```
// ----------------------------------------------------------------------------
// BSD 3-Clause License

// Copyright (c) 2016, qbrobotics
// Copyright (c) 2017-2020, Centro "E.Piaggio"
// All rights reserved.

// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are met:

// * Redistributions of source code must retain the above copyright notice, this
//   list of conditions and the following disclaimer.

// * Redistributions in binary form must reproduce the above copyright notice,
//   this list of conditions and the following disclaimer in the documentation
//   and/or other materials provided with the distribution.

// * Neither the name of the copyright holder nor the names of its
//   contributors may be used to endorse or promote products derived from
//   this software without specific prior written permission.

// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
// DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
// FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
// DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
// SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
// CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
// OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
// POSSIBILITY OF SUCH DAMAGE.
// ----------------------------------------------------------------------------


/**
* \file         command_processing.c
*

* \brief        Command processing functions.
* \date         March 20th, 2020
* \author       _Centro "E.Piaggio"_
* \copyright    (C) 2012-2016 qbrobotics. All rights reserved.
* \copyright    (C) 2017-2020 Centro "E.Piaggio". All rights reserved.
*/
//==============================================================     includes
#include "command_processing.h"
```

```
//============================================================      variables

reg8 * EEPROM_ADDR = (reg8 *) CYDEV_EE_BASE;

//
================================================================================
//                                                                RX DATA PROCESSING
//
================================================================================
//  This function checks for the availability of a data packet and process it:
//      - Verify checksum;
//      - Process commands;
//
================================================================================

void commProcess(void){

    uint8 CYDATA rx_cmd;
    rx_cmd = g_rx.buffer[0];

//============================================================      verify checksum

    if (!(LCRChecksum(g_rx.buffer, g_rx.length - 1) == g_rx.buffer[g_rx.length - 1])){
        // Wrong checksum
        g_rx.ready = 0;
        return;
    }

    switch(rx_cmd) {

//================================================================      CMD_ACTIVATE
        case CMD_ACTIVATE:
            cmd_activate();
            break;

//============================================================      CMD_SET_INPUTS

        case CMD_SET_INPUTS:
            cmd_set_inputs();
            break;

//==========================================================      CMD_GET_MEASUREMENTS

        case CMD_GET_MEASUREMENTS:
```

```
            cmd_get_measurements();
            break;

//=====================================================
CMD_GET_CURR_AND_MEAS

        case CMD_GET_CURR_AND_MEAS:
            cmd_get_curr_and_meas();
            break;

//=========================================================
CMD_GET_CURRENTS

        case CMD_GET_CURRENTS:
            cmd_get_currents();
            break;

//=========================================================
CMD_GET_CURR_DIFF

        case CMD_GET_CURR_DIFF:
            cmd_get_currents_for_cuff();
            break;

//=========================================================
CMD_GET_CURR_DIFF

        case CMD_GET_VELOCITIES:
            cmd_get_velocities();
            break;

//=========================================================
CMD_GET_CURR_DIFF

        case CMD_GET_ACCEL:
            cmd_get_accelerations();
            break;

//=========================================================
CMD_GET_JOYSTICK

        case CMD_GET_JOYSTICK:
            cmd_get_joystick();
        break;

//=========================================================     CMD_GET_EMG

        case CMD_GET_EMG:
            cmd_get_emg();
            break;

//=========================================================
```

```
CMD_GET_ACTIVATE

        case CMD_GET_ACTIVATE:
            cmd_get_activate();
            break;

//=========================================================
CMD_SET_BAUDRATE

        case CMD_SET_BAUDRATE:
            cmd_set_baudrate();
            break;

//=========================================================
CMD_GET_INPUT

        case CMD_GET_INPUTS:
            cmd_get_inputs();
            break;

//=========================================================
CMD_GET_INFO

        case CMD_GET_INFO:
            infoGet(  __REV16(*((uint16 *) &g_rx.buffer[1])) );
            break;

//=========================================================
CMD_SET_PARAM

        case CMD_SET_ZEROS:
            setZeros();
            break;

//=========================================================
CMD_GET_PARAM

        case CMD_GET_PARAM_LIST:
            manage_param_list( __REV16(*((uint16 *) &g_rx.buffer[1])) );
            break;

//=========================================================
CMD_PING

        case CMD_PING:
            cmd_ping();
            break;

//=========================================================
CMD_STORE_PARAMS

        case CMD_STORE_PARAMS:
```

```
            cmd_store_params();
            break;

//================================================
CMD_STORE_DEFAULT_PARAMS

        case CMD_STORE_DEFAULT_PARAMS:
            if(memStore(DEFAULT_EEPROM_DISPLACEMENT))
                sendAcknowledgment(ACK_OK);
            else
                sendAcknowledgment(ACK_ERROR);
            break;

//=======================================================
CMD_RESTORE_PARAMS

        case CMD_RESTORE_PARAMS:
            if(memRestore())
                sendAcknowledgment(ACK_OK);
            else
                sendAcknowledgment(ACK_ERROR);
            break;

//============================================================
CMD_INIT_MEM

        case CMD_INIT_MEM:
            if(memInit())
                sendAcknowledgment(ACK_OK);
            else
                sendAcknowledgment(ACK_ERROR);
            break;

//=============================================================
CMD_BOOTLOADER

        case CMD_BOOTLOADER:
            //Not sure if ACK_OK is correct, should check
            sendAcknowledgment(ACK_OK);
            CyDelay(1000);
            FTDI_ENABLE_Write(0x00);
            CyDelay(1000);
            Bootloadable_Load();
            break;

//============================================================
CMD_HAND_CALIBRATE

        case CMD_HAND_CALIBRATE:
            calib.speed = (int16)(g_rx.buffer[1]<<8 | g_rx.buffer[2]);
            calib.repetitions = (int16)(g_rx.buffer[3]<<8 | g_rx.buffer[4]);
```

```
            if(calib.speed == -1 && calib.repetitions == -1) {
                calib.enabled = FALSE;
                calib.speed = 0;
                calib.repetitions = 0;
                break;
            }
            // Speed & repetitions saturations
            if (calib.speed < 0) {
                calib.speed = 0;
            } else if (calib.speed > 200) {
                calib.speed = 200;
            }
            if (calib.repetitions < 0) {
                calib.repetitions = 0;
            } else if (calib.repetitions > 32767) {
                calib.repetitions = 32767;
            }

            g_refNew[0].pos = 0;                    // SoftHand is on motor 1
            calib.enabled = TRUE;

            sendAcknowledgment(ACK_OK);
            break;

//=================================================      ↵
CMD_GET_IMU_READINGS

        case CMD_GET_IMU_READINGS:
            cmd_get_imu_readings();
            break;

//=================================================      CMD_GET_IMU_PARAM

        case CMD_GET_IMU_PARAM:
            get_IMU_param_list( __REV16(*((uint16 *) &g_rx.buffer[1])) );
            break;

//=================================================      ↵
CMD_GET_ENCODER_CONF

        case CMD_GET_ENCODER_CONF:
            cmd_get_encoder_map();
            break;

//=================================================      CMD_GET_ENCODER_RAW

        case CMD_GET_ENCODER_RAW:
            cmd_get_encoder_raw();
            break;

//=================================================      CMD_GET_ADC_CONF
```

```
        case CMD_GET_ADC_CONF:
            cmd_get_ADC_map();
            break;

//=====================================================     CMD_GET_ADC_RAW

        case CMD_GET_ADC_RAW:
            cmd_get_ADC_raw();
            break;

//=====================================================     ↵
CMD_GET_SD_SINGLE_FILE

        case CMD_GET_SD_SINGLE_FILE:
            cmd_get_SD_file( __REV16(*((uint16 *) &g_rx.buffer[1])) );
            break;

//=====================================================     ↵
CMD_REMOVE_SD_SINGLE_FILE

        case CMD_REMOVE_SD_SINGLE_FILE:
            cmd_remove_SD_file( __REV16(*((uint16 *) &g_rx.buffer[1])) );
            break;

//=========================================================== ALL OTHER ↵
COMMANDS
        default:
            break;

    }
}

//↵
//////////////////////////////////////////////////////////////////////////////
//                                                                      INFO ↵
SEND
//↵
//////////////////////////////////////////////////////////////////////////////

void infoSend(void){
    char packet_string[1500];

    prepare_generic_info(packet_string);
    UART_RS485_PutString(packet_string);
}


//↵
//////////////////////////////////////////////////////////////////////////////
//                                                            COMMAND GET ↵
INFO
//↵
```

```
================================================================================

void infoGet(uint16 info_type) {
    char CYDATA packet_string[4000] = "";
    char CYDATA str_sd_data[20000] = "";
    //==================================     choose info type and prepare ↵
string

    switch (info_type) {
        case INFO_ALL:
            prepare_generic_info(packet_string);
            UART_RS485_ClearTxBuffer();
            UART_RS485_PutString(packet_string);
            break;
        case CYCLES_INFO:
            prepare_counter_info(packet_string);
            UART_RS485_ClearTxBuffer();
            UART_RS485_PutString(packet_string);
            break;
        case GET_SD_PARAM:
            Read_SD_Closed_File(sdParam, packet_string, sizeof(packet_string));
            UART_RS485_ClearTxBuffer();
            UART_RS485_PutString(packet_string);
            break;
        case GET_SD_DATA:
            Read_SD_Current_Data(str_sd_data, sizeof(str_sd_data));
            UART_RS485_ClearTxBuffer();
            UART_RS485_PutString(str_sd_data);
            break;
        case GET_SD_FS_TREE:
            Get_SD_FS(str_sd_data);
            UART_RS485_ClearTxBuffer();
            UART_RS485_PutString(str_sd_data);
            break;
        case GET_SD_EMG_HIST:
            // Send every single byte inside the function, since it could be ↵
a large file to send
            Read_SD_EMG_History_Data();
            break;
        case GET_SD_R01_SUMM:
            Read_SD_Closed_File(sdR01File, packet_string, sizeof(packet_string↵
));
            UART_RS485_ClearTxBuffer();
            UART_RS485_PutString(packet_string);
            break;
        default:
            break;
    }
}


//↵
```

```
//=============================================================================
//                                                                  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_GET[NUM_OF_PARAMS], const char* MENU_STR[NUM_OF_PARAMS_MENU]){

    //Package to be sent variables
    uint8 packet_data[PARAM_BYTE_SLOT*NUM_OF_DEV_PARAMS + PARAM_MENU_SLOT*NUM_OF_DEV_PARAM_MENUS + PARAM_BYTE_SLOT] = "";      //50*NUM_OF_DEV_PARAMS + 150*NUM_OF_DEV_PARAM_MENUS
    uint16 packet_lenght = PARAM_BYTE_SLOT*NUM_OF_DEV_PARAMS + PARAM_MENU_SLOT*NUM_OF_DEV_PARAM_MENUS + PARAM_BYTE_SLOT;

    //Auxiliary variables
    uint8 CYDATA i, j;
    uint8 CYDATA idx = 0;       //Parameter number
    uint8 CYDATA idx_menu = 0;
    uint8 CYDATA sod = 0;       //sizeof data
    uint8 CYDATA string_lenght;
    char CYDATA aux_str[50] = "";
    float aux_float;
    int16 aux_int16;
    uint16 aux_uint16;
    int32 aux_int32;
    uint32 aux_uint32;

    uint8 MOTOR_IDX = 0;
    uint8 SECOND_MOTOR_IDX = 1;

    uint8* m_addr = (uint8*)VAR_P[0];
    uint8* m_tmp = m_addr;

    packet_data[0] = CMD_GET_PARAM_LIST;
    packet_data[1] = NUM_OF_DEV_PARAMS;

    for (idx = 0; idx < NUM_OF_DEV_PARAMS; idx++) {

        // Assign m_addr memory address
        m_addr = (uint8*)VAR_P[idx];

        // Add parameter type and size to packet
        packet_data[2 + PARAM_BYTE_SLOT*idx] = TYPES[idx];
        packet_data[3 + PARAM_BYTE_SLOT*idx] = NUM_ITEMS[idx];

        // Find size of data
```

```
        switch (TYPES[idx]) {
            case TYPE_FLAG: case TYPE_INT8: case TYPE_UINT8: case TYPE_STRING:
                sod = 1; break;
            case TYPE_INT16: case TYPE_UINT16:
                sod = 2; break;
            case TYPE_INT32: case TYPE_UINT32: case TYPE_FLOAT:
                sod = 4; break;
        }

        if (!CUSTOM_PARAM_GET[idx]) {       // Default param get

            // Add parameter data to packet
            switch (TYPES[idx]) {
                case TYPE_FLAG: case TYPE_UINT8: case TYPE_STRING:
                    for (i=0; i<NUM_ITEMS[idx]; i++){
                        m_tmp = m_addr + i*sod;
                        packet_data[4 + PARAM_BYTE_SLOT*idx + i*sod] = *m_tmp;
                    }
                    break;
                case TYPE_INT8:
                    for (i=0; i<NUM_ITEMS[idx]; i++){
                        m_tmp = m_addr + i*sod;
                        packet_data[4 + PARAM_BYTE_SLOT*idx + i*sod] = *m_tmp;
                    }
                    break;
                case TYPE_INT16:
                    for (i=0; i<NUM_ITEMS[idx]; i++){
                        m_tmp = m_addr + i*sod;
                        aux_int16 = *((int16*)m_tmp);
                        for(j = 0; j < sod; j++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx + i*sod) + ↵
sod - j -1] = ((char*)(&aux_int16))[j];
                        }
                    }
                    break;
                case TYPE_UINT16:
                    for (i=0; i<NUM_ITEMS[idx]; i++){
                        m_tmp = m_addr + i*sod;
                        aux_uint16 = *((uint16*)m_tmp);
                        for(j = 0; j < sod; j++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx + i*sod) + ↵
sod - j -1] = ((char*)(&aux_uint16))[j];
                        }
                    }
                    break;
                case TYPE_INT32:
                    for (i=0; i<NUM_ITEMS[idx]; i++){
                        m_tmp = m_addr + i*sod;
                        aux_int32 = *((int32*)m_tmp);
                        for(j = 0; j < sod; j++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx + i*sod) + ↵
sod - j -1] = ((char*)(&aux_int32))[j];
```

```
                        }
                    }
                    break;
                case TYPE_UINT32:
                    for (i=0; i<NUM_ITEMS[idx]; i++){
                        m_tmp = m_addr + i*sod;
                        aux_uint32 = *((uint32*)m_tmp);
                        for(j = 0; j < sod; j++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx + i*sod) + sod - j -1] = ((char*)(&aux_uint32))[j];
                        }
                    }
                    break;

                case TYPE_FLOAT:
                    for (i=0; i<NUM_ITEMS[idx]; i++){
                        m_tmp = m_addr + i*sod;
                        aux_float = *((float*)m_tmp);
                        for(j = 0; j < sod; j++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx + i*sod) + sod - j -1] = ((char*)(&aux_float))[j];
                        }
                    }
                    break;
            }
        }
        else {

// DO NOT MODIFY THE FUNCTION BEFORE THIS LINE

// MODIFY CUSTOM PARAM
                switch(idx+1){
                    case 2:         // Position PID
                        if(c_mem.motor[MOTOR_IDX].control_mode != CURR_AND_POS_CONTROL) {
                            aux_float = (float) c_mem.motor[MOTOR_IDX].k_p / 65536;
                            for(i = 0; i < sod; i++) {
                                packet_data[(4 + PARAM_BYTE_SLOT*idx) + sod - i -1] = ((char*)(&aux_float))[i];
                            }
                            aux_float = (float) c_mem.motor[MOTOR_IDX].k_i / 65536;
                            for(i = 0; i < sod; i++) {
                                packet_data[(4 + PARAM_BYTE_SLOT*idx + sod) + sod - i -1] = ((char*)(&aux_float))[i];
                            }
                            aux_float = (float) c_mem.motor[MOTOR_IDX].k_d / 65536;
                            for(i = 0; i < sod; i++) {
                                packet_data[(4 + PARAM_BYTE_SLOT*idx + 2*sod) + sod - i -1] = ((char*)(&aux_float))[i];
```

```
                        }
                    }
                    else {
                        aux_float = (float) c_mem.motor[MOTOR_IDX].k_p_dl ↵
/ 65536;
                        for(i = 0; i < sod; i++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx) + sod - ↵
i -1] = ((char*)(&aux_float))[i];
                        }
                        aux_float = (float) c_mem.motor[MOTOR_IDX].k_i_dl ↵
/ 65536;
                        for(i = 0; i < sod; i++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx + sod) + ↵
sod - i -1] = ((char*)(&aux_float))[i];
                        }
                        aux_float = (float) c_mem.motor[MOTOR_IDX].k_d_dl ↵
/ 65536;
                        for(i = 0; i < sod; i++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx + 2*sod) ↵
+ sod - i -1] = ((char*)(&aux_float))[i];
                        }
                    }
                    break;

                case 3:         //Current PID
                    if(c_mem.motor[MOTOR_IDX].control_mode != ↵
CURR_AND_POS_CONTROL) {
                        aux_float = (float) c_mem.motor[MOTOR_IDX].k_p_c ↵
/ 65536;
                        for(i = 0; i < sod; i++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx) + sod - ↵
i -1] = ((char*)(&aux_float))[i];
                        }
                        aux_float = (float) c_mem.motor[MOTOR_IDX].k_i_c ↵
/ 65536;
                        for(i = 0; i < sod; i++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx + sod) + ↵
sod - i -1] = ((char*)(&aux_float))[i];
                        }
                        aux_float = (float) c_mem.motor[MOTOR_IDX].k_d_c ↵
/ 65536;
                        for(i = 0; i < sod; i++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx + 2*sod) ↵
+ sod - i -1] = ((char*)(&aux_float))[i];
                        }
                    }
                    else {
                        aux_float = (float) c_mem.motor[MOTOR_IDX].↵
k_p_c_dl / 65536;
                        for(i = 0; i < sod; i++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx) + sod - ↵
i -1] = ((char*)(&aux_float))[i];
```

```
                        }
                        aux_float = (float) c_mem.motor[MOTOR_IDX].k_i_c_dl / 65536;
                        for(i = 0; i < sod; i++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx + sod) + sod - i -1] = ((char*)(&aux_float))[i];
                        }
                        aux_float = (float) c_mem.motor[MOTOR_IDX].k_d_c_dl / 65536;
                        for(i = 0; i < sod; i++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx + 2*sod) + sod - i -1] = ((char*)(&aux_float))[i];
                        }
                    }
                    break;

                case 8:         //Measurement Offset
                    for (i=0; i<NUM_ITEMS[idx]; i++){
                        aux_int16 = (c_mem.enc[g_mem.motor[MOTOR_IDX].encoder_line].m_off[i] >> c_mem.enc[g_mem.motor[MOTOR_IDX].encoder_line].res[i]);
                        for(j = 0; j < sod; j++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx + i*sod) + sod - j -1] = ((char*)(&aux_int16))[j];
                        }
                    }
                    break;

                case 11:        //Position limits
                    aux_int32 = (c_mem.motor[MOTOR_IDX].pos_lim_inf >> c_mem.enc[g_mem.motor[MOTOR_IDX].encoder_line].res[0]);
                    for(j = 0; j < sod; j++) {
                        packet_data[(4 + PARAM_BYTE_SLOT*idx) + sod - j -1] = ((char*)(&aux_int32))[j];
                    }
                    aux_int32 = (c_mem.motor[MOTOR_IDX].pos_lim_sup >> c_mem.enc[g_mem.motor[MOTOR_IDX].encoder_line].res[0]);
                    for(j = 0; j < sod; j++) {
                        packet_data[(4 + PARAM_BYTE_SLOT*idx + sod) + sod - j -1] = ((char*)(&aux_int32))[j];
                    }
                    break;

                case 23:        //Rest Position
                    aux_int32 = (c_mem.SH.rest_pos >> c_mem.enc[g_mem.motor[MOTOR_IDX].encoder_line].res[0]);
                    for(j = 0; j < sod; j++) {
                        packet_data[(4 + PARAM_BYTE_SLOT*idx) + sod - j -1] = ((char*)(&aux_int32))[j];
                    }
                    break;
```

```
                case 44:          // Second Motor Position PID
                    if(c_mem.motor[SECOND_MOTOR_IDX].control_mode != CURR_AND_POS_CONTROL) {
                        aux_float = (float) c_mem.motor[SECOND_MOTOR_IDX].k_p / 65536;
                        for(i = 0; i < sod; i++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx) + sod - i -1] = ((char*)(&aux_float))[i];
                        }
                        aux_float = (float) c_mem.motor[SECOND_MOTOR_IDX].k_i / 65536;
                        for(i = 0; i < sod; i++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx + sod) + sod - i -1] = ((char*)(&aux_float))[i];
                        }
                        aux_float = (float) c_mem.motor[SECOND_MOTOR_IDX].k_d / 65536;
                        for(i = 0; i < sod; i++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx + 2*sod) + sod - i -1] = ((char*)(&aux_float))[i];
                        }
                    }
                    else {
                        aux_float = (float) c_mem.motor[SECOND_MOTOR_IDX].k_p_dl / 65536;
                        for(i = 0; i < sod; i++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx) + sod - i -1] = ((char*)(&aux_float))[i];
                        }
                        aux_float = (float) c_mem.motor[SECOND_MOTOR_IDX].k_i_dl / 65536;
                        for(i = 0; i < sod; i++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx + sod) + sod - i -1] = ((char*)(&aux_float))[i];
                        }
                        aux_float = (float) c_mem.motor[SECOND_MOTOR_IDX].k_d_dl / 65536;
                        for(i = 0; i < sod; i++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx + 2*sod) + sod - i -1] = ((char*)(&aux_float))[i];
                        }
                    }
                    break;

                case 45:          // Second Motor Current PID
                    if(c_mem.motor[SECOND_MOTOR_IDX].control_mode != CURR_AND_POS_CONTROL) {
                        aux_float = (float) c_mem.motor[SECOND_MOTOR_IDX].k_p_c / 65536;
                        for(i = 0; i < sod; i++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx) + sod - i -1] = ((char*)(&aux_float))[i];
```

```
                        }
                        aux_float = (float) c_mem.motor[SECOND_MOTOR_IDX].k_i_c / 65536;
                        for(i = 0; i < sod; i++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx + sod) + sod - i -1] = ((char*)(&aux_float))[i];
                        }
                        aux_float = (float) c_mem.motor[SECOND_MOTOR_IDX].k_d_c / 65536;
                        for(i = 0; i < sod; i++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx + 2*sod) + sod - i -1] = ((char*)(&aux_float))[i];
                        }
                    }
                    else {
                        aux_float = (float) c_mem.motor[SECOND_MOTOR_IDX].k_p_c_dl / 65536;
                        for(i = 0; i < sod; i++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx) + sod - i -1] = ((char*)(&aux_float))[i];
                        }
                        aux_float = (float) c_mem.motor[SECOND_MOTOR_IDX].k_i_c_dl / 65536;
                        for(i = 0; i < sod; i++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx + sod) + sod - i -1] = ((char*)(&aux_float))[i];
                        }
                        aux_float = (float) c_mem.motor[SECOND_MOTOR_IDX].k_d_c_dl / 65536;
                        for(i = 0; i < sod; i++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx + 2*sod) + sod - i -1] = ((char*)(&aux_float))[i];
                        }
                    }
                    break;

                case 50:          // Second Motor Measurement Offset
                    aux_int16 = (c_mem.enc[g_mem.motor[SECOND_MOTOR_IDX].encoder_line].m_off[i] >> c_mem.enc[g_mem.motor[SECOND_MOTOR_IDX].encoder_line].res[i]);
                    for(j = 0; j < sod; j++) {
                        packet_data[(4 + PARAM_BYTE_SLOT*idx + i*sod) + sod - j -1] = ((char*)(&aux_int16))[j];
                    }
                    break;

                case 53:         // Second Motor Position limits
                    aux_int32 = (c_mem.motor[SECOND_MOTOR_IDX].pos_lim_inf >> c_mem.enc[g_mem.motor[SECOND_MOTOR_IDX].encoder_line].res[0]);
                    for(j = 0; j < sod; j++) {
                        packet_data[(4 + PARAM_BYTE_SLOT*idx) + sod - j -1] = ((char*)(&aux_int32))[j];
```

```
                        }
                        aux_int32 = (c_mem.motor[SECOND_MOTOR_IDX].
