The iCub CAN protocol for the sensor boards

This document describes the iCub CAN protocol with focus on the sensor boards.

Approval History

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Version | Author | | Date | Approved | | Date |
| 1.0 | M. Accame | IIT – iCub Tech |  |  |  |  |
| 1.12 | V.Gaggero | IIT – iCub Tech |  |  |  |  |
|  |  |  |  |  |  |  |

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Comments |
| 0.5 | 14 Nov 18 | M. Accame | First WIP version. |
| 0.9 | 10 Dec 18 | M. Accame | Covered all sensor boards. Added most common use cases. |
| 1.0 | 25 Mar 19 | M. Accame | Added new boards (rfe, sg3, psc, mtb4w) and new POS messages |
| 1.1 | 11 Apr 19 | M. Accame | Clarified format of POS messages |
| 1.2 | 16 May 19 | M. Accame | Added picture of the skin triangle |
| 1.3 | 30 Jul 19 | M. Accame | Added a new mode in message config board skin |
| 1.4 | 2 Sept 19 | M. Accame | Corrected byte content of pictures of section: Typical Scenarios / The FT service. They are now coherent w/ the protocol. |
| 1.5 | 8 Nov 19 | M. Accame | Added a new section which helps the casual reader called **Where to find what you are looking for**. Changed author’s organisation to iCub Tech. |
| 1.6 | 21 Feb 2020 | M. Accame | Corrected format of reply message in GET\_ADC message. |
| 1.7 | 6 Aug 2020 | M. Accame | Added multiple reply to GET\_ADC message for the case of strain2 boards with version >= 2.0.8 |
| 1.8 | 30 May 2022 | M. Accame | Added CLS 110b for the streaming of battery data w/ TYP BAT\_PACKINFO. |
| 1.9 | 16 Sep 2022 | M. Accame | Added field ID in message POS\_CONFIG |
| 1.10 | 22 Feb 2023 | M. Accame | Added more configurability in IMU: in mode and orientation |
| 1.11 | 17 Sep 2024 | M. Accame | Clarified some management message + updated board support tables |
| 1.12 | 25 Nov 2024 | J.Losi V.Gaggero | Update BMS and BAT message protocol |

Table of Contents

[1 Introduction 1](#_Toc183426933)

[1.1 Where to find what you are looking for 1](#_Toc183426934)

[1.1.1 General details of the CAN protocol 1](#_Toc183426935)

[1.1.2 The reference binary format of the CAN messages 1](#_Toc183426936)

[1.1.3 The messages supported by which boards 1](#_Toc183426937)

[1.1.4 The services offered by the CAN protocol 1](#_Toc183426938)

[1.1.5 Typical use cases 1](#_Toc183426939)

[2 The iCub CAN protocol 2](#_Toc183426940)

[2.1 The structure of the CAN frame 2](#_Toc183426941)

[2.1.1 The ID 3](#_Toc183426942)

[2.1.2 The PAYLOAD 3](#_Toc183426943)

[2.2 The iCub network 4](#_Toc183426944)

[2.2.1 The iCub host 4](#_Toc183426945)

[2.2.1.1 Examples of possible hosts 4](#_Toc183426946)

[2.2.2 The iCub CAN nodes 6](#_Toc183426947)

[2.2.2.1 Valid addresses 6](#_Toc183426948)

[2.2.2.2 Processes executed by a CAN node: bootloader and application 6](#_Toc183426949)

[2.2.2.3 The board types 7](#_Toc183426950)

[2.2.2.4 Icons used to represent the nodes 8](#_Toc183426951)

[2.3 The supported communication modes 10](#_Toc183426952)

[2.3.1 Transport modes 10](#_Toc183426953)

[2.3.1.1 Unicast transport 10](#_Toc183426954)

[2.3.1.2 Broadcast transport 10](#_Toc183426955)

[2.3.1.3 Streaming transport 11](#_Toc183426956)

[2.3.2 Operations 11](#_Toc183426957)

[2.3.2.1 The SET<> operation is used to impose a value 11](#_Toc183426958)

[2.3.2.2 The GET<> operation is used to retrieve a value with REPLY<> 12](#_Toc183426959)

[2.3.2.3 The ORDER<> operation is used to impose an action or a behaviour 12](#_Toc183426960)

[2.3.2.4 The STREAM<> operation is used to transport a value 12](#_Toc183426961)

[2.3.3 Messages accepted by a CAN node 13](#_Toc183426962)

[2.4 The iCub CAN protocol classes 14](#_Toc183426963)

[2.4.1 The command classes 14](#_Toc183426964)

[2.4.2 The streaming classes 14](#_Toc183426965)

[2.5 The services offered by the CAN protocol 16](#_Toc183426966)

[2.5.1 The management service 16](#_Toc183426967)

[2.5.2 The MC service 16](#_Toc183426968)

[2.5.3 The FT service 16](#_Toc183426969)

[2.5.4 The HES service 16](#_Toc183426970)

[2.5.5 The Skin service 17](#_Toc183426971)

[2.5.6 The basic inertial service 17](#_Toc183426972)

[2.5.7 The IMU service 17](#_Toc183426973)

[2.5.8 The THERMO service 17](#_Toc183426974)

[2.5.9 The POS service 17](#_Toc183426975)

[2.5.10 The BAT service 17](#_Toc183426976)

[3 The messages in the command classes 19](#_Toc183426977)

[3.1 Class management / bootloader 19](#_Toc183426978)

[3.1.1 Messages used for general management of boards 20](#_Toc183426979)

[3.1.2 Messages used in the firmware update procedure 25](#_Toc183426980)

[3.2 Class polling motor control 30](#_Toc183426981)

[3.2.1 Messages used for general management of boards 30](#_Toc183426982)

[3.2.2 All the other messages 31](#_Toc183426983)

[3.3 Class polling analog sensors 32](#_Toc183426984)

[3.3.1 Messages used for general management of boards 33](#_Toc183426985)

[3.3.2 Messages for managing the FT and HES data service 36](#_Toc183426986)

[3.3.3 Messages for managing the IMU service 41](#_Toc183426987)

[3.3.4 Messages for managing the THERMO service 45](#_Toc183426988)

[3.3.5 Messages for managing the Skin service 48](#_Toc183426989)

[3.3.6 Messages for managing the accel / gyro in mtb boards 50](#_Toc183426990)

[3.3.7 Messages for advanced configuration of the FT service 51](#_Toc183426991)

[3.3.7.1 Backwards compatibility of messages that contain a regulation set 51](#_Toc183426992)

[3.3.7.2 Messages that select/query a regulation set 52](#_Toc183426993)

[3.3.7.3 Messages that operate on the content of a given regulation set 54](#_Toc183426994)

[3.3.7.4 Messages for a generic amplifier 65](#_Toc183426995)

[3.3.7.5 Messages specific for the PGA308 amplifiers in strain2 boards 71](#_Toc183426996)

[3.3.7.6 Other messages 74](#_Toc183426997)

[3.3.8 Messages for managing the POS service 75](#_Toc183426998)

[4 The messages of the streaming classes 80](#_Toc183426999)

[4.1 Class periodic motor control 80](#_Toc183427000)

[4.1.1 Messages supported by all boards 81](#_Toc183427001)

[4.1.2 All the other messages 81](#_Toc183427002)

[4.2 Class periodic analog sensors 82](#_Toc183427003)

[4.2.1 Messages which stream user-defined data 83](#_Toc183427004)

[4.2.2 Messages which stream POS data 84](#_Toc183427005)

[4.2.3 Messages which stream FT data 85](#_Toc183427006)

[4.2.4 Messages which stream HES data 89](#_Toc183427007)

[4.2.5 Messages which stream THERMO data 90](#_Toc183427008)

[4.3 Class periodic skin data 92](#_Toc183427009)

[4.3.1 Messages which stream SKIN data 93](#_Toc183427010)

[4.3.2 Messages which stream POS data 94](#_Toc183427011)

[4.4 Class periodic inertial data 95](#_Toc183427012)

[4.4.1 Messages which stream IMU data 96](#_Toc183427013)

[4.4.2 Messages which stream accel and gyro in mtb boards 99](#_Toc183427014)

[4.5 Class periodic battery service data 101](#_Toc183427015)

[4.5.1 Messages which stream Battery data 102](#_Toc183427016)

[5 Which messages on which board 107](#_Toc183427017)

[5.1 The parsers and the formers 107](#_Toc183427018)

[5.2 The formers / parsers on each board 111](#_Toc183427019)

[6 Details of the services 114](#_Toc183427020)

[6.1 The FT in strain / strain2 boards 114](#_Toc183427021)

[6.1.1 The FT data flow 114](#_Toc183427022)

[6.1.2 Used data formats 116](#_Toc183427023)

[6.1.3 The FT settings of strain / strain2 boards 116](#_Toc183427024)

[6.1.4 The values of the regulation set 118](#_Toc183427025)

[6.1.5 Which regulation set is applied 118](#_Toc183427026)

[6.1.6 Modifications to the values of the regulation set 118](#_Toc183427027)

[6.1.7 Differences in the analog stage between strain and strain2 119](#_Toc183427028)

[6.1.7.1 The offset in strain 119](#_Toc183427029)

[6.1.7.2 The programmable amplifiers PGA308 in strain2 119](#_Toc183427030)

[6.1.8 Streaming rate 121](#_Toc183427031)

[6.2 The IMU in mtb4 / strain2 / rfe boards 122](#_Toc183427032)

[6.2.1 The used IMU: Bosch BNO055 122](#_Toc183427033)

[6.2.2 Configurability via CAN 122](#_Toc183427034)

[6.2.2.1 The operating modes 122](#_Toc183427035)

[6.2.2.2 The orientation 124](#_Toc183427036)

[6.2.3 Used data formats 126](#_Toc183427037)

[6.2.4 Streaming rate 127](#_Toc183427038)

[6.3 The SKIN in mtb / mtb4 boards 128](#_Toc183427039)

[6.3.1 The skin patch 128](#_Toc183427040)

[6.3.2 Used data formats 128](#_Toc183427041)

[6.3.3 Streaming rate 128](#_Toc183427042)

[6.4 The accelerometer and gyroscope in mtb boards 130](#_Toc183427043)

[6.4.1 The used chips: LIS331DLH and L3G4200D by STMicroelectronics 130](#_Toc183427044)

[6.4.2 Used data formats 130](#_Toc183427045)

[6.4.3 Streaming rate 130](#_Toc183427046)

[6.5 The THERMO in mtb4 / strain2 boards 131](#_Toc183427047)

[6.5.1 The used chip: SI7051 by Silicon Labs 131](#_Toc183427048)

[6.5.2 Used data formats 131](#_Toc183427049)

[6.5.3 Streaming rate 131](#_Toc183427050)

[6.6 The POS in psc board 132](#_Toc183427051)

[6.6.1 The used chip: Infineon TLV493D 132](#_Toc183427052)

[6.6.2 Used data formats 132](#_Toc183427053)

[6.6.3 Streaming rate 132](#_Toc183427054)

[7 Typical scenarios 133](#_Toc183427055)

[7.1 A CAN network 133](#_Toc183427056)

[7.2 Discovery of nodes 133](#_Toc183427057)

[7.2.1 Basic discovery 133](#_Toc183427058)

[7.2.2 More detailed discovery 135](#_Toc183427059)

[7.3 Management of nodes 137](#_Toc183427060)

[7.3.1 Change the additional information 137](#_Toc183427061)

[7.3.2 Change the CAN address 139](#_Toc183427062)

[7.3.3 Change the CAN address in a random way 140](#_Toc183427063)

[7.3.4 The firmware update procedure 142](#_Toc183427064)

[7.4 The FT service 143](#_Toc183427065)

[7.4.1 Configuration of FT service 143](#_Toc183427066)

[7.4.2 Start of the FT service 145](#_Toc183427067)

[7.4.3 Stop of the FT service 146](#_Toc183427068)

[7.5 The IMU service 148](#_Toc183427069)

[7.5.1 Configuration of IMU service 148](#_Toc183427070)

[7.5.2 Start of IMU service 150](#_Toc183427071)

[7.5.3 Stop of IMU service 153](#_Toc183427072)

[7.6 The THERMO service 154](#_Toc183427073)

[7.6.1 Configuration of THERMO service 154](#_Toc183427074)

[7.6.2 Start of THERMO service 156](#_Toc183427075)

[7.6.3 Stop of THERMO service 158](#_Toc183427076)

[7.7 The SKIN service 159](#_Toc183427077)

[7.7.1 Configuration of the SKIN service 159](#_Toc183427078)

[7.7.2 Start of the SKIN service 161](#_Toc183427079)

[7.7.3 Stop of the SKIN service 164](#_Toc183427080)

[7.8 Advanced configuration of the strain2 for FT service 165](#_Toc183427081)

[7.8.1 Use of a different regulation set at bootstrap 165](#_Toc183427082)

[7.8.2 Saving to EEPROM the changes done on a regulation set 168](#_Toc183427083)

[7.8.3 Use of a different temporary regulation set 169](#_Toc183427084)

[7.8.4 Tuning of the analog front end of a given regulation set 171](#_Toc183427085)

[7.8.5 Tune the offset of the analog front end to move the ADC acquisition at midrange 173](#_Toc183427086)

[7.9 The POS service 175](#_Toc183427087)

[7.9.1 Configuration of POS service 175](#_Toc183427088)

[7.9.2 Start of POS service 177](#_Toc183427089)

[7.9.3 Stop of POS service 180](#_Toc183427090)

[7.10 The BAT service 181](#_Toc183427091)

[7.10.1 Configuration of BAT service 181](#_Toc183427092)

[7.10.2 Start of BAT service 182](#_Toc183427093)

[7.10.3 Logging of error and diagnostic messages 183](#_Toc183427094)

[7.10.4 Stop of BAT service 184](#_Toc183427095)

[8 Conclusions 185](#_Toc183427096)

[9 References 186](#_Toc183427097)

# Introduction

This document describes the iCub CAN protocol used by the sensor boards: strain, strain2, mtb, mtb4, mais, et cetera.

A first part deals with the generalities of the CAN protocol. It tells the format of the CAN frame in the iCub protocol, tells about the CAN network and which communication paradigms can be used. It also tells how the messages are classified and which services they offer. It is formed by section 2.

The second part is the core part of this documents and explains the details of data format used in the CAN messages. It is formed by sections 3 and 4.

Finally, a third part describe how to use the messages. It includes details of the offered service and examples of how to use the messages in the most common scenarios. It is formed by sections 5, 6 and 7.

## Where to find what you are looking for

To better clarify where to find information, please read the following.

### General details of the CAN protocol

See Section 2 (**The iCub CAN protocol)**.

### The reference binary format of the CAN messages

See Section 3 (**The messages in the command classes)** and Chapter 4 (**The messages of the streaming classes**).

### The messages supported by which boards

See Section 5 (**Which messages on which board)**.

### The services offered by the CAN protocol

See Section 2.5 (**The services offered by the CAN protocol)** for an introduction and Section 6 (**Details of the services**) for full details.

### Typical use cases

In Section 7 (**Typical scenarios**) there are examples of how to use the CAN messages for starting a service, receiving its data, stopping the service.

However, take the pictures and data formats in this section as examples. Hence, ***always*** check the reference binary format of the CAN message in sections 3 and 4.

The iCub CAN protocol

The iCub CAN protocol follows the standard CAN frame format, which has an 11-bits identifier and up to 8 bytes of payload. Transmission rate is 1mbps.

It uses the 11 bits of the identifier to manage up to 15 nodes with messages belonging to 7 classes, each one with an associated *mode* (command or streaming). The classes resemble the type of services used inside the robot: motion control, analog sensors, skin sensors, inertial sensors and board management (discovery, firmware update, etc.).

## The structure of the CAN frame

The following figure shows full details of how the CAN frame is organized.

CAN FRAME

ID

PAYLOAD

11 BITS

0 / 8 BYTES

Its format depends on the service mode of CLS.

CLS

SRC

DST/TYP

3 BITS 4 BITS 4 BITS

Its meaning depends on the value of CLS and in particular if it is a command or streaming class. This 4bits field may contain:

* The destination address (DST) for command classes CLS = {000b, 010b, 111b}.
* The type of content (TYP) for streaming classes, hence if CLS = {001b, 011b, 100b, 101b, 110b}

It is the source address of the CAN frame.

It is the class of the message. Here are the classes together with the associated service mode (command or streaming):

* 000b: polling motor control [command]
* 001b: periodic motor control [streaming]
* 010b: polling analog sensors [command]
* 011b: periodic analog sensors [streaming]
* 100b: periodic skin data [streaming]
* 101b: periodic inertial sensors [streaming]
* 110b: periodic battery data [streaming]
* 111b: management/bootloader [command]

CMD

ARG

DATA

command

streaming

1 BYTE

**Figure 1**: Partitioning of a CAN frame in the iCub CAN protocol.

### The ID

In the iCub CAN protocol the 11 bits of the ID are divided in three parts: CLS, SRC and DST/TYP.

