime, struct timeval ∗finishtime)
- char **checksum** (char ∗data_buffer, int data_length)

Functions for other devices

- int **commExtDrive** (**comm_settings** ∗comm_settings_t, int id, char ext_input)
- void **commSetCuffInputs** (**comm_settings** ∗comm_settings_t, int id, int flag)
- int **commGetJoystick** (**comm_settings** ∗comm_settings_t, int id, short int joystick[2])

5.6.1 Detailed Description

Library of functions for SERIAL PORT communication with a board. Function Prototypes.

Date

May 03, 2018

Author

Centro "E.Piaggio"

Copyright

(C) 2012-2016 qbrobotics. All rights reserved.
(C) 2017-2019 Centro "E.Piaggio". All rights reserved.

This library contains all necessary functions for communicating with a board when using a USB to RS485 connector that provides a Virtual COM interface.

5.6.2 Function Documentation

5.6.2.1 checksum()

```
char checksum (
            char * data_buffer,
            int data_length )
```

This functions returns an 8 bit LCR checksum over the lenght of a buffer.

Parameters

data_buffer	Buffer.
data_length	Buffer length.

Example

```
char    aux;
char    buffer[5];

buffer  = "abcde";
aux     = checksum(buffer,5);
printf("Checksum: %d", (int) aux)
```

5.6.2.2 closeRS485()

```
void closeRS485 (
              comm_settings * comm_settings_t )
```

This function is used to close a serial port being used with the board.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(*p. *7)* structure containing info about the communication settings.

Example

```
comm_settings   comm_settings_t;

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");
closeRS485(&comm_settings_t);
```

5.6.2.3 commActivate()

```
void commActivate (
              comm_settings * comm_settings_t,
             int id,
             char activate )
```

This function activates or deactivates a board connected to the serial port.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(*p. *7)* structure containing info about the communication settings.
id	The device's id number.
activate	TRUE to turn motors on. FALSE to turn motors off.

Example

```
comm_settings   comm_settings_t;
int             device_id = 65;

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");
commActivate(&comm_settings_t, device_id, TRUE);
closeRS485(&comm_settings_t);
```

5.6.2.4 commBootloader()

```
int commBootloader (
                comm_settings * comm_settings_t,
                int id )
```

This function sends the board in bootloader modality in order to update the firmware on the board

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.

Returns

Return 0 on success, -1 otherwise

Example

```
comm_settings comm_settings_t;
int     device_id = 65;

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");
commBootloader(&comm_settings_t, device_id);
closeRS485(&comm_settings_t);
```

5.6.2.5 commCalibrate()

```
int commCalibrate (
                comm_settings * comm_settings_t,
                int id )
```

This function is used to calibrate the maximum stiffness value of the board

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.

Returns

Returns 0 on success, -1 otherwise

Example

```
comm_settings comm_settings_t;
int     device_id = 65;

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");
commCalibrate(&comm_settings_t, device_id);
closeRS485(&comm_settings_t);
```

5.6.2.6 commExtDrive()

```
int commExtDrive (
              comm_settings * comm_settings_t,
             int id,
             char ext_input )
```

This function is used with the armslider device. Is used to drive another board with the inputs of the first one

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the comunication settings.
id	The id of the board drive.
ext_input	A flag used to activate the external drive functionality of the board.

Returns

A negative value if something went wrong, a zero if everything went fine.

5.6.2.7 commGetAccelerations()

```
int commGetAccelerations (
              comm_settings * comm_settings_t,
             int id,
             short int measurements[] )
```

This function gets the acceleration of the qbHand motor

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
measurements	Velocity measurements.

Returns

Returns 0 if communication was ok, -1 otherwise.

Example

```
comm_settings    comm_settings_t;
int              device_id = 65;
short int        acc_measurements[3];

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");

if(!commGetAccelerations(&comm_settings_t, device_id, acc_measurements))
    printf("Measurements: %d\t%d\t%d\n", acc_measurements[0], acc_measurements[1], acc_measurements[2]);
else
    puts("Couldn't retrieve accelerations.");

closeRS485(&comm_settings_t);
```

5.6.2.8 commGetActivate()

```
int commGetActivate (
               comm_settings * comm_settings_t,
              int id,
              char * activate )
```

This function gets the activation status of a board connected to the serial port.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
activation	Activation status.