pos_lim_sup >> c_mem.enc[g_mem.motor[SECOND_MOTOR_IDX].encoder_line].res[0]);
                        for(j = 0; j < sod; j++) {
                            packet_data[(4 + PARAM_BYTE_SLOT*idx + sod) + sod 
- j -1] = ((char*)(&aux_int32))[j];
                        }
                        break;

                    default:
                        break;
                }
// END OF MODIFY CUSTOM PARAM

// DO NOT MODIFY THE FUNCTION UNDER THIS LINE
        }

        sprintf(aux_str, (char*)PARAMS_STR[idx]);
        string_lenght = strlen(aux_str);

        // Parameters with a menu
        if (TYPES[idx] == TYPE_FLAG){
            switch(NUM_MENU[idx_menu]){
                case 1:     // input mode menu
                    switch(*m_addr) {
                        case INPUT_MODE_EXTERNAL:
                            strcat(aux_str, " Usb");
                        break;
                        case INPUT_MODE_ENCODER3:
                            strcat(aux_str, " Handle");
                        break;
                        case INPUT_MODE_EMG_PROPORTIONAL:
                            strcat(aux_str, " EMG proportional");
                        break;
                        case INPUT_MODE_EMG_INTEGRAL:
                            strcat(aux_str, " EMG integral");
                        break;
                        case INPUT_MODE_EMG_FCFS:
                            strcat(aux_str, " EMG FCFS");
                        break;
                        case INPUT_MODE_EMG_FCFS_ADV:
                            strcat(aux_str, " EMG FCFS Advanced");
                        break;
                        case INPUT_MODE_JOYSTICK:
                            strcat(aux_str, " Joystick");
                        break;
                        case INPUT_MODE_EMG_PROPORTIONAL_NC:
                            strcat(aux_str, " EMG proportional Normally Closed
");
                        break;
                    }
                    break;
```

```
            case 2:     // control mode menu
                switch(*m_addr){
                    case CONTROL_ANGLE:
                        strcat(aux_str, " Position");
                    break;
                    case CONTROL_PWM:
                        strcat(aux_str, " PWM");
                    break;
                    case CONTROL_CURRENT:
                        strcat(aux_str, " Current");
                    break;
                    case CURR_AND_POS_CONTROL:
                        strcat(aux_str, " Position and Current");
                    break;
                }
                break;
            case 3:     // yes/no menu
                if(*m_addr){
                    strcat(aux_str, " YES\0");
                }
                else {
                    strcat(aux_str, " NO\0");
                }
                break;
            case 4:     // right/lef menu
                switch(*m_addr){
                    case RIGHT_HAND:
                        strcat(aux_str, " Right\0");
                    break;
                    case LEFT_HAND:
                        strcat(aux_str, " Left\0");
                    break;
                }
                break;
            case 5:     // on/off menu
                switch(*m_addr){
                    case 0:
                        strcat(aux_str, " OFF\0");
                    break;
                    case 1:
                        strcat(aux_str, " ON\0");
                    break;
                }
                break;
            case 6:     // expansion port menu
                switch(*m_addr){
                    case EXP_NONE:
                        strcat(aux_str, " None\0");
                        break;
                    case EXP_SD_RTC:
                        strcat(aux_str, " SD/RTC board\0");
                        break;
```

```
                case EXP_WIFI:
                    strcat(aux_str, " WiFi board [N/A]\0");
                    break;
                case EXP_OTHER:
                    strcat(aux_str, " Other [N/A]\0");
                    break;
            }
            break;
        case 7:     // spi read delay menu
            switch(*m_addr){
                case 0:
                    strcat(aux_str, " None\0");
                    break;
                case 1:
                    strcat(aux_str, " Low\0");
                    break;
                case 2:
                    strcat(aux_str, " High\0");
                    break;
                default:
                    break;
            }
            break;
        case 8:     // user menu
            switch(*m_addr){
                case GENERIC_USER:
                    strcat(aux_str, " GENERIC USER\0");
                break;
                case MARIA:
                    strcat(aux_str, " MARIA\0");
                break;
                case R01:
                    strcat(aux_str, " R01\0");
                break;
            }
            break;
        case 9:     // driver type menu
            switch(*m_addr){
                case 0:
                    strcat(aux_str, " MC33887 (Standard)\0");
                break;
                case 1:
                    strcat(aux_str, " VNH5019 (High power)\0");
                break;
                case 2:
                    strcat(aux_str, " ESC (Brushless)\0");
                break;
            }
            break;
        case 10:    // device type menu
            switch(*m_addr){
                case 0:
```

```
                    strcat(aux_str, " SOFTHAND PRO\0");
                break;
                case 1:
                    strcat(aux_str, " GENERIC 2 MOTORS\0");
                break;
                case 2:
                    strcat(aux_str, " AIR CHAMBERS\0");
                break;
                case 3:
                    strcat(aux_str, " OTTOBOCK WRIST\0");
                break;
                case 4:
                    strcat(aux_str, " SOFTHAND 2 MOTORS\0");
                break;
            }
            break;
        case 11:    // fsm activation mode menu
            switch(*m_addr){
                case 0:
                    if (c_mem.dev.dev_type == SOFTHAND_2_MOTORS){
                        strcat(aux_str, " Fast:syn2, Slow:syn1\0");
                    }
                    else {
                        strcat(aux_str, " Fast:wrist,Slow:hand\0");
                    }
                break;
                case 1:
                    if (c_mem.dev.dev_type == SOFTHAND_2_MOTORS){
                        strcat(aux_str, " Slow:syn2, Fast:syn1\0");
                    }
                    else {
                        strcat(aux_str, " Slow:wrist,Fast:hand\0");
                    }
                break;
            }
            break;
        case 12:    // wrist direction association menu
            switch(*m_addr){
                case 0:
                    strcat(aux_str, " Close:CW, Open:CCW\0");
                break;
                case 1:
                    strcat(aux_str, " Close:CCW, Open:CW\0");
                break;
            }
            break;
    }
    //Recomputes string lenght
    string_lenght = strlen(aux_str)+1;
}

// Add parameter string to packet
```

```
        for(i = string_lenght; i != 0; i--)
            packet_data[(4 + PARAM_BYTE_SLOT*idx) + (sod*NUM_ITEMS[idx]) + string_lenght - i] = aux_str[string_lenght - i];
        //The following byte indicates the number of menus at the end of the packet to send
        if (TYPES[idx] == TYPE_FLAG){
            packet_data[(4 + PARAM_BYTE_SLOT*idx) + (sod*NUM_ITEMS[idx]) + string_lenght] = NUM_MENU[idx_menu];
            idx_menu = idx_menu + 1;
        }

        // Add struct index after an empty bit
        // Note: added here at the end of packets is transparent to old parameters retrieving version
        if (TYPES[idx] == TYPE_FLAG){
            packet_data[(4 + PARAM_BYTE_SLOT*idx) + (sod*NUM_ITEMS[idx]) + string_lenght + 2] = NUM_STRUCT[idx];
        }
        else {
            packet_data[(4 + PARAM_BYTE_SLOT*idx) + (sod*NUM_ITEMS[idx]) + string_lenght + 1] = NUM_STRUCT[idx];
        }
    }

    // Add menu
    for (j = 0; j < NUM_OF_DEV_PARAM_MENUS; j++) {
        string_lenght = strlen((char*)MENU_STR[j]);
        for(i = string_lenght; i != 0; i--)
            packet_data[PARAM_BYTE_SLOT*NUM_OF_DEV_PARAMS + 2 + j*PARAM_MENU_SLOT + string_lenght - i] = MENU_STR[j][string_lenght - i];
    }

    packet_data[packet_lenght - 1] = LCRChecksum(packet_data,packet_lenght - 1);
    commWrite(packet_data, packet_lenght);
}

//
//////////////////////////////////////////////////////////////////////////////
//                                                                  MANAGE PARAM LIST
//
//////////////////////////////////////////////////////////////////////////////

void manage_param_list(uint16 index) {
    uint8 CYDATA i, j;
    uint8 CYDATA sod;
    uint8 PARAM_IDX;
    int16 aux_int16;
    uint16 aux_uint16;
    int32 aux_int32;
    uint32 aux_uint32;
```

```
    float aux_float;

    uint8 MOTOR_IDX = 0;
    uint8 SECOND_MOTOR_IDX = 1;

    // Arrays
    struct st_eeprom* MEM_P = &c_mem;   // c_mem is used for param reading

    if (index){                         // Switch from c_mem to g_mem
        MEM_P = &g_mem;                 // g_mem is used for param setting
    }


//------------------- BEGIN OF PARAMETERS VARIABLES --------------------//
    uint8* VAR_P[NUM_OF_PARAMS] = {
        (uint8*)&(MEM_P->dev.id)↵
,
        (uint8*)&(MEM_P->motor[MOTOR_IDX].k_p),
        (uint8*)&(MEM_P->motor[MOTOR_IDX].k_p_c),
        (uint8*)&(MEM_P->motor[MOTOR_IDX].activ),
        (uint8*)&(MEM_P->motor[MOTOR_IDX].input_mode),
        (uint8*)&(MEM_P->motor[MOTOR_IDX].control_mode),
        (uint8*)&(MEM_P->enc[MEM_P->motor[MOTOR_IDX].encoder_line].res),
        (uint8*)&(MEM_P->enc[MEM_P->motor[MOTOR_IDX].encoder_line].m_off[0]),
        (uint8*)&(MEM_P->enc[MEM_P->motor[MOTOR_IDX].encoder_line].m_mult[0]),
        (uint8*)&(MEM_P->motor[MOTOR_IDX].pos_lim_flag)↵
,                                   //10
        (uint8*)&(MEM_P->motor[MOTOR_IDX].pos_lim_inf),
        (uint8*)&(MEM_P->motor[MOTOR_IDX].max_step_neg),
        (uint8*)&(MEM_P->motor[MOTOR_IDX].current_limit),
        (uint8*)&(MEM_P->emg.emg_threshold[0]),
        (uint8*)&(MEM_P->emg.emg_calibration_flag),
        (uint8*)&(MEM_P->emg.emg_max_value[0]),
        (uint8*)&(MEM_P->emg.emg_speed[0]),
        (uint8*)&(MEM_P->enc[MEM_P->motor[MOTOR_IDX].encoder_line].↵
double_encoder_on_off),
        (uint8*)&(MEM_P->enc[MEM_P->motor[MOTOR_IDX].encoder_line].↵
motor_handle_ratio),
        (uint8*)&(MEM_P->motor[MOTOR_IDX].activate_pwm_rescaling)↵
,                           //20
        (uint8*)&(MEM_P->motor[MOTOR_IDX].curr_lookup[0]),
        (uint8*)&(MEM_P->dev.hw_maint_date),
        (uint8*)&(MEM_P->SH.rest_pos),
        (uint8*)&(MEM_P->SH.rest_delay),
        (uint8*)&(MEM_P->SH.rest_vel),
        (uint8*)&(MEM_P->SH.rest_position_flag),
        (uint8*)&(MEM_P->emg.switch_emg),
        (uint8*)&(MEM_P->dev.right_left),
        (uint8*)&(MEM_P->imu.read_imu_flag),
        (uint8*)&(MEM_P->exp.read_exp_port_flag)↵
,                                   //30
        (uint8*)&(MEM_P->dev.reset_counters),
```

```
        (uint8*)&(MEM_P->exp.curr_time[0]),
        (uint8*)&(MEM_P->imu.SPI_read_delay),
        (uint8*)&(MEM_P->imu.IMU_conf[0][0]),
        (uint8*)&(MEM_P->dev.user_id),
        (uint8*)&(MEM_P->user[MEM_P->dev.user_id].user_code_string),

        // GENERIC PARAMS
        (uint8*)&(MEM_P->motor[MOTOR_IDX].encoder_line),            // other params of 1st motor
        (uint8*)&(MEM_P->motor[MOTOR_IDX].motor_driver_type),
        (uint8*)&(MEM_P->motor[MOTOR_IDX].pwm_rate_limiter),
        (uint8*)&(MEM_P->motor[MOTOR_IDX].not_revers_motor_flag),   //40
        (uint8*)&(MEM_P->enc[MEM_P->motor[MOTOR_IDX].encoder_line].Enc_idx_use_for_control),
        (uint8*)&(MEM_P->enc[MEM_P->motor[MOTOR_IDX].encoder_line].gears_params),
        (uint8*)&(MEM_P->dev.use_2nd_motor_flag),                   // second motor config and params
        (uint8*)&(MEM_P->motor[SECOND_MOTOR_IDX].k_p),
        (uint8*)&(MEM_P->motor[SECOND_MOTOR_IDX].k_p_c),
        (uint8*)&(MEM_P->motor[SECOND_MOTOR_IDX].activ),
        (uint8*)&(MEM_P->motor[SECOND_MOTOR_IDX].input_mode),
        (uint8*)&(MEM_P->motor[SECOND_MOTOR_IDX].control_mode),
        (uint8*)&(MEM_P->enc[MEM_P->motor[SECOND_MOTOR_IDX].encoder_line].res),
        (uint8*)&(MEM_P->enc[MEM_P->motor[SECOND_MOTOR_IDX].encoder_line].m_off[0]),    //50
        (uint8*)&(MEM_P->enc[MEM_P->motor[SECOND_MOTOR_IDX].encoder_line].m_mult[0]),
        (uint8*)&(MEM_P->motor[SECOND_MOTOR_IDX].pos_lim_flag),
        (uint8*)&(MEM_P->motor[SECOND_MOTOR_IDX].pos_lim_inf),
        (uint8*)&(MEM_P->motor[SECOND_MOTOR_IDX].max_step_neg),
        (uint8*)&(MEM_P->motor[SECOND_MOTOR_IDX].current_limit),
        (uint8*)&(MEM_P->enc[MEM_P->motor[SECOND_MOTOR_IDX].encoder_line].double_encoder_on_off),
        (uint8*)&(MEM_P->enc[MEM_P->motor[SECOND_MOTOR_IDX].encoder_line].motor_handle_ratio),
        (uint8*)&(MEM_P->motor[SECOND_MOTOR_IDX].activate_pwm_rescaling),
        (uint8*)&(MEM_P->motor[SECOND_MOTOR_IDX].curr_lookup[0]),
        (uint8*)&(MEM_P->motor[SECOND_MOTOR_IDX].encoder_line),         //60
        (uint8*)&(MEM_P->motor[SECOND_MOTOR_IDX].motor_driver_type),
        (uint8*)&(MEM_P->motor[SECOND_MOTOR_IDX].pwm_rate_limiter),
        (uint8*)&(MEM_P->motor[SECOND_MOTOR_IDX].not_revers_motor_flag),
        (uint8*)&(MEM_P->enc[MEM_P->motor[SECOND_MOTOR_IDX].encoder_line].Enc_idx_use_for_control),
        (uint8*)&(MEM_P->enc[MEM_P->motor[SECOND_MOTOR_IDX].encoder_line].gears_params),

        (uint8*)&(MEM_P->enc[0].Enc_raw_read_conf[0]),                  // additional generic params
        (uint8*)&(MEM_P->enc[1].Enc_raw_read_conf[0]),
        (uint8*)&(MEM_P->exp.read_ADC_sensors_port_flag),
        (uint8*)&(MEM_P->exp.ADC_conf[0]),
```

```
        (uint8*)&(MEM_P->exp.ADC_conf[6]),                              //70
        (uint8*)&(MEM_P->exp.record_EMG_history_on_SD),
        (uint8*)&(MEM_P->JOY_spec.joystick_closure_speed),
        (uint8*)&(MEM_P->JOY_spec.joystick_threshold),
        (uint8*)&(MEM_P->JOY_spec.joystick_gains[0]),
        (uint8*)&(MEM_P->dev.dev_type),
        (uint8*)&(MEM_P->WR.activation_mode),
        (uint8*)&(MEM_P->WR.fast_act_threshold[0]),

        (uint8*)&(MEM_P->WR.wrist_direction_association),               // ↵
additional wrist params

        (uint8*)&(MEM_P->MS.slave_comm_active),                         // ↵
additional master params
        (uint8*)&(MEM_P->MS.slave_ID),

        (uint8*)&(MEM_P->FB.max_residual_current),                      // ↵
additional feedback params
        (uint8*)&(MEM_P->FB.maximum_pressure_kPa),
        (uint8*)&(MEM_P->FB.prop_err_fb_gain)


    };

    uint8 TYPES[NUM_OF_PARAMS] = {
        TYPE_UINT8, TYPE_FLOAT, TYPE_FLOAT, TYPE_FLAG,
        TYPE_FLAG, TYPE_FLAG, TYPE_UINT8, TYPE_INT16,
        TYPE_FLOAT, TYPE_FLAG, TYPE_INT32, TYPE_INT32,
        TYPE_INT16, TYPE_UINT16, TYPE_FLAG, TYPE_UINT32,
        TYPE_UINT8, TYPE_FLAG, TYPE_INT8, TYPE_FLAG,
        TYPE_FLOAT, TYPE_UINT8, TYPE_INT32, TYPE_INT32,
        TYPE_INT32, TYPE_FLAG, TYPE_FLAG, TYPE_FLAG,
        TYPE_FLAG, TYPE_FLAG, TYPE_FLAG, TYPE_UINT8,
        TYPE_FLAG, TYPE_UINT8, TYPE_FLAG, TYPE_STRING,

        // GENERIC PARAMS
        TYPE_UINT8, TYPE_FLAG, TYPE_UINT8, TYPE_FLAG,
        TYPE_UINT8, TYPE_INT8, TYPE_FLAG, TYPE_FLOAT,
        TYPE_FLOAT, TYPE_FLAG, TYPE_FLAG, TYPE_FLAG,
        TYPE_UINT8, TYPE_INT16, TYPE_FLOAT, TYPE_FLAG,
        TYPE_INT32, TYPE_INT32, TYPE_INT16, TYPE_FLAG,
        TYPE_INT8,  TYPE_FLAG, TYPE_FLOAT, TYPE_UINT8,
        TYPE_FLAG,  TYPE_UINT8, TYPE_FLAG, TYPE_UINT8,
        TYPE_INT8, TYPE_UINT8, TYPE_UINT8, TYPE_FLAG,
        TYPE_UINT8, TYPE_UINT8, TYPE_FLAG, TYPE_UINT16,
        TYPE_INT16, TYPE_UINT16, TYPE_FLAG, TYPE_FLAG,
        TYPE_UINT16,

                    TYPE_FLAG, TYPE_FLAG, TYPE_UINT8,
        TYPE_INT32, TYPE_FLOAT, TYPE_FLOAT
    };
```

```
    uint8 NUM_ITEMS[NUM_OF_PARAMS] = {
        1, 3, 3, 1,
        1, 1, 3, 3,
        3, 1, 2, 2,
        1, 2, 1, 2,
        2, 1, 1, 1,
        6, 3, 1, 1,
        1, 1, 1, 1,
        1, 1, 1, 6,
        1, 5, 1, 6,

        // GENERIC PARAMS
        1, 1, 1, 1,
        3, 3, 1, 3,
        3, 1, 1, 1,
        3, 3, 3, 1,
        2, 2, 1, 1,
        1, 1, 6, 1,
        1, 1, 1, 3,
        3, N_Encoder_Line_Connected[0], N_Encoder_Line_Connected[1], 1,
        6, 6, 1, 1,
        1, 2, 1, 1,
        2,

           1, 1, 1,
        1, 1, 1
    };

    uint8 NUM_STRUCT[NUM_OF_PARAMS] = {     // see STRUCTURES INDEX in globals.h
        ST_DEVICE, ST_MOTOR+MOTOR_IDX, ST_MOTOR+MOTOR_IDX, ST_MOTOR+MOTOR_IDX,
        ST_MOTOR+MOTOR_IDX, ST_MOTOR+MOTOR_IDX, ST_ENCODER+(MEM_P->motor[MOTOR_IDX].encoder_line), ST_ENCODER+(MEM_P->motor[MOTOR_IDX].encoder_line),
        ST_ENCODER+(MEM_P->motor[MOTOR_IDX].encoder_line), ST_MOTOR+MOTOR_IDX, ST_MOTOR+MOTOR_IDX, ST_MOTOR+MOTOR_IDX,
        ST_MOTOR+MOTOR_IDX, ST_EMG, ST_EMG, ST_EMG,
        ST_EMG, ST_ENCODER+(MEM_P->motor[MOTOR_IDX].encoder_line), ST_ENCODER+(MEM_P->motor[MOTOR_IDX].encoder_line), ST_MOTOR+MOTOR_IDX,
        ST_MOTOR+MOTOR_IDX, ST_DEVICE, ST_SH_SPEC, ST_SH_SPEC,
        ST_SH_SPEC, ST_SH_SPEC, ST_EMG, ST_DEVICE,
        ST_IMU, ST_EXPANSION, ST_DEVICE, ST_EXPANSION,
        ST_IMU, ST_IMU, ST_DEVICE, ST_USER+(MEM_P->dev.user_id),

        // GENERIC PARAMS
        ST_MOTOR+MOTOR_IDX, ST_MOTOR+MOTOR_IDX, ST_MOTOR+MOTOR_IDX, ST_MOTOR+MOTOR_IDX,
        ST_ENCODER+(MEM_P->motor[MOTOR_IDX].encoder_line), ST_ENCODER+(MEM_P->motor[MOTOR_IDX].encoder_line), ST_DEVICE, ST_MOTOR+SECOND_MOTOR_IDX,
        ST_MOTOR+SECOND_MOTOR_IDX, ST_MOTOR+SECOND_MOTOR_IDX, ST_MOTOR+SECOND_MOTOR_IDX, ST_MOTOR+SECOND_MOTOR_IDX,
        ST_ENCODER+(MEM_P->motor[SECOND_MOTOR_IDX].encoder_line), ST_ENCODER+(MEM_P->motor[SECOND_MOTOR_IDX].encoder_line), ST_ENCODER+(MEM_P->motor[
```

```
SECOND_MOTOR_IDX].encoder_line), ST_MOTOR+SECOND_MOTOR_IDX,
        ST_MOTOR+SECOND_MOTOR_IDX, ST_MOTOR+SECOND_MOTOR_IDX, ST_MOTOR
+SECOND_MOTOR_IDX, ST_ENCODER+(MEM_P->motor[SECOND_MOTOR_IDX].encoder_line),
        ST_ENCODER+(MEM_P->motor[SECOND_MOTOR_IDX].encoder_line), ST_MOTOR
+SECOND_MOTOR_IDX, ST_MOTOR+SECOND_MOTOR_IDX, ST_MOTOR+SECOND_MOTOR_IDX,
        ST_MOTOR+SECOND_MOTOR_IDX, ST_MOTOR+SECOND_MOTOR_IDX, ST_MOTOR
+SECOND_MOTOR_IDX, ST_ENCODER+(MEM_P->motor[SECOND_MOTOR_IDX].encoder_line),
        ST_ENCODER+(MEM_P->motor[SECOND_MOTOR_IDX].encoder_line), ST_ENCODER+0
, ST_ENCODER+1, ST_EXPANSION,
        ST_EXPANSION, ST_EXPANSION, ST_EXPANSION, ST_JOY_SPEC,
        ST_JOY_SPEC, ST_JOY_SPEC, ST_DEVICE, ST_WR_SPEC,
        ST_WR_SPEC,

                ST_WR_SPEC, ST_MS_SPEC, ST_MS_SPEC,
        ST_FB_SPEC, ST_FB_SPEC, ST_FB_SPEC

    };

    const char* PARAMS_STR[NUM_OF_PARAMS] = {
        "1 - Device ID:", "2 - Position PID [P, I, D]:", "3 - Current PID [P,
I, D]:", "4 - Startup Activation:",
        "5 - Input mode:", "6 - Control mode:", "7 - Resolutions:", "8 -
Measurement Offsets:",
        "9 - Multipliers:", "10 - Pos. limit active:", "11 - Pos. limits [inf
, sup]:", "12 - Max steps [neg, pos]:",
        "13 - Current limit:", "14 - EMG thresholds:", "15 - EMG calibration
on startup:", "16 - EMG max values:",
        "17 - EMG max speeds:", "18 - Absolute encoder position:", "19 -
Motor handle ratio:", "20 - PWM rescaling:",
        "21 - Current lookup:", "22 - Date of maintenance [D/M/Y]:", "23 -
Rest position:", "24 - Rest position time delay (ms):",
        "25 - Rest vel closure (ticks/sec):", "26 - Rest position enabled:", "
27 - EMG inversion:",  "28 - Hand side:",
        "29 - Enable IMUs:", "30 - Read Expansion port:", "31 - Reset counters
:", "32 - Last checked Time [D/M/Y H:M:S]:",
        "33 - SPI read delay (IMU):", "34 - On board IMU conf. [a,g,m,q,t]:",
"35 - User ID:", "36 - User code:",

        // GENERIC PARAMS
        "37 - Associated encoder line:", "38 - Driver type:", "39 - PWM rate
limiter:", "40 - Not reversible:",
        "41 - Enc idx used for control:", "42 - Gear params[N1, N2, I1]:", "
43 - Use second motor:", "44 - Position PID [P, I, D]:",
        "45 - Current PID [P, I, D]:", "46 - Startup Activation:", "47 -
Input mode:", "48 - Control mode:",
        "49 - Resolutions:", "50 - Measurement Offsets:", "51 - Multipliers:"
, "52 - Pos. limit active:",
        "53 - Pos. limits [inf, sup]:", "54 - Max steps [neg, pos]:", "55 -
Current limit:", "56 - Absolute encoder position:",
        "57 - Motor handle ratio:", "58 - PWM rescaling:", "59 - Current
lookup:", "60 - Associated encoder line:",
        "61 - Driver type:", "62 - PWM rate limiter:", "63 - Not reversible:"
```

```
, "64 - Enc idx used for control:",
        "65 - Gear params[N1, N2, I1]:", "66 - Read enc raw line 0:", "67 - Read enc raw line 1:", "68 - Read additional ADC port:",
        "69 - ADC channel [1-6]:", "70 - ADC channel [7-12]:", "71 - Record EMG on SD card:", "72 - Joystick closure speed:",
        "73 - Joystick threshold:", "74 - Joystick gains:", "75 - Device type:", "76 - EMG FSM act.mode:",
        "77 - Fast act.thresholds:",
                                                "78 - Wrist direction:", "79 - Slave communication active:", "80 - Slave ID:",