* CLS is 3 bits long and specifies the class. Each class has an associated service mode: either command or streaming. The command messages use both unicast and broadcast transport. The streaming messages always use broadcast transport.
* SRC has 4 bits and specifies the address of the sender.
* The third part is of 4 bits and its meaning depends on CLS.
* If CLS has a command service mode, then the 4 bits contain the destination address DST. In such a case, the first byte of the payload contains the command code CMD and the remaining 7 bytes contain the required parameters.
* If CLS has a streaming service mode, then the 4 bits contain the type TYP of data contained in the payload.

### The PAYLOAD

The content of the payload depends on the value of ID.CLS and in particular on its service mode.

* If ID.CLS is of command service mode, we the payload must contain a command and its arguments. The first byte is PAYLOAD.CMD and specifies which operation to apply. The remaining 7 bytes PAYLOAD.ARG are available for hosting the parameters of the command.
* If ID.CLS is of streaming service mode, we identify what we transport by ID.TYP and then we can use the full capacity of the payload to contain only data. The number of used bytes and their format depends on ID.TYP.

## The iCub network

The iCub network is formed by up to 15 devices each with a given CAN address ADDR in range [0, 14]. The device with address = 0 is called host, the others are called nodes.

The iCub CAN network

HOST @ 0

node @ 1

The HOST is responsible to manage all the nodes and to receive data.

node @ 3

node @ 13

node @ 14

node @ 2

Each node is specialized for a given task: motion control, analog sensors etc.

**Figure 2**: The iCub network.

### The iCub host

The iCub host is a computing device with a CAN address = 0 which is capable to manage all the nodes in its network. As such, it must be able to send commands and to parse the messages sent by the nodes (replies to commands or streaming messages).

#### Examples of possible hosts

There are several implementations of the host, which reflect the evolution of iCub and the required use-cases. Here are two examples.

***The host is a PC with a USB-CAN dongle****.* The PC runs the CANreal GUI, which allows to edit CAN frames (with ID.SRC = 0), transmit them, and see replies through the CAN-USB dongle.

The CANreal used as a HOST



The HOST sends a discovery command

A node @ 1 replies: it is a strain2.

The node @ 1 sends back the string:

“I am a strain2”.

The HOST asks node @ 1 for more information

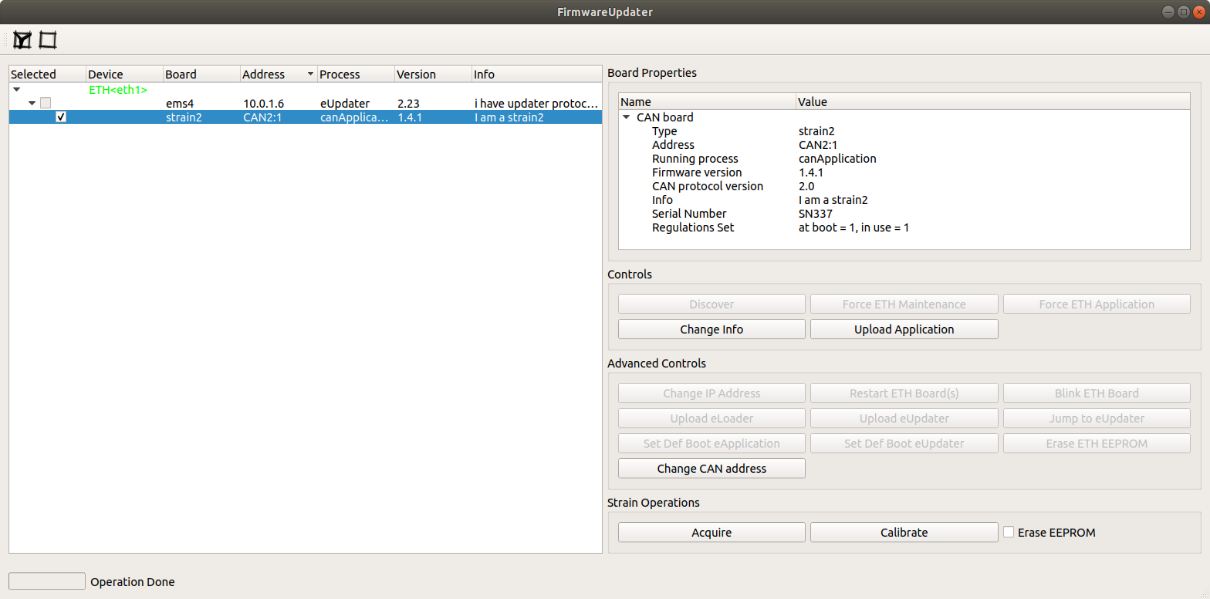
**Figure 3**: *The host is a PC with a USB-CAN dongle*. The CANreal GUI manages raw frames.

***The host is an ETH-board inside iCub***. In ETH-based iCub robots, the PC104 communicates via IP to the ETH-boards (e.g., the EMS) and each of them acts as a HOST able to send and receive CAN frames.

In maintenance mode, FirmwareUpdater runs on the PC104 and acts as a former / parser of CAN frames. The ETH-board just acts as a physical host with CAN ADDR = 0 that passes through the frames for other nodes.

The following figure shows the FirmwareUpdater to perform the same discovery operation already seen with the CANreal. FirmwareUpdater hides the direct view of CAN frames ad shows only high-level information.

The ems board is the physical HOST



This button launches discovery

This panel shows properties of the found boards

**Figure 4**: *The host is an ETH-board inside iCub*. The FirmwareUpdater GUI uses high-level concepts and hides the handling of CAN frames in the supporting libraries.

### The iCub CAN nodes

An iCub CAN node is characterised by a unique CAN address and a proper board type. It may run either the bootloader process or the application process.

The iCub CAN node

The CAN node executes a process (bootloader or application) that runs a basic parser plus other parsers which depends on the board type. The CAN PHY delivers the CAN frames to those parsers on the basis of the ID of the frame and the ADDR of the board.

NODE

EEPROM

process

other parsers

basic parser

ADDR

BOARDTYPE

CAN PHY

**Figure 5**: The iCub CAN node.

#### Valid addresses

We refer to the CAN address of a node with ADDR which has a range [1, 14]. The address 15, called EVERYONE, is reserved for broadcasting.

#### Processes executed by a CAN node: bootloader and application

The iCub CAN node runs the bootloader or the application process. At bootstrap it executes the bootloader, and if the bootloader does not detect any CAN frame activity, it executes the application after 5 seconds.

The application is the one that does the real job. However, the bootloader is useful as well because it offers recovery possibility in case of a buggy application. It also performs the firmware update of the application process.

The behaviour at bootstrap of the CAN node

Countdown expired

bootloader

Received a CAN frame: stop countdown.

application

Starts 5 sec countdown

**Figure 6**: Behaviour at bootstrap of the CAN node.

#### The board types

Each board runs a specific application that does the proper job. We roughly divide the boards based on their principal activity: motor-control boards (MOT) and sensor boards (SNS).

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Name | Activity | Description |
| 0x00 | dsp | MOT | DSP motor controlling board  **OBSOLETE** |
| 0x01 | pic | ? | PIC board  **OBSOLETE** |
| 0x02 | 2dc | MOT | DSP motor controlling board  **OBSOLETE** |
| 0x03 | mc4 | MOT | Motor control board for 4 DC motors  It is used in the CAN-based forearms of iCub  Its particularity is that it responds to two addresses: its primary address ADR and ADR+1. |
| 0x04 | bll | MOT | Motor control board for 2 brushless motors  It is used in CAN-based iCub. |
| 0x05 | mtb | SNS | mtb  It manages skin data and basic inertial sensors. |
| 0x06 | strain | SNS | strain  It manages FT data. |
| 0x07 | mais | SNS | MAIS board  It is used in the forearms of iCub to acquire positions of the hand. |
| 0x08 | foc | MOT | FOC  It is hosted in a 2FOC board and is used in ETH-based robots to drive one brushless motor. The board 2FOC mounts electronics to host two foc boards. |
| 0x09 | 6sg | SNS | 6SG  **OBSOLETE** |
| 0x0A | jog | MOT | JOG  A test board used to verify hardware in the robot, typically AEA etc. |
| 0x0B | mtb4 | SNS | MTB4  An enhancement of the mtb board. It interfaces skin patches, it has an IMU and temperature sensors. |
| 0x0C | strain2 | SNS | STRAIN2  An enhancement of the strain board. It manages FT data, it has an IMU and temperature sensors. |
| 0x0D | rfe | SNS | Robot Face Expression Board  It handles LEDs for face expression in iCub. |
| 0x0E | sg3 | SNS | Strain Gauges 3  It handles acquisition of three values in R1 hand. |
| 0x0F | psc | SNS | Proximal sensor collector  It handles acquisition of positions (in angles) in the R1 hand. |
| 0x10 | mtb4w | SNS | MTB4 for Waseda University  It handles sensors acquisition via emulated I2C. |
| 0x12 | amcbldc | MOT | The first MBD (model based designed) motor control board |
| 0x13 | bms | SNS |  |
| 0x14 | mtb4c | SNS | The same as mtb4 but w/ another MPU |
| 0x15 | amc2c | MOT | The second MBD motor control board, second core of the amc ETH board |
| 0x16 | strain2c | SNS | The same as strain2 but w/ another MPU |
| 0x17 | bat | SNS | Manages a battery |

**Table 1** – The board types supported by the iCub CAN protocol.

#### Icons used to represent the nodes

The following of the document will use the icons in the table to refer to a given node, to its running process and its board type. We shall use the yellow colour to represent a node running the bootloader and a green colour to represent its application.

See following table for examples.

|  |  |
| --- | --- |
| Icon | Description |
| bootloader | Generic bootloader  It represents a node of any board type running the bootloader. |
| application | Generic application  It represents a node of any board type running the application. |
| strain2  app<strain2> | A particular application  It represents a node of board type strain2 running the application. |
| mtb4  btl<mtb4> | A particular bootloader  It represents a node of board type strain2 running the application. |
| MOT  app<MOT> | A particular class of applications  It represents any node of a motor-control board running the application. |
| MOT  btl<MOT> | A particular class of applications  It represents any node of every motor-control board running the application. |
| SNS  app<SNS> | A particular class of applications  It represents any node of a sensor board running the application. |
| SNS  btl<SNS> | A particular class of applications  It represents any node of every sensor board running the application. |

**Table 2** – The icons describing the nodes with running process and type of board.

## The supported communication modes

The iCub CAN protocol supports three transport modes: unicast, broadcast and streaming (which is a periodic broadcast). We can use them to transport the following operations: SET<>, GET<>, REPLY<>, ORDER<>, and STREAM<>.

Here is description of them.

### Transport modes

They are the following: unicast, broadcast and streaming.

#### Unicast transport

Unicast transport mode

node @ 1

HOST @ 0

This is a frame with ID.CLS of type command and ID.DST = 1

accepted

node @ 2

discarted

**Figure 7**: A unicast frame is sent to a single address. It is represented with an arrow point.

#### Broadcast transport

Broadcast transport mode

node @ 1

HOST @ 0

This is a frame with ID.CLS of type command and ID.DST = 15

accepted

node @ 2

accepted

**Figure 8**: A broadcast frame is sent to every address. It is represented with a round point.

#### Streaming transport

Streaming transport mode

node @ 1

HOST @ 0

These frames have ID.CLS of type streaming

accepted

node @ 2

accepted

**Figure 9**: A streaming frame is sent to everybody (but typically, only the host is able to decode it). We represent a streaming frame with a diamond shaped point.

### Operations

They are: SET<>, GET<>, REPLY<>, ORDER<>, and STREAM<>. Here are examples of their use.

#### The SET<> operation is used to impose a value

Sequence diagram of SET<> operation

node @ 1

HOST @ 0

Its sets a value in a way expressed by PAYLOAD.CMD and PAYLOAD.ARG

It accepts the command and uses what in PAYLOAD.ARG to change an internal value

SET<>

**Figure 10**: The SET<> operation. By it the HOST writes a value inside the node.

#### The GET<> operation is used to retrieve a value with REPLY<>

Sequence diagram of GET<> operation

node @ 1

HOST @ 0

Its sends a request expressed by PAYLOAD.CMD

It sends back the value in PAYLOAD.ARG

GET<>

REPLY<>

**Figure 11**: The GET<> operation. By it, the HOST reads a value that the node sends with a REPLY<>.

#### The ORDER<> operation is used to impose an action or a behaviour

Sequence diagram of ORDER<> operation

node @ 1

HOST @ 0

Its sends an order expressed by PAYLOAD.CMD and PAYLOAD.ARG

It accepts the command and does an action.

A reply is possible, but depends on each CMD. It may be an OK/KO.

ORDER<>

**Figure 12**: The ORDER<> operation. Used to start an action, for instance the start of streaming.

#### The STREAM<> operation is used to transport a value

Sequence diagram of STREAM<> operation

node @ 1

HOST @ 0

It send data stored in PAYLOAD.DATA with periodic frames

STREAM<>

**Figure 13**: The STREAM<> operation. By it, the node sends acquired data in a periodic and spontaneous way.

### Messages accepted by a CAN node

Every CAN node is able to accept (but maybe not decode):

* All frames that belong to one of the command message classes that are sent in unicast to it or in broadcast. In other words, if CLS = {000b, 010b, 111b} and ID.DST equal to the node address ADR or equal to EVERYONE = 15.
* All frames that belong to one of the streaming classes CLS = {001b, 011b, 100b, 101b}.

CAN frames accepted by a CAN node

This is a streaming frame

node @ addr = 1

This frame is propagated to the CAN parser

This is a broadcast frame with ID.DST = 15

This frame is propagated to the CAN parser

This is a unicast frame with ID.DST = 1

This frame is ignored

This frame is propagated to the CAN parser

This is a unicast frame with ID.DST = 2

**Figure 14**: A node accepts three types of CAN frames: unicast class sent directly to its address (arrow point), unicast class sent to the EVERYONE address = 15 (diamond shape point), and broadcast class (diamond point). All unicast class frames sent to other addresses are ignored.

## The iCub CAN protocol classes

Here are the classes used by the iCub CAN protocol. They are divided according to their service: command and streaming classes.

### The command classes

These classes are used for SET<>, GET<>, REPLY<> or ORDER<> operations. Typically, we use messages belonging to these classes to configure or start/stop a service. The flow of these commands typically originates from the host and terminates on each board.

These classes contain up to 255 possible CMDs each. All the CMD fields available for each class will be shown later in the document.

|  |  |
| --- | --- |
| CLS | Description |
| **000b** | **Polling motor control**  [mode: command]  It is used to send messages related to motion control from one node, typically the host, which has address SRC = 000b, to another node with address DST = xxxb. |
| **010b** | **Polling analog sensors**  [mode: command]  It is used to send messages related to analog sensors from one node, typically the host which has address SRC = 000b, to another node with address DST = xxxb. |
| **111b** | **Management/bootloader**  [mode: command]  It is used to send messages related to management issues from one node, typically the host which has address SRC = 000b, to another node which address DST = xxxb.  This class contains messages for discovery of the nodes in the network, for setting and asking user-defined descriptive information of the nodes, and for performing firmware update. |

**Table 3** – The command classes in the iCub CAN protocol.

### The streaming classes

These classes are used STREAM<> operations. They stream data from the boards towards other boards, typically to the host.

These classes contain up to 16 TYP of messages each. All the TYP fields available for each class will be shown later in the document.

|  |  |
| --- | --- |
| CLS | Description |
| **001b** | **Periodic motor control**  [mode: streaming]  It is used to send periodic messages related to motion control from one node with address SRC = xxxb to every other node.  Typically, the host commands the node to begin broadcasting with a message of CLS = 000b, then it listens and decodes. |
| **011b** | **Periodic analog sensors**  [mode: streaming]  It is used to send periodic messages related to analog sensors from one node with address SRC = xxxb to every other node.  Typically, the node begins broadcasting after it has received a message of CLS = 010b with an order to transmit at a given rate |
| **100b** | **Periodic skin data**  [transport: streaming]  It is used to send periodic messages related to skin data from one node with address SRC = xxxb to every other node.  Typically, the host commands the node to begin broadcasting with a message of CLS = 010b, then it listens and decodes. |
| **101b** | **Periodic inertial data**  [transport: streaming]  It is used to send periodic messages related to inertial data from one node with address SRC = xxxb to every other node.  Typically, the host commands the node to begin broadcasting with a message of CLS = 010b, then it listens and decodes. |
| **110b** | **Periodic battery data**  [transport: streaming]  It is used to send periodic messages related to battery status from one node with address SRC = xxxb to every other node.  Typically, the host commands the node to begin broadcasting with a message of CLS = 010b, then it listens and decodes. |

**Table 4** – The broadcast classes in the iCub CAN protocol.

## The services offered by the CAN protocol

The iCub CAN protocol has grown to manage services offered by the boards.