Returns

Returns 0 if communication was ok, -1 otherwise.

Example

```
comm_settings comm_settings_t;
int      device_id = 65;
char     activation_status;

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");

if(!commGetActivate(&comm_settings_t, DEVICE_ID, activation_status))
    printf("Activation status: %d\n", &activation_status);
else
    puts("Couldn't retrieve activation status.");

closeRS485(&comm_settings_t);
```

5.6.2.9 commGetCounters()

```
int commGetCounters (
               comm_settings * comm_settings_t,
              int id,
              short unsigned int counters[20] )
```

This function gets counters values from a board connected to the serial port.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
counters	Counters

Returns

Returns 0 if communication was ok, -1 otherwise.

Example

```
    comm_settings       comm_settings_t;
    int                 device_id = 65;
    short unsigned int  counters[20];

    openRS485(&comm_settings_t,"/dev/tty.usbserial-128");

    if(!commGetCounters(&comm_settings_t, DEVICE_ID, counters))
        printf("Counters: %d\t%d\t {...} %d\n", counters[0], counters[1], {...}, counters[20]);
    else
        puts("Couldn't retrieve counters.");

    closeRS485(&comm_settings_t);
```

5.6.2.10 commGetCurrAndMeas()

```
int commGetCurrAndMeas (
               comm_settings * comm_settings_t,
              int id,
              short int * values )
```

This function gets currents and position measurements from a board connected to the serial port

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
values	Current and position measurements. Currents are in first two positions

Returns

Returns 0 if communication was ok, -1 otherwise.

Example

```
comm_settings    comm_settings_t;
int              device_id = 65;
short int        values[5];

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");

 if(!commGetCurrAndMeas(&comm_settings_t, device_id, currents)){
     printf("Currents: %d\t%d\t%d\n",values[0], values[1]);
     printf("Measurements: %d\t%d\t%d\n", values[2], values[3], values[4]);
 }
 else
     puts("Couldn't retrieve currents.");

closeRS485(&comm_settings_t);
```

5.6.2.11 commGetCurrents()

```
int commGetCurrents (
               comm_settings * comm_settings_t,
              int id,
              short int currents[2] )
```

This function gets currents from a board connected to the serial port.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
currents	Currents.

Returns

Returns 0 if communication was ok, -1 otherwise.

Example

```
comm_settings    comm_settings_t;
int              device_id = 65;
short int        currents[2];

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");

if(!commGetCurrents(&comm_settings_t, device_id, currents))
    printf("Measurements: %d\t%d\t%d\n",currents[0], currents[1]);
else
    puts("Couldn't retrieve currents.");

closeRS485(&comm_settings_t);
```

5.6.2.12 commGetEmg()

```
int commGetEmg (
              comm_settings * comm_settings_t,
              int id,
              short int emg[2] )
```

This function gets measurements from electomyographics sensors connected to the qbHand. IS USED ONLY W↩
HEN THE BOARD IS USED FOR A QBHAND

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
values	Emg sensors measurements.

Returns

Returns 0 if communication was ok, -1 otherwise.

Example

```
comm_settings    comm_settings_t;
int              device_id = 65;
short int        values[2];

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");

if(!commGetEmg(&comm_settings_t, device_id, values));
    printf("Measurements: %d\t%d\t%d\n", values[0], values[1]);
else
    puts("Couldn't retrieve emg values.");

closeRS485(&comm_settings_t);
```

5.6.2.13 commGetInfo()

```
int commGetInfo (
              comm_settings * comm_settings_t,
              int id,
              short int info_type,
              char * info )
```

This function is used to ping the board and get information about the device.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
buffer	Buffer that stores a string with information about the device. BUFFER SIZE MUST BE AT LEAST 500.
info_type	Information to be retrieved.