        "81 - Maximum slave residual current:", "82 - Maximum pressure feedback (kPa):", "83 - Proportional pressure error gain:",
    };

    //Parameters menu
    char spi_delay_menu[118]    = "";
    sprintf(spi_delay_menu, "0 -> None\n1 -> Low (%u us delay for each 8-bit register read)\n2 -> High (%u us delay for each 8-bit register read)\n", (int)SPI_DELAY_LOW, (int)SPI_DELAY_HIGH);

    char fsm_activation_mode_menu[56] = "";
    if (MEM_P->dev.dev_type == SOFTHAND_2_MOTORS){
        sprintf(fsm_activation_mode_menu, "0 -> Fast:syn2, Slow:syn1\n1 -> Slow:syn2, Fast:syn1\n");
    }
    else {
        sprintf(fsm_activation_mode_menu, "0 -> Fast:wrist,Slow:hand\n1 -> Slow:wrist,Fast:hand\n");
    }

    const char* MENU_STR[NUM_OF_PARAMS_MENU] = {
        "0 -> Usb\n1 -> Handle\n2 -> EMG proportional\n3 -> EMG Integral\n4 -> EMG FCFS\n5 -> EMG FCFS Advanced\n6 -> Joystick\n7 -> EMG proportional NC\n",         //1 input_mode_menu
        "0 -> Position\n1 -> PWM\n2 -> Current\n3 -> Position and Current\n",                                                  //2 control_mode_menu
        "0 -> Deactivate [NO]\n1 -> Activate [YES]\n",                                                                                                  //3 yes_no_menu
        "0 -> Right\n1 -> Left\n",
                                                                                                                                                        //4 right_left_menu
        "0 -> OFF\n1 -> ON\nThe board will reset\n",                                                                                                    //5 on_off_menu
        "0 -> None\n1 -> SD/RTC board\n2 -> WiFi board [N/A]\n3 -> Other [N/A]\nThe board will reset\n",                      //6 exp_port_menu
        spi_delay_menu
                                                                                                                                                        //7 spi_delay_menu
        "0 -> Generic user\n1 -> Maria\n2 -> R01\nThe board will reset\n"
```

```
,                                                   //8 user_id_menu
        "0 -> MC33887 (Standard)\n1 -> VNH5019 (High power)\n2 -> ESC (↵
Brushless)\nThe board will reset\n",                                    //↵
9 motor_driver_type_menu
        "0 -> SOFTHAND PRO\n1 -> GENERIC 2 MOTORS\n2 -> AIR CHAMBERS\n3 -> ↵
OTTOBOCK WRIST\n4 -> SOFTHAND 2 MOTORS\nThe board will reset\n",        //10 ↵
device_type_menu
        fsm_activation_mode_menu↵
,                                                                       ↵
//11 fsm_activation_mode_menu
        "0 -> Close:CW, Open:CCW\n1 -> Close:CCW, Open:CW\n↵
"                                                   //12 wrist_direction_menu
    };

    uint8 NUM_MENU[32] = {3, 1, 2, 3, 3, 3, 3, 3, 3, 4, 5, 6, 3, 7, 8, 9, 3, 5↵
, 3, 1, 2, 3, 3, 3, 9, 3, 5, 3, 10, 11, 12, 3};
    uint8 CUSTOM_PARAM_GET_LIST[9]  = {2, 3, 8, 11, 23, 44, 45, 50, 53};
    uint8 CUSTOM_PARAM_SET_LIST[18] = {2, 3, 5, 8, 11, 23, 24, 28, 31, 32, 38↵
, 44, 45, 47, 50, 53, 61, 75};
    uint8 USER_ID_PARAM = 35;

// Note: If a custom parameter change is needed, add to CUSTOM_PARAM_LIST, ↵
then change it
// in the dedicated function set_custom_param()

//------------------ END OF PARAMETERS VARIABLES --------------------//

// DO NOT MODIFY THE FUNCTION UNDER THIS LINE

    uint8 CUSTOM_PARAM_GET[NUM_OF_PARAMS];
    j = 0;
    for (i=0; i<NUM_OF_PARAMS; i++) {
        if (CUSTOM_PARAM_GET_LIST[j] == i+1) {
            CUSTOM_PARAM_GET[i] = TRUE;
            j++;
        }
        else {
            CUSTOM_PARAM_GET[i] = FALSE;
        }
    }   // All parameters can be get with default settings, except the ↵
following ones
    uint8 CUSTOM_PARAM_SET[NUM_OF_PARAMS];
    j = 0;
    for (i=0; i<NUM_OF_PARAMS; i++) {
        if (CUSTOM_PARAM_SET_LIST[j] == i+1) {
            CUSTOM_PARAM_SET[i] = TRUE;
            j++;
        }
        else {
            CUSTOM_PARAM_SET[i] = FALSE;
        }
    }   // All parameters can be setted with default settings, except the ↵
```

```
following ones

    if (!index) {
        // Get parameters list with relative types
        get_param_list(VAR_P, TYPES, NUM_ITEMS, NUM_STRUCT, NUM_MENU, ↵
PARAMS_STR, CUSTOM_PARAM_GET, MENU_STR);
    }
    else {
        // Set specific parameter
        PARAM_IDX = index -1;       // Get right vector param index

        // Find size of data
        switch (TYPES[PARAM_IDX]) {
            case TYPE_FLAG: case TYPE_INT8: case TYPE_UINT8: case TYPE_STRING:
                sod = 1; break;
            case TYPE_INT16: case TYPE_UINT16:
                sod = 2; break;
            case TYPE_INT32: case TYPE_UINT32: case TYPE_FLOAT:
                sod = 4; break;
        }

        if (!CUSTOM_PARAM_SET[PARAM_IDX]) {
            // Use default specifications for param setting
            switch(TYPES[PARAM_IDX]) {
                case TYPE_FLAG: case TYPE_UINT8:
                    for (i=0; i<NUM_ITEMS[PARAM_IDX]; i++){
                        *(VAR_P[PARAM_IDX] + i*sod) = g_rx.buffer[3+i];
                    }
                    break;
                case TYPE_STRING:
                    for (i=0; i<NUM_ITEMS[PARAM_IDX]; i++){
                        *(VAR_P[PARAM_IDX] + i*sod) = g_rx.buffer[3+i];
                    }
                    *(VAR_P[PARAM_IDX] + i*sod) = '\0';
                break;
                case TYPE_INT8:
                    for (i=0; i<NUM_ITEMS[PARAM_IDX]; i++){
                        *(VAR_P[PARAM_IDX] + i*sod) = *((int8*) &g_rx.buffer[↵
3 + i]);
                    }
                    break;
                case TYPE_INT16:
                    for (i=0; i<NUM_ITEMS[PARAM_IDX]; i++){
                        aux_int16 = *((int16 *) &g_rx.buffer[3 + i*sod]);
                        for(j = 0; j < sod; j++) {
                            ((char*)(VAR_P[PARAM_IDX] + i*sod))[sod - j -1] = ↵
((char*)(&aux_int16))[j];
                        }
                    }
                    break;
                case TYPE_UINT16:
                    for (i=0; i<NUM_ITEMS[PARAM_IDX]; i++){
```

```
                        aux_uint16 = *((uint16 *) &g_rx.buffer[3 + i*sod]);
                        for(j = 0; j < sod; j++) {
                            ((char*)(VAR_P[PARAM_IDX] + i*sod))[sod - j -1] = ((char*)(&aux_uint16))[j];
                        }
                    }
                    break;
                case TYPE_INT32:
                    for (i=0; i<NUM_ITEMS[PARAM_IDX]; i++){
                        aux_int32 = *((int32 *) &g_rx.buffer[3 + i*sod]);
                        for(j = 0; j < sod; j++) {
                            ((char*)(VAR_P[PARAM_IDX] + i*sod))[sod - j -1] = ((char*)(&aux_int32))[j];
                        }
                    }
                    break;
                case TYPE_UINT32:
                    for (i=0; i<NUM_ITEMS[PARAM_IDX]; i++){
                        aux_uint32 = *((uint32 *) &g_rx.buffer[3 + i*sod]);
                        for(j = 0; j < sod; j++) {
                            ((char*)(VAR_P[PARAM_IDX] + i*sod))[sod - j -1] = ((char*)(&aux_uint32))[j];
                        }
                    }
                    break;
                case TYPE_FLOAT:
                    for (i=0; i<NUM_ITEMS[PARAM_IDX]; i++){
                        aux_float = *((float *) &g_rx.buffer[3 + i*sod]);
                        for(j = 0; j < sod; j++) {
                            ((char*)(VAR_P[PARAM_IDX] + i*sod))[sod - j -1] = ((char*)(&aux_float))[j];
                        }
                    }
                    break;
                default:
                    break;
            }
        }
        else {
            // Use custom specifications for param setting
            set_custom_param(index);
        }

        // Store param also in user_emg structure
        if (index != USER_ID_PARAM) {        // Not when changing user id
            memcpy( &(MEM_P->user[MEM_P->dev.user_id].user_emg), &(MEM_P->emg), sizeof(MEM_P->emg) );
        }

        // Perform chip reset if needed
        if (TYPES[PARAM_IDX] == TYPE_FLAG){
            uint8 idx = 0, menu_idx = -1;
```

```
            do {
                if (TYPES[idx] == TYPE_FLAG) menu_idx++;    // Increment idx to find the right NUM_MENU entry
                idx++;
            } while (idx <= PARAM_IDX);

            if (NUM_MENU[menu_idx] == 5 || NUM_MENU[menu_idx] == 6 || NUM_MENU[menu_idx] == 8 || NUM_MENU[menu_idx] == 9 || NUM_MENU[menu_idx] == 10) {
                reset_PSoC_flag = TRUE;
            }
        }
    }
}

//
//////////////////////////////////////////////////////////////////////////////
//                                                                  SET CUSTOM PARAM
//
//////////////////////////////////////////////////////////////////////////////

void set_custom_param(uint16 index) {

    uint8 CYDATA i, j;
    uint8 aux_uchar;
    float aux_float, aux_float2;

    uint8 MOTOR_IDX = 0;
    uint8 SECOND_MOTOR_IDX = 1;

    switch(index){
        case 2:         // Position PID
            if(c_mem.motor[MOTOR_IDX].control_mode != CURR_AND_POS_CONTROL) {
                aux_float = *((float *) &g_rx.buffer[3]);
                for(j = 0; j < 4; j++) {
                    ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j];
                }
                g_mem.motor[MOTOR_IDX].k_p = aux_float2 * 65536;

                aux_float = *((float *) &g_rx.buffer[3 + 4]);
                for(j = 0; j < 4; j++) {
                    ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j];
                }
                g_mem.motor[MOTOR_IDX].k_i = aux_float2 * 65536;

                aux_float = *((float *) &g_rx.buffer[3 + 8]);
                for(j = 0; j < 4; j++) {
                    ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j];
                }
```

```
            g_mem.motor[MOTOR_IDX].k_d = aux_float2 * 65536;
        }
        else {
            aux_float = *((float *) &g_rx.buffer[3]);
            for(j = 0; j < 4; j++) {
                ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j
];
            }
            g_mem.motor[MOTOR_IDX].k_p_dl = aux_float2 * 65536;

            aux_float = *((float *) &g_rx.buffer[3 + 4]);
            for(j = 0; j < 4; j++) {
                ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j
];
            }
            g_mem.motor[MOTOR_IDX].k_i_dl = aux_float2 * 65536;

            aux_float = *((float *) &g_rx.buffer[3 + 8]);
            for(j = 0; j < 4; j++) {
                ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j
];
            }
            g_mem.motor[MOTOR_IDX].k_d_dl = aux_float2 * 65536;
        }
        break;

    case 3:         //Current PID
        if(c_mem.motor[MOTOR_IDX].control_mode != CURR_AND_POS_CONTROL) {
            aux_float = *((float *) &g_rx.buffer[3]);
            for(j = 0; j < 4; j++) {
                ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j
];
            }
            g_mem.motor[MOTOR_IDX].k_p_c = aux_float2 * 65536;

            aux_float = *((float *) &g_rx.buffer[3 + 4]);
            for(j = 0; j < 4; j++) {
                ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j
];
            }
            g_mem.motor[MOTOR_IDX].k_i_c = aux_float2 * 65536;

            aux_float = *((float *) &g_rx.buffer[3 + 8]);
            for(j = 0; j < 4; j++) {
                ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j
];
            }
            g_mem.motor[MOTOR_IDX].k_d_c = aux_float2 * 65536;
        }
        else {
            aux_float = *((float *) &g_rx.buffer[3]);
            for(j = 0; j < 4; j++) {
```

```
                ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j
];
            }
            g_mem.motor[MOTOR_IDX].k_p_c_dl = aux_float2 * 65536;

            aux_float = *((float *) &g_rx.buffer[3 + 4]);
            for(j = 0; j < 4; j++) {
                ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j
];
            }
            g_mem.motor[MOTOR_IDX].k_i_c_dl = aux_float2 * 65536;

            aux_float = *((float *) &g_rx.buffer[3 + 8]);
            for(j = 0; j < 4; j++) {
                ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j
];
            }
            g_mem.motor[MOTOR_IDX].k_d_c_dl = aux_float2 * 65536;
        }
        break;

    case 5:         //Input mode
        g_mem.motor[MOTOR_IDX].input_mode = g_rx.buffer[3];

        // Hold the actual position
        g_refNew[MOTOR_IDX].pos = g_meas[g_mem.motor[MOTOR_IDX].
encoder_line].pos[0];
        break;

    case 8:         //Measurement Offset
        for(i = 0; i < NUM_OF_SENSORS; i++) {
            g_mem.enc[g_mem.motor[MOTOR_IDX].encoder_line].m_off[i] = (
int16)(g_rx.buffer[3 + i*2]<<8 | g_rx.buffer[4 + i*2]);
            g_mem.enc[g_mem.motor[MOTOR_IDX].encoder_line].m_off[i] =
g_mem.enc[g_mem.motor[MOTOR_IDX].encoder_line].m_off[i] << g_mem.enc[g_mem.
motor[MOTOR_IDX].encoder_line].res[i];

            g_meas[g_mem.motor[MOTOR_IDX].encoder_line].rot[i] = 0;
        }
        reset_last_value_flag[MOTOR_IDX] = 1;
        break;

    case 11:        //Position limits
        g_mem.motor[MOTOR_IDX].pos_lim_inf = (int32)(g_rx.buffer[3]<<24 |
g_rx.buffer[4]<<16 | g_rx.buffer[5]<<8 | g_rx.buffer[6]);
        g_mem.motor[MOTOR_IDX].pos_lim_sup = (int32)(g_rx.buffer[7]<<24 |
g_rx.buffer[8]<<16 | g_rx.buffer[9]<<8 | g_rx.buffer[10]);

        g_mem.motor[MOTOR_IDX].pos_lim_inf = g_mem.motor[MOTOR_IDX].
pos_lim_inf << g_mem.enc[g_mem.motor[MOTOR_IDX].encoder_line].res[0];
        g_mem.motor[MOTOR_IDX].pos_lim_sup = g_mem.motor[MOTOR_IDX].
pos_lim_sup << g_mem.enc[g_mem.motor[MOTOR_IDX].encoder_line].res[0];
```

```
            break;

        case 23:        //Rest Position
            g_mem.SH.rest_pos = (int32)(g_rx.buffer[3]<<24 | g_rx.buffer[4]<<16 | g_rx.buffer[5]<<8 | g_rx.buffer[6]);
            g_mem.SH.rest_pos = g_mem.SH.rest_pos << g_mem.enc[g_mem.motor[MOTOR_IDX].encoder_line].res[0];
            break;

        case 24:        //Rest Position Time Delay
           g_mem.SH.rest_delay = (int32)(g_rx.buffer[3]<<24 | g_rx.buffer[4]<<16 | g_rx.buffer[5]<<8 | g_rx.buffer[6]);
           if (g_mem.SH.rest_delay < 10) g_mem.SH.rest_delay = 10;
           break;

        case 28:        //Right/Left hand flag
            aux_uchar = *((uint8*) &g_rx.buffer[3]);
            if (aux_uchar) {    // 1
                g_mem.dev.right_left = LEFT_HAND;
            } else {            // 0
                g_mem.dev.right_left = RIGHT_HAND;
            }
            reset_last_value_flag[MOTOR_IDX] = 1;

            if (g_mem.dev.dev_type == SOFTHAND_PRO){
                // Change also default encoder line (only with SoftHand FW)
                g_mem.motor[MOTOR_IDX].encoder_line = g_mem.dev.right_left;

                // Change also gears parameters
                g_mem.enc[g_mem.motor[MOTOR_IDX].encoder_line].double_encoder_on_off = TRUE;
                g_mem.enc[g_mem.motor[MOTOR_IDX].encoder_line].gears_params[0] = SH_N1;
                g_mem.enc[g_mem.motor[MOTOR_IDX].encoder_line].gears_params[1] = SH_N2;
                g_mem.enc[g_mem.motor[MOTOR_IDX].encoder_line].gears_params[2] = SH_I1;

                // Get CS0 encoder line for RIGHT HAND and CS1 line for LEFT HAND as default
                g_mem.motor[MOTOR_IDX].pwm_rate_limiter = PWM_RATE_LIMITER_MAX;
                g_mem.motor[MOTOR_IDX].not_revers_motor_flag = TRUE;       // SoftHand not reversible motor
                g_mem.motor[MOTOR_IDX].pos_lim_inf = 0;
                g_mem.motor[MOTOR_IDX].pos_lim_sup = (int32)16000 << g_mem.enc[g_mem.motor[0].encoder_line].res[0];
            }

            break;

        case 31:        //Reset counters - uint8
            aux_uchar = *((uint8*) &g_rx.buffer[3]);
```

```
            if (aux_uchar) {
                reset_counters();
                g_mem.dev.reset_counters = FALSE;
            }

            if (c_mem.exp.read_exp_port_flag == EXP_SD_RTC) {
                // Set date of maintenance from RTC
                aux_uchar = DS1302_read(DS1302_DATE_RD);
                g_mem.dev.stats_period_begin_date[0] = (aux_uchar/16) * 10 + ↵
aux_uchar%16;     //day
                aux_uchar = DS1302_read(DS1302_MONTH_RD);
                g_mem.dev.stats_period_begin_date[1] = (aux_uchar/16) * 10 + ↵
aux_uchar%16;     // month
                aux_uchar = DS1302_read(DS1302_YEAR_RD);
                g_mem.dev.stats_period_begin_date[2] = (aux_uchar/16) * 10 + ↵
aux_uchar%16;           // year
            }
            break;

        case 32:          //Current Time
            for (uint8 i=0; i<6; i++){
                g_mem.exp.curr_time[i] = g_rx.buffer[3 + i];
            }

            if (g_mem.exp.read_exp_port_flag == EXP_SD_RTC) {
                set_RTC_time();
            }
            break;

        case 38:         // First Motor Driver Type
            g_mem.motor[MOTOR_IDX].motor_driver_type = g_rx.buffer[3];
            set_motor_driver_type();
            break;

        case 44:          // Second Motor Position PID
            if(c_mem.motor[SECOND_MOTOR_IDX].control_mode != ↵
CURR_AND_POS_CONTROL) {
                aux_float = *((float *) &g_rx.buffer[3]);
                for(j = 0; j < 4; j++) {
                    ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j↵
];
                }
                g_mem.motor[SECOND_MOTOR_IDX].k_p = aux_float2 * 65536;

                aux_float = *((float *) &g_rx.buffer[3 + 4]);
                for(j = 0; j < 4; j++) {
                    ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j↵
];
                }
                g_mem.motor[SECOND_MOTOR_IDX].k_i = aux_float2 * 65536;

                aux_float = *((float *) &g_rx.buffer[3 + 8]);
```

```
                for(j = 0; j < 4; j++) {
                    ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j
];
                }
                g_mem.motor[SECOND_MOTOR_IDX].k_d = aux_float2 * 65536;
            }
            else {
                aux_float = *((float *) &g_rx.buffer[3]);
                for(j = 0; j < 4; j++) {
                    ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j
];
                }
                g_mem.motor[SECOND_MOTOR_IDX].k_p_dl = aux_float2 * 65536;

                aux_float = *((float *) &g_rx.buffer[3 + 4]);
                for(j = 0; j < 4; j++) {
                    ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j
];
                }
                g_mem.motor[SECOND_MOTOR_IDX].k_i_dl = aux_float2 * 65536;

                aux_float = *((float *) &g_rx.buffer[3 + 8]);
                for(j = 0; j < 4; j++) {
                    ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j
];
                }
                g_mem.motor[SECOND_MOTOR_IDX].k_d_dl = aux_float2 * 65536;
            }
            break;

        case 45:          // Second Motor Current PID
            if(c_mem.motor[SECOND_MOTOR_IDX].control_mode !=
CURR_AND_POS_CONTROL) {
                aux_float = *((float *) &g_rx.buffer[3]);
                for(j = 0; j < 4; j++) {
                    ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j
];
                }
                g_mem.motor[SECOND_MOTOR_IDX].k_p_c = aux_float2 * 65536;

                aux_float = *((float *) &g_rx.buffer[3 + 4]);
                for(j = 0; j < 4; j++) {
                    ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j
];
                }
                g_mem.motor[SECOND_MOTOR_IDX].k_i_c = aux_float2 * 65536;

                aux_float = *((float *) &g_rx.buffer[3 + 8]);
                for(j = 0; j < 4; j++) {
                    ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j
];
                }
```

```
                g_mem.motor[SECOND_MOTOR_IDX].k_d_c = aux_float2 * 65536;
            }
            else {
                aux_float = *((float *) &g_rx.buffer[3]);
                for(j = 0; j < 4; j++) {
                    ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j
];
                }
                g_mem.motor[SECOND_MOTOR_IDX].k_p_c_dl = aux_float2 * 65536;

                aux_float = *((float *) &g_rx.buffer[3 + 4]);
                for(j = 0; j < 4; j++) {
                    ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j
];
                }
                g_mem.motor[SECOND_MOTOR_IDX].k_i_c_dl = aux_float2 * 65536;

                aux_float = *((float *) &g_rx.buffer[3 + 8]);
                for(j = 0; j < 4; j++) {
                    ((char*)(&aux_float2))[4 - j -1] = ((char*)(&aux_float))[j
];
                }
                g_mem.motor[SECOND_MOTOR_IDX].k_d_c_dl = aux_float2 * 65536;
            }
            break;

        case 47:          // Second Motor Input mode
            g_mem.motor[SECOND_MOTOR_IDX].input_mode = g_rx.buffer[3];

            // Hold the actual position
            g_refNew[SECOND_MOTOR_IDX].pos = g_meas[g_mem.motor[
SECOND_MOTOR_IDX].encoder_line].pos[0];
            break;

        case 50:          // Second Motor Measurement Offset
            for(i = 0; i < NUM_OF_SENSORS; i++) {
                g_mem.enc[g_mem.motor[SECOND_MOTOR_IDX].encoder_line].m_off[i
] = (int16)(g_rx.buffer[3 + i*2]<<8 | g_rx.buffer[4 + i*2]);
                g_mem.enc[g_mem.motor[SECOND_MOTOR_IDX].encoder_line].m_off[i
] = g_mem.enc[g_mem.motor[SECOND_MOTOR_IDX].encoder_line].m_off[i] << g_mem.