At the beginning, one board was almost entirely dedicated to offer a unique service. For instance, the strain board is dedicated entirely to force-torque acquisition. However, with time we have integrated more and more functionalities inside a board. It is the case of the strain2, which hosts force-torque acquisition, IMU, and temperature.

The iCub CAN protocol originally attempted to classify messages according to their principal use. It is for this reason that we have a motion control classes for commands and streaming and analog sensors classes.

Here is a tentative classification of such services

### The management service

Used to discover nodes, change their addresses, impose and retrieve user-defined information, perform firmware update. It is available on every board type by bootloader and applications.

It uses messages of classes: management/bootloader (CLS = 111b), but also, some messages of polling motion control (CLS = 000b) and polling analog sensors (CLS = 010b)

### The MC service

Used to move motors. It is available on the application of every board of type motor control.

It uses messages of classes: polling motion control (CLS = 000b) and periodic motor control (CLS = 001b).

### The FT service

Used to acquire force-torque values. It is available on the application of boards of type strain and strain2.

It uses messages of classes: polling analog sensors (CLS = 010b) and periodic analog sensors (CLS = 011b).

### The HES service

Used to acquire Hall-effect sensor values. It is available on the application of boards of type mais.

It uses messages of classes: polling analog sensors (CLS = 010b) and periodic analog sensors (CLS = 011b).

### The Skin service

Used to acquire pressure values from skin patches. It is available on the application of boards of type mtb, mtb4 and psc.

It uses messages of classes: polling analog sensors (CLS = 010b) and periodic skin data (CLS = 100b).

### The basic inertial service

Used to acquire accelerometer and gyroscope values. It is available on the application of mtb boards only. It has now evolved into the IMU service.

It uses messages of classes: polling analog sensors (CLS = 010b) and periodic inertial data (CLS = 101b).

### The IMU service

Used to acquire IMU values. It is available on the application of boards of type mtb4 and strain2.

It uses messages of classes: polling analog sensors (CLS = 010b) and periodic inertial data (CLS = 101b).

### The THERMO service

Used to acquire temperature values. It is available on the application of boards of type mtb4 and strain2.

It uses messages of classes: polling analog sensors (CLS = 010b) and periodic analog sensors (CLS = 011b).

### The POS service

Used to acquire position values. It is available on the application of boards of type psc only for angle acquisition.

It uses messages of classes: polling analog sensors (CLS = 010b) and periodic analog sensors (CLS = 011b).

### The BAT service

Used to acquire the status of a battery.

It uses messages of classes: polling analog sensors (CLS = 010b) and periodic battery data (CLS = 011b).

The messages in the command classes

Here is description of the messages in the command classes.

## Class management / bootloader

In this class (CLS = 111b) there are the commands used to perform basic management services which are common to every board. The name of the class is originally bootloader because this class contains commands used for the firmware updater procedure that the bootloader executes. However, this name is misleading because also the applications use its commands for other purposes such as discovery and change of address.

Here is the complete list.

|  |  |
| --- | --- |
| CMD | Description |
| 0x00 | BOARD |
| 0x01 | ADDRESS |
| 0x02 | START |
| 0x03 | DATA |
| 0x04 | END |
| 0x0C | GET\_ADDITIONAL\_INFO |
| 0x0D | SET\_ADDITIONAL\_INFO |
| 0x32 | SET\_ADDRESS |
| 0xFF | BROADCAST (or DISCOVER) |

**Table 5** – The CMDs of the bootloader / management class.

### Messages used for general management of boards

Here are the messages used for management. They allow discovering the nodes in the network, to retrieve user-defined information, to modify it, and to change the address of the nodes.

|  |  |
| --- | --- |
| CMD | Description |
| **0xFF** | **BROADCAST (or DISCOVER)**  **In brief**  It is used to discover the nodes in the CAN network. It is typically transmitted to all nodes in the network using ID.DST = 15 = EVERYONE.  **Parsed by**    bootloader  application  **Format of PAYLOAD**  0xFF   * sizeof(ARG) = 0.   **Actions on reception**    bootloader  application  It just replies to the sender.  **Reply**  bootloader  It replies to the sender with the same CMD and following format.  0xFF  BRD  BL.maj  BL.min   * sizeof(ARG) = 3 * BRD is the board type. See table 1 for values. * BL.maj is the major number of the bootloader version. * BL.min is the minor number of the bootloader version.   application  It replies to the sender with the same CMD and following format.  0xFF  BRD  AP.maj  AP.min  AP.bld   * sizeof(ARG) = 4 * BRD is the board type. See table 1 for values. * AP.maj is the major number of the application version. * AP.min is the minor number of the application version. * AP.bld is the build number of the application version.   **Note**  It is possible to detect if a node runs the bootloader or the application simply by looking at the size of the payload of the reply to a DISCOVER message. If it is 5, it is running the application, else the bootloader. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x0C** | **GET\_ADDITIONAL\_INFO**  **In brief**  Used to get additional information that is stored in storage of the node. The additional information is a string of up to 32 characters.  **Parsed by**    bootloader  application  **Format of PAYLOAD**  0x0C   * sizeof(ARG) = 0.   **Actions on reception**    bootloader  application  It just replies to the sender.  **Reply**    bootloader  application  It sends back a burst of M = 8 messages which describe the string of the additional info, let’s say contained inside char info32[32].  It replies to the sender with the same CMD and following format.  0x0C  SEQ  C0  C1  C2  C3   * sizeof(ARG) = 5 * SEQ is the sequence number of the burst and has values in [0, M-1]. * The Cx bytes contain the value of info32[4\*SEQ+x]. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x0D** | **SET\_ADDITIONAL\_INFO**  **In brief**  Used to set additional information that is stored in storage of the node. The additional information is a string of up to 32 characters.  The way this command is used is through a burst of M = 8 messages which describe the string of the additional info, let’s say contained inside char info32[32].  **Parsed by**    bootloader  application  **Format of PAYLOAD**  0x0D  SEQ  C0  C1  C2  C3   * sizeof(ARG) = 5. * SEQ is the sequence number of the burst and has values in [0, M-1]. * The Cx bytes contain the value to copy into info32[4\*SEQ+x].   **Actions on reception**    bootloader  application  It copies in a RAM buffer info32tmp[] the four characters contained in [C0, C1, C2, C3] and when all the M = 8 messages are received (so that all the 32 characters are inside the RAM buffer), it copies info32tmp[] into the final info32[] location, possibly on EEPROM.  **Reply**    bootloader  application  It does not reply. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x32** | **SET\_ADDRESS**  **In brief**  Used to change the CAN address. It is a new message that enables both explicit and random address assignment modes. This latter is useful when two or more nodes accidentally have the same address.  **Parsed by**    bl<mtb4>  bl<strain2>  bl<rfe>  mtb4  strain2  rfe  **Format of PAYLOAD**  0x32  ADR  INVALIDMASK   * sizeof(ARG) = 3. * ADR is the new address. Valid values are [1, 14] and 0xff. This latter specify to use the random assignment mode. * INVALIDMASK contains a bitmask stored as a std::uint16\_t in little endian. Tis mask is used only by the random assignment mode to tell which addresses to exclude: if the bit in position x is 1, then the address x must be excluded. The mask should not be 0x   **Actions on reception**    bootloader  application  If ADR is valid and not 0xff then we assign the new address equal to ADR. Else, if it is equal to 0xff we attempt to assign a random address allowed by the INVALIDMASK.  **Reply**    bootloader  application  It does not reply.  **Note**  The legacy mode to change address is to use a message of a different unicast class for each specific node. For instance the 2foc requires the message CMD = 0x32 of CLS = 000b (polling motion control) and the strain requires the message CMD = 0x32 of CLS = 010b (polling analog sensors). |

### Messages used in the firmware update procedure

The firmware updater procedure allows the bootloader to change the code space of the application based on messages received by a host.

Here are details of the messages.

|  |  |
| --- | --- |
| CMD | Description |
| **0x00** | **BOARD**  **In brief**  It commands the start of the firmware update procedure.  **Parsed by**    bootloader  application  **Format of PAYLOAD**    0x00  EE   * sizeof(ARG) = 1. * EE is a boolean (0 /1) which tells if after firmware update it is required to erase all user-defined storage (EEPROM).   **Actions on reception**  application  It just resets to execute the bootloader.  bootloader  It stores the EE info and enters the firmware update procedure.  **Reply**  application  It does not reply.  bootloader  It replies with the same CMD and empty ARG.  0x00   * sizeof(ARG) = 0 |

|  |  |
| --- | --- |
| CMD | Description |
| **0x01** | **ADDRESS**  **In brief**  Used for the firmware updater procedure. It tells: how many bytes of code data the DATA command will send, their starting address in program space, and if the code data belongs to program memory or data memory.  **Parsed by**  bootloader  **Format of PAYLOAD**  0x01  N  ADDR-L  T  ADDR-H   * sizeof(ARG) = 6. * N contains the number of bytes of program data that the command DATA is going to transmit. * ADDR-L contains the less significant part of the starting address (ADDR) in code space of those bytes. It is expressed as a std::uint16\_t in little endian. For 16-bits addressing modes that is enough. * T contains the type of code space we want to manage. A value of 0 means program memory, a value of 1 means data memory. In hex format it is always 0. * ADDR-H contains the most significant part of the starting address (ADDR) in code space. It is expressed as a std::uint16\_t in little endian. It is used in all the 32-bits addressing modes (as in hex format).   **Actions on reception**  bootloader  It memorises N and ADDR = (ADDR-L + (ADDR-H >> 16)) in a new section of code space to program. We call it:  struct Section { uint32\_t start; uint32\_t end; uint32\_t curr; };  Section section = {ADDR, ADDR+N, ADDR};  **Reply**  bootloader  It does not reply |

|  |  |
| --- | --- |
| CMD | Description |
| **0x03** | **DATA**  **In brief**  Used for the firmware updater procedure. It contains the code data that the bootloader will burn in the section of code space previously descripted by the message ADDRESS.  **Parsed by**  bootloader  **Format of PAYLOAD**  0x03  CODEDATA   * sizeof(ARG) = S. It is variable. * CODEDATA contains the raw bytes that the bootloader will burn in code space. Its number is sizeof(PAYLOAD) – 1, which typically is equal to 6.   **Actions on reception**  bootloader  It burns CODEDATA in code space at address section.curr, but only if it does not go beyond section.end. It increments current address with: section.curr += S, so that next DATA command can write in the proper address.  **Reply**  bootloader  It may reply or not.  It replies to the sender with the same CMD and OK/KO information only when the node has received all the data as indicated by previous ADDRESS command.  0x03  OKKO   * sizeof(ARG) = 1 * OKKO is 1 is everything was OK, 0 otherwise.   **Note**  Typically the firmware update procedure uses sequences formed by messages {ADDRESS, {DATA}D}C. As an example, to write 8 code sections of 64 bytes each, one can use messages with ADDRESS.N = 64, sizeof(DATA.CODEDATA) = 4, D = 16, and C = 8. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x02** | **START**  **In brief**  Used for the firmware updater procedure. It tells to … start the termination of the procedure.  **Parsed by**  bootloader  **Format of PAYLOAD**  0x02  CODEDATA   * sizeof(ARG) = S. It is variable. * CODEDATA contains the raw bytes that the bootloader will burn in code space. Its number is sizeof(PAYLOAD) – 1, which typically is equal to 6.   **Actions on reception**  bootloader  It ensures that all received data is burned in code space and, if requested by command BOARD with its BOARD.EE, it erases the EEPROM.  **Reply**  bootloader  It replies to the sender with the same CMD and OK/KO information.  0x02  OKKO   * sizeof(ARG) = 1 * OKKO is 1 is everything was OK, 0 otherwise. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x04** | **END**  **In brief**  Used for the firmware updater procedure. It tells that the procedure is over and to restart to run the application.  **Parsed by**  bootloader  **Format of PAYLOAD**  0x04   * sizeof(ARG) = 0.   **Actions on reception**  bootloader  It sends back a reply and then it restarts.  **Reply**  bootloader  It replies to the sender with the same CMD and OK/KO information.  0x04  OKKO   * sizeof(ARG) = 1 * OKKO is 1 is everything was OK, 0 otherwise. |

## Class polling motor control

The messages of class polling motor control (CLS = 000b) are used mostly for motor control boards (MOT). However, some sensor boards support some of them.

|  |  |
| --- | --- |
| CMD | Description |
|  | … others |
| 0x32 | SET\_BOARD\_ID |
|  | … others |
| 0x52 | GET\_FIRMWARE\_VERSION |
|  | … others |

**Table 6** – The CMDs of the polling motor control class (supported by some sensor boards).

### Messages used for general management of boards

They are messages originally developed for MC boards that have similar versions for sensor boards. The new boards (mtb4, strain2, rfe) all support these messages to enhance the freedom of the host and avoid confusion on what message send to what board. However, their right place is in the bootloader/management class, where we already have added some of them.

|  |  |
| --- | --- |
| CMD | Description |
| **0x32** | **SET\_BOARD\_ID**  **In brief**  It commands a change of CAN address to motor control boards (but also of strain2, mtb4 and rfe).  **Parsed by**    btl<MC>  appl<MC>  strain2  mtb4  rfe  **Format of PAYLOAD**    0x32  ADR   * sizeof(ARG) = 1. * ADR is the new address. Valid values are [1, 14].   **Actions on reception**    btl<MC>  appl<MC>  strain2  mtb4  rfe  It imposes a new address to be ARG.ADR.  **Reply**    btl<MC>  appl<MC>  strain2  mtb4  rfe  It does not reply. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x5B** | **GET\_FIRMWARE\_VERSION**  **In brief**  Used to get information about application and CAN protocol versions of motor control boards (but also of strain2, mtb4 and rfe).  **Parsed by**    appl<MC>  strain2  mtb4  rfe  **Format of PAYLOAD**  0x5B  REQ.ma  REQ.mi   * sizeof(ARG) = 0. * (REQ.ma, REQ.mi) is the CAN protocol version required by the host to do effective communication.   **Actions on reception**    appl<MC>  strain2  mtb4  rfe  It just replies to the sender.  **Reply**    appl<MC>  strain2  mtb4  rfe  It replies to the sender with the same CMD and following format.  0x5B  BRD  APP.ma  APP.mi  APP.bu  CAN.ma  CAN.mi  COMP   * sizeof(ARG) = 7 * BRD is the type of board. * (APP.ma, APP.mi, APP.bu) contains the application version in major, minor, build. * (CAN.ma, CAM.mi) contains the CAN protocol version in major, minor. * COMP = ((CAN.ma == REG.ma) && (CAN.mi >= REQ.mi)) ? (1) : (0) |

### All the other messages

The description of every MC command message is out of the scope of this document.