Example

```
comm_settings comm_settings_t;
char    auxstring[500];
int     device_id = 65;

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");
commGetInfo(&comm_settings_t, device_id, INFO_ALL, auxstring);
puts(auxstring);
closeRS485(&comm_settings_t);
```

5.6.2.14 commGetInputs()

```
int commGetInputs (
               comm_settings * comm_settings_t,
              int id,
              short int inputs[2] )
```

This function gets input references from a board connected to the serial port.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
inputs	Input references.

Returns

Returns 0 if communication was ok, -1 otherwise.

Example

```
comm_settings   comm_settings_t;
int             device_id = 65;
short int       inputs[2];

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");

if(!commGetInputs(&comm_settings_t, DEVICE_ID, inputs))
    printf("Inputs: %d\t%d\n",inputs[0], inputs[1]);
else
    puts("Couldn't retrieve device inputs.");

closeRS485(&comm_settings_t);
```

5.6.2.15 commGetJoystick()

```
int commGetJoystick (
               comm_settings * comm_settings_t,
              int id,
              short int joystick[2] )
```

This function gets joystick measurementes from a softhand connected to the serial port.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
joystick	Joystick analog measurements.

Returns

Returns 0 if communication was ok, -1 otherwise.

Example

```
comm_settings    comm_settings_t;
int              device_id = 65;
short int        joystick[2];

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");

if(!commGetJoystick(&comm_settings_t, device_id, joystick))
    printf("Measurements: %d\t%d\t%d\n",joystick[0], joystick[1]);
else
    puts("Couldn't retrieve joystick measurements.");

closeRS485(&comm_settings_t);
```

5.6.2.16 commGetMeasurements()

```
int commGetMeasurements (
              comm_settings * comm_settings_t,
              int id,
              short int measurements[3] )
```

This function gets position measurements from a board connected to the serial port.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
measurements	Measurements.

Returns

Returns 0 if communication was ok, -1 otherwise.

Example

```
comm_settings   comm_settings_t;
int             device_id = 65;
short int       measurements[3];

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");

if(!commGetMeasurements(&comm_settings_t, DEVICE_ID, measurements))
    printf("Measurements: %d\t%d\t%d\n",measurements[0], measurements[1], measurements[2]);
else
    puts("Couldn’t retrieve measurements.");

closeRS485(&comm_settings_t);
```

5.6.2.17 commGetParamList()

```
int commGetParamList (
              comm_settings * comm_settings_t,
              int id,
              unsigned short index,
              void * values,
              unsigned short value_size,
              unsigned short num_of_values,
              uint8_t * buffer )
```

This function gets all the parameters that are stored in the board memory and sets one of them if requested

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
index	The index relative to the parameter to be get.
values	An array with the parameter values.
value_size	The byte size of the parameter to be get
num_of_values	The size of the array of the parameter to be get
buffer	The array where the parameters' values and descriptions are saved

Example

```
    comm_settings   comm_settings_t;
    int             device_id = 65;
    unsigned char   aux_string[2000];
    int             index = 0;
    int             value_size = 0;
    int             num_of_values = 0;

    // Get parameters
    commGetParamList(&comm_settings_t, device_id, index, NULL, value_size, num_of_values, aux_string);
    string_unpacking_and_printing(aux_string);

    // Set parameters

    float           pid[3];
    pid[0] = 0.1;
    pid[1] = 0.2;
    pid[2] = 0.3;
    index = 2;
    value_size = 4;
    num_of_values = 3;
    commGetParamList(&comm_settings_t, device_id, index, pid, value_size, num_of_values, NULL);
```

5.6.2.18 commGetVelocities()

```
int commGetVelocities (
              comm_settings * comm_settings_t,
             int id,
             short int measurements[] )
```

This function gets velocities of the two motors and the shaft from a board connected to a serial port or from the only shaft of the qbHand

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
measurements	Velocity measurements.

Returns

Returns 0 if communication was ok, -1 otherwise.