enc[g_mem.motor[SECOND_MOTOR_IDX].encoder_line].res[i];

                g_meas[g_mem.motor[SECOND_MOTOR_IDX].encoder_line].rot[i] = 0;
            }
            reset_last_value_flag[SECOND_MOTOR_IDX] = 1;
            break;

        case 53:         // Second Motor Position limits
            g_mem.motor[SECOND_MOTOR_IDX].pos_lim_inf = (int32)(g_rx.buffer[3]
<<24 | g_rx.buffer[4]<<16 | g_rx.buffer[5]<<8 | g_rx.buffer[6]);
            g_mem.motor[SECOND_MOTOR_IDX].pos_lim_sup = (int32)(g_rx.buffer[7]
<<24 | g_rx.buffer[8]<<16 | g_rx.buffer[9]<<8 | g_rx.buffer[10]);
```

```
            g_mem.motor[SECOND_MOTOR_IDX].pos_lim_inf = g_mem.motor[↵
SECOND_MOTOR_IDX].pos_lim_inf << g_mem.enc[g_mem.motor[SECOND_MOTOR_IDX].↵
encoder_line].res[0];
            g_mem.motor[SECOND_MOTOR_IDX].pos_lim_sup = g_mem.motor[↵
SECOND_MOTOR_IDX].pos_lim_sup << g_mem.enc[g_mem.motor[SECOND_MOTOR_IDX].↵
encoder_line].res[0];
            break;

        case 61:        // Second Motor Driver Type
            g_mem.motor[SECOND_MOTOR_IDX].motor_driver_type = g_rx.buffer[3];
            set_motor_driver_type();
            break;

        case 75:        // Device type
            g_mem.dev.dev_type = g_rx.buffer[3];

            if (g_mem.dev.dev_type == SOFTHAND_PRO){    // change also gears ↵
parameters
                g_mem.enc[g_mem.motor[MOTOR_IDX].encoder_line].↵
double_encoder_on_off = TRUE;
                g_mem.enc[g_mem.motor[MOTOR_IDX].encoder_line].gears_params[0↵
] = SH_N1;
                g_mem.enc[g_mem.motor[MOTOR_IDX].encoder_line].gears_params[1↵
] = SH_N2;
                g_mem.enc[g_mem.motor[MOTOR_IDX].encoder_line].gears_params[2↵
] = SH_I1;

                // Get CS0 encoder line for RIGHT HAND and CS1 line for LEFT ↵
HAND as default
                g_mem.motor[MOTOR_IDX].encoder_line  = g_mem.dev.right_left;
                g_mem.motor[MOTOR_IDX].pwm_rate_limiter = PWM_RATE_LIMITER_MAX;
                g_mem.motor[MOTOR_IDX].not_revers_motor_flag = TRUE;       // ↵
SoftHand not reversible motor
                g_mem.motor[MOTOR_IDX].pos_lim_inf = 0;
                g_mem.motor[MOTOR_IDX].pos_lim_sup = (int32)16000 << g_mem.enc↵
[g_mem.motor[MOTOR_IDX].encoder_line].res[0];
            }

            if (g_mem.dev.dev_type == SOFTHAND_2_MOTORS){    // activate also ↵
2nd motor and double_encoder
                g_mem.dev.use_2nd_motor_flag = TRUE;
                g_mem.enc[g_mem.motor[MOTOR_IDX].encoder_line].↵
double_encoder_on_off = TRUE;
                g_mem.enc[g_mem.motor[SECOND_MOTOR_IDX].encoder_line].↵
double_encoder_on_off = TRUE;
            /*
                g_mem.motor[MOTOR_IDX].pos_lim_inf = 0;
                g_mem.motor[MOTOR_IDX].pos_lim_sup = (int32)18000 << g_mem.enc↵
[g_mem.motor[MOTOR_IDX].encoder_line].res[0];
                g_mem.motor[SECOND_MOTOR_IDX].pos_lim_inf = ((int32)(-25000)) ↵
<< g_mem.enc[g_mem.motor[SECOND_MOTOR_IDX].encoder_line].res[0];
```

```
                g_mem.motor[SECOND_MOTOR_IDX].pos_lim_sup = 0;
            */
                for (i=0; i< NUM_OF_MOTORS; i++) {                      // ↵
Maxon DCX16S
                    g_mem.motor[i].current_limit = 800;                 // [mA]
                    g_mem.motor[i].k_p           = 0.12 * 65536;
                    g_mem.motor[i].k_i           =    0 * 65536;
                    g_mem.motor[i].k_d           = 0.05 * 65536;
                }
            }

            break;

        default:
            break;
    }
}

//↵
================================================================================
//                                                          GET IMU PARAMETER ↵
LIST
//↵
================================================================================

void get_IMU_param_list(uint16 index)
{
    //Package to be sent variables
    uint8 packet_data[  PARAM_BYTE_SLOT   +     // Number of connected IMUs
                        7*PARAM_BYTE_SLOT +     // IMUs ID  (1 port instead ↵
of 7)
                        9*PARAM_BYTE_SLOT +     // Mag cal parameters (1 row ↵
instead of 9)
                        PARAM_BYTE_SLOT   +     // 1 - Device ID
                        PARAM_BYTE_SLOT*N_IMU_MAX + // IMU configurations
                        PARAM_BYTE_SLOT   +     // SPI read delay
                        PARAM_MENU_SLOT   + PARAM_BYTE_SLOT + 1 ] = "";
    uint16 num_imus_id_params = 7;      //(1 port instead of 7)
    uint16 num_mag_cal_params = 0;
    uint16 first_imu_parameter = 2;
    uint16 packet_length = PARAM_BYTE_SLOT +
                           num_imus_id_params*PARAM_BYTE_SLOT +
                           num_mag_cal_params*PARAM_BYTE_SLOT +
                           PARAM_BYTE_SLOT   +
                           (uint16)(PARAM_BYTE_SLOT*N_IMU_Connected) +
                           PARAM_BYTE_SLOT + PARAM_MENU_SLOT + ↵
PARAM_BYTE_SLOT + 1;

    //Auxiliary variables
    uint16 CYDATA i, j, k, h;
    uint16 start_byte = 0;
```

```
    //Parameters menu string definitions
    char n_imu_str[26]          = "Number of connected IMUs:";
    char ids_str[11]            = "";
    char mag_param_str[20]      = "Mag cal parameters:";
    char id_str[16]             = "";
    char imu_table_str[42]      = "";
    char spi_read_delay_str[26] = "";

    //Strings lenghts
    uint8 CYDATA id_str_len = strlen(id_str);
    uint8 CYDATA n_imu_str_len = strlen(n_imu_str);
    uint8 CYDATA ids_str_len = strlen(ids_str);
    uint8 CYDATA mag_param_str_len = strlen(mag_param_str);
    uint8 CYDATA imu_table_str_len = strlen(imu_table_str);
    uint8 CYDATA spi_read_delay_str_len = strlen(spi_read_delay_str);

    char spi_delay_menu[118]    = "";
    uint8 CYDATA spi_delay_menu_len;

    sprintf(spi_delay_menu, "0 -> None\n1 -> Low (%u us delay for each 8-bit ↵
register read)\n2 -> High (%u us delay for each 8-bit register read)\n", (int)↵
SPI_DELAY_LOW, (int)SPI_DELAY_HIGH);
    spi_delay_menu_len = strlen(spi_delay_menu);

    // Compute number of read parameters depending on N_IMU_Connected and
    // update packet_length
    num_mag_cal_params = (uint16)(N_IMU_Connected / 2);
    if ( (N_IMU_Connected - num_mag_cal_params*2) > 0 ) num_mag_cal_params++;

    packet_length = PARAM_BYTE_SLOT +
                    num_imus_id_params*PARAM_BYTE_SLOT +
                    num_mag_cal_params*PARAM_BYTE_SLOT +
                    PARAM_BYTE_SLOT   +
                    (uint16)(PARAM_BYTE_SLOT*N_IMU_Connected) +
                    PARAM_MENU_SLOT + PARAM_BYTE_SLOT + 1;

    first_imu_parameter = 1 + num_imus_id_params + num_mag_cal_params + 2;
    packet_data[0] = CMD_GET_IMU_PARAM;
    packet_data[1] = 1 + num_imus_id_params + num_mag_cal_params + 1 + (uint8)↵
N_IMU_Connected + 1;        // NUM_PARAMS

    switch(index) {
        case 0:         //List of all parameters with relative types
            /*-------------N IMU--------------*/
            start_byte = 0;
            packet_data[2] = TYPE_UINT8;
            packet_data[3] = 1;
            packet_data[4] = (uint8)N_IMU_Connected;
            for(i = n_imu_str_len; i != 0; i--)
                packet_data[5 + n_imu_str_len - i] = n_imu_str[n_imu_str_len ↵
- i];
```

```
            /*-------------IMUS ID--------------*/
            start_byte = start_byte + PARAM_BYTE_SLOT;
            i = 0;
            for (k = 0; k < num_imus_id_params; k++){
                sprintf(ids_str, "Port %u ID:", k);
                h = 4;
                ids_str_len = strlen(ids_str);
                packet_data[2+start_byte + PARAM_BYTE_SLOT*k] = TYPE_UINT8;
                packet_data[3+start_byte + PARAM_BYTE_SLOT*k] = 3;

                for (j = 3*k; j <= 3*k+2; j++) {  // for each possible imu on ↵
port k
                    if (IMU_connected[i] == j) {
                        packet_data[h+start_byte + PARAM_BYTE_SLOT*k] = (uint8↵
)IMU_connected[i];
                        i++;
                    }
                    else {
                        packet_data[h+start_byte + PARAM_BYTE_SLOT*k] = 255;
                    }
                    h++;
                }

                //if (IMU_connected[i] >= 3*k && IMU_connected[i] <= 3*k + 2)
                for(j = ids_str_len; j != 0; j--)
                    packet_data[7+start_byte + PARAM_BYTE_SLOT*k + ↵
ids_str_len - j] = ids_str[ids_str_len - j];
            }

            /*-------------GET MAG PARAM--------------*/
            start_byte = start_byte + PARAM_BYTE_SLOT*num_imus_id_params;
            for (k = 0; k < num_mag_cal_params; k++){
                packet_data[2+start_byte + PARAM_BYTE_SLOT*k] = TYPE_UINT8;

                packet_data[3+start_byte + PARAM_BYTE_SLOT*k] = 3;
                packet_data[4+start_byte + PARAM_BYTE_SLOT*k] = (uint8) MagCal↵
[IMU_connected[2*k]][0];
                packet_data[5+start_byte + PARAM_BYTE_SLOT*k] = (uint8) MagCal↵
[IMU_connected[2*k]][1];
                packet_data[6+start_byte + PARAM_BYTE_SLOT*k] = (uint8) MagCal↵
[IMU_connected[2*k]][2];

                // check if there is a second value
                if ( N_IMU_Connected < 2*(k+1) ) {
                    // there is only one value
                    for(j = mag_param_str_len; j != 0; j--)
                        packet_data[7+start_byte + PARAM_BYTE_SLOT*k + ↵
mag_param_str_len - j] = mag_param_str[mag_param_str_len - j];
                }
                else {
                    // fill the second value
                    packet_data[3+start_byte + PARAM_BYTE_SLOT*k] = 6;
```

```
                    packet_data[7+start_byte + PARAM_BYTE_SLOT*k] = (uint8) ↵
MagCal[IMU_connected[2*k+1]][0];
                    packet_data[8+start_byte + PARAM_BYTE_SLOT*k] = (uint8) ↵
MagCal[IMU_connected[2*k+1]][1];
                    packet_data[9+start_byte + PARAM_BYTE_SLOT*k] = (uint8) ↵
MagCal[IMU_connected[2*k+1]][2];

                    for(j = mag_param_str_len; j != 0; j--)
                        packet_data[10+start_byte + PARAM_BYTE_SLOT*k + ↵
mag_param_str_len - j] = mag_param_str[mag_param_str_len - j];
                }
            }

            /*-----------------ID-----------------*/

            start_byte = start_byte + PARAM_BYTE_SLOT*num_mag_cal_params;
            sprintf(id_str, "%u - Device ID:", first_imu_parameter-1);
            id_str_len = strlen(id_str);
            packet_data[2+start_byte] = TYPE_UINT8;
            packet_data[3+start_byte] = 1;
            packet_data[4+start_byte] = c_mem.dev.id;
            for(i = id_str_len; i != 0; i--)
                packet_data[5+start_byte + id_str_len - i] = id_str[↵
id_str_len - i];

            /*-------------GET IMUS MODE-------------*/

            start_byte = start_byte + PARAM_BYTE_SLOT;
            for (i = 0; i < (uint8)N_IMU_Connected; i++){
                sprintf(imu_table_str, "%u - IMU %d configuration:", ↵
first_imu_parameter + i, (int) IMU_connected[i]);
                imu_table_str_len = strlen(imu_table_str);

                packet_data[(uint16)(2 + start_byte + PARAM_BYTE_SLOT*i)] = ↵
TYPE_UINT8;
                packet_data[(uint16)(3 + start_byte + PARAM_BYTE_SLOT*i)] = 5;

                packet_data[(uint16)(4 + start_byte + PARAM_BYTE_SLOT*i)] = (↵
uint8)(c_mem.imu.IMU_conf[IMU_connected[i]][0]);
                packet_data[(uint16)(5 + start_byte + PARAM_BYTE_SLOT*i)] = (↵
uint8)(c_mem.imu.IMU_conf[IMU_connected[i]][1]);
                packet_data[(uint16)(6 + start_byte + PARAM_BYTE_SLOT*i)] = (↵
uint8)(c_mem.imu.IMU_conf[IMU_connected[i]][2]);
                packet_data[(uint16)(7 + start_byte + PARAM_BYTE_SLOT*i)] = (↵
uint8)(c_mem.imu.IMU_conf[IMU_connected[i]][3]);
                packet_data[(uint16)(8 + start_byte + PARAM_BYTE_SLOT*i)] = (↵
uint8)(c_mem.imu.IMU_conf[IMU_connected[i]][4]);

                for(j = imu_table_str_len; j != 0; j--)
                    packet_data[(uint16)(9 + start_byte + PARAM_BYTE_SLOT*i + ↵
imu_table_str_len - j)] = imu_table_str[imu_table_str_len - j];
            }
```

```
            /*-----------------SPI DELAY-----------------*/

            start_byte = start_byte + (uint16)(PARAM_BYTE_SLOT*N_IMU_Connected
);
            sprintf(spi_read_delay_str, "%u - SPI read delay:", 
first_imu_parameter+N_IMU_Connected);
            packet_data[2+start_byte] = TYPE_FLAG;
            packet_data[3+start_byte] = 1;
            packet_data[4+start_byte] = c_mem.imu.SPI_read_delay;
            switch(c_mem.imu.SPI_read_delay) {
                case 0:
                    strcat(spi_read_delay_str, " None");
                    spi_read_delay_str_len = 26;
                    break;
                case 1:
                    strcat(spi_read_delay_str, " Low");
                    spi_read_delay_str_len = 25;
                    break;
                case 2:
                    strcat(spi_read_delay_str, " High");
                    spi_read_delay_str_len = 26;
                    break;
                default:
                    break;
            }
            for(i = spi_read_delay_str_len; i != 0; i--)
                packet_data[5+start_byte + spi_read_delay_str_len - i] = 
spi_read_delay_str[spi_read_delay_str_len - i];
            //The following byte indicates the number of menus at the end of 
the packet to send
            packet_data[5+start_byte + spi_read_delay_str_len] = 1;


            /*------------PARAMETERS MENU-----------*/
            start_byte = start_byte + PARAM_BYTE_SLOT;
            for(i = spi_delay_menu_len; i!= 0; i--)
                packet_data[(uint16)(2 + start_byte) + spi_delay_menu_len - i
] = spi_delay_menu[spi_delay_menu_len - i];

            packet_data[packet_length - 1] = LCRChecksum(packet_data,
packet_length - 1);
            commWrite(packet_data, packet_length);
            UART_RS485_ClearTxBuffer();
        break;

//============================================================     other_params
        default:

            if (index < first_imu_parameter-1)
                break;
```

```
            if (index == first_imu_parameter+N_IMU_Connected) {
                g_mem.imu.SPI_read_delay = g_rx.buffer[3];  //SPI read delay
- uint8
                break;
            }

            if (index == first_imu_parameter-1) {
                g_mem.dev.id = g_rx.buffer[3];          //ID - uint8
            }
            else {

                //Set Imu table (index > = first_imu_parameter)
                g_mem.imu.IMU_conf[IMU_connected[index-first_imu_parameter]][0
] = g_rx.buffer[3];
                g_mem.imu.IMU_conf[IMU_connected[index-first_imu_parameter]][1
] = g_rx.buffer[4];
                g_mem.imu.IMU_conf[IMU_connected[index-first_imu_parameter]][2
] = g_rx.buffer[5];
                g_mem.imu.IMU_conf[IMU_connected[index-first_imu_parameter]][3
] = g_rx.buffer[6];
                g_mem.imu.IMU_conf[IMU_connected[index-first_imu_parameter]][4
] = g_rx.buffer[7];

                // Recompute IMU packets dimension
                imus_data_size = 1; //header
                for (i = 0; i < N_IMU_Connected; i++)
                {
                    single_imu_size[IMU_connected[i]] = 1 + 6*g_mem.imu.