## Class polling analog sensors

The class polling analog sensors (CLS = 010b) contains messages dedicated to sensor boards.

|  |  |
| --- | --- |
| CMD | Description |
| 0x03 | SET\_MATRIX\_RC |
| 0x04 | SET\_CH\_DAC |
| 0x07 | SET\_TXMODE |
| 0x08 | SET\_CANDATARATE |
| 0x09 | SAVE2EE |
| 0x0A | GET\_MATRIX\_RC |
| 0x0B | GET\_CH\_DAC |
| 0x0C | GET\_CH\_ADC |
| 0x10 | SET\_RESOLUTION (SET\_MAIS\_RESOLUTION) |
| 0x11 | SET\_MATRIX\_G |
| 0x12 | GET\_MATRIX\_G |
| 0x13 | SET\_CALIB\_TARE |
| 0x14 | GET\_CALIB\_TARE |
| 0x15 | SET\_CURR\_TARE |
| 0x16 | GET\_CURR\_TARE |
| 0x17 | SET\_FULLSCALE |
| 0x18 | GET\_FULLSCALE |
| 0x19 | SET\_SERIAL\_NO |
| 0x1A | GET\_SERIAL\_NO |
| 0x1B | GET\_EEPROM\_STATUS |
| 0x1C | GET\_FIRMWARE\_VERSION |
| 0x1D | AMPLIFIER\_RESET |
| 0x1E | AMPLIFIER\_RANGE\_OF\_GAIN\_GET |
| 0x1F | AMPLIFIER\_RANGE\_OF\_OFFSET\_GET |
| 0x20 | AMPLIFIER\_GAINOFFSET\_GET |
| 0x21 | AMPLIFIER\_GAINOFFSET\_SET |
| 0x22 | AMPLIFIER\_OFFSET\_AUTOCALIB |
| 0x2A | AMPLIFIER\_PGA308\_CFG1\_GET |
| 0x2B | AMPLIFIER\_PGA308\_CFG1\_SET |
| 0x32 | SET\_BOARD\_ADX |
| 0x33 | IMU\_CONFIG\_GET |
| 0x34 | IMU\_CONFIG\_SET |
| 0x35 | IMU\_TRANSMIT |
| 0x38 | THERMOMETER\_CONFIG\_GET |
| 0x39 | THERMOMETER\_CONFIG\_SET |
| 0x3A | THERMOMETER\_TRANSMIT |
| 0x3D | REGULATIONSET\_SET |
| 0x3E | REGULATIONSET\_GET |
| 0x4D | SKIN\_SET\_BRD\_CFG |
| 0x4F | SKIN\_ACC\_GYRO\_SETUP |
| 0x50 | SET\_TRIANG\_CFG |
| 0x51 | POS\_CONFIG\_GET |
| 0x52 | POS\_CONFIG\_SET |
| 0x53 | POS\_TRANSMIT |

**Table 7** – The CMDs of the polling analog sensor class.

### Messages used for general management of boards

They are command messages that have similar one in the polling MC class. The host must send the command of the correct class after it has detected the board type with a DISCOVER message of the bootloader / management class.

|  |  |
| --- | --- |
| CMD | Description |
| **0x32** | **SET\_BOARD\_ADX**  **In brief**  It commands a change of CAN address in sensor boards.  **Parsed by**    btl<SNS>  app<SNS>  **Format of PAYLOAD**    0x32  ADR   * sizeof(ARG) = 1. * ADR is the new address. Valid values are [1, 14].   **Actions on reception**    btl<SNS>  app<SNS>  It imposes a new address to be ARG.ADR.  **Reply**    btl<SNS>  app<SNS>  It does not reply.  **Note**  The class motor control polling has a similar message.  The bootloader / management class also has a similar message with extended features. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x1C** | **GET\_FIRMWARE\_VERSION**  **In brief**  Used to get information about application and CAN protocol versions of sensor boards only.  **Parsed by**    strain  strain2  mtb  mtb4  mais  **Format of PAYLOAD**  0x5B  REQ.ma  REQ.mi   * sizeof(ARG) = 0. * (REQ.ma, REQ.mi) is the CAN protocol version required by the host to do effective communication.   **Actions on reception**    strain  strain2  mtb  mtb4  mais  It just replies to the sender.  **Reply**    strain  strain2  mtb  mtb4  mais  It replies to the sender with the same CMD and following format.  0x5B  BRD  APP.ma  APP.mi  APP.bu  CAN.ma  CAN.mi  COMP   * sizeof(ARG) = 7 * BRD is the type of board. * (APP.ma, APP.mi, APP.bu) contains the application version in major, minor, build. * (CAN.ma, CAM.mi) contains the CAN protocol version in major, minor. * COMP = ((CAN.ma == REG.ma) && (CAN.mi >= REQ.mi)) ? (1) : (0)   **Note**  The class motor control polling has a similar message. But … **beware**: the message in class polling motor control has a different CMD value. |

### Messages for managing the FT and HES data service

In here are messages used by boards to configure rate and start/stop of their main service: FT for strain/strain2 boards and HES (Hall-effect sensor) data in the mais board.

|  |  |
| --- | --- |
| CMD | Description |
| **0x08** | **SET\_CANDATARATE**  **In brief**  Used to set the transmission rate of some services in sensor boards: hall effect sensor data in mais board, FT data in strain / strain2 boards.  **Parsed by**    strain  strain2  mais  **Format of PAYLOAD**  0x07  RATE   * sizeof(ARG) = 1. * RATE contains the tx rate in milli-seconds.   **Actions on reception**    strain  strain2  mais  It memorises the TX rate for the FT or hall effect sensor.  **Reply**    strain  strain2  mais  It does not reply. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x10** | **SET\_MAIS\_RESOLUTION (or SET\_ RESOLUTION)**  **In brief**  Used to set the resolution of the acquired ADC samples by the mais board. If the board does not receive this command, the default resolution would be 8 bits.  **Parsed by**  mais  **Format of PAYLOAD**  0x10  RES   * sizeof(ARG) = 1. * RES contains one of the following values: enum class MaisRes { eightbits = 2 }.   **Actions on reception**  mais  It memorises the resolution of hall effect sensor.  **Reply**    strain  strain2  mais  It does not reply.  **Note**  Previous resolution modes with 16 bits (RES = 0) and debug mode (RES = 1) are deprecated. They would stream messages with TYP = 12, 13, 14, and 15 in former case and with TYP = 12, 13 in the latter case. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x18** | **GET\_FULLSCALE**  **In brief**  Used to retrieve the fullscale stored in the FT settings of strain / strain2 boards.  **Parsed by**    strain  strain2  **Format of PAYLOAD**  0x18  RS  CH   * sizeof(ARG) = 1. * RS is a nibble that contains the target regulation set. Its usual value should be 0 which means the regulation set currently in use. If we have a strain2 and we want to query one of its three regulation sets, we can also use 1, 2, or 3. * CH is a nibble that contains the target channel expressed as 0, 1, 2, 3, 4, or 5.   **Actions on reception**    strain  strain2  It sends a reply with the wanted fullscale.  **Reply**    strain  strain2  It replies to the sender with the same CMD and following format.    0x18  RS  CH  FS   * sizeof(ARG) = 3 * RS and CH are the same as those in the received message. * FS contains the value of the fullscale as a std::uint16\_t in **BIG** **ENDIAN** (sic!). |

|  |  |
| --- | --- |
| CMD | Description |
| **0x0C** | **GET\_ADC**  **In brief**  Used to retrieve the value of the channel, either in its ADC or in calibrated value.  **Parsed by**    strain  strain2  mais  **Format of PAYLOAD**  0x0C  CH  MODE   * sizeof(ARG) = 2 * CH is a nibble that contains the target channel. For strain and strain2 its value is from enum class Channel { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, all = 0xf } with exclusion of Channel::all, apart from strain2 which accepts this value staring from application version 2.08. For mais its value is in range [0, 14]. * MODE is a std::uint8\_t which tells if the value is the raw ADC (=0) or calibrated (=1). The board mais does not use MODE.   **Actions on reception**    strain  strain2  mais  It sends back a reply with the value.  **Reply**    strain  It replies to the sender with a single reply message with the same CMD and following format.    0x0C  CH  MODE  VALUE   * sizeof(ARG) = 4 * CH and MODE are the same as in the received frame. * VALUE contains the value of the acquisition as a std::uint16\_t in **BIG** **ENDIAN** (sic!).     strain2  It replies to the sender with one or six reply messages with same CMD and following format. The number of replies is six only if the received CH has value equal Channel::all and application version of the strain2 is >= 2.0.8.    0x0C  CH  MODE  VALUE   * sizeof(ARG) = 4 * CH and MODE are the same as in the received frame. * VALUE contains the value of the acquisition as a std::uint16\_t in **BIG** **ENDIAN** (sic!).   mais  It replies to the sender with a single reply message with same CMD and following format.  0x0C  CH  VALUE   * sizeof(ARG) = 3 * CH is the same as in the received frame. * VALUE contains the value of the acquisition as a std::uint16\_t in **BIG** **ENDIAN** (sic!). |

|  |  |
| --- | --- |
| CMD | Description |
| **0x07** | **SET\_TXMODE**  **In brief**  Used to start transmission of some services in sensor boards: FT data in strain / strain2 boards, hall effect sensor data in mais board, skin in mtb / mtb4 boards.  **Parsed by**    strain  strain2  mais  mtb  mtb4  **Format of PAYLOAD**  0x07  MODE   * sizeof(ARG) = 1. * MODE uses different values in mais and strain/strain2 boards: * strain/strain2: the values are from enum class StrainMode { txCalibrated = 0, txNone = 1, txUncalibrated = 3, txAll = 4 }. * mais: the values are from enum class MaisMode { txHallEffectData = 0, txNone = 1 }. * mtb / mtb4: the values are from enum class SkinMode { txData = 0, txNone = 1 }.   **Actions on reception**    strain  strain2  If ARG.MODE is StrainMode::txCalibrated it starts emitting messages of the class periodic analog sensors with TYPs FORCE\_VECTOR and TORQUE\_VECTOR containing calibrated data.  If ARG.MODE is StrainMode::txUncalibrated it starts emitting messages of the class periodic analog sensors with TYPs FORCE\_VECTOR and TORQUE\_VECTOR containing uncalibrated data.  If ARG.MODE is StrainMode::txAll it starts emitting messages of the class periodic analog sensors with TYPs FORCE\_VECTOR and TORQUE\_VECTOR containing calibrated data and with TYPs UNCALIBFORCE\_VECTOR\_DEBUGMODE and UNCALIBTORQUE\_VECTOR\_DEBUGMODE with uncalibrated data.  If ARG.MODE is StrainMode::txNone it stops any transmission.  mais  If ARG.MODE is MaisMode:: txHallEffectData it starts emitting messages of the class periodic analog sensors with TYPs HES0TO6 and HES7TO14.  If ARG.MODE is MaisMode::txNone it stops any transmission.    mtb  mtb4  If ARG.MODE is SkinMode:: txData it starts emitting messages of the class periodic skin with TYPs TRG0 up to TRG15, depending on which triangles are configured.  If ARG.MODE is SkinMode::txNone it stops any transmission.  **Reply**    strain  strain2  mais  It does not reply. |

### Messages for managing the IMU service

In here are messages that mtb4 and strain2 boards use to configure rate to and start/stop the IMU service.

|  |  |
| --- | --- |
| CMD | Description |
| **0x34** | **IMU\_CONFIG\_SET**  **In brief**  Used to impose the configuration of the IMU.  **Parsed by**    mtb4  strain2  **Format of PAYLOAD**    0x34  MASK  MODE  ORIENT  FFU08  FFU16   * sizeof(ARG) = 7 * MASK contains a bitmask of the configured sensors inside the IMU expressed as a std::uint16\_t. Supported sensors are from enum class imuSensor { acc = 0, mag = 1, gyr = 2, eul = 3, qua = 4, lia = 5, grv = 6, status = 15, none = 16 }. * MODE is 1 byte long and contains the computation mode to be used by the IMU.   *IN SHORT*. MODE = 0x01 imposes default data fusion, MODE = 0x00 default NO data fusion.  *IN MORE DETAILS*.  MODE is split into two groups: T (type) of 1 bit and PAR (parameter) of 7 bits.    T  PAR  Possible values for MODE.T are TYPE::basic = 0 and TYPE::advanced = 1.  If MODE.T is TYPE::basic, then MODE.PARAM can assume only the values 0 or 1 to mean the use of data fusion (=1) from the accelerometer, magnetometer and gyroscope or the use of raw data (=0).  If MODE.T is TYPE::advance, then MODE.PARAM contains a mode which is specific of the used IMU chip. Details for the case of the boards w/ the Bosch BNO055 chip are reported in the specific section.   * ORIENT is 1 byte long and allows to configure the orientations of the axis of the IMU.   *IN SHORT*. ORIENT = 0x00 imposes the default orientation of the IMU chip.  *IN MORE DETAILS*.  ORIENT is split into two groups: T (type) of 1 bit and PAR (parameter) of 7 bits.    T  PAR  Possible values for MODE.T are TYPE::factorydefault = 0 and TYPE::advanced = 1.  If ORIENT.T is Type::factorydefault, then the IMU keeps its standard orientation (see datasheet of the used chip for details). In this case ORIENT.PAR is discarded.  Else if ORIENT.T TYPE::advanced, then the IMU chip uses what inside ORIENT.PAR to change its orientation.  Examples for the case of the boards w/ the Bosch BNO055 chip are reported in the specific section.   * FFU08 and FFU16 are for future use.   **Actions on reception**    mtb4  strain2  rfe  It loads the received configuration.  **Reply**    mtb4  strain2  rfe  It does not reply. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x33** | **IMU\_CONFIG\_GET**  **In brief**  Used to retrieve the configuration of the IMU.  **Parsed by**    mtb4  strain2  **Format of PAYLOAD**  0x33   * sizeof(ARG) = 0.   **Actions on reception**    mtb4  strain2  It sends a reply with the wanted configuration.  **Reply**    mtb4  strain2  It replies to the sender with the same CMD and following format.    0x33  MASK  MODE  ORIENT  FFU08  FFU16   * sizeof(ARG) = 7 * MASK, MODE, ORIENT, FF08 and FF16 are the same as for IMU\_CONFIG\_SET |

|  |  |
| --- | --- |
| CMD | Description |
| **0x35** | **IMU\_TRANSMIT**  **In brief**  Starts the IMU service at a specified rate or it stops it.  **Parsed by**    mtb4  strain2  **Format of PAYLOAD**  0x35  RATE   * sizeof(ARG) = 1 * RATE contains the transmission rate expressed as a std::uint8\_t in milli-seconds.   **Actions on reception**    mtb4  strain2  If ARG.RATE is > 0 it starts transmission of the relevant message of class periodic inertial (IMU\_TRIPLE, IMU\_QUATERNION, or IMU\_STATUS), else if = 0 it stops it.  **Reply**    mtb4  strain2  It does not reply. |

### Messages for managing the THERMO service

In here are messages that the mtb4 and strain2 boards use to configure rate and to start/stop the THERMO service.

|  |  |
| --- | --- |
| CMD | Description |
| **0x39** | **THERMOMETER\_CONFIG\_SET**  **In brief**  Used to impose the configuration of the THERMO service.  **Parsed by**    mtb4  strain2  **Format of PAYLOAD**    0x39  MASK   * sizeof(ARG) = 1 * MASK contains a bitmask of the configured temperature sensors expressed as a std::uint8\_t. So far the only possible value is 0x01 because the only supported sensor is from enum class Thermo { one = 0, none = 8 }.   **Actions on reception**    mtb4  strain2  It loads the received configuration.  **Reply**    mtb4  strain2  It does not reply. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x38** | **THERMOMETER\_CONFIG\_GET**  **In brief**  Used to retrieve the configuration of the THERMO.  **Parsed by**    mtb4  strain2  **Format of PAYLOAD**  0x38   * sizeof(ARG) = 0.   **Actions on reception**    mtb4  strain2  It sends a reply with the wanted configuration.  **Reply**    mtb4  strain2  It replies to the sender with the same CMD and following format.    0x38  MASK   * sizeof(ARG) = 1 * MASK contains a bitmask of the configured temperature sensors expressed as a std::uint8\_t. So far the only possible value is 0x01 because the only supported sensor is from enum class Thermo { one = 0, none = 8 }. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x3A** | **THERMOMETER\_TRANSMIT**  **In brief**  Starts the THERMO service at a specified rate or it stops it.  **Parsed by**    mtb4  strain2  **Format of PAYLOAD**  0x3A  RATE   * sizeof(ARG) = 1 * RATE contains the transmission rate expressed as a std::uint8\_t in seconds.   **Actions on reception**    mtb4  strain2  If ARG.RATE is > 0 it starts transmission of the message of class periodic analog and TYP = THERMOMETER\_MEASURE, else if = 0 it stops it.  **Reply**    mtb4  strain2  It does not reply. |

### Messages for managing the Skin service

In here are messages that the mtb, mtb4 and psc boards use to configure the Skin service.

They instead start / stop the service with the same message **SET\_TXMODE** also used for start/stop of the HES and FT services.

|  |  |
| --- | --- |
| CMD | Description |
| **0x4D** | **SKIN\_SET\_BRD\_CFG**  **In brief**  Used to configure the acquisition of skin data in mtb/mtb4 boards. It contains settings which are general to every triangle.  **Parsed by**    mtb  mtb4  **Format of PAYLOAD**  0x4D  COMP  RATE  NOLD   * sizeof(ARG) = 3 * COMP is the compensation mode for the skin. Its value can be enum class SkinComp { withTemperatureCompensation = 0, palmFingerTip = 1, withoutTempCompensation = 2, testmodeRAW = 7, withTemperatureCompensationV2 = 8 }. * RATE is the period of transmission in milli-seconds. * NOLD is the value of the no-load that the board uses. Typical value is: 245.   **Actions on reception**    mtb  mtb4  It applies the received values.  **Reply**    mtb  mtb4  It does not send a reply. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x50** | **SET\_TRIANG\_CFG**  **In brief**  Used to configure the acquisition of skin data in mtb/mtb4 boards. It operates on skin triangles.  **Parsed by**    mtb  mtb4  **Format of PAYLOAD**  0x50  FIRST  LAST  SHIFT  ENA  CDCOFFSET   * sizeof(ARG) = 6 * FIRST is number of the first triangle to which apply the values. Valid values are: [0, 15]. * LAST is number of the last triangle to which apply the values. Valid values are: [FIRST, 15]. * SHIFT is the shift value applied to the triangles. Default value is 2. * ENA = 1 enables the triangles, else if = 0 it disables them. * CDCOFFSET is the value of the cdcoffset to be applied for the triangles. It is a std::uint16\_t value in little endian. Default value is: 0x2200.   **Actions on reception**    mtb  mtb4  It applies the received values.  **Reply**    mtb  mtb4  It does not send a reply. |

### Messages for managing the accel / gyro in mtb boards

In board mtb the only mode to configure the accelerometer or gyroscope is to use the command ACC\_GYRO\_SETUP. This command is also supported by mtb4 and strain2 which both mount a full IMU, however its use is discouraged in favor of the IMU\_\* commands.

|  |  |
| --- | --- |
| CMD | Description |
| **0x4F** | **ACC\_GYRO\_SETUP**  **In brief**  Starts the accelerometer and/or gyroscope on mtb boards. The mtb4 and strain2 boards support this command for legacy purposes only.  **Parsed by**    mtb  mtb4  strain2  **Format of PAYLOAD**  0x4F  MASK  RATE   * sizeof(ARG) = 2 * MASK contains a bitmask with the sensors to transmit. The valid bits are: enum class InertialTypeBit { internaldigitalaccelerometer = 1, externaldigitalgyroscope = 2, externaldigitalaccelerometer = 3 }. * RATE contains the transmission rate expressed as a std::uint8\_t in milli-seconds.   **Actions on reception**    mtb  mtb4  strain2  If ARG.RATE is > 0 and if mask is not 0, it starts transmission of the relevant message of class periodic inertial (DIGITAL\_ACCELEROMETER, DIGITAL\_GYROSCOPE), else it stops transmission.  **Reply**    mtb  mtb4  strain2  It does not reply.  **Notes**  In the mtb board, the two accelerometers (internal and external) cannot be used at the same time, if for an error both are selected the internal one is used.  The mtb4 and strain2 board will use the accelerometer / gyroscope inside their IMU if at least one of the relevant values of InertialTypeBit is set. |

### Messages for advanced configuration of the FT service

This section contains messages that the strain/strain2 board use for advance management of the FT service.