Example

```
comm_settings    comm_settings_t;
int              device_id = 65;
short int        vel_measurements[3];

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");

if(!commGetVelocities(&comm_settings_t, device_id, vel_measurements))
    printf("Measurements: %d\t%d\t%d\n", vel_measurements[0], vel_measurements[1], vel_measurements[2]);
else
    puts("Couldn't retrieve velocities.");

closeRS485(&comm_settings_t);
```

5.6.2.19 commHandCalibrate()

```
int commHandCalibrate (
             comm_settings * comm_settings_t,
            int id,
            short int speed,
            short int repetitions )
```

This function is used to make a series of opening and closures of the qbHand

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
speed	The speed of hand closure and opening [0 - 200]
repetitions	The nnumber of closures needed to be done [0 - 32767]

Example

```
comm_settings comm_settings_t;
```

```
    int     speed = 200
    int     repetitions = 400;
    int     device_id = 65;

    openRS485(&comm_settings_t,"/dev/tty.usbserial-128");
    commHandCalibrate(&comm_settings_t, device_id, speed, repetitions);
    closeRS485(&comm_settings_t);
```

5.6.2.20 commInitMem()

```
int commInitMem (
              comm_settings * comm_settings_t,
              int id )
```

This function initialize the EEPROM memory of the board by loading the default factory parameters. After the initialization a flag is set.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(*p. *7)* structure containing info about the communication settings.
id	The device's id number.

Example

```
    comm_settings comm_settings_t;
    int     device_id = 65;

    openRS485(&comm_settings_t,"/dev/tty.usbserial-128");

    commInitMem(&comm_settings_t, device_id)

    closeRS485(&comm_settings_t);
```

5.6.2.21 commPing()

```
int commPing (
              comm_settings * comm_settings_t,
              int id )
```

This function is used to ping the board.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(*p. *7)* structure containing info about the communication settings.
id	The device's id number.
buffer	Buffer that stores a string with information about the device. BUFFER SIZE MUST BE AT LEAST 500.

Returns

Returns 0 if ping was ok, -1 otherwise.

Example

```
comm_settings   comm_settings_t;
int             device_id = 65;

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");
if ( commPing(&comm_settings_t, device_id) )
    puts("Device exists.");
else
    puts("Device does not exist.");

closeRS485(&comm_settings_t);
```

5.6.2.22 commRestoreParams()

```
int commRestoreParams (
        comm_settings * comm_settings_t,
        int id )
```

This function restores the factory default parameters.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.

Example

```
comm_settings comm_settings_t;
int     device_id = 65;

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");

commRestoreParams(&comm_settings_t, device_id)

closeRS485(&comm_settings_t);
```

5.6.2.23 commSetBaudRate()

```
void commSetBaudRate (
        comm_settings * comm_settings_t,
        int id,
        short int baudrate )
```

This function sets the baudrate of communication.

Parameters

comm_↵ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
baudrate	BaudRate requested 0 = 2M baudrate, 1 = 460.8k baudrate

Example

```
comm_settings   comm_settings_t;
int             device_id = 65;
short int       baudrate = 0;

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");
commSetBaudRate(&comm_settings_t, global_args.device_id, baudrate);
closeRS485(&comm_settings_t);
```

5.6.2.24 commSetCuffInputs()

```
void commSetCuffInputs (
              comm_settings * comm_settings_t,
             int id,
             int flag )
```

This function send reference inputs to a board connected to the serial port. Is used only when the device is a Cuff.

Parameters

comm_↵ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
flag	A flag that indicates used to activate the cuff driving functionality of the board.

Example

```
comm_settings   comm_settings_t;
int             device_id = 65;
short int       cuff_inputs[2];

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");

int flag = 1;
commSetCuffInputs(&comm_settings_t, device_id, flag);
closeRS485(&comm_settings_t);
```

5.6.2.25 commSetInputs()

```
void commSetInputs (
              comm_settings * comm_settings_t,
             int id,
             short int inputs[] )
```

This function send reference inputs to a board connected to the serial port.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
inputs	Input references.

Example

```
comm_settings   comm_settings_t;
int             device_id = 65;
short int       inputs[2];

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");

inputs[0]   = 1000;
inputs[1]   = -1000;
commSetInputs(&comm_settings_t, device_id, inputs);
closeRS485(&comm_settings_t);
```

5.6.2.26 commSetPosStiff()

```
void commSetPosStiff (
              comm_settings * comm_settings_t,
             int id,
             short int inputs[] )
```

This function send reference inputs to a board connected to the serial port. The reference is in shaft position and stiffness preset.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
inputs	Input references.