IMU_conf[IMU_connected[i]][0] + 6*g_mem.imu.IMU_conf[IMU_connected[i]][1] + 6
*g_mem.imu.IMU_conf[IMU_connected[i]][2] + 16*g_mem.imu.IMU_conf[IMU_connected
[i]][3] + 2*g_mem.imu.IMU_conf[IMU_connected[i]][4]+ 1;
                    imus_data_size = imus_data_size + single_imu_size[
IMU_connected[i]];
                }
                imus_data_size = imus_data_size + 1;    //checksum

            }

        break;
    }
}

//
================================================================================
//                                                          COMMAND SET ZEROS
//
================================================================================

void setZeros()
{
    uint8 CYDATA i, j;        // iterator
```

```
    for (j = 0; j < N_ENCODER_LINE_MAX; j++) {
        for(i = 0; i < NUM_OF_SENSORS; i++) {
            g_mem.enc[j].m_off[i] = (int32)(data_encoder_raw[j][i]);
            g_meas[j].rot[i] = (int8)0;
        }
        reset_last_value_flag[j] = 1;
    }

    sendAcknowledgment(ACK_OK);
}

//
================================================================================
//                                                    PREPARE GENERIC DEVICE
INFO
//
================================================================================

void prepare_generic_info(char *info_string)
{
    int i;

    struct st_eeprom* MEM_P = &c_mem;

    if(c_mem.dev.id != 250){                //To avoid dummy board ping
        char str[100];
        strcpy(info_string, "");
        strcat(info_string, "\r\n");
        strcat(info_string, "Firmware version: ");
        strcat(info_string, VERSION);
        strcat(info_string, ".\r\n\r\n");

        strcat(info_string, "DEVICE INFO\r\n");
        sprintf(str, "ID: %d\r\n", (int) MEM_P->dev.id);
        strcat(info_string, str);
        switch(MEM_P->dev.dev_type){
            case SOFTHAND_PRO:
                strcat(info_string, "Device: SOFTHAND PRO\r\n");
                break;
            case GENERIC_2_MOTORS:
                strcat(info_string, "Device: GENERIC 2 MOTORS\r\n");
                break;
            case AIR_CHAMBERS_FB:
                strcat(info_string, "Device: AIR CHAMBERS HAPTIC FEEDBACK\r\n"
);
                break;
            case OTBK_ACT_WRIST_MS:
                strcat(info_string, "Device: OTTOBOCK 6v ACTIVE WRIST MASTER\r
\n");
                break;
            case SOFTHAND_2_MOTORS:
```

```
            strcat(info_string, "Device: SOFTHAND 2 MOTORS\r\n");
            break;
        default:
            break;
    }

    switch(MEM_P->dev.right_left){
        case RIGHT_HAND:
            strcat(info_string, "Hand side: RIGHT\r\n");
            break;
        case LEFT_HAND:
            strcat(info_string, "Hand side: LEFT\r\n");
            break;
    }

    switch(MEM_P->dev.user_id) {
        case MARIA:
            strcat(info_string, "User: MARIA\r\n");
            break;
        case R01:
            strcat(info_string, "User: R01\r\n");
            break;
        default:
            strcat(info_string, "User: GENERIC USER\r\n");
            break;
    }
    strcat(info_string, "\r\n");

    for (uint8 k = 0; k <= MEM_P->dev.use_2nd_motor_flag; k++) {

        uint8 MOTOR_IDX = k;
        struct st_motor* MOT = &(MEM_P->motor[MOTOR_IDX]);      // ↵
Default motor
        uint8 ENC_L = MOT->encoder_line;             // Associated ↵
encoder line

        sprintf(str, "MOTOR %d INFO\r\n", MOTOR_IDX+1);
        strcat(info_string, str);

        strcat(info_string, "Motor reference");
        if(MOT->control_mode == CONTROL_CURRENT)
            strcat(info_string," - Currents: ");
        else {
            if (MOT->control_mode == CONTROL_PWM)
                strcat(info_string," - Pwm: ");
            else
                strcat(info_string," - Position: ");
        }
        if(MOT->control_mode == CONTROL_CURRENT) {
            sprintf(str, "%d ", (int)(g_refOld[MOTOR_IDX].curr));
            strcat(info_string,str);
        }
```

```
            else {
                if(MOT->control_mode == CONTROL_PWM) {
                    sprintf(str, "%d ", (int)(g_refOld[MOTOR_IDX].pwm));
                    strcat(info_string,str);
                }
                else {
                    sprintf(str, "%d ", (int)(g_refOld[MOTOR_IDX].pos >> MEM_P
->enc[ENC_L].res[0]));
                    strcat(info_string,str);
                }
            }
            strcat(info_string,"\r\n");

            sprintf(str, "Motor enabled: ");
            if (g_ref[MOTOR_IDX].onoff & 0x01) {
                strcat(str, "YES\r\n");
            } else {
                strcat(str, "NO\r\n");
            }
            strcat(info_string, str);

            strcat(info_string, "PWM rescaling activation: ");
            if(MOT->activate_pwm_rescaling == MAXON_12V)
                strcat(info_string, "YES\n");
            else
                strcat(info_string, "NO\n");

            sprintf(str, "PWM Limit: %d\r\n", (int) dev_pwm_limit[MOTOR_IDX]);
            strcat(info_string, str);

            strcat(info_string, "\r\nMEASUREMENTS INFO\r\n");
            strcat(info_string, "Sensor value: ");
            for (i = 0; i < NUM_OF_SENSORS; i++) {
                sprintf(str, "%d", (int)(g_meas[ENC_L].pos[i] >> MEM_P->enc[
ENC_L].res[i]));
                strcat(info_string, str);
                if (i != NUM_OF_SENSORS-1){
                    strcat(info_string, ", ");
                }
            }
            strcat(info_string, "\r\n");

            if (MOT->input_mode == INPUT_MODE_JOYSTICK){
                sprintf(str, "Joystick measurements: %d, %d", (int)
g_adc_measOld.joystick[0], (int)g_adc_measOld.joystick[1]);
                strcat(info_string, str);
                strcat(info_string, "\r\n");
            }

            sprintf(str, "Battery %d Voltage (mV): %ld", MOTOR_IDX+1, (int32)
dev_tension[MOTOR_IDX] );
            strcat(info_string, str);
```

```
            strcat(info_string, "\r\n");

            sprintf(str, "Full charge power tension %d (mV): %ld", MOTOR_IDX+1
, (int32) pow_tension[MOTOR_IDX] );
            strcat(info_string, str);
            strcat(info_string, "\r\n");

            sprintf(str, "Current %d (mA): %ld", MOTOR_IDX+1, (int32) g_meas[
ENC_L].curr );
            strcat(info_string, str);
            strcat(info_string, "\r\n");

            if (MOT->not_revers_motor_flag == TRUE){
                sprintf(str, "Last Grasp Hold Current %d (mA): %ld", MOTOR_IDX
+1, (int32) g_meas[ENC_L].hold_curr );
                strcat(info_string, str);
                strcat(info_string, "\r\n");
            }

            sprintf(str, "\r\nMOTOR %d CONFIGURATION\r\n", MOTOR_IDX+1);
            strcat(info_string, str);

            strcat(info_string, "PID Controller: ");
            if(MOT->control_mode != CURR_AND_POS_CONTROL) {
                sprintf(str, "P -> %f  ", ((double) MOT->k_p / 65536));
                strcat(info_string, str);
                sprintf(str, "I -> %f  ", ((double) MOT->k_i / 65536));
                strcat(info_string, str);
                sprintf(str, "D -> %f\r\n", ((double) MOT->k_d / 65536));
                strcat(info_string, str);
            }
            else {
                sprintf(str, "P -> %f  ", ((double) MOT->k_p_dl / 65536));
                strcat(info_string, str);
                sprintf(str, "I -> %f  ", ((double) MOT->k_i_dl / 65536));
                strcat(info_string, str);
                sprintf(str, "D -> %f\r\n", ((double) MOT->k_d_dl / 65536));
                strcat(info_string, str);
            }

            strcat(info_string, "Current PID Controller: ");
            if(MOT->control_mode != CURR_AND_POS_CONTROL) {
                sprintf(str, "P -> %f  ", ((double) MOT->k_p_c / 65536));
                strcat(info_string, str);
                sprintf(str, "I -> %f  ", ((double) MOT->k_i_c / 65536));
                strcat(info_string, str);
                sprintf(str, "D -> %f\r\n", ((double) MOT->k_d_c / 65536));
                strcat(info_string, str);

            }
            else {
                sprintf(str, "P -> %f  ", ((double) MOT->k_p_c_dl / 65536));
```

```
            strcat(info_string, str);
            sprintf(str, "I -> %f  ", ((double) MOT->k_i_c_dl / 65536));
            strcat(info_string, str);
            sprintf(str, "D -> %f\r\n", ((double) MOT->k_d_c_dl / 65536));
            strcat(info_string, str);
        }
        if (MOT->activ == 0x01)
            strcat(info_string, "Startup activation: YES\r\n");
        else
            strcat(info_string, "Startup activation: NO\r\n");

        switch(MOT->input_mode) {
            case INPUT_MODE_EXTERNAL:
                strcat(info_string, "Input mode: USB\r\n");
                break;
            case INPUT_MODE_ENCODER3:
                strcat(info_string, "Input mode: Handle\r\n");
                break;
            case INPUT_MODE_EMG_PROPORTIONAL:
                strcat(info_string, "Input mode: EMG proportional\r\n");
                break;
            case INPUT_MODE_EMG_INTEGRAL:
                strcat(info_string, "Input mode: EMG integral\r\n");
                break;
            case INPUT_MODE_EMG_FCFS:
                strcat(info_string, "Input mode: EMG FCFS\r\n");
                break;
            case INPUT_MODE_EMG_FCFS_ADV:
                strcat(info_string, "Input mode: EMG FCFS ADV\r\n");
                break;
            case INPUT_MODE_JOYSTICK:
                strcat(info_string, "Input mode: Joystick\r\n");
                break;
            case INPUT_MODE_EMG_PROPORTIONAL_NC:
                strcat(info_string, "Input mode: EMG proportional ↵
Normally Closed\r\n");
                break;
        }

        switch(MOT->control_mode) {
            case CONTROL_ANGLE:
                strcat(info_string, "Control mode: Position\r\n");
                break;
            case CONTROL_PWM:
                strcat(info_string, "Control mode: PWM\r\n");
                break;
            case CONTROL_CURRENT:
                strcat(info_string, "Control mode: Current\r\n");
                break;
            case CURR_AND_POS_CONTROL:
                strcat(info_string, "Control mode: Position and Current\r\↵
n");
```

```
                    break;
                default:
                    break;
            }

            if (MEM_P->enc[ENC_L].double_encoder_on_off)
                strcat(info_string, "Absolute encoder position: YES\r\n");
            else
                strcat(info_string, "Absolute encoder position: NO\r\n");

            sprintf(str, "Motor-Handle Ratio: %d\r\n", (int)MEM_P->enc[ENC_L].motor_handle_ratio);
            strcat(info_string, str);
#ifdef GENERIC_FW       // decided not to show when using SOFTHAND_FW to streamline ping, since these parameters are not settable
            strcat(info_string, "Encoder indices used for motor control: ");
            for (i = 0; i < NUM_OF_SENSORS; ++i) {
                sprintf(str, "%d", (int) MEM_P->enc[ENC_L].Enc_idx_use_for_control[i]);
                strcat(info_string, str);
                if (i != NUM_OF_SENSORS-1){
                    strcat(info_string, ", ");
                }
            }
            strcat(info_string, "\r\n");
            sprintf(str, "First Gear: %d teeth\r\n", (int)MEM_P->enc[ENC_L].gears_params[0]);
            strcat(info_string, str);
            sprintf(str, "Second Gear: %d teeth\r\n", (int)MEM_P->enc[ENC_L].gears_params[1]);
            strcat(info_string, str);
            sprintf(str, "Gear invariant: %d\r\n", (int)MEM_P->enc[ENC_L].gears_params[2]);
            strcat(info_string, str);
#endif
            strcat(info_string, "\r\n");

            strcat(info_string, "Sensor resolution: ");
            for (i = 0; i < NUM_OF_SENSORS; ++i) {
                sprintf(str, "%d", (int) MEM_P->enc[ENC_L].res[i]);
                strcat(info_string, str);
                if (i != NUM_OF_SENSORS-1){
                    strcat(info_string, ", ");
                }
            }
            strcat(info_string, "\r\n");

            strcat(info_string, "Measurement Offset: ");
            for (i = 0; i < NUM_OF_SENSORS; ++i) {
                sprintf(str, "%ld", (int32) MEM_P->enc[ENC_L].m_off[i] >> MEM_P->enc[ENC_L].res[i]);
                strcat(info_string, str);
```

```
            if (i != NUM_OF_SENSORS-1){
                strcat(info_string, ", ");
            }
        }
        strcat(info_string, "\r\n");

        strcat(info_string, "Measurement Multiplier: ");
        for (i = 0; i < NUM_OF_SENSORS; ++i) {
            sprintf(str,"%f", (float) MEM_P->enc[ENC_L].m_mult[i]);
            strcat(info_string, str);
            if (i != NUM_OF_SENSORS-1){
                strcat(info_string, ", ");
            }
        }
        strcat(info_string, "\r\n");

        sprintf(str, "Current lookup table: %f, %f, %f, %f, %f, %f\r\n",
            MOT->curr_lookup[0], MOT->curr_lookup[1], MOT->curr_lookup[2],
            MOT->curr_lookup[3], MOT->curr_lookup[4], MOT->curr_lookup[5]);
        strcat(info_string, str);

        sprintf(str, "Position limit active: %d", (int)MOT->pos_lim_flag);
        strcat(info_string, str);
        strcat(info_string, "\r\n");

        sprintf(str, "Position limit motor: inf -> %ld  ", (int32)MOT->
pos_lim_inf >> MEM_P->enc[ENC_L].res[0]);
        strcat(info_string, str);
        sprintf(str, "sup -> %ld\r\n", (int32)MOT->pos_lim_sup >> MEM_P->
enc[ENC_L].res[0]);
        strcat(info_string, str);

        sprintf(str, "Max step pos and neg: %d %d", (int)MOT->max_step_pos
, (int)MOT->max_step_neg);
        strcat(info_string, str);
        strcat(info_string, "\r\n");

        sprintf(str, "Current limit: %d\r\n", (int)MOT->current_limit);
        strcat(info_string, str);
#ifdef GENERIC_FW       // decided not to show when using SOFTHAND_FW to 
streamline ping, since these parameters are not settable
        sprintf(str, "Motor board associated encoder line: %d\r\n", (int)
MOT->encoder_line);
        strcat(info_string, str);
        switch(MOT->motor_driver_type) {
            case DRIVER_MC33887:
                strcat(info_string, "Driver type: MC33887 (Standard)\r\n");
                break;
            case DRIVER_VNH5019:
                strcat(info_string, "Driver type: VNH5019 (High power)\r\n
");
                break;
```

```
            case DRIVER_BRUSHLESS:
                strcat(info_string, "Driver type: ESC (Brushless)\r\n");
                break;
            default:
                break;
            }
            sprintf(str, "PWM rate limiter value: %d\r\n", (int)MOT->
pwm_rate_limiter);
            strcat(info_string, str);
            if (MOT->not_revers_motor_flag)
                strcat(info_string, "Not reversible motor: YES\r\n");
            else
                strcat(info_string, "Not reversible motor: NO\r\n");
#endif
            strcat(info_string, "\r\n");
        }

        strcat(info_string, "EMG CONFIGURATION\r\n");
        sprintf(str, "EMG thresholds [0 - 1024]: %u, %u", MEM_P->emg.
emg_threshold[0], MEM_P->emg.emg_threshold[1]);
        strcat(info_string, str);
        strcat(info_string, "\r\n");

        sprintf(str, "EMG max values [0 - 4096]: %lu, %lu", MEM_P->emg.
emg_max_value[0], MEM_P->emg.emg_max_value[1]);
        strcat(info_string, str);
        strcat(info_string, "\r\n");

        if (MEM_P->emg.switch_emg)
            strcat(info_string, "EMG inversion: YES\r\n");
        else
            strcat(info_string, "EMG inversion: NO\r\n");

        if (MEM_P->emg.emg_calibration_flag)
            strcat(info_string, "Calibration enabled: YES\r\n");
        else
            strcat(info_string, "Calibration enabled: NO\r\n");

        sprintf(str, "EMG max speed: %d %d", (int)MEM_P->emg.emg_speed[0], (
int)MEM_P->emg.emg_speed[1]);
        strcat(info_string, str);
        strcat(info_string, "\r\n");

        if (MEM_P->exp.read_ADC_sensors_port_flag == TRUE){
            strcat(info_string, "Additional ADC sensors value:\r\n");
            for (i = 0; i < NUM_OF_ADDITIONAL_EMGS; ++i) {
                sprintf(str,"ADC %d -> %d", (int)(i + 1), (int) g_adc_meas.
add_emg[i]);
                strcat(info_string, str);
                strcat(info_string,"\r\n");
            }
            for (i = 0; i < NUM_OF_INPUT_EMGS; ++i) {
```

```
                sprintf(str,"EMG input %d -> %d", (int)(i + 1), (int) g_adc_meas.emg[i]);
                strcat(info_string, str);
                strcat(info_string,"\r\n");
            }
        }

#ifdef GENERIC_FW

        strcat(info_string, "\r\n");
        strcat(info_string, "JOYSTICK CONFIGURATION\r\n");
        sprintf(str, "Closure speed: %d", c_mem.JOY_spec.joystick_closure_speed);
        strcat(info_string, str);
        strcat(info_string, "\r\n");

        sprintf(str, "Joystick Threshold: %d", c_mem.JOY_spec.joystick_threshold);
        strcat(info_string, str);
        strcat(info_string, "\r\n");
        sprintf(str, "Joystick Gains - X:%hu  Y:%hu", c_mem.JOY_spec.joystick_gains[0], c_mem.JOY_spec.joystick_gains[1]);
        strcat(info_string, str);
        strcat(info_string, "\r\n");

#endif

        if (MEM_P->SH.rest_position_flag) {
            sprintf(str, "Rest time delay (ms): %d", (int)MEM_P->SH.rest_delay);
            strcat(info_string, str);
            strcat(info_string, "\r\n");

            sprintf(str, "Rest velocity closure (ticks/sec): %d", (int)MEM_P->SH.rest_vel);
            strcat(info_string, str);
            strcat(info_string, "\r\n");

            sprintf(str, "Rest position: %d", (int)(MEM_P->SH.rest_pos >> MEM_P->enc[MEM_P->motor[0].encoder_line].res[0]));
            strcat(info_string, str);
            strcat(info_string, "\r\n");
        }

        if (MEM_P->imu.read_imu_flag) {
            sprintf(str, "IMU Connected: %d\r\n", (int) N_IMU_Connected);
            strcat(info_string, str);

            strcat(info_string, "\r\n");

            strcat(info_string, "IMUs CONFIGURATION\r\n");
            for (i=0; i<N_IMU_Connected; i++){
```

```
            sprintf(str, "Imu %d \r\n\tID: %d\r\n", i, (int) IMU_connected[i]);
            strcat(info_string, str);

            sprintf(str, "\tAccelerometers: ");
            if ((MEM_P->imu.IMU_conf[IMU_connected[i]][0]))
                strcat(str, "YES\r\n");
            else
                strcat(str, "NO\r\n");
            strcat(str, "\tGyroscopes: ");
            if ((MEM_P->imu.IMU_conf[IMU_connected[i]][1]))
                strcat(str, "YES\r\n");
            else
                strcat(str, "NO\r\n");
            strcat(str, "\tMagnetometers: ");
            if ((MEM_P->imu.IMU_conf[IMU_connected[i]][2]))
                strcat(str, "YES\r\n");
            else
                strcat(str, "NO\r\n");
            strcat(str, "\tQuaternion: ");
            if ((MEM_P->imu.IMU_conf[IMU_connected[i]][3]))
                strcat(str, "YES\r\n");
            else
                strcat(str, "NO\r\n");
            strcat(str, "\tTemperature: ");
            if ((MEM_P->imu.IMU_conf[IMU_connected[i]][4]))
                strcat(str, "YES\r\n");
            else
                strcat(str, "NO\r\n");

            strcat(info_string, str);
        }

        strcat(info_string, "\r\n");

        IMU_reading_info(info_string);
    }

    strcat(info_string, "\r\n");

#ifdef GENERIC_FW
    int j;
    strcat(info_string, "ENCODER CONFIGURATION\r\n");
    for (i = 0; i < N_ENCODER_LINE_MAX; i++) {
        sprintf(str, "Encoder Connected Line %d: %d", (int) i, (int) N_Encoder_Line_Connected[i]);
        strcat(info_string, str);
        if (MEM_P->dev.right_left == i) {
            sprintf(str, " [%s HAND main encoder line]", (MEM_P->dev.right_left?"LEFT":"RIGHT"));
            strcat(info_string, str);
        }
```

```
            strcat(info_string, "\r\n");
            if (N_Encoder_Line_Connected[i] > 0) {
                strcat(info_string, "Raw value [status]:\r\n");
                for (j = 0; j < N_Encoder_Line_Connected[i]; j++) {
                    sprintf(str, "%d\t[%s]\r\n", (uint16) Encoder_Value[i][j]
, (Encoder_Check[i][j]==16?"OK":"X"));
                    strcat(info_string, str);
                }
                strcat(info_string, "\r\n");
            }
        }
#endif

#ifdef MASTER_FW
        if (MEM_P->MS.slave_comm_active)
            strcat(info_string, "Slave communication active: YES\r\n");
        else
            strcat(info_string, "Slave communication active: NO\r\n");

        sprintf(str, "Slave ID: %d\r\n", (int)MEM_P->MS.slave_ID);
        strcat(info_string, str);
#endif

        sprintf(str, "Last FW cycle time: %u us\r\n", (uint16)timer_value0 - (
uint16)timer_value);
        strcat(info_string, str);

        strcat(info_string, "\r\n\0");      // End of info_string
    }
}

//
//============================================================================
//                                                  PREPARE GENERIC COUNTERS
INFO
//
//============================================================================

void prepare_counter_info(char *info_string)
{
    char str[100];
    int i;
    int step;

    struct st_eeprom* MEM_P = &g_mem;
    struct st_motor* MOT = &(MEM_P->motor[0]);      // Default motor
    uint8 ENC_L = MOT->encoder_line;                // Associated encoder line

    strcpy(info_string, "");

    strcat(info_string, "\r\nUSAGE STATISTICS\r\n");
    strcat(info_string, "\r\n");
```

```
    sprintf(str, "Date of HW maintenance: %02d/%02d/20%02d\r\n", (int)MEM_P->dev.hw_maint_date[0], (int)MEM_P->dev.hw_maint_date[1], (int)MEM_P->dev.hw_maint_date[2]);
    strcat(info_string, str);

    sprintf(str, "Date of usage stats period begin: %02d/%02d/20%02d\r\n", (int)MEM_P->dev.stats_period_begin_date[0], (int)MEM_P->dev.stats_period_begin_date[1], (int)MEM_P->dev.stats_period_begin_date[2]);
    strcat(info_string, str);

    sprintf(str, "Last checked Time: %02d/%02d/20%02d %02d:%02d:%02d\r\n", (int)MEM_P->exp.curr_time[0], (int)MEM_P->exp.curr_time[1], (int)MEM_P->exp.curr_time[2], (int)MEM_P->exp.curr_time[3], (int)MEM_P->exp.curr_time[4], (int)MEM_P->exp.curr_time[5]);
    strcat(info_string, str);

    sprintf(str, "Positions histogram (ticks):\r\n");
    strcat(info_string, str);
    step = ( (int)(MOT->pos_lim_sup >> MEM_P->enc[ENC_L].res[0]) / 10);
    for (i=1; i<=10;i++){
        sprintf(str, "Bin %d [%d-%d]: %lu\r\n", i, step*(i-1)+1, step*(i), MEM_P->cnt.position_hist[i-1]);
        strcat(info_string, str);
    }
    strcat(info_string, "\r\n");

    sprintf(str, "Current histogram (mA):\r\n");
    strcat(info_string, str);
    step = ( (int)(MOT->current_limit) / 4);
    for (i=1; i<=4;i++){
        sprintf(str, "Threshold %d [%d-%d]: %lu\r\n", i, step*(i-1), step*(i), MEM_P->cnt.current_hist[i-1]);
        strcat(info_string, str);
    }
    strcat(info_string, "\r\n");

    sprintf(str, "Motions through EMG counter: %lu, %lu", MEM_P->cnt.motion_counter[0], MEM_P->cnt.motion_counter[1]);
    strcat(info_string, str);
    strcat(info_string, "\r\n");

    sprintf(str, "Rest position occurrences: %lu", MEM_P->cnt.rest_counter);
    strcat(info_string, str);
    strcat(info_string, "\r\n");

    sprintf(str, "Angle total displacement (ticks): %lu", MEM_P->cnt.wire_disp);
    strcat(info_string, str);
    strcat(info_string, "\r\n");

    sprintf(str, "Total power on time (sec): %lu", MEM_P->cnt.total_runtime);
```

```
    strcat(info_string, str);
    strcat(info_string, "\r\n");

    sprintf(str, "Total rest position time (sec): %lu", MEM_P->cnt.
total_time_rest);
    strcat(info_string, str);
    strcat(info_string, "\r\n");


    // R01 Project statistics (some are duplicated)
    char CYDATA R01_str[300];
    prepare_R01_info(R01_str);
    strcat(info_string, "\r\n");
    strcat(info_string, R01_str);
}

//
================================================================================
//                                                    PREPARE GENERIC COUNTERS
INFO
//
================================================================================

void prepare_R01_info(char *info_string)
{
    char str[150];

    struct st_eeprom* MEM_P = &g_mem;

    strcpy(info_string, "");

    strcat(info_string, "R01 PROJECT STATISTICS\r\n");
    strcat(info_string, "\r\n");

    sprintf(str, "Power cycles: %lu", MEM_P->cnt.power_cycles);
    strcat(info_string, str);
    strcat(info_string, "\r\n");

    sprintf(str, "EMG activations counter: %lu, %lu", MEM_P->cnt.
emg_act_counter[0], MEM_P->cnt.emg_act_counter[1]);
    strcat(info_string, str);
    strcat(info_string, "\r\n");

    sprintf(str, "Number of motions (close/open): %lu, %lu", MEM_P->cnt.