The messages use some enumerative types that describe the range of some variables. Here are variables, their associated types.

* Regulation set. Its value can be one of the following enum class regSet { theoneinuse = 0, first = 1, second = 2, third = 3 }.
* Validity of the regulation set. Its value can be one of the following enum class regSetValidity { temporary = 0, permanent = 1 }.
* Channel. Its value can be one of the following enum class Channel { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, all = 0xf }.

For details on the meaning of those variables, please refer to the relevant section later in the document.

#### Backwards compatibility of messages that contain a regulation set

The strain board does not have the concept of multiple regulation sets. Hence, nothing in the CAN frames that specifies it.

When at a later time, we developed the new strain2 boards which support multiple regulation sets, we needed to add a field RS with the regulation set onto which to perform a SET<> or an ORDER<> operation. But at the same time we wanted to keep unchanged the messages for strain 2 boards.

Hence, we decided to place RS inside bits that in the messages for strain board are always zero and to use for it the values of regSet { theoneinuse = 0, first = 1, second = 2, third = 3 }.

In strain boards, we always use RS = regSet::theoneinused, which is 0 and does not change the content of the message that the strain is able to parse.

In strain2 boards, we can work in the same ways and use RS = regSet::theoneinused. The message will operate on the regulation set that the board loaded at bootstrap or imposed by a previous message.

Bu we can also operate directly on a specific regulation set with regSet::one, regSet::two, and regSet::three.

#### Messages that select/query a regulation set

The strain2 boards support three regulation sets in their permanent storage (EEPROM). It is possible to impose the use of one over the other, so that the board can be more versatile versus several kind of scenarios.

The strain board does not support this message because it does not have multiple regulation sets.

|  |  |
| --- | --- |
| CMD | Description |
| **0x3D** | **REGULATIONSET\_SET**  **In brief**  Used to impose the regulation set of the strain/strain2 board.  **Parsed by**  strain2  **Format of PAYLOAD**  0x3D  RS  VA   * sizeof(ARG) = 1 * RS is a nibble that contains the target regulation set. Its value is from enum class regSet {} limited to regSet::first (= 1), regSet::second (= 2), and regSet::third (= 3). * VA is a nibble that contains the validity of the regulation set. Its values is from enum class regSetValidity { temporary = 0, permanent = 1 }.   **Actions on reception**  strain2  It applies the received ARG.RS value. If ARG.VA is regSetValidity::temporary, the board loads the regulation set specified by ARG.RS and uses it until a new command arrives or the board restarts. If ARG.VA is regSetValidity::permanent, the board saves the regulation set specified by ARG.RS in RAM. It will save it in EEPROM only after it receives a message SAVE2EE so that it can use it at the next bootstrap.  **Reply**  strain2  It does not send a reply. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x3E** | **REGULATIONSET\_GET**  **In brief**  Used to retrieve the regulation set of the strain2 board.  **Parsed by**  strain2  **Format of PAYLOAD**  0x3E  0x0  VA   * sizeof(ARG) = 1 * VA is a nibble that contains the validity of the requested regulation set. Its values can be regSetvalidity { temporary = 0, permanent = 1 }.   **Actions on reception**  strain2  It sends a reply with the requested regulation set value.  **Reply**  strain2  It replies to the sender with the same CMD and following format.  0x3E  RS  VA   * sizeof(ARG) = 1 * RS is a nibble that contains the requested regulation set. Its value can be regSet::first (= 1), regSet::second (= 2), or regSet ::third (= 3). In case of VA is regSetvalidity::temporary it sends the value of the regulation set effectively in use at the time of the request, else it sends back the value of the regulation set loaded at bootstrap |

#### Messages that operate on the content of a given regulation set

In here are messages that initialize and maintain the regulation sets of the strain / strain2 board.

They are backwards compatible with strain boards because the RS field uses the most significant nibble inside a byte that strain boards use for values = [0, 1, 2, 3, 4, 5, 5] as row, column, channel.

|  |  |
| --- | --- |
| CMD | Description |
| **0x03** | **SET\_MATRIX\_RC**  **In brief**  Used to change a value of the transformation matrix stored the regulation set of strain / strain2 boards. The matrix contains Q15 values.  **Parsed by**    strain  strain2  **Format of PAYLOAD**  0x03  RS  R  Z  C  VALUE   * sizeof(ARG) = 4. * RS is a nibble that contains the target regulation set. Its value is from enum class regSet{}. For strain boards it must be regSet::theoneinuse (=0). * R is the row of the matrix. It has values belonging to enum class Channel { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, all = 0xf }, with exclusion of Channel::all. * Z is always 0. * C is the column of the matrix. . It has values belonging to enum class Channel { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, all = 0xf }, with exclusion of Channel::all. * VALUE is the value of the matrix stored as a Q15 in **BIG** **ENDIAN** (sic!).   **Actions on reception**    strain  strain2  It assigns VALUE in position (R, C) of the transformation matrix of regulation set specified by RS.  **Reply**    strain  strain2  It does not reply.  **Note**  The value will be stored in permanent storage only when the board receives a message of type SAVE2EE. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x0A** | **GET\_MATRIX\_RC**  **In brief**  Used to retrieve a value of the transform matrix stored the regulation set of strain / strain2 boards.  **Parsed by**    strain  strain2  **Format of PAYLOAD**  0x0A  RS  R  Z  C   * sizeof(ARG) = 2. * RS is a nibble that contains the target regulation set. Its value is from enum class regSet{}. For strain boards it must be regSet::theoneinuse (=0). * R is the row of the matrix. It has values belonging to enum class Channel { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, all = 0xf }, with exclusion of Channel::all * Z is always 0. * C is the column of the matrix. It has values belonging to enum class Channel { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, all = 0xf }, with exclusion of Channel::all   **Actions on reception**    strain  strain2  It sends back the value.  **Reply**    strain  strain2  It replies to the sender with the same CMD and following format.  0x0A  RS  R  Z  C  VALUE   * sizeof(ARG) = 4 * RS, R, Z and C are the same as those in the received message. * VALUE is the value of the matrix stored as a Q15 value in **BIG** **ENDIAN** (sic!). |

|  |  |
| --- | --- |
| CMD | Description |
| **0x17** | **SET\_FULLSCALE**  **In brief**  Used to impose the fullscale of the regulation set of strain / strain2 boards.  **Parsed by**    strain  strain2  **Format of PAYLOAD**    0x17  RS  CH  FS   * sizeof(ARG) = 3 * RS is a nibble that contains the target regulation set. Its value is from enum class regSet{}. For strain boards it must be regSet::theoneinuse (=0). * CH is a nibble that contains the target channel. . It has values belonging to enum class Channel { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, all = 0xf }, with exclusion of Channel::all. * FS contains the value of the fullscale as a std::uint16\_t in **BIG** **ENDIAN** (sic!).   **Actions on reception**    strain  strain2  It stores the received fullscale.  **Reply**    strain  strain2  It does not send a reply.  **Note**  The value will be stored in permanent storage only when the board receives a message of type SAVE2EE. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x18** | **SET\_SERIAL\_NUMBER**  **In brief**  Used to impose the serial number that specifies the regulation set of the given strain / strain2 board.  **Parsed by**    strain  strain2  **Format of PAYLOAD**    0x18  SN   * sizeof(ARG) = sizeof(SN) * SN contains the raw string of the serial number. Its length is up to 7. Example: SN = { “SN123” } has length = 5.   **Actions on reception**  strain  strain2  It stores the received value .  **Reply**    strain  strain2  It does not send a reply.  **Note**  The value will not be stored in permanent storage until the board receives a message of type SAVE2EE. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x1A** | **GET\_SERIAL\_NUMBER**  **In brief**  Used to retrieve the serial number that specifies the regulation set of the given strain / strain2 board.  **Parsed by**    strain  strain2  **Format of PAYLOAD**  0x1A   * sizeof(ARG) = 0.   **Actions on reception**    strain  strain2  It sends a reply with the wanted serial number in form of a string.  **Reply**    strain  strain2  It replies to the sender with the same CMD and following format.    0x1A  SN   * sizeof(ARG) = sizeof(SN) * SN contains the raw string of the serial number. Its length is up to 7. Example: SN = { “SN123” } has length = 5. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x04** | **SET\_CH\_DAC**  **In brief**  Used to impose an offset in the analog front end of strain/strain2 boards, so that the acquired ADC values are shifted down or up in value.  **Parsed by**    strain  strain2  **Format of PAYLOAD**    0x04  RS  CH  OFFSET   * sizeof(ARG) = 3 * RS is a nibble that contains the target regulation set. Its value is from enum class regSet{}. For strain boards it must be regSet::theoneinuse (=0). * CH is a nibble that contains the target channel. It has values belonging to enum class Channel { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, all = 0xf }, with exclusion of Channel::all. * OFFSET contains the value of the offset as a std::uint16\_t in **BIG** **ENDIAN** (sic!).   **Actions on reception**  strain  It stored the received OFFSET and applies as a value of a DAC which moves up or down the analog value of the strain-gauge.  strain2  It stores the received offset and programs the registers of the PGA (programmable amplifier) at the analog front end so that its offset is equal to OFFSET.  **Reply**    strain  strain2  It does not send a reply.  **Note**  The value will not be stored in permanent storage until the board receives a message of type SAVE2EE. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x0B** | **GET\_CH\_DAC**  **In brief**  Used to retrieve the offset in the analog front end of strain/strain2 boards.  **Parsed by**    strain  strain2  **Format of PAYLOAD**  0x0B  RS  CH   * sizeof(ARG) = 1 * RS is a nibble that contains the target regulation set. Its value is from enum class regSet{}. For strain boards it must be regSet::theoneinuse (=0). * CH is a nibble that contains the target channel. It has values belonging to enum class Channel { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, all = 0xf }, with exclusion of Channel::all.   **Actions on reception**    strain  strain2  It sends back a reply with the offset.  **Reply**    strain  strain2  It replies to the sender with the same CMD and following format.    0x0B  RS  CH  OFFSET   * sizeof(ARG) = 3 * RS and CH as in the received message. * FS contains the value of the offset as a std::uint16\_t in **BIG** **ENDIAN** (sic!). |

|  |  |
| --- | --- |
| CMD | Description |
| **0x13** | **SET\_CALIB\_TARE**  **In brief**  Used to impose the tare (digital offset) of the regulation set of strain / strain2 boards.  **Parsed by**    strain  strain2  **Format of PAYLOAD**    0x13  MODE  RS  CH  TARE   * sizeof(ARG) = 2 or 4 * MODE tells how to change the tare. Possible values are: enum class Mode { everychannelreset = 0, everychannelnegativeofadc = 1, setchannelwithvalue = 2 }. * RS is a nibble that contains the target regulation set. Its value is from enum class regSet{}. For strain boards it must be regSet::theoneinuse (=0). * CH is a nibble that contains the target channel. It has values belonging to enum class Channel { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, all = 0xf }, with exclusion of Channel::all. This field is used only if MODE is Mode::setchannelwithvalue. * TARE contains the value of the tare expressed in Q15 format in **BIG** **ENDIAN** (sic!). This field is present only if MODE is Mode::setchannelwithvalue.   **Actions on reception**    strain  strain2  It modified the value of the tare in the digital regulation set specified by ARG.RS in a way that depends on the value of ARG.MODE.   * If = Mode::everychannelreset, the values of tare for all the six channels are imposed = 0. * If = Mode:: everychannelnegativeofadc, the values of tare for all the six channels are imposed = negative value of the ADC in Q15 format, so that U16toQ15(ADC[ch]) + TARE[ch] = 0 for every channel ch. * If = Mode::setchannelwithvalue, the value of channel specified by ARG.CH is assigned with ARG.TARE.   **Reply**    strain  strain2  It does not send a reply.  **Note**  The value will not be stored in permanent storage until the board receives a message of type SAVE2EE. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x14** | **GET\_CALIB\_TARE**  **In brief**  Used to retrieve a value of the calibration tare stored the regulation set of strain / strain2 boards.  **Parsed by**    strain  strain2  **Format of PAYLOAD**  0x14  RS  CH   * sizeof(ARG) = 1. * RS is a nibble that contains the target regulation set. Its value is from enum class regSet{}. For strain boards it must be regSet::theoneinuse (=0). * CH is a nibble that contains the target channel. It has values belonging to enum class Channel { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, all = 0xf }, with exclusion of Channel::all.   **Actions on reception**    strain  strain2  It sends back the value.  **Reply**    strain  strain2  It replies to the sender with the same CMD and following format.  0x14  RS  CH  VALUE   * sizeof(ARG) = 3 * RS and CH are the same as those in the received message. * VALUE is the value of the tare stored as a Q15 value in **BIG** **ENDIAN** (sic!). |

|  |  |
| --- | --- |
| CMD | Description |
| **0x1B** | **GET\_EEPROM\_STATUS**  **In brief**  Used in strain/strain2 boards to know if the RAM copy of the FT settings is equal to the one in EEPROM.  **Parsed by**    strain  strain2  **Format of PAYLOAD**  0x1B   * sizeof(ARG) = 0.   **Actions on reception**    strain  strain2  It sends back the required info.  **Reply**    strain  strain2  It replies to the sender with the same CMD and following format.  0x1B  EQUAL   * sizeof(ARG) = 1 * EQUAL is 1 if the EEPROM copy is equal to the RAM copy. It is 0 if any value of the FT settings was changed by a relevant SET\_ command without having saved that with the SAVE2EE command. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x09** | **SAVE2EE**  **In brief**  Used in strain/strain2 boards to save in permanent storage all RAM data belonging to the FT settings.  **Parsed by**    strain  strain2  **Format of PAYLOAD**  0x09   * sizeof(ARG) = 0.   **Actions on reception**    strain  strain2  It saves to EEPROM (permanent storage) all the changed data in the FT settings. Then it sends back a reply.  **Reply**    strain  strain2  It replies to the sender with the same CMD and following format.  0x09  OKKO   * sizeof(ARG) = 1 * OKKO is 1 if operation was successful, 0 if not. |

#### Messages for a generic amplifier

These messages offer agnostic configuration of the gain and offset of the amplifier.