Example

```
comm_settings   comm_settings_t;
int             device_id = 65;
short int       inputs[2];

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");

inputs[0]   = 100;          //Degrees
inputs[1]   = 30;           //stiffness preset
commSetPosStiff(&comm_settings_t, device_id, inputs);
closeRS485(&comm_settings_t);
```

5.6.2.27 commSetWatchDog()

```
void commSetWatchDog (
              comm_settings * comm_settings_t,
             int id,
             short int wdt )
```

This function sets watchdog timer of a board.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
wdt	Watchdog timer in [csec], max value: 500 [cs] / min value: 0 (disable) [cs]

Example

```
comm_settings   comm_settings_t;
int             device_id = 65;
short int       wdt = 60;

openRS485(&comm_settings_t,"/dev/tty.usbserial-128);
commSetWatchDog(&comm_settings_t, global_args.device_id, wdt);
closeRS485(&comm_settings_t);
```

5.6.2.28 commSetZeros()

```
int commSetZeros (
             comm_settings * comm_settings_t,
            int id,
            void * values,
            unsigned short num_of_values )
```

This function sets the encoders's zero positon value that remains stored in the board memory.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
value	An array with the encoder readings values.
num_of_values	The size of the values array, equal to the sensor number.

Example

```
comm_settings   comm_settings_t;
int             device_id = 65;
short int       measurements[3];


openRS485(&comm_settings_t,"/dev/tty.usbserial-128");
commGetMeasurements(comm_settings_t, device_id, measurements)
for(i = 0; i<3; i++)
    measurements[i] = -measurements[i];
commSetZeros(&comm_settings_t, global_args.device_id, measurements, 3);
closeRS485(&comm_settings_t);
```

5.6.2.29 commStoreDefaultParams()

```
int commStoreDefaultParams (
              comm_settings * comm_settings_t,
             int id )
```

This function stores the factory default parameters.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(*p. *7)* structure containing info about the communication settings.
id	The device's id number.

Example

```
comm_settings comm_settings_t;
int      device_id = 65;

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");
commStoreDefaultParams(&comm_settings_t, device_id)
closeRS485(&comm_settings_t);
```

5.6.2.30 commStoreParams()

```
int commStoreParams (
              comm_settings * comm_settings_t,
             int id )
```

This function stores all parameters that were set in the board memory.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(*p. *7)* structure containing info about the communication settings.
id	The device's id number.

Example

```
comm_settings comm_settings_t;
int      device_id = 65;

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");

commStoreParams(&comm_settings_t, device_id)

closeRS485(&comm_settings_t);
```

5.6.2.31 openRS485()

```
void openRS485 (
              comm_settings * comm_settings_t,
             const char * port_s,
             int BAUD_RATE = B2000000 )
```

This function is used to open a serial port for using with the board.

Parameters

comm_settings (p. *7)*	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
port_s	The string to the serial port path.
BAUD_RATE	The default baud rate value of the serial port

Returns

Returns the file descriptor associated to the serial port.

Example

```
comm_settings   comm_settings_t;

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");
if(comm_settings_t.file_handle == INVALID_HANDLE_VALUE)
{
// ERROR
}
```

5.6.2.32 RS485GetInfo()

```
void RS485GetInfo (
              comm_settings * comm_settings_t,
             char * buffer )
```

This function is used to ping the serial port for a board and to get information about the device. ONLY USE WHEN ONE DEVICE IS CONNECTED ONLY.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
buffer	Buffer that stores a string with information about the device. BUFFER SIZE MUST BE AT LEAST 500.

Example

```
comm_settings    comm_settings_t;
char             auxstring[500];

openRS485(&comm_settings_t,"/dev/tty.usbserial-128");
RS485GetInfo(&comm_settings_t, auxstring);
puts(auxstring);
closeRS485(&comm_settings_t);
```

5.6.2.33 RS485ListDevices()

```
int RS485ListDevices (
              comm_settings * comm_settings_t,
             char list_of_ids[255] )
```

This function is used to list the number of devices connected to the serial port and get their relative IDs

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
list_of_ids[255]	Buffer that stores a list of IDs to ping, in order to see which of those IDs is connected. Is then filled with the IDs connected to the serial port.