motion_counter[0], MEM_P->cnt.motion_counter[1]);
    strcat(info_string, str);
    strcat(info_string, "\r\n");

    sprintf(str, "Excessive signal activity (close/open): %lu, %lu", MEM_P->
cnt.excessive_signal_activity[0], MEM_P->cnt.excessive_signal_activity[1]);
    strcat(info_string, str);
    strcat(info_string, "\r\n");
```

```
    sprintf(str, "Total runtime (sec): %lu", MEM_P->cnt.total_runtime);
    strcat(info_string, str);
    strcat(info_string, "\r\n");

    sprintf(str, "Average duration of a powered-on session (sec): %.4f", (float)(MEM_P->cnt.total_runtime / MEM_P->cnt.power_cycles));
    strcat(info_string, str);
    strcat(info_string, "\r\n");

    sprintf(str, "Frequency of motions: %.4f", (float)((MEM_P->cnt.emg_act_counter[0] + MEM_P->cnt.emg_act_counter[1]) / (float)MEM_P->cnt.total_runtime));
    strcat(info_string, str);
    strcat(info_string, "\r\n");

}

//=============================================================================
//                                                    PREPARE SD CARD PARAM INFO
//=============================================================================

void prepare_SD_param_info(char *info_string)
{
    char str[100];
    int i;

    // NOTE: use g_mem structure instead of c_mem because when changing parameters c_mem struct is not updated yet

    struct st_eeprom* MEM_P = &g_mem;

    sprintf(info_string, "Firmware version: %s\r\n", VERSION);

    sprintf(str, "ID: %d\r\n", (int) MEM_P->dev.id);
    strcat(info_string, str);
    switch(MEM_P->dev.right_left){
        case RIGHT_HAND:
            strcat(info_string, "Hand side: RIGHT\r\n");
            break;
        case LEFT_HAND:
            strcat(info_string, "Hand side: LEFT\r\n");
            break;
    }

    sprintf(str, "Date of HW maintenance: %02d/%02d/20%02d\r\n", (int)MEM_P->dev.hw_maint_date[0], (int)MEM_P->dev.hw_maint_date[1], (int)MEM_P->dev.hw_maint_date[2]);
    strcat(info_string, str);
```

```
    sprintf(str, "Date of usage stats period begin: %02d/%02d/20%02d\r\n", (int)MEM_P->dev.stats_period_begin_date[0], (int)MEM_P->dev.stats_period_begin_date[1], (int)MEM_P->dev.stats_period_begin_date[2]);
    strcat(info_string, str);

    for (uint8 k = 0; k <= MEM_P->dev.use_2nd_motor_flag; k++) {

        uint8 MOTOR_IDX = k;
        struct st_motor* MOT = &(MEM_P->motor[MOTOR_IDX]);      // Default motor
        uint8 ENC_L = MOT->encoder_line;             // Associated encoder line

        sprintf(str, "MOTOR %d INFO\r\n", MOTOR_IDX+1);
        strcat(info_string, str);

        strcat(info_string, "PWM rescaling activation: ");
        if(MOT->activate_pwm_rescaling == MAXON_12V)
            strcat(info_string, "YES\r\n");
        else
            strcat(info_string, "NO\r\n");

        sprintf(str, "PWM Limit: %d\r\n", (int) dev_pwm_limit[MOTOR_IDX]);
        strcat(info_string, str);

        strcat(info_string, "Position PID: ");
        if(MOT->control_mode != CURR_AND_POS_CONTROL) {
            sprintf(str, "P -> %f  ", ((double) MOT->k_p / 65536));
            strcat(info_string, str);
            sprintf(str, "I -> %f  ", ((double) MOT->k_i / 65536));
            strcat(info_string, str);
            sprintf(str, "D -> %f\r\n", ((double) MOT->k_d / 65536));
            strcat(info_string, str);
        }
        else {
            sprintf(str, "P -> %f  ", ((double) MOT->k_p_dl / 65536));
            strcat(info_string, str);
            sprintf(str, "I -> %f  ", ((double) MOT->k_i_dl / 65536));
            strcat(info_string, str);
            sprintf(str, "D -> %f\r\n", ((double) MOT->k_d_dl / 65536));
            strcat(info_string, str);
        }

        strcat(info_string, "Current PID: ");
        if(MOT->control_mode != CURR_AND_POS_CONTROL) {
            sprintf(str, "P -> %f  ", ((double) MOT->k_p_c / 65536));
            strcat(info_string, str);
            sprintf(str, "I -> %f  ", ((double) MOT->k_i_c / 65536));
            strcat(info_string, str);
            sprintf(str, "D -> %f\r\n", ((double) MOT->k_d_c / 65536));
            strcat(info_string, str);
```

```
        }
        else {
            sprintf(str, "P -> %f  ", ((double) MOT->k_p_c_dl / 65536));
            strcat(info_string, str);
            sprintf(str, "I -> %f  ", ((double) MOT->k_i_c_dl / 65536));
            strcat(info_string, str);
            sprintf(str, "D -> %f\r\n", ((double) MOT->k_d_c_dl / 65536));
            strcat(info_string, str);
        }
        if (MOT->activ == 0x01)
            strcat(info_string, "Startup activation: YES\r\n");
        else
            strcat(info_string, "Startup activation: NO\r\n");

        switch(MOT->input_mode) {
            case INPUT_MODE_EXTERNAL:
                strcat(info_string, "Input mode: USB\r\n");
                break;
            case INPUT_MODE_ENCODER3:
                strcat(info_string, "Input mode: Handle\r\n");
                break;
            case INPUT_MODE_EMG_PROPORTIONAL:
                strcat(info_string, "Input mode: EMG proportional\r\n");
                break;
            case INPUT_MODE_EMG_INTEGRAL:
                strcat(info_string, "Input mode: EMG integral\r\n");
                break;
            case INPUT_MODE_EMG_FCFS:
                strcat(info_string, "Input mode: EMG FCFS\r\n");
                break;
            case INPUT_MODE_EMG_FCFS_ADV:
                strcat(info_string, "Input mode: EMG FCFS ADV\r\n");
                break;
            case INPUT_MODE_JOYSTICK:
                strcat(info_string, "Input mode: Joystick\r\n");
                break;
            case INPUT_MODE_EMG_PROPORTIONAL_NC:
                strcat(info_string, "Input mode: EMG proportional Normally ↵
Closed\r\n");
                break;
        }

        switch(MOT->control_mode) {
            case CONTROL_ANGLE:
                strcat(info_string, "Control mode: Position\r\n");
                break;
            case CONTROL_PWM:
                strcat(info_string, "Control mode: PWM\r\n");
                break;
            case CONTROL_CURRENT:
                strcat(info_string, "Control mode: Current\r\n");
                break;
```

```
            case CURR_AND_POS_CONTROL:
                strcat(info_string, "Control mode: Position and Current\r\n");
                break;
            default:
                break;
        }

        if (MEM_P->enc[ENC_L].double_encoder_on_off)
            strcat(info_string, "Absolute encoder position: YES\r\n");
        else
            strcat(info_string, "Absolute encoder position: NO\r\n");

        strcat(info_string, "Resolutions: ");
        for (i = 0; i < NUM_OF_SENSORS; ++i) {
            sprintf(str, "%d", (int) MEM_P->enc[ENC_L].res[i]);
            strcat(info_string, str);
            if (i != NUM_OF_SENSORS-1){
                strcat(info_string, ", ");
            }
        }
        strcat(info_string, "\r\n");

        strcat(info_string, "Offsets: ");
        for (i = 0; i < NUM_OF_SENSORS; ++i) {
            sprintf(str, "%ld", (int32) MEM_P->enc[ENC_L].m_off[i] >> MEM_P->
enc[ENC_L].res[i]);
            strcat(info_string, str);
            if (i != NUM_OF_SENSORS-1){
                strcat(info_string, ", ");
            }
        }
        strcat(info_string, "\r\n");

        strcat(info_string, "Multipliers: ");
        for (i = 0; i < NUM_OF_SENSORS; ++i) {
            sprintf(str,"%f", (float) MEM_P->enc[ENC_L].m_mult[i]);
            strcat(info_string, str);
            if (i != NUM_OF_SENSORS-1){
                strcat(info_string, ", ");
            }
        }
        strcat(info_string, "\r\n");

        sprintf(str, "Current lookup table p[0] - p[5]: %f, %f, %f, %f, %f, %f
\r\n", MOT->curr_lookup[0], MOT->curr_lookup[1], MOT->curr_lookup[2], MOT->
curr_lookup[3], MOT->curr_lookup[4], MOT->curr_lookup[5]);
        strcat(info_string, str);

        sprintf(str, "Position limit active: %d\r\n", (int)MOT->pos_lim_flag);
        strcat(info_string, str);

        sprintf(str, "Position limits: inf -> %ld, sup -> %ld\r\n", (int32)MOT
```

```
->pos_lim_inf >> MEM_P->enc[ENC_L].res[0], (int32)MOT->pos_lim_sup >> MEM_P->enc[ENC_L].res[0]);
        strcat(info_string, str);

        sprintf(str, "Current limit: %d\r\n", (int)MOT->current_limit);
        strcat(info_string, str);

        if ((MOT->input_mode == INPUT_MODE_EMG_PROPORTIONAL) ||
            (MOT->input_mode == INPUT_MODE_EMG_INTEGRAL) ||
            (MOT->input_mode == INPUT_MODE_EMG_FCFS) ||
            (MOT->input_mode == INPUT_MODE_EMG_FCFS_ADV) ||
            (MOT->input_mode == INPUT_MODE_EMG_PROPORTIONAL_NC)) {
            sprintf(str, "EMG thresholds [0 - 1024]: %u, %u", MEM_P->emg.emg_threshold[0], MEM_P->emg.emg_threshold[1]);
            strcat(info_string, str);
            strcat(info_string, "\r\n");

            sprintf(str, "EMG max values [0 - 4096]: %lu, %lu", MEM_P->emg.emg_max_value[0], MEM_P->emg.emg_max_value[1]);
            strcat(info_string, str);
            strcat(info_string, "\r\n");

            if (MEM_P->emg.emg_calibration_flag)
                strcat(info_string, "Calibration enabled: YES\r\n");
            else
                strcat(info_string, "Calibration enabled: NO\r\n");

            sprintf(str, "EMG max speed: %d %d", (int)MEM_P->emg.emg_speed[0], (int)MEM_P->emg.emg_speed[1]);
            strcat(info_string, str);
            strcat(info_string, "\r\n");
        }
    }

    if (MEM_P->SH.rest_position_flag) {
        sprintf(str, "Rest time delay (ms): %d", (int)MEM_P->SH.rest_delay);
        strcat(info_string, str);
        strcat(info_string, "\r\n");

        sprintf(str, "Rest velocity closure (ticks/sec): %d", (int)MEM_P->SH.rest_vel);
        strcat(info_string, str);
        strcat(info_string, "\r\n");

        sprintf(str, "Rest position: %d", (int)(MEM_P->SH.rest_pos >> MEM_P->enc[MEM_P->motor[0].encoder_line].res[0]));
        strcat(info_string, str);
        strcat(info_string, "\r\n");
    }

    if (MEM_P->imu.read_imu_flag) {
        sprintf(str, "IMU Connected: %d\r\n", (int) N_IMU_Connected);
```

```
        strcat(info_string, str);

        strcat(info_string, "\r\n");

        strcat(info_string, "IMUs CONFIGURATION\r\n");
        for (i=0; i<N_IMU_Connected; i++){
            sprintf(str, "Imu %d \r\n\tID: %d\r\n", i, (int) IMU_connected[i]);
            strcat(info_string, str);

            sprintf(str, "\tAccelerometers: ");
            if ((MEM_P->imu.IMU_conf[IMU_connected[i]][0]))
                strcat(str, "YES\r\n");
            else
                strcat(str, "NO\r\n");
            strcat(str, "\tGyroscopes: ");
            if ((MEM_P->imu.IMU_conf[IMU_connected[i]][1]))
                strcat(str, "YES\r\n");
            else
                strcat(str, "NO\r\n");
            strcat(str, "\tMagnetometers: ");
            if ((MEM_P->imu.IMU_conf[IMU_connected[i]][2]))
                strcat(str, "YES\r\n");
            else
                strcat(str, "NO\r\n");
            strcat(str, "\tQuaternion: ");
            if ((MEM_P->imu.IMU_conf[IMU_connected[i]][3]))
                strcat(str, "YES\r\n");
            else
                strcat(str, "NO\r\n");
            strcat(str, "\tTemperature: ");
            if ((MEM_P->imu.IMU_conf[IMU_connected[i]][4]))
                strcat(str, "YES\r\n");
            else
                strcat(str, "NO\r\n");

            strcat(info_string, str);
        }

        strcat(info_string, "\r\n");
    }
}


//↵
//============================================================================
//                                                          PREPARE SD CARD ↵
LEGEND
//↵
//============================================================================
void prepare_SD_legend(char *info_string)
{
    char str[140];
```

```
    int i;

    // Legend
    strcpy(info_string, "Hour,Min,Sec,");
    for (i=1; i<=10;i++){       // Position bins
        sprintf(str, "Bin_%d_Pos,", i);
        strcat(info_string, str);
    }
    for (i=1; i<=4;i++){        // Current bins
        sprintf(str, "Bin_%d_Curr,", i);
        strcat(info_string, str);
    }
    sprintf(str, "Rest_times,Wire_disp,Total_rest_time,Power_cycles,EMG_1_act,EMG_2_act,EMG_1_excess,EMG_2_excess,Motion_1,Motion_2,Total_runtime");
    strcat(info_string, str);
    strcat(info_string, "\r\n");
}


//=============================================================================
//                                                          PREPARE SD CARD INFO
//=============================================================================
void prepare_SD_info(char *info_string)
{
    char str[120];
    int i;

    strcpy(info_string, "");

    // Time
    strcat(info_string, "");
    sprintf(str, "%02d,%02d,%02d,", (int)g_mem.exp.curr_time[3], (int)g_mem.exp.curr_time[4], (int)g_mem.exp.curr_time[5]);
    strcat(info_string, str);

    // Pos_Bin
    for (i=1; i<=10;i++){
        sprintf(str, "%lu,", g_mem.cnt.position_hist[i-1]);
        strcat(info_string, str);
    }

    // Curr_Bin
    for (i=1; i<=4;i++){
        sprintf(str, "%lu,", g_mem.cnt.current_hist[i-1]);
        strcat(info_string, str);
    }

    // Rest_times, Wire_disp, Total_time_rest, Power_cycles
    sprintf(str, "%lu,%lu,%lu,%lu,", g_mem.cnt.rest_counter, g_mem.cnt.
```

```
wire_disp, g_mem.cnt.total_time_rest, g_mem.cnt.power_cycles);
    strcat(info_string, str);

    // EMG_1_act, EMG_2_act, EMG_1_excess, EMG_2_excess, Motion_1, Motion_2
    sprintf(str, "%lu,%lu,%lu,%lu,%lu,%lu,", g_mem.cnt.emg_act_counter[0], ↵
g_mem.cnt.emg_act_counter[1],
                                        g_mem.cnt.excessive_signal_activity[0], ↵
g_mem.cnt.excessive_signal_activity[1],
                                        g_mem.cnt.motion_counter[0], g_mem.cnt.↵
motion_counter[1]);
    strcat(info_string, str);

    // Total_runtime
    sprintf(str, "%lu", g_mem.cnt.total_runtime);
    strcat(info_string, str);

    strcat(info_string, "\r\n");
}

//↵
================================================================================
//                                                PREPARE SD CARD EMG HISTORY ↵
LEGEND
//↵
================================================================================
void prepare_SD_EMG_History_legend(char *info_string)
{

    // Legend
    strcpy(info_string, "Time,EMG1,EMG2\n");

}

//↵
================================================================================
//                                                        PREPARE SD CARD EMG ↵
HISTORY
//↵
================================================================================
void prepare_SD_EMG_history(char *info_string)
{
    char str_data[100] = "";
    uint16 v_idx = 0;
    static float h_time = 0.0;

    strcpy(info_string, "");

    // Oldest samples of the vector
    for (int i = 0; i < SAMPLES_FOR_EMG_HISTORY; i++){

        // Send line per line all the history vector
        // First line (oldest) is made by values of emg_history_next_idx ↵
```

```
index (they will be the next to be updated)
        v_idx = emg_history_next_idx + i;
        if (v_idx > SAMPLES_FOR_EMG_HISTORY){
            v_idx -= SAMPLES_FOR_EMG_HISTORY;
        }

        // Time vector is reconstructed setting oldest samples as t=0.0s
        sprintf(str_data, "%.1f,%u,%u\n", h_time, emg_history[v_idx][0], ↵
emg_history[v_idx][1]);

        strcat(info_string, str_data);

        h_time += 0.2;      // Row time interval is 200ms (5Hz)

    }

}

//↵
================================================================================
//                                                                  IMU READING ↵
INFO
//↵
================================================================================

void IMU_reading_info(char *info_string)
{
    char str[300];
    int i;

    strcat(info_string, "SENSORS INFO\r\n");
    for (i=0; i<N_IMU_Connected; i++){
        sprintf(str, "Imu %d \r\n\tID: %d\r\n", i, (int) IMU_connected[i]);
        strcat(info_string, str);

        if ((c_mem.imu.IMU_conf[IMU_connected[i]][0])){
            sprintf(str, "\tAcc: %d\t%d\t%d\r\n", (int16) g_imu[i].accel_value↵
[0], (int16) g_imu[i].accel_value[1],(int16) g_imu[i].accel_value[2]);
            strcat(info_string, str);
        }

        if ((c_mem.imu.IMU_conf[IMU_connected[i]][1])){
            sprintf(str, "\tGyro: %d\t%d\t%d\r\n", (int16) g_imu[i].gyro_value↵
[0], (int16) g_imu[i].gyro_value[1],(int16) g_imu[i].gyro_value[2]);
            strcat(info_string, str);
        }

        if ((c_mem.imu.IMU_conf[IMU_connected[i]][2])){
            sprintf(str, "\tMag: %d\t%d\t%d\r\n", (int16) g_imu[i].mag_value[0↵
], (int16) g_imu[i].mag_value[1],(int16) g_imu[i].mag_value[2]);
            strcat(info_string, str);
        }
```

```
        if ((c_mem.imu.IMU_conf[IMU_connected[i]][3])){
            sprintf(str, "\tQuat: %.3f\t%.3f\t%.3f\t%.3f\r\n", (float) g_imu[i
].quat_value[0], (float) g_imu[i].quat_value[1],(float) g_imu[i].quat_value[2]
, (float) g_imu[i].quat_value[3]);
            strcat(info_string, str);
        }

        if ((c_mem.imu.IMU_conf[IMU_connected[i]][4])){
            sprintf(str, "\tTemperature: %d\r\n", (int16) g_imu[i].temp_value);
            strcat(info_string, str);
        }

    }
    strcat(info_string, "\r\n");
}

//
================================================================================
//                                                    WRITE FUNCTIONS FOR
RS485
//
================================================================================

void commWrite_old_id(uint8 *packet_data, uint16 packet_lenght, uint8 old_id)
{
    uint16 CYDATA index;    // iterator

    // frame - start
    UART_RS485_PutChar(':');
    UART_RS485_PutChar(':');
    // frame - ID
    //if(old_id)
        UART_RS485_PutChar(old_id);
    //else
        //UART_RS485_PutChar(g_mem.id);

    // frame - length
    UART_RS485_PutChar((uint8)packet_lenght);
    // frame - packet data
    for(index = 0; index < packet_lenght; ++index) {
        UART_RS485_PutChar(packet_data[index]);
    }

    index = 0;

    while(!(UART_RS485_ReadTxStatus() & UART_RS485_TX_STS_COMPLETE) && index
++ <= 1000){}

    RS485_CTS_Write(1);
    RS485_CTS_Write(0);
}
```

```
void commWrite(uint8 *packet_data, uint16 packet_lenght)
{
    uint16 CYDATA index;    // iterator

    // frame - start
    UART_RS485_PutChar(':');
    UART_RS485_PutChar(':');
    // frame - ID
    UART_RS485_PutChar(g_mem.dev.id);
    // frame - length
    UART_RS485_PutChar((uint8)packet_lenght);
    // frame - packet data
    for(index = 0; index < packet_lenght; ++index) {
        UART_RS485_PutChar(packet_data[index]);
    }

    index = 0;

    while(!(UART_RS485_ReadTxStatus() & UART_RS485_TX_STS_COMPLETE) && index↵
++ <= 1000){}

    RS485_CTS_Write(1);
    RS485_CTS_Write(0);
}

//↵
================================================================================
//                                                  WRITE FUNCTION FOR ANOTHER ↵
DEVICE
//↵
================================================================================

void commWriteID(uint8 *packet_data, uint16 packet_lenght, uint8 id)
{
    static uint16 CYDATA i;    // iterator

    // frame - start
    UART_RS485_PutChar(':');
    UART_RS485_PutChar(':');
    // frame - ID
    UART_RS485_PutChar(id);
    // frame - length
    UART_RS485_PutChar((uint8)packet_lenght);
    // frame - packet data
    for(i = 0; i < packet_lenght; ++i) {
        UART_RS485_PutChar(packet_data[i]);
    }

    i = 0;

    while(!(UART_RS485_ReadTxStatus() & UART_RS485_TX_STS_COMPLETE) && i++ <= ↵
```

```
1000){}

    RS485_CTS_Write(1);
    RS485_CTS_Write(0);
}

//↵
=========================================================================
//                                                              CHECKSUM ↵
FUNCTION
//↵
=========================================================================

// Performs a XOR byte by byte on the entire vector

uint8 LCRChecksum(uint8 *data_array, uint8 data_length) {

    uint8 CYDATA i;
    uint8 CYDATA checksum = 0x00;

    for(i = 0; i < data_length; ++i)
       checksum ^= data_array[i];

    return checksum;
}


//↵
=========================================================================
//                                                      ACKNOWLEDGMENT ↵
FUNCTION
//↵
=========================================================================

void sendAcknowledgment(uint8 value) {
    int packet_lenght = 2;
    uint8 packet_data[2];

    packet_data[0] = value;
    packet_data[1] = value;

    commWrite(packet_data, packet_lenght);
}

//↵
=========================================================================
//                                                              STORE ↵
MEMORY
//↵
=========================================================================
```

```
uint8 memStore(int displacement)
{
    int i;  // iterator
    uint8 writeStatus;
    int pages;
    uint8 ret_val = 1;

    // Disable Interrupt
    ISR_RS485_RX_Disable();

    // Stop motor
    PWM_MOTORS_WriteCompare1(0);

    // Update temperature information for better writing performance
    EEPROM_UpdateTemperature();

    memcpy( &c_mem, &g_mem, sizeof(g_mem) );

    pages = sizeof(g_mem) / 16 + (sizeof(g_mem) % 16 > 0);

    for(i = 0; i < pages; ++i) {
        writeStatus = EEPROM_Write((uint8*)&g_mem.flag + 16 * i, i + displacement);
        if(writeStatus != CYRET_SUCCESS) {
            ret_val = 0;
            break;
        }
    }

    memcpy( &g_mem, &c_mem, sizeof(g_mem) );

    // Re-Enable Interrupt
    ISR_RS485_RX_Enable();

    return ret_val;
}


//
================================================================================
//                                                                    RECALL
MEMORY
//
================================================================================


void memRecall(void)
{
    uint16 i;

    for (i = 0; i < sizeof(g_mem); i++) {
        ((reg8 *) &g_mem.flag)[i] = EEPROM_ADDR[i];
```

```
    }

    // Recall saved user_emg structure
    memcpy( &(g_mem.emg), &(g_mem.user[g_mem.dev.user_id].user_emg), sizeof(g_mem.emg) );

    //check for initialization
    if (g_mem.flag == FALSE) {
        memRestore();
    } else {
        memcpy( &c_mem, &g_mem, sizeof(g_mem) );
    }
}


//============================================================================
//                                                                RESTORE MEMORY
//============================================================================


uint8 memRestore(void) {
    uint16 i;

    for (i = 0; i < sizeof(g_mem); i++) {
        ((reg8 *) (uint8*)&g_mem.flag)[i] = EEPROM_ADDR[i + (DEFAULT_EEPROM_DISPLACEMENT * 16)];
    }

    //check for initialization
    if (g_mem.flag == FALSE) {
        return memInit();
    } else {
        return memStore(0);
    }
}