However, the board may not support any range of offset or gain because of HW limitation or simplified software implementation. Hence, please use the suitable message to query the supported range before imposing values.

|  |  |
| --- | --- |
| CMD | Description |
| **0x1D** | **AMPLIFIER\_RESET**  **In brief**  Used to impose the default configuration of the amplifier.  **Parsed by**  strain2  **Format of PAYLOAD**  0x1D  RS  CH   * sizeof(ARG) = 1 * RS is a nibble that contains the target regulation set. Its value is from enum class regSet { theoneinuse = 0, first = 1, second = 2, third = 3 }. * CH is a nibble that contains the target channel. It has values belonging to enum class Channel { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, all = 0xf }.   **Actions on reception**  strain2  It forces the default configuration for the specified channel and target regulation set. In case of StrainChannel::all, the command is effective for every channel. In case of StrainRegulationSet::theoneinuse, then it also apply the content of the configuration to the amplifier of the target channel.  **Reply**  strain2  It does not reply.  **Notes**  The values will not be stored in permanent storage until the board receives a message of type SAVE2EE.  The default configuration for the case of the PGA308 is GDAC = 0x4000, ZDAC = 0x8000, CFG0.GO = 0x06, CFG0.GI = 0x04, CFG0.OS = 0x20, CFG0.GI4 = 0. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x1E** | **AMPLIFIER\_RANGE\_OF\_GAIN\_GET**  **In brief**  Used to ask the range of the gain supported by the amplifier.  **Parsed by**  strain2  **Format of PAYLOAD**  0x1E  RS  CH   * sizeof(ARG) = 1 * RS is a nibble that contains the target regulation set. Its value is from enum class regSet { theoneinuse = 0, first = 1, second = 2, third = 3 }. * CH is a nibble that contains the target channel. It has values belonging to enum class Channel { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, all = 0xf } with exclusion of Channel::all.   **Actions on reception**  strain2  It sends back a reply with the limits.  **Reply**  strain2  It replies to the sender with the same CMD and following format.    0x1E  RS  CH  LOWEST  HIGHEST   * sizeof(ARG) = 5 * RS and CH as in the received message. * LOWEST contains the lowest possible value of the gain as a std::uint16\_t in little endian, where each unit express a gain of 0.01 (1/100). * HIGHEST contains the highest possible value of the gain as a std::uint16\_t in little endian, where each unit express a gain of 0.01 (1/100). |

|  |  |
| --- | --- |
| CMD | Description |
| **0x1F** | **AMPLIFIER\_RANGE\_OF\_OFFSET\_GET**  **In brief**  Used to ask the range of the offset supported by the amplifier.  **Parsed by**  strain2  **Format of PAYLOAD**  0x1F  RS  CH   * sizeof(ARG) = 1 * RS is a nibble that contains the target regulation set. Its value is from enum class regSet { theoneinuse = 0, first = 1, second = 2, third = 3 }. * CH is a nibble that contains the target channel. It has values belonging to enum class Channel { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, all = 0xf } with exclusion of Channel::all.   **Actions on reception**  strain2  It sends back a reply with the limits.  **Reply**  strain2  It replies to the sender with the same CMD and following format.    0x1E  RS  CH  LOWEST  HIGHEST   * sizeof(ARG) = 5 * RS and CH as in the received message. * LOWEST contains the lowest possible value of the offset as a std::uint16\_t in little endian. * HIGHEST contains the highest possible value of the offset as a std::uint16\_t in little endian. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x20** | **AMPLIFIER\_GAINOFFSET\_GET**  **In brief**  Used to retrieve the pair gain-offset from an amplifier.  **Parsed by**  strain2  **Format of PAYLOAD**  0x20  RS  CH   * sizeof(ARG) = 1 * RS is a nibble that contains the target regulation set. Its value is from enum class regSet { theoneinuse = 0, first = 1, second = 2, third = 3 }. * CH is a nibble that contains the target channel. It has values belonging to enum class Channel { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, all = 0xf } with exclusion of Channel::all.   **Actions on reception**  strain2  It sends back a reply with the pair gain-offset.  **Reply**  strain2  It replies to the sender with the same CMD and following format.    0x20  RS  CH  GAIN  OFFSET   * sizeof(ARG) = 5 * RS and CH as in the received message. * GAIN contains the value of the gain as a std::uint16\_t in little endian, where each unit express a gain of 0.01 (1/100). * OFFSET contains the value of the offset as a std::uint16\_t in little endian. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x21** | **AMPLIFIER\_GAINOFFSET\_SET**  **In brief**  Used to impose the pair gain-offset to an amplifier.  **Parsed by**  strain2  **Format of PAYLOAD**    0x21  RS  CH  MODE  GAIN  OFFSET   * sizeof(ARG) = 6 * RS is a nibble that contains the target regulation set. Its value is from enum class regSet { theoneinuse = 0, first = 1, second = 2, third = 3 }. * CH is a nibble that contains the target channel. It has values belonging to enum class Channel { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, all = 0xf } with exclusion of Channel::all. * MODE is a byte with possible values 0 or 1. * GAIN contains the value of the gain as a std::uint16\_t in little endian, where each unit express a gain of 0.01 (1/100). * OFFSET contains the value of the offset as a std::uint16\_t in little endian.   **Actions on reception**  strain2  If MODE is 0, it attempts to apply the values of GAIN and OFFSET to the amplifier. The operation will be successful as long as they belong to the allowed limits retrieved with commands AMPLIFIER\_RANGE\_OF\_GAIN\_GET and AMPLIFIER\_RANGE\_OF\_OFFSET\_GET. Else, if MODE is 1, the amplifier will be applied its default values.  **Reply**  strain2  It does not reply. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x22** | **AMPLIFIER\_OFFSET\_AUTOCALIB**  **In brief**  Used to impose the pair gain-offset to an amplifier.  **Parsed by**  strain2  **Format of PAYLOAD**    0x22  RS  CH  MODE  TARGET  TOLERANCE  NSAM   * sizeof(ARG) = 7 * RS is a nibble that can contains only the value regSet::theoneinuse (= 0) because the command must start an acquisition procedure. * CH is a nibble that contains the target channel. It has values belonging to enum class Channel { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, all = 0xf }. It can be also Channel::all. * MODE is a byte with only one possible value (so far) = 0. It is a placeholder to future different autotune algorithms. * TARGET contains the target ADC value as a std::uint16\_t in little endian. Half scale is 32K. * TOLERANCE is the value of tolerance expressed as a std::uint16\_t in little endian. An high value makes the result more permissive. * NSAM is the number of ADC acquisition samples used to average the measure. If = 0, the board will use a default value (strain uses 4). A high value of NSAM requires more execution time but filters more noise away.   **Actions on reception**  strain2  The board start the autotune algorithm and returns a result.  **Reply**  strain2  It replies to the sender with the same CMD and following format.    0x22  RS  CH  CHMSK  ALGMSK  MEMSK  FFU  MAE   * sizeof(ARG) = 7 * RS and CH as in the received message. * CHMSK is a noisy channel mask. A bit in position x is 1 if the ADC acquisition on that channel is NOISY either in the first or second acquisition phase. It is noisy if the difference between the highest acquired value and the lowest is higher than the specified tolerance. The bit in position 7 is 1 if any noisy channel in the first acquisition phase. The bit in position 6 is 1 if any noisy channel in the second acquisition phase. * ALGMSK is a algorithm OK mask. A bit in position x is 1 if the algorithm is OK for channel x. * MEMSK is final measure mask. A bit in position x is 1 if the ADC acquisition of channel x after the change of offset is below tolerance. * FFU is for further use. * MAE is a std::uint16\_t in little endian which contains the MAE over all selected channels between the target and the final measure. |

#### Messages specific for the PGA308 amplifiers in strain2 boards

In here is description of the CAN messages that directly operates on the registers of the PGA308 amplifiers.

These messages offer full functionalities in programming the gain and offset of the amplifier. However, their use can be more difficult because one must compute exactly the effect of the values of the registers in the transfer function.

|  |  |
| --- | --- |
| CMD | Description |
| **0x2B** | **AMPLIFIER\_PGA308\_CFG1\_SET**  **In brief**  Used to load the configuration of the PGA308 at register-level.  **Parsed by**  strain2  **Format of PAYLOAD**    0x2B  RS  CH  GDAC  GI  I  GO  OS  ZDAC   * sizeof(ARG) = 7 * RS is a nibble that contains the target regulation set. Its value is from enum class regSet{ theoneinuse = 0, first = 1, second = 2, third = 3 }. * H is a nibble that contains the target channel. Its value is from enum class Channel { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, all = 0xf }. * GDAC is a std::uint16\_t in little endian format with the content of the GDAC register (Fine Gain Adjust). * GI is a nibble with the content of the CFG0.GI register (Front-End PGA Gain Select). * I is a single bit with the content of the CFG0.GI4 register (Front-End PGA Input Mux Control). * GO is formed by three bits and is the content of the CFG0.GO register (Output Amplifier Gain Select). * OS is a std::uint8\_t which with the content of the CFG0.OS register (Coarse Offset Adjust on Front-End PGA). * ZDAC is a std::uint16\_t in little endian format with the content of the ZDAC register (Fine Offset Adjust).   **Actions on reception**  strain2  It loads the received configuration for the specified channel and target regulation set. In case of Channel::all, the command is effective for every channel.  **Reply**  strain2  It does not reply.  **Note**  The values will not be stored in permanent storage until the board receives a message of type SAVE2EE. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x2A** | **AMPLIFIER\_PGA308\_CFG1\_GET**  **In brief**  Used to retrieve the configuration of the PGA308 at register level.  **Parsed by**  strain2  **Format of PAYLOAD**    0x2A  RS  CH   * sizeof(ARG) = 1 * RS is a nibble that contains the target regulation set. Its value is from enum class regSet { theoneinuse = 0, first = 1, second = 2, third = 3 }. * CH is a nibble that contains the target channel. Its value is from enum class Channel { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, all = 0xf } with exclusion of Channel::all.   **Actions on reception**  strain2  It sends a reply with the wanted configuration.  **Reply**  strain2  It replies to the sender with the same CMD and following format.    0x2A  RS  CH  GDAC  GI  I  GO  OS  ZDAC   * sizeof(ARG) = 7 * RS and CH are the same as the received message. * GDAC is a std::uint16\_t in little endian format with the content of the GDAC register (Fine Gain Adjust). * GI is a nibble with the content of the CFG0.GI register (Front-End PGA Gain Select). * I is a single bit with the content of the CFG0.GI4 register (Front-End PGA Input Mux Control). * GO is formed by three bits and is the content of the CFG0.GO register (Output Amplifier Gain Select). * OS is a std::uint8\_t which with the content of the CFG0.OS register (Coarse Offset Adjust on Front-End PGA). * ZDAC is a std::uint16\_t in little endian format with the content of the ZDAC register (Fine Offset Adjust). |

#### Other messages

Messages SET\_MATRIX\_G, GET\_MATRIX\_G are useless because neither strain nor strain2 use the gain for the matrix.

Messages SET\_CURR\_TARE and GET\_CURR\_TARE are used only for debug purposes. They have the same format as SET\_CALIB\_TARE and GET\_CALIB\_TARE and apply an extra offset after the matrix multiplication. However, such an offset is not stored in EEPROM.

### Messages for managing the POS service

In here are messages that the psc board uses to configure acquisition and calibration of positions of the POS service, as well as rate and start/stop of the service.

The protocol for these messages is in draft status.

|  |  |
| --- | --- |
| CMD | Description |
| **0x52** | **POS\_CONFIG\_SET**  **In brief**  Used to impose the configuration of the POS service.  **Parsed by**  ,  psc  mtb4fap  **Format of PAYLOAD**    0x52  ID  TY  SETTINGS   * sizeof(ARG) = 7 * ID is a nibble that contains the ID of the sensor that we want to configure. This nibble is not used by the psc board, for which it must be = 0x0. For the mtb4fap board valid values are from 0 up to 14. The value of 15 indicates *all the sensors*. * TY is a single byte that contains the type of sensors that we want to configure. For now, for the case of the psc board it is = 0x0. The mtb4fap board also use the value 14 to indicate that the sensor type is NONE and that the sensor specified by ID must be disabled * SETTINGS is a six-bytes field that contains the configuration of the sensors. For now, for the case of the psc board (but also of mtb4fap board), the field SETTINGS contains two DECIDEG-STTNG items, each of three bytes. Details of DECIDEG-STTNG is in the following.     DECIDEG-STTNG  DECIDEG-STTNG  **Actions on reception**  ,  psc  mtb4fap  It loads the received configuration.  **Reply**  ,  psc  mtb4fap  It does not reply.  NOTE  The psc board does not use the field ID but a unified field TYPE which is always = 0. |

|  |  |
| --- | --- |
|  | Format of DECIDEG-STT as used for the psc and mtb4fap board  The deci-degree is an angle expressed as an int16\_t value with resolution 0.1 degrees.  DECIDEG-STTNG  Its configuration DECIDEG-STTNG contains:   * A boolean which tells if the value is **enabled** or not. * A **label** that identifies it (we allow up to 16 labels). * Some calibration value such as **rotation** (of 0, +180, +90, -90 degrees) to move the measure away of a discontinuity, a **zero** reference (expressed as a deci-degree), a **direction** (expressed as a bool) to define increasing or decreasing values. These values are filled into three bytes.   E  D  R  LBL  ZERO   * size = 3 * E is one bit that tells if the configuration is enabled. If E is 0 then none of the remaining fields have meaning. * D is one bit that expresses the direction with respect to the raw acquisition from sensor. Use values from enum class DIR : uint8\_t { same = 0, inverted = 1 }. * R is two bits long and expresses rotation. Use values from enum class ROT : uint8\_t { none = 0, plus180 = 1, plus090 = 2, minus090 = 3 }. * LBL is a nibble that contains a label, which univocally identifies the measure. This label shall be used in the streaming message POS\_MEASURE. We allow 16 different labels so that we can express many measures. Use values in enum class posLABEL : uint8\_t { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, six = 6, seven = 7, eight = 8, nine = 9, ten = 10, eleven = 11, twelve = 12, thirteen = 13, fourteen = 14, fifteen = 15 }. * ZERO is two bytes long and contains a deci-degree expressed as a int16\_t in little endian ordering. It contains the value of the angle that we want to consider at zero degrees. We must acquire this value in raw mode with {E = true, D = DIR::same, R = ROT::none, LBL = any value, ZERO = 0}.   The transformation from the acquired value vraw is performed by the function ***transform***() as follows:  static const deciDeg rotationmap[4]; // = {0, 1800, 900, -900};  deciDeg rotateclip(deciDeg v) const { return ( (v+rotationmap[static\_cast<uint8\_t>(rotation)]) % 3600 ); }  deciDeg ***transform***(const deciDeg vraw) const  {  deciDeg z = rotateclip(vraw) - rotateclip(zero);  return (invertdirection) ? (-z) : (+z);  } |

|  |  |
| --- | --- |
|  | Typical values of DECIDEG-STTNG as used for the psc board  The **default** configuration at startup of the two sensors inside the psc board is: E = true, D = DIR::same, R = ROT::none, ZERO = 0. The first will have posLABEL::zero, the second posLABEL::one.    0x{80|00|00}  0x{81|00|00}  The configuration for the three psc boards used in the PSC service in the R1 hand should have proper labels to allow an easy identification of the six required values. If no angle compensation is required (D = DIR::same, R = ROT::none, ZERO = 0) one should use the following settings.    PSC @ CAN1:1  0x{80|00|00}  0x{81|00|00}    PSC @ CAN1:2  0x{82|00|00}  0x{83|00|00}    PSC @ CAN1:3  0x{84|00|00}  0x{85|00|00} |

|  |  |
| --- | --- |
| CMD | Description |
| **0x51** | **POS\_CONFIG\_GET**  **In brief**  Used to retrieve the configuration of the POS service.  **Parsed by**  psc  **Format of PAYLOAD**    0x51  TYPE   * sizeof(ARG) = 1. * TYPE is a single byte that contains the type of sensors that we want to ask for. For now, for the case of the psc board it is = 0x00.   **Actions on reception**  psc  It sends a reply with the wanted configuration.  **Reply**  psc  It replies to the sender with the same CMD and following format.    0x51  TYPE  SETTINGS   * sizeof(ARG) = 7 * TYPE and SETTINGS have the same format as in POS\_CONFIG\_SET. |

|  |  |
| --- | --- |
| CMD | Description |
| **0x53** | **POS\_TRANSMIT**  **In brief**  Starts the POS service at a specified rate or it stops it.  **Parsed by**  psc  **Format of PAYLOAD**  0x53  RATE   * sizeof(ARG) = 1 * RATE contains the transmission rate expressed as a std::uint8\_t in milli-seconds.   **Actions on reception**  psc  If ARG.RATE is > 0 it starts transmission of the message of class periodic analog and TYP = POS\_MEASURE, else if = 0 it stops it.  **Reply**  psc  It does not reply. |

The messages of the streaming classes

Here is description of the messages in each class.

## Class periodic motor control

The class periodic motor control (CLS = 001b) contains messages used by motor control boards to stream their status to the host or also to other motor control boards.

|  |  |
| --- | --- |
| TYP | Description |
| 0x0 | 2FOC |
| 0x1 | POSITION |
| 0x2 | PID\_VAL |
| 0x3 | STATUS |
| 0x4 | CURRENT |
| 0x5 | OVERFLOW |
| 0x6 | PRINT |
| 0x7 | VELOCITY |
| 0x8 | PID\_ERROR |
| 0x9 | DEBUG |
| 0xA | MOTOR\_POSITION |
| 0xB | MOTOR\_SPEED |
| 0xC | ADDITIONAL\_STATUS |
| 0xF | EMSTO2FOC\_DESIRED\_CURRENT |

**Table 8** – The TYPs of the periodic motor control class.

Full description of these messages is out of the scope of this document. In here, we shall describe only the messages also supported by sensor boards.