Returns

Returns the number of devices connected

Example

```
comm_settings   comm_settings_t;
int             device_id = 65;
int             device_num;
char            list_of_ids[255];

openRS485(&comm_settings_t, device_id);
device_num = RS485ListDevices(&comm_settings_t, &list_of_ids);
closeRS485(&comm_settings_t);
printf("Number of devices connected: %d", i);
```

5.6.2.34 RS485listPorts()

```
int RS485listPorts (
             char list_of_ports[60][255] )
```

This function is used to return a list of available serial ports. A maximum of 10 ports are found.

Parameters

list_of_ports	An array of strings with the serial ports paths.

Returns

Returns the number of serial ports found.

Example

```
int     i, num_ports;
char    list_of_ports[10][255];

num_ports = RS485listPorts(ports);

for(i = 0; i < num_ports; ++i)
```

```
    {
        puts(ports[i]);
    }
```

5.6.2.35 RS485read()

```
int RS485read (
            comm_settings * comm_settings_t,
            int id,
            char * package )
```

This function is used to read a package from the device.

Parameters

comm_↩ settings_t	A ***comm_settings*** *(p. 7)* structure containing info about the communication settings.
id	The device's id number.
package	Package will be stored here.

Returns

Returns package length if communication was ok, -1 otherwise.

Example

```
    comm_settings   comm_settings_t;
    int             device_id = 65;
    char            data_read[1000];

    openRS485(&comm_settings_t, "/dev/tty.usbserial-128");
    commPing(&comm_settings_t, device_id);
    RS485read(&comm_settings_t, device_id, data_read);
    closeRS485(&comm_settings_t);

    printf(data_read);
```

5.6.2.36 timevaldiff()

```
long timevaldiff (
            struct timeval * starttime,
            struct timeval * finishtime )
```

This functions returns a difference between two timeval structures in order to obtain time elapsed between the two timeval;

Parameters

starttime	The timeval structure containing the start time
finishtime	The timeval structure containing the finish time

Returns

Returns the elapsed time between the two timeval structures.

Example

```
struct timeval start, finish;
gettimeofday(&start, NULL);
// other instructions
gettimeofday(&now, NULL);
long diff = timevaldiff(&start, &now);

printf(Time elapsed: %ld, diff);
```

Index

Software documentation - Command-line tools

Generated by Doxygen 1.8.13

Contents

Chapter 1

Command line tools

Those functions allows to use the board through a serial port

Author

Centro "E.Piaggio"

Copyright

Date

January 25th, 2019

This is a set of functions that allows to use the boards via a serial port.

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 global_args Struct Reference

Data Fields

- int **device_id**
- int **flag_set_inputs**

  ./qbadmin -s option
- int **flag_get_measurements**

  ./qbadmin -g option
- int **flag_activate**

  ./qbadmin -a option
- int **flag_deactivate**

  ./qbadmin -d option
- int **flag_ping**

  ./qbadmin -p option
- int **flag_reading_ping**

  ./qbmove -r option
- int **flag_serial_port**

  ./qbadmin -t option
- int **flag_verbose**

  ./qbadmin -v option
- int **flag_file**

  ./qbadmin -f option
- int **flag_log**

  ./qbadmin -l option
- int **flag_get_emg**

  ./qbadmin -q option to get the EMG sensors measurements
- int **flag_set_zeros**

  ./qbadmin -z option
- int **flag_use_gen_sin**

  ./qbadmin -y option
- int **flag_calibration**

  ./qbadmin -k option to start a series of hand closures and openings
- int **flag_get_currents**