//============================================================================
//                                                                  MEMORY INIT
//============================================================================

uint8 memInit(void)
{
    uint8 i, j;

    //initialize memory settings
    for (i=0; i<15; i++){
```

```
        g_mem.unused_bytes[i] = 0;
    }

    // DEV STRUCT
    g_mem.dev.id                = 1;
    g_mem.dev.right_left        = RIGHT_HAND;
    g_mem.dev.dev_type          = GENERIC_2_MOTORS;
    g_mem.dev.reset_counters    = FALSE;
    reset_counters();                       //Initialize counters
    for (i=0; i<EEPROM_BYTES_ROW*EEPROM_AFTER_CNT_FREE_ROWS; i++){
        g_mem.unused_bytes1[i] = 0;
    }
    g_mem.dev.use_2nd_motor_flag = FALSE;

    // MOTOR STRUCT
    for (i=0; i< NUM_OF_MOTORS; i++) {
        g_mem.motor[i].k_p          =0.0165 * 65536;
        g_mem.motor[i].k_i          =     0 * 65536;
        g_mem.motor[i].k_d          = 0.007 * 65536;  // changed in order to ↵
avoid metallic clatter, previous value 0.2
        g_mem.motor[i].k_p_c        =     1 * 65536;
        g_mem.motor[i].k_i_c        = 0.001 * 65536;
        g_mem.motor[i].k_d_c        =     0 * 65536;

        g_mem.motor[i].k_p_dl       =   0.1 * 65536;
        g_mem.motor[i].k_i_dl       =     0 * 65536;
        g_mem.motor[i].k_d_dl       =     0 * 65536;
        g_mem.motor[i].k_p_c_dl     =   0.3 * 65536;
        g_mem.motor[i].k_i_c_dl     =0.0002 * 65536;
        g_mem.motor[i].k_d_c_dl     =     0 * 65536;

        g_mem.motor[i].activ        = 1;
        g_mem.motor[i].activate_pwm_rescaling = MAXON_24V;      //rescaling ↵
active for 12V motor
        g_mem.motor[i].motor_driver_type = DRIVER_MC33887;      //SoftHand ↵
standard driver
        g_mem.motor[i].input_mode    = INPUT_MODE_EXTERNAL;
        g_mem.motor[i].control_mode  = CONTROL_ANGLE;
        g_mem.motor[i].max_step_pos = 0;
        g_mem.motor[i].max_step_neg = 0;
        for(j = 0; j < LOOKUP_DIM; j++) {
            g_mem.motor[i].curr_lookup[j] = 0;
        }
        g_mem.motor[i].current_limit = DEFAULT_CURRENT_LIMIT;
        g_mem.motor[i].encoder_line = i;
        g_mem.motor[i].pos_lim_flag = 1;

        g_mem.motor[i].pwm_rate_limiter = PWM_RATE_LIMITER_MAX;
        g_mem.motor[i].not_revers_motor_flag = FALSE;       // Generic ↵
reversible motor
    }
```

```
    // ENC STRUCT
    for (i = 0; i< N_ENCODER_LINE_MAX; i++){
        for (j = 0; j<N_ENCODERS_PER_LINE_MAX; j++) {
            g_mem.enc[i].Enc_raw_read_conf[j] = 0;
        }
        for(j = 0; j < NUM_OF_SENSORS; j++){
            g_mem.enc[i].res[j] = 3;
            g_mem.enc[i].m_mult[j] = 1;
            g_mem.enc[i].m_off[j] = (int32)0 << g_mem.enc[i].res[j];
        }
        g_mem.enc[i].double_encoder_on_off = FALSE;
        g_mem.enc[i].motor_handle_ratio = 22;
        for(j = 0; j < NUM_OF_SENSORS; j++){
            g_mem.enc[i].Enc_idx_use_for_control[j] = j;    // First encoder ↵
is that with index 0 as default, then with index 1 and so on
        }
        g_mem.enc[i].gears_params[0] = 15;
        g_mem.enc[i].gears_params[1] = 14;
        g_mem.enc[i].gears_params[2] = 1;
    }

    for (i=0; i< NUM_OF_MOTORS; i++) {
        g_mem.motor[i].pos_lim_inf = 0;
        g_mem.motor[i].pos_lim_sup = (int32)19000 << g_mem.enc[g_mem.motor[i].↵
encoder_line].res[0];
    }

    // EMG STRUCT
    g_mem.emg.emg_threshold[0] = 200;
    g_mem.emg.emg_threshold[1] = 200;
    g_mem.emg.emg_max_value[0] = 1024;
    g_mem.emg.emg_max_value[1] = 1024;
    g_mem.emg.emg_speed[0] = 100;
    g_mem.emg.emg_speed[1] = 100;
    g_mem.emg.emg_calibration_flag = 0;      // EMG calibration disabled by ↵
default
    g_mem.emg.switch_emg = 0;

    // IMU STRUCT
    g_mem.imu.read_imu_flag = FALSE;
    g_mem.imu.SPI_read_delay = 0;       // 0 - No delay
    for (i = 0; i< N_IMU_MAX; i++){
        g_mem.imu.IMU_conf[i][0] = 1;   // Accelerometers
        g_mem.imu.IMU_conf[i][1] = 1;   // Gyroscopes
        g_mem.imu.IMU_conf[i][2] = 0;   // Magnetometers
        g_mem.imu.IMU_conf[i][3] = 0;   // Quaternions
        g_mem.imu.IMU_conf[i][4] = 0;   // Temperatures
    }

    // EXP STRUCT
    g_mem.exp.read_exp_port_flag = EXP_NONE;       // 0 - None
    strcpy(g_mem.user[g_mem.dev.user_id].user_code_string, "GEN001");
```

```
    if (g_mem.exp.read_exp_port_flag == EXP_SD_RTC) {
        // Set date of maintenance from RTC
        store_RTC_current_time();

        g_mem.dev.stats_period_begin_date[0] = g_mem.exp.curr_time[0];
        g_mem.dev.stats_period_begin_date[1] = g_mem.exp.curr_time[1];
        g_mem.dev.stats_period_begin_date[2] = g_mem.exp.curr_time[2];
    }
    g_mem.exp.read_ADC_sensors_port_flag = FALSE;
    for (i = 0; i < NUM_OF_ADC_CHANNELS_MAX; i++) {
        g_mem.exp.ADC_conf[i] = 0;
    }
    //Activate only the two EMG channels by default for every firmware ↵
configuration
    g_mem.exp.ADC_conf[2] = 1;
    g_mem.exp.ADC_conf[3] = 1;
    g_mem.exp.record_EMG_history_on_SD = FALSE;

    // WR STRUCT (default in generic fw)
    g_mem.WR.activation_mode = 0;                // Default Fast:wrist/syn2, ↵
Slow:hand/syn1
    g_mem.WR.fast_act_threshold[0] = 800;

#ifdef SOFTHAND_FW
    // Override memory values with specific ones for SoftHand Pro device
    memInit_SoftHandPro();
#endif

#ifdef MASTER_FW
    // Override memory values with specific ones for Master device
    memInit_Master();
#endif

#ifdef AIR_CHAMBERS_FB_FW
    // Override memory values with specific ones for Air Chambers device
    memInit_AirChambersFb();
#endif

#ifdef OTBK_ACT_WRIST_MS_FW
    // Override memory values with specific ones for Ottobock Wrist device
    memInit_OtbkActWristMs();
#endif

    // JOYSTICK STRUCT
    g_mem.JOY_spec.joystick_closure_speed = 150;
    g_mem.JOY_spec.joystick_threshold = 100;
    g_mem.JOY_spec.joystick_gains[0] = 1024;
    g_mem.JOY_spec.joystick_gains[1] = 1024;

    // Default generic user_id
    g_mem.dev.user_id = GENERIC_USER;
```

```
    // set the initialized flag to show EEPROM has been populated
    g_mem.flag = TRUE;

    //write that configuration to EEPROM
    return ( memStore(0) && memStore(DEFAULT_EEPROM_DISPLACEMENT) );
}

//
//////////////////////////////////////////////////////////////////////////////
//                                                                 MEMORY INIT
SOFTHAND
//
//////////////////////////////////////////////////////////////////////////////
void memInit_SoftHandPro(void)
{
    uint8 MOTOR_IDX = 0;

    //initialize memory settings ( Specific for SoftHand Pro device )
    g_mem.dev.right_left = LEFT_HAND;
    g_mem.dev.dev_type = SOFTHAND_PRO;

    g_mem.motor[MOTOR_IDX].activ         = 1;
    g_mem.motor[MOTOR_IDX].input_mode    = INPUT_MODE_EXTERNAL;
    g_mem.motor[MOTOR_IDX].control_mode  = CONTROL_ANGLE;

    // Get CS0 encoder line for RIGHT HAND and CS1 line for LEFT HAND as
default
    g_mem.motor[MOTOR_IDX].encoder_line  = g_mem.dev.right_left;
    g_mem.motor[MOTOR_IDX].pwm_rate_limiter = PWM_RATE_LIMITER_MAX;
    g_mem.motor[MOTOR_IDX].not_revers_motor_flag = TRUE;       // SoftHand
not reversible motor
    g_mem.motor[MOTOR_IDX].pos_lim_inf = 0;
    g_mem.motor[MOTOR_IDX].pos_lim_sup = (int32)16000 << g_mem.enc[g_mem.motor
[MOTOR_IDX].encoder_line].res[0];

    for (int i=0; i < N_ENCODER_LINE_MAX; i++) {
        // Initialize parameters for each encoder line (either for RIGHT and
for LEFT hand)
        g_mem.enc[i].double_encoder_on_off = TRUE;
        g_mem.enc[i].gears_params[0] = SH_N1;
        g_mem.enc[i].gears_params[1] = SH_N2;
        g_mem.enc[i].gears_params[2] = SH_I1;
    }

    g_mem.emg.emg_max_value[0] = 1024;
    g_mem.emg.emg_max_value[1] = 1024;
    g_mem.emg.emg_threshold[0] = 200;
    g_mem.emg.emg_threshold[1] = 200;
    g_mem.emg.emg_speed[0] = 100;
    g_mem.emg.emg_speed[1] = 100;
    g_mem.emg.emg_calibration_flag = 0;      // EMG calibration disabled by
default
```

```
    g_mem.emg.switch_emg = 0;

    //Initialize rest position parameters
    g_mem.SH.rest_position_flag = FALSE;
    g_mem.SH.rest_pos = (int32)7000 << g_mem.enc[g_mem.motor[MOTOR_IDX].↵
encoder_line].res[0]; // 56000
    g_mem.SH.rest_delay = 10;
    g_mem.SH.rest_vel = 10000;

    g_mem.imu.read_imu_flag = FALSE;
    g_mem.exp.read_exp_port_flag = EXP_NONE;       // 0 - None
    g_mem.exp.record_EMG_history_on_SD = FALSE;
    strcpy(g_mem.user[g_mem.dev.user_id].user_code_string, "USR001");
}

//↵
================================================================================
//                                                                MEMORY INIT ↵
MASTER
//↵
================================================================================
void memInit_Master(void)
{
    g_mem.dev.id                = 2;

    // MS STRUCT
    g_mem.MS.slave_ID = 1;
    g_mem.MS.slave_comm_active = FALSE;
}

//↵
================================================================================
//                                                    MEMORY INIT AIR CHAMBERS ↵
FB
//↵
================================================================================
void memInit_AirChambersFb(void)
{
    // Default configuration with Air Chambers Haptic feedback
    g_mem.dev.dev_type          = AIR_CHAMBERS_FB;

    g_mem.motor[0].control_mode  = CONTROL_PWM;

    // Drive slave with reference generated on second motor index
    // Default slave configuration for haptic feedback with SoftHand 2.0.↵
3
    g_mem.motor[1].input_mode = INPUT_MODE_EMG_FCFS;
    g_mem.motor[1].pos_lim_inf = 0;
    g_mem.motor[1].pos_lim_sup = (int32)22000 << g_mem.enc[g_mem.motor[1].↵
encoder_line].res[0];

    // FB STRUCT
```

```
    g_mem.FB.max_residual_current = 450;
    g_mem.FB.maximum_pressure_kPa = 25.0;
    g_mem.FB.prop_err_fb_gain = 1.0;
}

//
//==========================================================================
//                                  MEMORY INIT OTTOBOCK ACTIVE WRIST MASTER
FB
//
//==========================================================================
void memInit_OtbkActWristMs(void)
{
    // Default configuration with Ottobock Active Wrist feedback
    g_mem.dev.dev_type  = OTBK_ACT_WRIST_MS;
    g_mem.dev.right_left = LEFT_HAND;

    g_mem.motor[0].control_mode = CONTROL_PWM;
    g_mem.motor[0].pwm_rate_limiter = 100;
    g_mem.motor[0].not_revers_motor_flag = FALSE;

    // Drive slave with reference generated on second motor index
    // Default slave configuration for SoftHand 3.0
    g_mem.motor[1].input_mode = INPUT_MODE_EMG_FCFS;
    g_mem.motor[1].encoder_line  = g_mem.dev.right_left;
    g_mem.motor[1].pwm_rate_limiter = PWM_RATE_LIMITER_MAX;
    g_mem.motor[1].not_revers_motor_flag = FALSE;       // False, because it
is important only to configure motor parameters to compute position reference
    g_mem.motor[1].pos_lim_inf = 0;
    g_mem.motor[1].pos_lim_sup = (int32)16000 << g_mem.enc[g_mem.motor[1].
encoder_line].res[0];

    // WR STRUCT
    g_mem.WR.activation_mode = 0;                // Default Fast:wrist/syn2,
Slow:hand/syn1
    g_mem.WR.fast_act_threshold[0] = 800;
    g_mem.WR.fast_act_threshold[1] = 800;
    g_mem.WR.wrist_direction_association = 0;   // Default Close:CW, Open:CCW
}

//
//==========================================================================
//                                                  ROUTINE INTERRUPT
FUNCTION
//
//==========================================================================
/**
* Bunch of functions used on request from UART communication
**/

void cmd_get_measurements(){
```

```
    uint8 CYDATA index;
    int16 aux_int16;
    // Packet: header + measure(int16) + crc

    uint8 packet_data[8];

    //Header package
    packet_data[0] = CMD_GET_MEASUREMENTS;

    for (index = NUM_OF_SENSORS; index--;) {
        aux_int16 = (int16)(g_measOld[g_mem.motor[0].encoder_line].pos[index] ↵
>> g_mem.enc[g_mem.motor[0].encoder_line].res[index]);
        packet_data[(index << 1) + 2] = ((char*)(&aux_int16))[0];
        packet_data[(index << 1) + 1] = ((char*)(&aux_int16))[1];
    }

    if (g_mem.dev.use_2nd_motor_flag == TRUE){
        //Overwrite only second measure with first encoder on second motor ↵
line (just to have a measure also of second motor line on API)
        index = 1;
        aux_int16 = (int16)(g_measOld[g_mem.motor[1].encoder_line].pos[0] >> ↵
g_mem.enc[g_mem.motor[1].encoder_line].res[0]);
        packet_data[(index << 1) + 2] = ((char*)(&aux_int16))[0];
        packet_data[(index << 1) + 1] = ((char*)(&aux_int16))[1];
    }

    // Calculate Checksum and send message to UART

    packet_data[7] = LCRChecksum (packet_data, 7);

    commWrite(packet_data, 8);

}

void cmd_get_velocities(){

    uint8 CYDATA index;
    int16 aux_int16;

    // Packet: header + measure(int16) + crc

    uint8 packet_data[8];

    //Header package
    packet_data[0] = CMD_GET_VELOCITIES;

    for (index = NUM_OF_SENSORS; index--;) {
        aux_int16 = (int16)(g_measOld[g_mem.motor[0].encoder_line].vel[index] ↵
>> g_mem.enc[g_mem.motor[0].encoder_line].res[index]);
        packet_data[(index << 1) + 2] = ((char*)(&aux_int16))[0];
        packet_data[(index << 1) + 1] = ((char*)(&aux_int16))[1];
    }
```

```
    // Calculate Checksum and send message to UART

    packet_data[7] = LCRChecksum (packet_data, 7);

    commWrite(packet_data, 8);
}

void cmd_get_accelerations(){

    uint8 CYDATA index;
    int16 aux_int16;

    // Packet: header + measure(int16) + crc

    uint8 packet_data[8];

    //Header package
    packet_data[0] = CMD_GET_ACCEL;

    for (index = NUM_OF_SENSORS; index--;) {
        aux_int16 = (int16)(g_measOld[g_mem.motor[0].encoder_line].acc[index] ↵
>> g_mem.enc[g_mem.motor[0].encoder_line].res[index]);
        packet_data[(index << 1) + 2] = ((char*)(&aux_int16))[0];
        packet_data[(index << 1) + 1] = ((char*)(&aux_int16))[1];
    }

    // Calculate Checksum and send message to UART

    packet_data[7] = LCRChecksum (packet_data, 7);

    commWrite(packet_data, 8);
}

void cmd_get_joystick() {

    int16 aux_int16;

    // Packet: header + measure(int16) + crc

    uint8 packet_data[6];

    // Header
    packet_data[0] = CMD_GET_JOYSTICK;

    aux_int16 = (int16) g_adc_measOld.joystick[0];
    packet_data[2] = ((char*)(&aux_int16))[0];
    packet_data[1] = ((char*)(&aux_int16))[1];

    aux_int16 = (int16) g_adc_measOld.joystick[1];
    packet_data[4] = ((char*)(&aux_int16))[0];
    packet_data[3] = ((char*)(&aux_int16))[1];
```

```
    packet_data[5] = LCRChecksum (packet_data, 5);

    commWrite(packet_data, 6);
}

void cmd_set_inputs(){

    // Store position setted in right variables
    int16 aux_int16[NUM_OF_MOTORS];
    static int16 last_aux_int16[NUM_OF_MOTORS];

    aux_int16[0] = (int16)(g_rx.buffer[1]<<8 | g_rx.buffer[2]);
    aux_int16[1] = (int16)(g_rx.buffer[3]<<8 | g_rx.buffer[4]);

    // Check if last command received was the same as this
    //(Note: last command not last motor reference in g_ref)
    for (uint8 i = 0; i < (1 + c_mem.dev.use_2nd_motor_flag); i++) {
       if(last_aux_int16[i] != aux_int16[i]){
            change_ext_ref_flag = TRUE;
       }
        // Update last command
       last_aux_int16[i] = aux_int16[i];
    }

    // Update g_refNew in case a new command has been received
    if (change_ext_ref_flag) {
        for (uint8 i = 0; i< NUM_OF_MOTORS; i++) {
            if(g_mem.motor[i].control_mode == CONTROL_CURRENT) {
                g_refNew[i].curr = aux_int16[i];
            }
            else {
                if(g_mem.motor[i].control_mode == CONTROL_PWM) {
                    g_refNew[i].pwm = aux_int16[i];
                }
                else {
                    g_refNew[i].pos = aux_int16[i];   // motor ref
                    g_refNew[i].pos = g_refNew[i].pos << g_mem.enc[c_mem.motor
[i].encoder_line].res[0];
                }
            }

            // Check if the reference is nor higher or lower than the
position limits
            if (c_mem.motor[i].pos_lim_flag && (g_mem.motor[i].control_mode
== CURR_AND_POS_CONTROL || g_mem.motor[i].control_mode == CONTROL_ANGLE)) {

                if (g_refNew[i].pos < c_mem.motor[i].pos_lim_inf)
                    g_refNew[i].pos = c_mem.motor[i].pos_lim_inf;

                if (g_refNew[i].pos > c_mem.motor[i].pos_lim_sup)
                    g_refNew[i].pos = c_mem.motor[i].pos_lim_sup;
```

```
            }
        }
    }
}

void cmd_activate(){

    // Store new value reads
    uint8 aux = g_rx.buffer[1];

    // Check type of control mode enabled
    if ((g_mem.motor[0].control_mode == CONTROL_ANGLE) || (g_mem.motor[0].
control_mode == CURR_AND_POS_CONTROL)) {
        g_refNew[0].pos = g_meas[g_mem.motor[0].encoder_line].pos[0];
    }
    g_refNew[0].onoff = (aux & 0x01);

#ifdef AIR_CHAMBERS_FB_FW
    if (g_mem.dev.dev_type == AIR_CHAMBERS_FB){
        // Send PWM 0 to the PUMP in case a deactivation command arrives
        // [There is no driver activation, so g_refNew[i].onoff is useless]
        if (!(g_refNew[0].onoff)) {
            g_refNew[0].pwm = 0;
        }

        // Activate or deactivate the valve
        VALVE_Write((aux >> 1) & 0x01);
    }
#endif

    // Activate/Deactivate motor
    enable_motor(0, g_refNew[0].onoff);

    if (g_mem.dev.use_2nd_motor_flag == TRUE) {
        if ((g_mem.motor[1].control_mode == CONTROL_ANGLE) || (g_mem.motor[1].
control_mode == CURR_AND_POS_CONTROL)) {
            g_refNew[1].pos = g_meas[g_mem.motor[1].encoder_line].pos[0];
        }
        g_refNew[1].onoff = ((aux >> 1) & 0x01);
        enable_motor(1, g_refNew[1].onoff);
    }

}

void cmd_get_activate(){

    uint8 packet_data[3];

    // Header
    packet_data[0] = CMD_GET_ACTIVATE;

    // Fill payload
```

```
    if (g_mem.dev.use_2nd_motor_flag == TRUE) {
        packet_data[1] = ((g_ref[1].onoff << 1) | g_ref[0].onoff);
    }
    else {
        packet_data[1] = g_ref[0].onoff;
    }

    // Calculate checksum
    packet_data[2] = LCRChecksum(packet_data, 2);

    // Send package to UART
    commWrite(packet_data, 3);

}

void cmd_get_curr_and_meas(){

    uint8 CYDATA index;
    int16 aux_int16;

    //Packet: header + curr_meas(int16) + pos_meas(int16) + CRC

    uint8 packet_data[12];

    //Header package
    packet_data[0] = CMD_GET_CURR_AND_MEAS;

    // Currents
    aux_int16 = (int16) g_measOld[g_mem.motor[0].encoder_line].curr; //Real ↵
current motor1
    packet_data[2] = ((char*)(&aux_int16))[0];
    packet_data[1] = ((char*)(&aux_int16))[1];

    if (c_mem.dev.use_2nd_motor_flag == TRUE) {
        aux_int16 = (int16) g_measOld[g_mem.motor[1].encoder_line].curr; //↵
Real current motor 2
    }
    else {
        aux_int16 = (int16) g_measOld[g_mem.motor[0].encoder_line].estim_curr↵
; //Estimated current
    }
    packet_data[4] = ((char*)(&aux_int16))[0];
    packet_data[3] = ((char*)(&aux_int16))[1];

    // Positions
    for (index = NUM_OF_SENSORS; index--;) {
        aux_int16 = (int16)(g_measOld[g_mem.motor[0].encoder_line].pos[index] ↵
>> g_mem.enc[g_mem.motor[0].encoder_line].res[index]);
        packet_data[(index << 1) + 6] = ((char*)(&aux_int16))[0];
        packet_data[(index << 1) + 5] = ((char*)(&aux_int16))[1];
    }
    // Calculate Checksum and send message to UART
```

```
    packet_data[11] = LCRChecksum (packet_data, 11);
    commWrite(packet_data, 12);

}

void cmd_get_currents(){

    // Packet: header + motor_measure(int16) + crc

    uint8 packet_data[6];
    int16 aux_int16;

    //Header package

    packet_data[0] = CMD_GET_CURRENTS;

    if (c_mem.dev.dev_type != AIR_CHAMBERS_FB){
        // Currents
        aux_int16 = (int16) g_measOld[g_mem.motor[0].encoder_line].curr; //Real current
    }
    else {
        // Send pressure times 100 here instead of current (Simulink use)
        aux_int16 = (int16)(g_fb_meas.pressure*100.0); //Pressure
    }
    packet_data[2] = ((char*)(&aux_int16))[0];
    packet_data[1] = ((char*)(&aux_int16))[1];

    if (c_mem.dev.use_2nd_motor_flag == TRUE) {
        aux_int16 = (int16) g_measOld[g_mem.motor[1].encoder_line].curr; //Real current motor 2
    }
    else {
        aux_int16 = (int16) g_measOld[g_mem.motor[0].encoder_line].estim_curr; //Estimated current
    }
    packet_data[4] = ((char*)(&aux_int16))[0];
    packet_data[3] = ((char*)(&aux_int16))[1];

    // Calculate Checksum and send message to UART

    packet_data[5] = LCRChecksum (packet_data, 5);

    commWrite(packet_data, 6);
}

void cmd_get_currents_for_cuff(){

    // Packet: header + motor_measure(int16) + crc

    uint8 packet_data[4];
```

```
    int16 aux_int16;

    //Header package

    packet_data[0] = CMD_SET_CUFF_INPUTS;

    aux_int16 = (int16) g_measOld[g_mem.motor[0].encoder_line].estim_curr; //
Estimated Current
    packet_data[2] = ((char*)(&aux_int16))[0];
    packet_data[1] = ((char*)(&aux_int16))[1];

    // Calculate Checksum and send message to UART

    packet_data[3] = LCRChecksum (packet_data, 3);

    commWriteID(packet_data, 4, g_mem.dev.id -1);
}

//
=============================================================================
//                                    READ RESIDUAL CURRENT FUNCTION FROM
SOFTHAND
//
=============================================================================

int16 commReadResCurrFromSH()
{
    uint8 packet_data[16];
    uint8 packet_lenght;
    int16 curr_diff = 0;
    uint32 t_start, t_end;
    uint8 read_flag = TRUE;

    packet_lenght = 2;
    packet_data[0] = CMD_GET_CURR_DIFF;
    packet_data[1] = CMD_GET_CURR_DIFF;
    commWriteID(packet_data, packet_lenght, c_mem.MS.slave_ID);

    t_start = (uint32) MY_TIMER_ReadCounter();
    while(g_rx.buffer[0] != CMD_SET_CUFF_INPUTS) {
        if (interrupt_flag){
            interrupt_flag = FALSE;
            interrupt_manager();
        }

        t_end = (uint32) MY_TIMER_ReadCounter();
        if((t_start - t_end) > 4500000){            // 4.5 s timeout
            read_flag = FALSE;
            master_mode = 0;                // Exit from master mode
            break;
        }
    }
```

```
    if (read_flag) {
        curr_diff = (int16)(g_rx.buffer[1]<<8 | g_rx.buffer[2])
;
    }

    return curr_diff;
}

void cmd_set_baudrate(){

    // Set BaudRate
    c_mem.dev.baud_rate = g_rx.buffer[1];

    switch(g_rx.buffer[1]){
        case 13:
            CLOCK_UART_SetDividerValue(13);
            break;
        default:
            CLOCK_UART_SetDividerValue(3);
    }
}

void cmd_ping(){

    uint8 packet_data[2];

    // Header
    packet_data[0] = CMD_PING;

    // Load Payload
    packet_data[1] = CMD_PING;

    // Send Package to uart
    commWrite(packet_data, 2);
}

void cmd_get_inputs(){

    // Packet: header + motor_measure(int16) + crc

    uint8 packet_data[6];
    int16 aux_int16;

    //Header package

    packet_data[0] = CMD_GET_INPUTS;

    aux_int16 = (int16)(g_refOld[0].pos >> g_mem.enc[g_mem.motor[0].