### Messages supported by all boards

|  |  |
| --- | --- |
| TYP | Description |
| **0x6** | **PRINT**  **In brief**  The board may emit a burst of such messages to transport a human readable string when it wants to communicate a status or a warning or an error. The string is long up to 30 characters.  **Emitted by**  application  The board starts a burst of messages when it wants to transmit a string of up to 30 characters.  **Format of DATA**    CODE  S  N  C0  C1  C2  C3  C4  C5   * sizeof(DATA) = [3, 8] * CODE is = 6 for all frames except for the last frame where it is = 128+8. * S is the identifier of the string inside the burst of frames. We have a different one for each string so that the receiver can assemble the string from the CAN frames. * N is the number of the frame inside the burst. Its value can be [0, 5]. * {C0, C1, C2, C3, C4, C5} are the characters of the string. The last frame may have only C0.   **Example**  For the sake of protocol description let char str30[31] = { ‘h’, ‘e’, ‘l’, ‘l’, ‘o’, ‘ ‘,‘w’, ‘o’, ‘r’, ‘l’, ‘d’, ‘!’, ‘!’, 0}; be the null-terminated string. In this case, there will be 13 characters, hence 3 CAN frames. In case the emitting board has ADDR = 3, and we assume an identifier of the string S = 0x7, then we have the following frames:  ID = {0x136}, DATA = {0x06, 0x70, ‘h’, ‘e’, ‘l’, ‘l’, ‘o’, ‘ ‘}  ID = {0x136}, DATA = {0x06, 0x71, ‘w’, ‘o’, ‘r’, ‘l’, ‘d’, ‘!’}  ID = {0x136}, DATA = {0x86, 0x72, ‘!’} |

### All the other messages

The description of every MC streaming message is out of the scope of this document.

## Class periodic analog sensors

The class periodic analog sensors (CLS = 011b) contains messages used by sensor boards to stream their data to the host.

|  |  |
| --- | --- |
| TYP | Description |
| 0x0 | USER\_DEF |
| 0x1 | POS\_MEASURE |
| 0x8 | UNCALIBFORCE\_VECTOR\_DEBUGMODE |
| 0x9 | UNCALIBTORQUE\_VECTOR\_DEBUGMODE |
| 0xA | FORCE\_VECTOR |
| 0xB | FORCE\_TORQUE |
| 0xC | HES0TO6 |
| 0xD | HES7TO14 |
| 0xE | THERMOMETER\_MEASURE |

**Table 9** – The TYPs of the periodic analog sensors class.

Here are details of their format.

### Messages which stream user-defined data

|  |  |
| --- | --- |
| TYP | Description |
| **0x0** | **USER\_DEF**  **In brief**  It transports data according to a user-defined agreement. It MUST NOT be implemented in iCub in presence of other boards, unless for debug purposes.  **Emitted by**  ?  Transmission is started at a given rate by a user-defined strategy.  **Format of DATA**  VALUES   * sizeof(DATA) = 8. * DATA is a space of 8 bytes that the user can fill as need. |
|  |  |

### Messages which stream POS data

|  |  |
| --- | --- |
| TYP | Description |
| **0x1** | **POS\_MEASURE**  **In brief**  It transports data from position sensors.  **Emitted by**  psc  Transmission is started at a given rate by a message of CLS = 0x2 (polling analog) and CMD = 0x53 (POS\_TRANSMIT) with ARG.RATE = rate (in [millisec]) and stopped with ARG.RATE = 0. One must first configure and enable the chosen sensors with a message of CLS = 0x2 (polling analog) and CMD = 0x52 (POS\_CONFIG\_SET).  **Format of DATA**    TYPE  LBL  N  VALUES   * sizeof(DATA) = 8. * TYPE is a byte with the type of data that the message transports. It has values in enum class posTYPE : uint8\_t { angleDeciDeg = 0, unknown = 255 }. For the psc board which transports angles in deci-degrees the values is always posTYPE::angleDeciDeg = 0. * LBL is a nibble used to identify the label of the first angle contained inside the message. It assumes a value from enum class posLABEL : uint8\_t { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, six = 6, seven = 7, eight = 8, nine = 9, ten = 10, eleven = 11, twelve = 12, thirteen = 13, fourteen = 14, fifteen = 15 }. * N is a nibble which tells how many values are contained inside the message. The first value will have label = LBL, the second (if present) will have label = LBL+1, etc. * VALUES is a six bytes space which holds the values transported by the message. The number of values is expressed by N and their format is expressed by TYPE.   + If TYPE is posTYPE::angleDeciDeg as in the case of the psc, then each value is a std::int16\_t in little endian at a resolution of 0.1 degrees. The psc board usually sends two values (N = 2).   With N = 2 only the first four bytes of VALUES will be meaningful.    VALUE-(LBL)  VALUE-(LBL+1)  0x0000 |

### Messages which stream FT data

|  |  |
| --- | --- |
| TYP | Description |
| **0x8** | **UNCALIBFORCE\_VECTOR\_DEBUGMODE**  **In brief**  It transports the raw ADC values of channels 0, 1 and 2. Used by the strain/strain2 in debug mode only.  **Emitted by**    strain  strain2  Transmission is started by a message of CLS = 0x2 (polling analog) and CMD = 0x07 (SET\_TXMODE) with ARG.MODE = strainMode:: txAll (= 4) and stopped with ARG.MODE = strainMode::txNone (= 1). The transmission rate is configured with a message of CLS = 0x2 (polling analog) and CMD = 0x08 (SET\_CANDATARATE) with ARG.PERIOD = period in [msec].  **Format of DATA**    X  Y  Z  C  2  C1  C0  S   * sizeof(DATA) = 6 or 7. * X is the first component expressed as a std::uint16\_t in little endian. * Y is the second component expressed as a std::uint16\_t in little endian. * Z is the third component expressed as a std::uint16\_t in little endian. * {C2, C1, C0, S} are an optional byte which is transmitted in case of a saturation in ADC acquisition of at least one of the six ADC channels. S has value 1 of any of the six channels saturated (if sizeof(DATA) is 7 S is always 1). Cx contains the kind of saturation in channel x: NONE = 0, atLOWscale = 1, atHIGHscale = 2.   **Note**  A raw ADC value has a range [0, 64K) and represents the output of an ADC of the same range. It is saturated at low scale if is in range [0, 1000). It is saturated at high scale if it is in range [64000, 64K) |

|  |  |
| --- | --- |
| TYP | Description |
| **0x9** | **UNCALIBFORCE\_TORQUE\_DEBUGMODE**  **In brief**  It transports the raw ADC values of channels 3, 4 and 5. Used by the strain/strain2 in debug mode only.  This message is activated by …  **Emitted by**    strain  strain2  Transmission is started by a message of CLS = 0x2 (polling analog) and CMD = 0x07 (SET\_TXMODE) with ARG.MODE = strainMode:: txAll (= 4) and stopped with ARG.MODE = strainMode::txNone (= 1). The transmission rate is configured with a message of CLS = 0x2 (polling analog) and CMD = 0x08 (SET\_CANDATARATE) with ARG.PERIOD = period in [msec].  **Format of DATA**    X  Y  Z  C  5  C4  C3  S   * sizeof(DATA) = 6 or 7. * X is the first component expressed as a std::uint16\_t in little endian. * Y is the second component expressed as a std::uint16\_t in little endian. * Z is the third component expressed as a std::uint16\_t in little endian. * {C5, C4, C3, S} are an optional byte which is transmitted in case of a saturation in ADC acquisition of at least one of the six ADC channels. S has value 1 of any of the six channels saturated (if sizeof(DATA) is 7 S is always 1). Cx contains the kind of saturation in channel x: NONE = 0, atLOWscale = 1, atHIGHscale = 2.   **Note**  A raw ADC value has a range [0, 64K) and represents the output of an ADC of the same range. It is saturated at low scale if is in range [0, 1000). It is saturated at high scale if it is in range [64000, 64K) |

|  |  |
| --- | --- |
| TYP | Description |
| **0xA** | **FORCE\_VECTOR**  **In brief**  It transports either the X Y Z force components or the raw ADC values of channels 0, 1 and 2.  **Emitted by**    strain  strain2  Transmission is started by a message of CLS = 0x2 (polling analog) and CMD = 0x07 (SET\_TXMODE). If ARG.MODE = strainMode::txCalibrated (= 0) the board will transmit the X Y Z force components; else if ARG.MODE = strainMode::txUncalibrated (= 3) the board will transmit the ADC values of channels 0, 1, and 2. ). Transmission is stopped with ARG.MODE = strainMode::txNone (its value is 1). The transmission rate is configured with a message of CLS = 0x2 (polling analog) and CMD = 0x08 (SET\_CANDATARATE) with ARG.PERIOD = period in [msec].  **Format of DATA**    X  Y  Z  C  2  C1  C0  S   * sizeof(DATA) = 6 or 7. * X is the first component expressed as a std::uint16\_t in little endian. * Y is the second component expressed as a std::uint16\_t in little endian. * Z is the third component expressed as a std::uint16\_t in little endian. * {C2, C1, C0, S} are an optional byte which is transmitted in case of a saturation in ADC acquisition of at least one of the six ADC channels. S has value 1 of any of the six channels saturated (if sizeof(DATA) is 7 S is always 1). Cx contains the kind of saturation in channel x: NONE = 0, atLOWscale = 1, atHIGHscale = 2.   **Note**  A raw ADC value has a range [0, 64K) and represents the output of an ADC of the same range. It is saturated at low scale if is in range [0, 1000). It is saturated at high scale if it is in range [64000, 64K).  Force data in X, Y, Z has a range [0, 64K) and contains a 1Q15 number (an integer representation of values in the range [-1, +1-2-15]) which is the result of the calibration inside the board. It actually contains 1Q15 + 0x8000. To get the 1Q15 value for the first component one need to do:  std::int16\_t q15X = static\_cast<std::int16\_t>(X – 0x8000);  Its floating point representation is obtained with:  double dX = static\_cast<double>(q15X) / 32768.0;  And to finally retrieve the actual force on X component measured in [N] one must multiply for the proper fullscale:  double fX = fullscale[0] \* dX; |

|  |  |
| --- | --- |
| TYP | Description |
| **0xB** | **TORQUE\_VECTOR**  **In brief**  It transports either the X Y Z torque components or the raw ADC values of channels 3, 4 and 5.  **Emitted by**    strain  strain2  Transmission is started by a message of CLS = 0x2 (polling analog) and CMD = 0x07 (SET\_TXMODE). If ARG.MODE = TxMode:: calibrated (= 0) the board will transmit the X Y Z torque components; else if ARG.MODE = strainMode::txUncalibrated (= 3) the board will transmit the ADC values of channels 3, 4, and 5. Transmission is stopped with ARG.MODE = strainMode::txNone (its value is 1). The transmission rate is configured with a message of CLS = 0x2 (polling analog) and CMD = 0x08 (SET\_CANDATARATE) with ARG.PERIOD = period in [msec].  **Format of DATA**    X  Y  Z  C  5  C4  C3  S   * sizeof(DATA) = 6 or 7. * X is the first component expressed as a std::uint16\_t in little endian. * Y is the second component expressed as a std::uint16\_t in little endian. * Z is the third component expressed as a std::uint16\_t in little endian. * {C5, C4, C3, S} are an optional byte which is transmitted in case of a saturation in ADC acquisition of at least one of the six ADC channels. S has value 1 of any of the six channels saturated (if sizeof(DATA) is 7 S is always 1). Cx contains the kind of saturation in channel x: NONE = 0, atLOWscale = 1, atHIGHscale = 2.   **Note**  A raw ADC value has a range [0, 64K) and represents the output of an ADC of the same range. It is saturated at low scale if is in range [0, 1000). It is saturated at high scale if it is in range [64000, 64K).  Torque data in X, Y, Z has a range [0, 64K) and contains a 1Q15 number (an integer representation of values in the range [-1, +1-2-15]) which is the result of the calibration inside the board. It actually contains 1Q15 + 0x8000. To get the 1Q15 value for the first component one need to do:  std::int16\_t q15X = static\_cast<std::int16\_t>(X – 0x8000);  Its floating point representation is obtained with:  double dX = static\_cast<double>(q15X) / 32768.0;  And to finally retrieve the actual torque on X component measured in [N m] one must multiply for the proper fullscale:  double fX = fullscale[3] \* dX; |

### Messages which stream HES data

|  |  |
| --- | --- |
| TYP | Description |
| **0xC** | **HES0TO6**  **In brief**  It transports data from the Hall effect sensors: channel 0 to 6.  **Emitted by**    mais  Transmission is started by a message of CLS = 0x2 (polling analog) and CMD = 0x07 (SET\_TXMODE) with ARG.MODE = maisMode:: txHallEffectData (= 0) and stopped with ARG.MODE = maisMode::txNone (its value is 1). The transmission rate is configured with a message of CLS = 0x2 (polling analog) and CMD = 0x08 (SET\_CANDATARATE) with ARG.PERIOD = period in [msec].  **Format of DATA**  S0  S1  S2  S3  S4  S5  S6   * sizeof(DATA) = 7. * Sx is data from sensor x expressed as a std::uint8\_t. |

|  |  |
| --- | --- |
| TYP | Description |
| **0xD** | **HES7TO14**  **In brief**  It transports data from the Hall effect sensors: channel 7 to 14.  **Emitted by**    mais  Transmission is started by a message of CLS = 0x2 (polling analog) and CMD = 0x07 (SET\_TXMODE) with ARG.MODE = maisMode:: txHallEffectData (= 0) and stopped with ARG.MODE = maisMode::txNone (its value is 1). The transmission rate is configured with a message of CLS = 0x2 (polling analog) and CMD = 0x08 (SET\_CANDATARATE) with ARG.PERIOD = period in [msec].  **Format of DATA**  S7  S8  S9  S10  S11  S12  S13  S14   * sizeof(DATA) = 7. * Sx is data from sensor x expressed as a std::uint8\_t. |

### Messages which stream THERMO data

|  |  |
| --- | --- |
| TYP | Description |
| **0xE** | **THERMOMETER\_MEASURE**  **In brief**  It transports data from temperature sensors. The used resolution is of 0.1 Celsius degrees.  **Emitted by**    mtb4  strain2  Transmission is started at a given rate by a message of CLS = 0x2 (polling analog) and CMD = 0x3A (THERMOMETER\_TRANSMIT) with ARG.RATE = rate (in [sec]) and stopped with ARG.RATE = 0. One must first configure and enable the chosen thermometer with a message of CLS = 0x2 (polling analog) and CMD = 0x39 (THERMOMETER\_CONFIG\_SET).  **Format of DATA**  M  T  FFU   * sizeof(DATA) = 5. * M is a bit mask, which express the used thermometer. So far we only have one thermometer, hence the mask will be always 0x01 * T is the temperature value expressed by a std::int16\_t in little endian at a resolution of 0.1 Celsius degrees. * FFU is for future use. It will probably contain temperature from another thermometer. |

## Class periodic skin data

The class periodic skin data (CLS = 100b) contains messages used by mtb / mtb4 boards to stream pressure data to the host.

|  |  |
| --- | --- |
| TYP | Description |
| 0x0 | TRG00 |
| 0x1 | TRG01 |
| 0x2 | TRG02 |
| 0x3 | TRG03 |
| 0x4 | TRG04 |
| 0x5 | TRG05 |
| 0x6 | TRG06 |
| 0x7 | TRG07 |
| 0x8 | TRG08 |
| 0x9 | TRG09 |
| 0xA | TRG10 |
| 0xB | TRG11 |
| 0xC | TRG12 |
| 0xD | TRG13 |
| 0xE | TRG14 |
| 0xF | TRG15 |

**Table 10** – The TYPs of the periodic skin data class.

### Messages which stream SKIN data

The format is similar for all the messages; hence, in here we show details of the TRG00 only.

|  |  |
| --- | --- |
| TYP | Description |
| **0x0** | **TRG00**  **In brief**  It transports in two frames the skin data of the 12 points belonging to the 0-th triangle.  **Emitted by**    mtb  mtb4  Transmission is started at a given rate by a message of CLS = 0x2 (polling analog) and CMD = 0x4D (SKIN\_SET\_BRD\_CFG) with ARG.RATE = rate (in [msec]) and stopped with ARG.RATE = 0.  **Format of DATA**  C1  P0  P1  P2  P3  P4  P5  P6   * sizeof(DATA) = 8. * C1 = 0x40 tells that the frame contains data of the first 7 points. * Px contains the skin value of point x.   C2  P7  P8  P9  P10  P11  STATUS   * sizeof(DATA) = 8. * C1 = 0xC0 tells that the frame contains data of the remaining 5 points. * Px contains the skin value of point x. * STATUS contains debug information about the acquired values. |

### Messages which stream POS data

|  |  |
| --- | --- |
| TYP | Description |
| **0x1** | **POS\_MEASURE**  **In brief**  It transports data from position sensors.  **Emitted by**  psc  Transmission is started at a given rate by a message of CLS = 0x2 (polling analog) and CMD = 0x53 (POS\_TRANSMIT) with ARG.RATE = rate (in [millisec]) and stopped with ARG.RATE = 0. One must first configure and enable the chosen sensors with a message of CLS = 0x2 (polling analog) and CMD = 0x52 (POS\_CONFIG\_SET).  **Format of DATA**    TYPE  LBL  N  VALUES   * sizeof(DATA) = 8. * TYPE is a byte with the type of data that the message transports. It has values in enum class posTYPE : uint8\_t { angleDeciDeg = 0, unknown = 255 }. For the psc board which transports angles in deci-degrees the values is always posTYPE::angleDeciDeg = 0. * LBL is a nibble used to identify the label of the first angle contained inside the message. It assumes a value from enum class posLABEL : uint8\_t { zero = 0, one = 1, two = 2, three = 3, four = 4, five = 5, six = 6, seven = 7, eight = 8, nine = 9, ten = 10, eleven = 11, twelve = 12, thirteen = 13, fourteen = 14, fifteen = 15 }. * N is a nibble which tells how many values are contained inside the message. The first value will have label = LBL, the second (if present) will have label = LBL+1, etc. * VALUES is a six bytes space which holds the values transported by the message. The number of values is expressed by N and their format is expressed by TYPE.   + If TYPE is posTYPE::angleDeciDeg as in the case of the psc, then each value is a std::int16\_t in little endian at a resolution of 0.1 degrees. The psc board usually sends two values (N = 2).   With N = 2 only the first four bytes of VALUES will be meaningful.    VALUE-(LBL)  VALUE-(LBL+1)  0x0000 |

## Class periodic inertial data

The class periodic inertial data (CLS = 101b) contains messages used by sensor boards to stream their inertial data to the host.

|  |  |
| --- | --- |
| TYP | Description |
| 0x0 | DIGITAL\_GYROSCOPE |
| 0x1 | DIGITAL\_ACCELEROMETER |
| 0x3 | IMU\_TRIPLE |
| 0x4 | IMU\_QUATERNION |
| 0x5 | IMU\_STATUS |

**Table 11** – The TYPs of the periodic inertial data class.

Here are details of their format.