  ./qbadmin -c option

- int **flag_bootloader_mode**

  ./qbadmin -b option
- int **flag_set_pos_stiff**

  ./qbadmin -e option
- int **flag_get_velocities**

  ./qbadmin -i option
- int **flag_get_accelerations**

  ./qbadmin -o option
- int **flag_set_cuff_inputs**

  ./qbadmin -u option
- int **flag_set_baudrate**

  ./qbadmin -R option
- int **flag_set_watchdog**

  ./qbadmin -W option
- int **flag_polling**

  ./qbadmin -P option
- int **flag_baudrate**

  ./qbadmin -B option
- int **flag_get_joystick**

  ./qbadmin -j option
- int **flag_ext_drive**

  ./qbadmin -x option
- int **flag_get_imu_readings**

  Additional -Q option.
- int **flag_get_adc_raw**

  Additional -A option.
- int **flag_get_encoder_raw**

  Additional -E option.
- int **flag_get_SD_files**

  Additional -S option.
- short int **inputs** [NUM_OF_MOTORS]
- short int **measurements** [4]
- short int **velocities** [4]
- short int **accelerations** [4]
- short int **measurement_offset** [4]
- short int **currents** [NUM_OF_MOTORS]
- char **filename** [255]
- char **log_file** [255]
- short int **calib_speed**

  Calibration speed.
- short int **calib_repetitions**

  Calibration repetitions.
- short int **emg** [NUM_OF_EMGS]

  Emg sensors values read from the device.
- short int **joystick** [2]

  Analog joystick measurements.
- short int **ext_drive**
- int **n_imu**
- uint8_t ∗ **ids**
- uint8_t ∗ **imu_table**
- uint8_t ∗ **mag_cal**
- short int **BaudRate**

- int **save_baurate**
- short int **WDT**
- short int ∗ **adc_raw**
- FILE ∗ **SD_param_file**
- FILE ∗ **SD_data_file**
- FILE ∗ **emg_file**
- FILE ∗ **log_file_fd**

The documentation for this struct was generated from the following file:

- **qbadmin.c**

4.2 position Struct Reference

Data Fields

- float **prec**
- float **act**

The documentation for this struct was generated from the following file:

- **qbadmin.c**

Chapter 5

File Documentation

5.1 definitions.h File Reference

Definitions for board commands, parameters and packages.

```
#include <math.h>
```
Include dependency graph for definitions.h:

5.2 nmmi_param.c File Reference

Command line tools file.

```
#include "../../qbAPI/src/qbmove_communications.h"
#include "definitions.h"
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <getopt.h>
#include <stdlib.h>
```
Include dependency graph for nmmi_param.c:

Macros

- #define **NUM_OF_MAX_PARAMS** 100

Functions

- int **port_selection** ()
- int **open_port** ()
- int **initMemory** ()
- void **printMainMenu** ()
- void **printVersion** ()
- void **sort_params_asc** (int ∗, int ∗, int ∗, int)
- void **retrieve_section_str** (int, char ∗)
- int **baudrate_reader** ()
- int **main** (int argc, char ∗∗argv)
- static int **compar** (const void ∗a, const void ∗b)

Variables

- char **get_or_set**
- comm_settings **comm_settings_t**
- uint8_t **device_id** = BROADCAST_ID
- int ∗ **base_arr**
- int ∗ **param_idx_arr**

5.2.1 Detailed Description

Command line tools file.

Author

Centro "E.Piaggio"

Copyright

(C) 2019 Centro "E.Piaggio". All rights reserved.

With this file is possible to get or set firmware parameters with a new interface based on old qbparam tool.

5.2.2 Function Documentation

5.2.2.1 baudrate_reader()

```
int baudrate_reader ( )
```

Baudrate functions

5.3 nmmi_param_imu.c File Reference

Command line tools file.

```
#include "../../qbAPI/src/qbmove_communications.h"
#include "../../qbAPI/src/cp_communications.h"
#include "definitions.h"
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <getopt.h>
#include <stdint.h>
```

Include dependency graph for nmmi_param_imu.c:

Functions

- int **port_selection** ()
- int **open_port** ()
- int **initMemory** ()
- void **printMainMenu** ()
- void **printVersion** ()
- int **calibrate** ()
- int **baudrate_reader** ()
- int **main** (int argc, char ∗∗argv)

Variables

- char **get_or_set**
- comm_settings **comm_settings_t**
- uint8_t **device_id** = BROADCAST_ID

5.3.1 Detailed Description

Command line tools file.