encoder_line].res[0]);
    packet_data[2] = ((char*)(&aux_int16))[0];
    packet_data[1] = ((char*)(&aux_int16))[1];
```

```
    aux_int16 = (int16)(g_refOld[1].pos >> g_mem.enc[g_mem.motor[1].encoder_line].res[0]);
    packet_data[4] = ((char*)(&aux_int16))[0];
    packet_data[3] = ((char*)(&aux_int16))[1];

    // Calculate Checksum and send message to UART

    packet_data[5] = LCRChecksum(packet_data, 5);

    commWrite(packet_data, 6);
}

void cmd_store_params(){

    // Check input mode enabled
    uint32 off_1;
    float mult_1;
    uint8 CYDATA packet_lenght = 2;
    uint8 CYDATA packet_data[2];
    uint8 CYDATA old_id;

    if( c_mem.motor[0].input_mode == INPUT_MODE_EXTERNAL ) {
        off_1 = c_mem.enc[c_mem.motor[0].encoder_line].m_off[0];
        mult_1 = c_mem.enc[c_mem.motor[0].encoder_line].m_mult[0];

        g_refNew[0].pos = (int32)((float)g_refNew[0].pos / mult_1);

        g_refNew[0].pos = (int32)((float)g_refNew[0].pos * g_mem.enc[c_mem.motor[0].encoder_line].m_mult[0]);

        g_refNew[0].pos += (g_mem.enc[c_mem.motor[0].encoder_line].m_off[0] - off_1);

        // Check position Limits

        if (c_mem.motor[0].pos_lim_flag) {                    // position limiting
            if (g_refNew[0].pos < c_mem.motor[0].pos_lim_inf)
                g_refNew[0].pos = c_mem.motor[0].pos_lim_inf;

            if (g_refNew[0].pos > c_mem.motor[0].pos_lim_sup)
                g_refNew[0].pos = c_mem.motor[0].pos_lim_sup;
        }
    }

    // If SD is used, create new param and data file
    if (c_mem.exp.read_exp_port_flag == EXP_SD_RTC) {
        FS_FClose(pFile);

        InitSD_FS();
    }
```

```
    // Store params

    if (c_mem.dev.id != g_mem.dev.id) {     //If a new id is going to be set ↵
we will lose communication
        old_id = c_mem.dev.id;          //after the memstore(0) and the ACK ↵
won't be recognised
        if(memStore(0)) {
            packet_data[0] = ACK_OK;
            packet_data[1] = ACK_OK;
            commWrite_old_id(packet_data, packet_lenght, old_id);
        }
        else{
            packet_data[0] = ACK_ERROR;
            packet_data[1] = ACK_ERROR;
            commWrite_old_id(packet_data, packet_lenght, old_id);
        }
    }
    else {
        if(memStore(0))
            sendAcknowledgment(ACK_OK);
        else
            sendAcknowledgment(ACK_ERROR);
    }

    // FW reset (if necessary)
    if (reset_PSoC_flag == TRUE) {
        CySoftwareReset();
    }
}

void cmd_get_emg(){

    uint8 packet_data[6];
    int16 aux_int16;

    // Header
    packet_data[0] = CMD_GET_EMG;

    aux_int16 = (int16) g_adc_measOld.emg[0];
    packet_data[2] = ((char*)(&aux_int16))[0];
    packet_data[1] = ((char*)(&aux_int16))[1];

    aux_int16 = (int16) g_adc_measOld.emg[1];
    packet_data[4] = ((char*)(&aux_int16))[0];
    packet_data[3] = ((char*)(&aux_int16))[1];

    packet_data[5] = LCRChecksum (packet_data, 5);

    commWrite(packet_data, 6);

}
```

```
void cmd_get_imu_readings(){
    //Retrieve accelerometers, gyroscopes, magnetometers, quaternions and temperatures readings

    uint8 k_imu;
    uint16 c = 1;
    uint8 k = 0;
    uint16 gl_c = 1;
    int16 aux_int16;
    float aux_float;

    // Packet: header + imu id(uint8) + imu flags(uint8) + crc
    uint8 packet_data[350];
    uint8 single_packet[32];

    //Header package
    packet_data[0] = CMD_GET_IMU_READINGS;

    for (k_imu = 0; k_imu < N_IMU_Connected; k_imu++)
    {

        single_packet[0] = (uint8) 0x3A; //':';
        if (c_mem.imu.IMU_conf[IMU_connected[k_imu]][0]){
            aux_int16 = (int16) g_imu[k_imu].accel_value[0];
            single_packet[c + 1] = ((char*)(&aux_int16))[0];
            single_packet[c] = ((char*)(&aux_int16))[1];

            aux_int16 = (int16) g_imu[k_imu].accel_value[1];
            single_packet[c + 3] = ((char*)(&aux_int16))[0];
            single_packet[c + 2] = ((char*)(&aux_int16))[1];

            aux_int16 = (int16) g_imu[k_imu].accel_value[2];
            single_packet[c + 5] = ((char*)(&aux_int16))[0];
            single_packet[c + 4] = ((char*)(&aux_int16))[1];

            c = c + 6;
        }
        if (c_mem.imu.IMU_conf[IMU_connected[k_imu]][1]){
            aux_int16 = (int16) g_imu[k_imu].gyro_value[0];
            single_packet[c + 1] = ((char*)(&aux_int16))[0];
            single_packet[c] = ((char*)(&aux_int16))[1];

            aux_int16 = (int16) g_imu[k_imu].gyro_value[1];
            single_packet[c + 3] = ((char*)(&aux_int16))[0];
            single_packet[c + 2] = ((char*)(&aux_int16))[1];

            aux_int16 = (int16) g_imu[k_imu].gyro_value[2];
            single_packet[c + 5] = ((char*)(&aux_int16))[0];
            single_packet[c + 4] = ((char*)(&aux_int16))[1];

            c = c + 6;
        }
```

```
        if (c_mem.imu.IMU_conf[IMU_connected[k_imu]][2]){
            aux_int16 = (int16) g_imu[k_imu].mag_value[0];
            single_packet[c + 1] = ((char*)(&aux_int16))[0];
            single_packet[c] = ((char*)(&aux_int16))[1];

            aux_int16 = (int16) g_imu[k_imu].mag_value[1];
            single_packet[c + 3] = ((char*)(&aux_int16))[0];
            single_packet[c + 2] = ((char*)(&aux_int16))[1];

            aux_int16 = (int16) g_imu[k_imu].mag_value[2];
            single_packet[c + 5] = ((char*)(&aux_int16))[0];
            single_packet[c + 4] = ((char*)(&aux_int16))[1];

            c = c + 6;
        }
        if (c_mem.imu.IMU_conf[IMU_connected[k_imu]][3]){
            aux_float = (float) g_imu[k_imu].quat_value[0];
            for(k = 0; k < 4; k++) {
                single_packet[c + 4 - k -1] = ((char*)(&aux_float))[k];
            }

            aux_float = (float) g_imu[k_imu].quat_value[1];
            for(k = 0; k < 4; k++) {
                single_packet[c + 8 - k -1] = ((char*)(&aux_float))[k];
            }

            aux_float = (float) g_imu[k_imu].quat_value[2];
            for(k = 0; k < 4; k++) {
                single_packet[c + 12 - k -1] = ((char*)(&aux_float))[k];
            }

            aux_float = (float) g_imu[k_imu].quat_value[3];
            for(k = 0; k < 4; k++) {
                single_packet[c + 16 - k -1] = ((char*)(&aux_float))[k];
            }
            c = c + 16;
        }
        if (c_mem.imu.IMU_conf[IMU_connected[k_imu]][4]){
            aux_int16 = (int16) g_imu[k_imu].temp_value;
            single_packet[c + 1] = ((char*)(&aux_int16))[0];
            single_packet[c] = ((char*)(&aux_int16))[1];
            c = c + 2;
        }
        single_packet[single_imu_size[IMU_connected[k_imu]] - 1] = (uint8) ↵
0x3A; //':';
        c = 1;

        for(k=0; k < single_imu_size[IMU_connected[k_imu]]; k++) {
            packet_data[gl_c + k] = (uint8) single_packet[k];
        }
        gl_c = gl_c + single_imu_size[IMU_connected[k_imu]];
```

```
        memset(&single_packet, 0, sizeof(single_packet));

    }

    // Calculate Checksum and send message to UART
    packet_data[imus_data_size-1] = LCRChecksum (packet_data, imus_data_size-1
);
    commWrite(packet_data, imus_data_size);
}

void cmd_get_encoder_map(){
    //Retrieve Encoder map

    uint8 packet_data[4+N_ENCODER_LINE_MAX*N_ENCODERS_PER_LINE_MAX];
    uint8 CYDATA i, j;

    // Header
    packet_data[0] = CMD_GET_ENCODER_CONF;

    // Fill payload
    packet_data[1] = N_ENCODER_LINE_MAX;
    packet_data[2] = N_ENCODERS_PER_LINE_MAX;
    for (i=0; i<N_ENCODER_LINE_MAX; i++) {
        for (j=0; j < N_ENCODERS_PER_LINE_MAX; j++) {
            packet_data[3 + i*N_ENCODERS_PER_LINE_MAX + j] = c_mem.enc[i].
Enc_raw_read_conf[j];
        }
    }

    // Calculate checksum
    packet_data[3+N_ENCODER_LINE_MAX*N_ENCODERS_PER_LINE_MAX] = LCRChecksum(
packet_data, 3+N_ENCODER_LINE_MAX*N_ENCODERS_PER_LINE_MAX);

    // Send package to UART
    commWrite(packet_data, 4+N_ENCODER_LINE_MAX*N_ENCODERS_PER_LINE_MAX);
}

void cmd_get_encoder_raw(){
    //Retrieve all Encoders raw values

    uint8 packet_data[2+2*N_ENCODER_LINE_MAX*N_ENCODERS_PER_LINE_MAX];
    uint8 i, j, idx;
    uint16 aux_uint16;

    //Header package
    packet_data[0] = CMD_GET_ENCODER_RAW;

    // Fill payload
    idx = 0;
    for (i=0; i<N_ENCODER_LINE_MAX; i++) {
        for (j=0; j < N_Encoder_Line_Connected[i]; j++) {
            if (c_mem.enc[i].Enc_raw_read_conf[j] == 1) {
```

```
                aux_uint16 = (uint16)Encoder_Value[i][j];
                packet_data[(idx << 1) + 2] = ((char*)(&aux_uint16))[0];
                packet_data[(idx << 1) + 1] = ((char*)(&aux_uint16))[1];
                idx++;
            }
        }
    }

    // Calculate checksum
    packet_data[1+2*idx] = LCRChecksum(packet_data, 1+2*idx);

    // Send package to UART
    commWrite(packet_data, 2+2*idx);
}

void cmd_get_ADC_map(){
    //Retrieve Encoder map

    uint8 packet_data[3+NUM_OF_ADC_CHANNELS_MAX];
    uint8 CYDATA i;

    // Header
    packet_data[0] = CMD_GET_ADC_CONF;

    // Fill payload
    packet_data[1] = NUM_OF_ADC_CHANNELS_MAX;
    for (i=0; i<NUM_OF_ADC_CHANNELS_MAX; i++) {
        packet_data[2 + i] = c_mem.exp.ADC_conf[i];
    }

    // Calculate checksum
    packet_data[2+NUM_OF_ADC_CHANNELS_MAX] = LCRChecksum(packet_data, 2↵
+NUM_OF_ADC_CHANNELS_MAX);

    // Send package to UART
    commWrite(packet_data, 3+NUM_OF_ADC_CHANNELS_MAX);

}

void cmd_get_ADC_raw(){
    //Retrieve Additional EMG port raw values

    uint8 packet_data[2+2*NUM_OF_ADC_CHANNELS_MAX];
    uint8 CYDATA i, idx = 0;
    int16 aux_int16;

    // Header
    packet_data[0] = CMD_GET_ADC_RAW;

    // Fill payload
    for (i = 0; i < NUM_OF_ANALOG_INPUTS; i++) {
        if (c_mem.exp.ADC_conf[i] == 1) {
```

```
            aux_int16 = (int16) ADC_buf[i];
            packet_data[(idx << 1) + 2] = ((char*)(&aux_int16))[0];
            packet_data[(idx << 1) + 1] = ((char*)(&aux_int16))[1];
            idx++;
        }
    }

    // Calculate checksum
    packet_data[1+2*idx] = LCRChecksum(packet_data, 1+2*idx);

    // Send package to UART
    commWrite(packet_data, 2+2*idx);
}

void cmd_get_SD_file( uint16 filename_length ){

    uint8 i = 0;
    char CYDATA filename[50] = "";
    char CYDATA str_sd_data[20000] = "";
    strcpy(filename, "");
    strcpy(str_sd_data, "");

    for (i=0; i<filename_length; i++){
        *((uint8*)filename + i) = (char)g_rx.buffer[3+i];
    }
    *((uint8*)filename + i) = '\0';

    // Check if the file is the one currently opened or not
    if (strcmp(filename, sdFile)){
        Read_SD_Closed_File(filename, str_sd_data, sizeof(str_sd_data));
    }
    else {
        //It is the currently open working file
        Read_SD_Current_Data(str_sd_data, sizeof(str_sd_data));
    }

    //itoa(filename_length, filename, 10);
    // Send the file to API that receives packet as a ping string
    UART_RS485_ClearTxBuffer();
    UART_RS485_PutString(str_sd_data);
}

void cmd_remove_SD_file( uint16 filename_length ){

    uint8 i = 0;
    char CYDATA filename[50] = "";
    strcpy(filename, "");

    for (i=0; i<filename_length; i++){
        *((uint8*)filename + i) = (char)g_rx.buffer[3+i];
    }
    *((uint8*)filename + i) = '\0';
```

```
    // Check if the file is the one currently opened or not
    uint8 res = Remove_SD_File(filename);


    uint8 packet_data[3];

    //Header package

    packet_data[0] = CMD_REMOVE_SD_SINGLE_FILE;
    packet_data[1] = res;

    // Calculate Checksum and send message to UART
    packet_data[2] = LCRChecksum (packet_data, 2);

    commWrite(packet_data, 3);
}

//
//============================================================================
//                                                          AIR CHAMBERS
CONTROL
//
//============================================================================
/* It asks current difference to the SoftHand and sets force feedback device
inputs proportionally to this difference.*/

void air_chambers_control(int slave_motor_idx) {

#ifdef AIR_CHAMBERS_FB_FW

    int16 curr_diff;
    int32 pressure_reference, err_pressure, pressure_value;
    int32 valve_command;
    int16 x_value;

    // Use pressure and residual current read from the SoftHand

    curr_diff = (int16)commReadResCurrFromSH();

    // Current difference saturation old mapping
//    if(curr_diff > c_mem.FB.max_residual_current) {
//        curr_diff = c_mem.FB.max_residual_current;
//    }
//    if(curr_diff < 0) {
//        curr_diff = 0;
//    }

    // Compute pressure reference

    x_value = curr_diff - 50.0;
    if (x_value < 0)
```

```
        x_value = 0;

    // old mapping --- linear mapping
    //pressure_reference = (int32)(curr_diff * (c_mem.FB.maximum_pressure_kPa/c_mem.FB.max_residual_current)); // normalization by maximum values (200 mmHg->26.6 kPa)
    pressure_reference = (int32)((int32)(-30.0*x_value*x_value + 55.0*c_mem.FB.max_residual_current*x_value)/(c_mem.FB.max_residual_current*c_mem.FB.max_residual_current));
    if (pressure_reference < 0)
        pressure_reference = 0;
    if (pressure_reference > c_mem.FB.maximum_pressure_kPa)
        pressure_reference = c_mem.FB.maximum_pressure_kPa;

    pressure_value = (int32)g_fb_meas.pressure;
    err_pressure = pressure_reference - pressure_value;       // error in kPa
//    if (err_pressure < 0){
//        err_pressure = 0;
//    }

    if (x_value <= 0){
        //i.e the hand is opening
        valve_command = 0;  //valve open: air passes
    }
    else {
        //i.e the hand is closing, so valve should stay closed independently from the pressure error
        //if err_pressure greater than 0, it means pressure should increase, so valve should stay closed
        //if err_pressure==0, it means you reached the right pressure, so valve should stay closed
        valve_command = 1;  //3.6V (5V - 2 diodes) - valve close: air doesn't pass
    }

    // Pump control
    g_refNew[0].pwm = (int32)(c_mem.FB.prop_err_fb_gain*err_pressure);
    //c_mem.FB.prop_err_fb_gain default 1.0 gain since, max err_pressure is 25 and pwm range is approx. 25 ticks [45=2V,70=3V]

    // Limit output voltage
    if (g_refNew[0].pwm > 80) // 80 (3.5V) 80% of 4.3V (5V - 1 diode)
        g_refNew[0].pwm = 80; // 80
    if (g_refNew[0].pwm < 20)
        g_refNew[0].pwm = 0;

    VALVE_Write(valve_command);


    // Drive slave with reference generated on second motor index
    // Use second motor structures and parameters, only to generate position reference not for PID control
```

```
    // IMPORTANT: configure second motor parameters with proper slave parameters
    motor_control_generic(slave_motor_idx);
#endif
}


//============================================================================
//                                                    EMG ACTIVATION VELOCITY FSM
//============================================================================
/* It decides which is the current emg activation velocity.*/

uint8 emg_activation_velocity_fsm() {

    static uint8 fsm_state = RELAX_STATE;     // Wrist FSM state
    static int32 cnt = 0;
    int32 CYDATA err_emg_1, err_emg_2;
    int32 CYDATA f_err_emg_1, f_err_emg_2;

    err_emg_1 = g_adc_meas.emg[0] - c_mem.emg.emg_threshold[0];
    err_emg_2 = g_adc_meas.emg[1] - c_mem.emg.emg_threshold[1];
    f_err_emg_1 = g_adc_meas.emg[0] - c_mem.WR.fast_act_threshold[0];
    f_err_emg_2 = g_adc_meas.emg[1] - c_mem.WR.fast_act_threshold[1];


    // State machine - Evaluate emg activation status
    // Note: in this way, diff_emg_1 and diff_emg_2 are for sure differences between two consecutive activated values
    switch (fsm_state){
        case RELAX_STATE:
            if (err_emg_1 > 0 || err_emg_2 > 0){
                fsm_state = TIMER_STATE;
            }

            break;
        case TIMER_STATE:     // Timer
            if (err_emg_1 > 0 || err_emg_2 > 0){
                cnt = cnt +1;
            }
            else {
                fsm_state = RELAX_STATE;
            }

            if (cnt > 100){
                if ((err_emg_1 > 0 && f_err_emg_1 > 0) || (err_emg_2 > 0 && f_err_emg_2 > 0)){
                    // Fast activation
                    if (c_mem.WR.activation_mode == 0){
                        fsm_state = MOVE_FAST_ACT;
```

```
                }
                else{
                    fsm_state = MOVE_SLOW_ACT;
                }
            }
            if ((err_emg_1 > 0 && f_err_emg_1 < 0) || (err_emg_2 > 0 && ↵
f_err_emg_2 < 0)){
                // Slow activation
                if (c_mem.WR.activation_mode == 0){
                    fsm_state = MOVE_SLOW_ACT;
                }
                else{
                    fsm_state = MOVE_FAST_ACT;
                }
            }
            if (err_emg_1 < 0 && err_emg_2 < 0){
                // Involuntary activation
                fsm_state = RELAX_STATE;
            }
            cnt = 0;
        }

        break;
    case MOVE_FAST_ACT:

        if (err_emg_1 < 0 && err_emg_2 < 0){
            fsm_state = RELAX_STATE;
        }

        break;
    case MOVE_SLOW_ACT:

        if (err_emg_1 < 0 && err_emg_2 < 0){
            fsm_state = RELAX_STATE;
        }

        break;
}

    return fsm_state;

}

//↵
==========================================================================
//                                          OTTOBOCK ACTIVE WRIST MASTER ↵
CONTROL
//↵
==========================================================================
/* It moves Ottobock active wrist (as master) and connected SoftHand slave ↵
according to emg activation velocity.*/
```

```
void otbk_act_wrist_control(int slave_motor_idx) {

#ifdef OTBK_ACT_WRIST_MS_FW

    uint8 fsm_state = emg_activation_velocity_fsm();

    // State machine - Evaluate emg activation status
    // Note: in this way, diff_emg_1 and diff_emg_2 are for sure differences between two consecutive activated values
    switch (fsm_state){
        case RELAX_STATE:

            // Wrist stop
            g_refNew[0].pwm = 0;

            // Softhand stop
            // Do not update the motor reference, so the SoftHand stays still
            g_ref[slave_motor_idx].pos = g_refOld[slave_motor_idx].pos;
            g_refNew[slave_motor_idx].pos = g_ref[slave_motor_idx].pos;

            break;
        case TIMER_STATE:     // Timer

            break;
        case MOVE_FAST_ACT: // Wrist movement

            // Wrist movement
            if (g_adc_meas.emg[0] > g_adc_meas.emg[1]){
                if (c_mem.WR.wrist_direction_association == 0){
                    g_refNew[0].pwm = 60;       //Rotate CW
                }
                else {
                    g_refNew[0].pwm = -60;   // Rotate CCW
                }
            }
            else {
                if (c_mem.WR.wrist_direction_association == 0){
                    g_refNew[0].pwm = -60;   // Rotate CCW
                }
                else {
                    g_refNew[0].pwm = 60;       //Rotate CW
                }
            }

            // Softhand stop
            // Do not update the motor reference, so the SoftHand stays still
            g_ref[slave_motor_idx].pos = g_refOld[slave_motor_idx].pos;
            g_refNew[slave_motor_idx].pos = g_ref[slave_motor_idx].pos;

            break;
        case MOVE_SLOW_ACT: // Hand movement
```

```
            // Wrist stop
            g_refNew[0].pwm = 0;

            // SoftHand movement
            // Drive slave with reference generated on second motor index
            // Use second motor structures and parameters, only to generate position reference not for PID control
            // IMPORTANT: configure second motor parameters with proper slave parameters
            motor_control_generic(slave_motor_idx);

            break;
    }

    // Limit output voltage
    if (g_refNew[0].pwm > 67) // 67 (8.4V max of 2S ottobock battery) 66.6% of 12.6V
        g_refNew[0].pwm = 67; // 67
    if (g_refNew[0].pwm < -67)
        g_refNew[0].pwm = -67;

#endif
}

//
//////////////////////////////////////////////////////////////////////////////
//                                                                      DRIVE SLAVE
//
//////////////////////////////////////////////////////////////////////////////

void drive_slave(uint8 motor_idx, uint8 slave_ID) {

#ifdef MASTER_FW
    uint8 packet_data[6];
    uint8 packet_lenght;
    int16 aux_int16;

    // If not the use of handle or an emg input mode is set, exit from master_mode
    if( c_mem.motor[motor_idx].input_mode != INPUT_MODE_ENCODER3          &&
        c_mem.motor[motor_idx].input_mode != INPUT_MODE_EMG_PROPORTIONAL  &&
        c_mem.motor[motor_idx].input_mode != INPUT_MODE_EMG_INTEGRAL      &&
        c_mem.motor[motor_idx].input_mode != INPUT_MODE_EMG_FCFS          &&
        c_mem.motor[motor_idx].input_mode != INPUT_MODE_EMG_FCFS_ADV          &&
        c_mem.motor[motor_idx].input_mode != INPUT_MODE_EMG_PROPORTIONAL_NC     ){
        master_mode = 0;
        return;
    }
```

```
    if (dev_tension[0] >= 5000 && dev_tension[0] < 7000){
        master_mode = 0;
        return;
    }


    //Sends a Set inputs command to a second board
    packet_data[0] = CMD_SET_INPUTS;
    aux_int16 = (int16) (g_ref[motor_idx].pos >> g_mem.enc[g_mem.motor[
motor_idx].encoder_line].res[0]);
    packet_data[2] = ((char*)(&aux_int16))[0];
    packet_data[1] = ((char*)(&aux_int16))[1];
    *((int16 *) &packet_data[3]) = 0;

    packet_lenght = 6;
    packet_data[packet_lenght - 1] = LCRChecksum(packet_data,packet_lenght - 1
);
    commWriteID(packet_data, packet_lenght, slave_ID);

#endif
}

//
//============================================================================
//                                                          STOP MASTER
DEVICE
//
//============================================================================

void stop_master_device() {

#ifdef AIR_CHAMBERS_FB_FW
    if (c_mem.dev.dev_type == AIR_CHAMBERS_FB){
        // Stop pump and open valve
        g_refNew[0].pwm = 0;
        VALVE_Write(0);
    }
#endif

#ifdef OTBK_ACT_WRIST_MS_FW
    if (c_mem.dev.dev_type == OTBK_ACT_WRIST_MS){
        //Stop wrist motor
        g_refNew[0].pwm = 0;
    }
#endif
}

//
//============================================================================
//                                                          DEACTIVATE
SLAVES
//
```

```
================================================================================

void deactivate_slaves() {

#ifdef MASTER_FW

    uint8 packet_data[10];
    uint8 packet_lenght;

    // If not a emg input mode is set, exit from master_mode
    if(c_mem.motor[0].input_mode != INPUT_MODE_EMG_PROPORTIONAL  &&
        c_mem.motor[0].input_mode != INPUT_MODE_EMG_INTEGRAL     &&
        c_mem.motor[0].input_mode != INPUT_MODE_EMG_FCFS          &&
        c_mem.motor[0].input_mode != INPUT_MODE_EMG_FCFS_ADV      &&
        c_mem.motor[0].input_mode != INPUT_MODE_EMG_PROPORTIONAL_NC     ){
        master_mode = 0;
        return;
    }

    //Sends a Set inputs command to a second board
    packet_data[0] = CMD_ACTIVATE;

    *((int16 *) &packet_data[1]) = 0;   //3 to activate, 0 to deactivate
    packet_lenght = 3;
    packet_data[packet_lenght - 1] = LCRChecksum(packet_data,packet_lenght - 1
);

    commWrite(packet_data, packet_lenght);

#endif
}

/* [] END OF FILE */
```