Author

Centro "E.Piaggio"

Copyright

(C) 2012-2016 qbrobotics. All rights reserved.
(C) 2017-2018 Centro "E.Piaggio". All rights reserved.

With this file is possible to get or set IMU parameters.

5.3.2 Function Documentation

5.3.2.1 baudrate_reader()

```
int baudrate_reader ( )
```

Baudrate functions

5.4 qbadmin.c File Reference

Command line tools file.

```
#include "../../qbAPI/src/qbmove_communications.h"
#include "../../qbAPI/src/cp_communications.h"
#include "definitions.h"
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <getopt.h>
#include <string.h>
#include <sys/time.h>
#include <math.h>
#include <signal.h>
#include <assert.h>
```

Include dependency graph for qbadmin.c:

Data Structures

- struct **global_args**
- struct **position**

Functions

- int **open_port** ()
- int **port_selection** ()
- int **polling** ()
- void **display_usage** (void)
- float ∗∗ **file_parser** (char ∗, int ∗, int ∗)
- void **int_handler** (int sig)
- void **int_handler_2** (int sig)
- void **int_handler_3** (int sig)
- int **baudrate_reader** ()
- int **baudrate_writer** (const int)
- int **main** (int argc, char ∗∗argv)

Variables

- static const struct option **longOpts** [ ]
- static const char ∗ **optString** = "s:adgprtvh?f:ljqxzkycbe:uoiW:PB:QAES"
- struct **global_args global_args**
- struct **position p1**
- struct **position p2**
- uint8_t **resolution** [4]
- int **ret**
- int **aux_int**
- comm_settings **comm_settings_1**

5.4.1 Detailed Description

Command line tools file.

Author

Centro "E.Piaggio"

Copyright

(C) 2012-2016 qbrobotics. All rights reserved.
(C) 2017-2019 Centro "E.Piaggio". All rights reserved.

With this file is possible to command a terminal device.

5.4.2 Function Documentation

5.4.2.1 baudrate_reader()

```
int baudrate_reader ( )
```

Baudrate functions

5.4.2.2 display_usage()

```
void display_usage (
            void )
```

Display program usage, and exit.

5.4.2.3 file_parser()

```
float ** file_parser (
            char * filename,
            int * deltat,
            int * num_values )
```

Parse csv input file with values to be sent to the motors

Parse CSV file and return a pointer to a matrix of float dinamically allocated. Remember to use free(pointer) in the caller

5.4.2.4 int_handler()

```
void int_handler (
            int sig )
```

CTRL-c handler 1

handle CTRL-C interruption 1

5.4.2.5 int_handler_2()

```
void int_handler_2 (
            int sig )
```

CTRL-c handler 2

handle CTRL-C interruption 2

5.4.2.6 int_handler_3()

```
void int_handler_3 (
            int sig )
```

CTRL-c handler 3

Handles the ctrl+c interruption to save the emg sensors measurements into a file

5.4.2.7 main()

```
int main (
            int argc,
            char ** argv )
```

main loop

5.5 qbparam.c File Reference

Command line tools file.

```
#include "../../qbAPI/src/qbmove_communications.h"
#include "definitions.h"
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <getopt.h>
#include <stdint.h>
```

Include dependency graph for qbparam.c:

Macros

- #define **NUM_OF_MAX_PARAMS** 100

Functions

- int **port_selection** ()
- int **open_port** ()
- int **initMemory** ()
- void **printMainMenu** ()
- void **printVersion** ()
- int **calibrate** ()
- int **baudrate_reader** ()
- int **main** (int argc, char **argv)

Variables

- char **get_or_set**
- comm_settings **comm_settings_t**
- uint8_t **device_id** = BROADCAST_ID

5.5.1 Detailed Description

Command line tools file.

Author

Centro "E.Piaggio"

Copyright

With this file is possible to get or set firmware parameters.

5.5.2 Function Documentation

5.5.2.1 baudrate_reader()

```
int baudrate_reader ( )
```

Baudrate functions

Index