### Messages which stream IMU data

In here there are message used by the new mtb4 and strain2 boards that mount a proper IMU chip. They are able to transport a wide range of sensor types.

|  |  |
| --- | --- |
| TYP | Description |
| **0x3** | **IMU\_TRIPLE**  **In brief**  It transports 3-axis data originated from an IMU. See **Note** for details on type of data and format for the IMU.  **Emitted by**    mtb4  strain2  rfe  Transmission is started at a given rate by a message of CLS = 0x2 (polling analog) and CMD = 0x35 (IMU\_TRANSMIT) with ARG.RATE = rate (in [msec]) and stopped with ARG.RATE = 0.  One must first configure and enable the IMU for the target measures with a message of CLS = 0x2 (polling analog) and CMD = 0x34 (IMU\_CONFIG\_SET).  **Format of DATA**    SEQ  SNS  X  Y  Z   * sizeof(DATA) = 8. * SEQ is a sequence number expressed in std::uint8\_t that is incremented at each acquisition. It is used to matching 3-axis data amongst them and with status messages. * SNS is the type of IMU sensor and with value amongst enum class imuSensor but limited to those w/ 3 axis * X is the first component expressed as a std::int16\_t in little endian. * Y is the second component expressed as a std::int16\_t in little endian. * Z is the third component expressed as a std::int16\_t in little endian.   **Note**  The IMU supports the data types from enum class imuSensor { acc = 0, mag = 1, gyr = 2, eul = 3, qua = 4, lia = 5, grv = 6, status = 15, none = 16 }.  Each component is expressed as a std::int16\_t with following resolution and measurement units:   * imuSensor::acc in 0.01 [m/s2] * imuSensor::mag in 0.0625 [microTesla] * imuSensor::gyr in 0.0625 [deg/s] * imuSensor::eul in 0.0625 [deg] * imuSensor::qua in 0.00006103515625 [quaternion] (=1/16384) * imuSensor::lia in 0.01 [m/s2] * imuSensor::grv in 0.01 [m/s2] |

|  |  |
| --- | --- |
| TYP | Description |
| **0x4** | **IMU\_QUATERNION**  **In brief**  It transports quaternion data originated from an IMU. See **Note** of **IMU\_TRIPLE** for details on type of data and format for the IMU.  **Emitted by**    mtb4  strain2  rfe  Transmission is started at a given rate by a message of CLS = 0x2 (polling analog) and CMD = 0x35 (IMU\_TRANSMIT) with ARG.RATE = rate (in [msec]) and stopped with ARG.RATE = 0.  One must first configure and enable the IMU for the target measures with a message of CLS = 0x2 (polling analog) and CMD = 0x34 (IMU\_CONFIG\_SET\_CONFIG\_SET).  **Format of DATA**    W  X  Y  Z   * sizeof(DATA) = 8. * W is the first component expressed as a std::int16\_t in little endian. * X is the second component expressed as a std::int16\_t in little endian. * Y is the third component expressed as a std::int16\_t in little endian. * Z is the fourth component expressed as a std::int16\_t in little endian. |

|  |  |
| --- | --- |
| TYP | Description |
| **0x5** | **IMU\_STATUS**  **In brief**  It contains acquisition status of the IMU.  **Emitted by**    mtb4  strain2  rfe  Transmission is started at a given rate by a message of CLS = 0x2 (polling analog) and CMD = 0x35 (IMU\_TRANSMIT) with ARG.RATE = rate (in [msec]) and stopped with ARG.RATE = 0.  One must first configure and enable the IMU for the target measures with a message of CLS = 0x2 (polling analog) and CMD = 0x34 (IMU\_CONFIG\_SET\_CONFIG\_SET).  **Format of DATA**  SEQ  SGY  SAC  SMA  ACQDURATION   * sizeof(DATA) = 8. * SEQ is a sequence number expressed in std::uint8\_t that is incremented at each acquisition. It is used to match 3-axis data and status messages amongst them. * SGY contains the calibration status of the gyroscope * SAC contains the calibration status of the accelerometer. * SMA contains the calibration status of the magnetometer. * ACQDURATION contains the time the IMU has spent for the acquisition expressed as a std::uint32\_t in little endian in micro-sec units.   **Note**  The IMU performs an internal autocalibration for accelerometer, gyroscope and magnetometer the result of which is expressed with enum class Calib { none = 0, poor = 1, medium = 2, good = 3 }.  The content of the message is still WIP. We could merge {SGY, SAC, SMA} in a single byte, reduce ACQDURATIONtime to 2 bytes and add a new ACQTIME expressed in milli-sec with an absolute generation time which could help putting all measures in the correct time scale. |

### Messages which stream accel and gyro in mtb boards

In here there are message used by mtb boards (and for legacy purposes by the mtb4 and strain2) which transports data from accelerometer and gyroscope.

If there are not mtb boards, we recommend using the new IMU messages.

|  |  |
| --- | --- |
| TYP | Description |
| **0x0** | **DIGITAL\_GYROSCOPE**  **In brief**  It transports gyroscope data as emitted by the mtb board (and for legacy purposes by the mtb4 and strain2).  **Emitted by**    mtb  mtb4  strain2  Transmission is started with a given rate by a message of CLS = 0x2 (polling analog) and CMD = 0x3A (ACC\_GYRO\_SETUP) with ARG.RATE = rate (in [sec]) if the ARG.MASK enables the gyroscope, and stopped with ARG.RATE = 0.  **Format of DATA**    X  Y  Z   * sizeof(DATA) = 6. * X is the first component expressed as a std::int16\_t in little endian. * Y is the first component expressed as a std::int16\_t in little endian. * Z is the first component expressed as a std::int16\_t in little endian.   **Note**  TBD: find out the resolution of the measure. |

|  |  |
| --- | --- |
| TYP | Description |
| **0x1** | **DIGITAL\_ACCELEROMETER**  **In brief**  It transports accelerometer data as emitted by the mtb board (and for legacy purposes by the mtb4 and strain2).  **Emitted by**    mtb  mtb4  strain2  Transmission is started with a given rate by a message of CLS = 0x2 (polling analog) and CMD = 0x3A (ACC\_GYRO\_SETUP) with ARG.RATE = rate (in [sec]) if the ARG.MASK enables the accelerometer, and stopped with ARG.RATE = 0.  **Format of DATA**    X  Y  Z   * sizeof(DATA) = 6. * X is the first component expressed as a std::int16\_t in little endian. * Y is the first component expressed as a std::int16\_t in little endian. * Z is the first component expressed as a std::int16\_t in little endian.   **Note**  TBD: find out the resolution of the measure. |

## Class periodic battery service data

The class periodic battery data (CLS = 110b) contains messages used by sensor boards to stream the status of a battery to the host.

|  |  |
| --- | --- |
| TYP | Description |
| 0x0 | BMS\_PACK\_INFO |
| 0x1 | Unused |
| 0x2 | Unused |
| 0x3 | Unused |
| 0x4 | Unused |
| 0x5 | Unused |
| 0x6 | Unused |
| 0x7 | Unused |
| 0x8 | BMS\_BMS\_STATUS\_INFO |
| 0x9 | BMS\_BAT\_STATUS\_INFO |
| 0xA | BMS\_BMS\_ADVANCED\_STATUS\_INFO |
| 0xB | Unused |
| 0xC | Unused |
| 0xD | Unused |
| 0xE | Unused |
| 0xF | BMS\_DIAGN\_LOG\_INFO |

**Table 12** – The TYPs of the periodic battery data class.

Here are details of their format.

### Messages which stream Battery data

In here there are messages used to transport battery data.

|  |  |
| --- | --- |
| TYP | Description |
| **0x0** | **BMS\_PACK\_INFO**  **In brief**  It transports information about the whole battery pack (which is formed by multiple cells).  **Emitted by**  BMS\_15586  BMS\_18239  BAT  On the BMS and BAT, transmission is started at the starting TX command.  **Format of DATA**    VOLTAGE  CURRENT  STATE\_OF\_CHARGE  TEMPERATURE   * sizeof(DATA) = 8. * VOLTAGE is the emitted voltage in units of 0.1V and expressed and a std::uint16\_t in little endian. * CURRENT is the emitted current in units of 0.1A and expressed and a std::uint16\_t in little endian. * CHARGE is the average battery-pack charge in units of 1% and expressed as a std::int16\_t in little endian. * TEMPERATURE is average temperature of the battery-pack in units of 1 ℃ and expressed as a std::int16\_t in little endian. Temperature is available only for BMS. BAT cannot stream the battery-pack average temperature.   **Note**  The BMS coded as 15586 in Wingst and the BAT board can assume only an address = 2, so the ID will be 0x620. |

|  |  |
| --- | --- |
| TYP | Description |
| **0x8** | **BMS\_BAT\_STATUS\_INFO**  **In brief**  It transports information and status about the battery pack for the BMS coded as 15586 (which is formed by multiple cells of type 10s3pNCR18650BL-01). This message won’t be available for the new BMS with code 18239, which will integrate this info in message 0x62A.  **Emitted by**  BMS\_15586  On the BMS, transmission is continuous at a fixed rate of 1 Hz. This BMS cannot start or stop the transmission based on an ENABLE or DISABLE command but when the battery pack is on it send all messages over CAN.  **Format of DATA**    GENERAL\_STATUS   * sizeof(DATA) = 8 but only the first byte is filled. * GENERAL\_STATUS is formed as:   + Bit0: low voltage   + Bit1: high voltage   + Bit2: Overcurrent in discharge   + Bit3: Overcurrent in charge   + Bit4: Low StateOfCharge   + Bit5: Low temperature   + Bit6: High temperature   **Note**  The BMS coded as 15586 in Wingst and the BAT board can assume only an address = 2, so the ID will be 0x620. |

|  |  |
| --- | --- |
| TYP | Description |
| 0x9 | BMS\_BAT\_STATUS\_INFO  In brief  It transports information about the status of the BAT board only.  Emitted by  BAT  On the BAT only, transmission is started on the reception of TX command.  Format of DATA    DCDC\_STATUS\_A  DCDC\_STATUS\_B   * sizeof(DATA) = 8. (Only the first 2 bytes are filled, the others are zeros) * STATUS\_A contains boolean flags regarding status and errors on the DCDC for the BAT board. Precisely we have, from rightmost bit to leftmost one:   + HSM broken error   + HSM fault   + HSM power good   + HSM started   + MOTORS fault error (42V side)   + MOTORS started and on (42V side)   + CPU fault error (12V side)   + CPU started and on (12V side) * STATUS\_B contains status and errors when BAT board gets HW or SW problems or gets restarted. Specifically, starting from the rightmost bit to the leftmost, we have:   + BUTTON 2 (MOTOR) restarting   + BUTTON 1 (CPU) restarting   + HSM HW fault error (hw overcurrent/overvoltage on the hot swap manager)   + HSM SW fault error (sw overcurrent/overvoltage on the host swap manager)   Note  The BAT board can assume only an address = 2, so the ID will be 0x629. |

|  |  |
| --- | --- |
| TYP | Description |
| **0xA** | **BMS\_ADVANCED\_STATUS\_INFO**  **In brief**  It transports advanced information about the BMS status. This message won’t be available for the BMS with Wingst code 15586 and BAT.  **Emitted by**  BMS\_18239  On the BMS, transmission is started at the starting TX command.  **Format of DATA**    CMD\_FLAG\_BL  CMS\_FLAG\_BH  EQL\_FLAG\_BL  EQL\_FLAG\_BH  BMS\_VOUT\_BL  BMS\_VOUT\_BH  ALARM\_FLAG\_BL  ALARM\_FLAG\_BL   * sizeof(DATA) = 8. * Command Flag byte low. * Command Flag byte high. * Balance Flag byte low. * Balance Flag byte high. * Bms Vout (mV) byte low. * Bms Vout (mV) byte high. * Alarm Flag byte low. * Alarm Flag byte high. |

|  |  |
| --- | --- |
| TYP | Description |
| **0xF** | **BMS\_DIAGN\_LOG\_INFO**  **In brief**  It transports information about the diagnostic info (latest errors) on the battery pack (which is formed by multiple cells). This message won’t be available for the BMS with Wingst code 15586 and BAT.  **Emitted by**  BMS\_18239  On the BMS, this message is sent if and only if an error happened in the last run. Basically, at the restart of the communication, the host board asks to the BMS if we got any error in the previous run before closing the communication. If that happened the latest errors are send as payload.  This type of message is a polling message, but, since a battery polling class message doesn’t’ t exist, this message has the last available value of the periodic message class .  **Format of DATA**    COMMAND\_IDX (0x73)  RECORD\_IDX (0-23) | LAST\_SESSION\_BIT (0x80)  TSMP (UNIX) LSB  TSMP (UNIX)  TSMP (UNIX)  TSMP (UNIX) MSB  RECORD\_ LSB  RECORD\_ MSB   * sizeof(DATA) = 8. * Command Index. Always 0x73 for this CAN message. * Record Index (0-23) | LastSessionBit (always 0x80) * Unix Timestamp LSB * Unix Timestamp * Unix Timestamp * Unix Timestamp MSB * Record data LSB * Record data MSB |

Which messages on which board

In here, it is summarised which messages are supported by which boards.

At first, we group the messages in small groups: parsers and formers. We use the term parser for one object that is able to decode a group of messages and former for an object that is able to emit messages.

Then we tell for which board which are the supported parsers or formers.

## The parsers and the formers

We can group together in some messages that have a similar use and that the boards supports in full.

The CAN parsers and formers in the nodes

Every bootloader is able to manage the same subset of messages. However, some boards may manage some extra messages.

bootloader

BTL EXT parser

BTL common parser

application

APP common parser

Serv-IMU parser

APP EXT parser

Every application is able to manage the same subset of messages. There are some other groups of messages dedicated to a specific task

Serv-IMU former

**Figure 15**: Example of parsers and formers in a CAN node.

Here is a list of the possible ones with the associated messages.

|  |  |  |
| --- | --- | --- |
| Parser / Former | Supported message | Description |
| **BTL common parser**  bootloader | **Management class:**  BOARD  ADDRESS  START  DATA  END  GET\_ADDITIONAL\_INFO  SET\_ADDITIONAL\_INFO  BROADCAST (or DISCOVER) | Used in every bootloader for decoding and reply to messages that offer discovery, management and firmware update service. |
| **BTL SNS-specific parser**  btl<SNS> | **Polling analog sensor class:**  SET\_BOARD\_ADX | Used in every bootloader of sensor boards for decoding messages that are specific to sensors only. |
| **BTL EXT parser**  btl<mtb4>  btl<strain2>  btl<rfe>  btl<psc> | **Management class:**  SET\_ADDRESS  **Polling analog sensor class:**  SET\_BOARD\_ADX  **Polling motion control class:**  SET\_BOARD\_ID | Used in every bootloader of new sensor boards (mtb4 and strain2) as an extension of the **BTL SNS-specific class**. |
| **APP common parser**  application | **Management class:**  BOARD  GET\_ADDITIONAL\_INFO  SET\_ADDITIONAL\_INFO  BROADCAST (or DISCOVER) | Used in every application for decoding and reply to messages that offer discovery, management and start of firmware update service. |
| **APP SNS-specific parser**  app<SNS> | **Polling analog sensor class:**  SET\_BOARD\_ADX  GET\_FIRMWARE\_VERSION | Used in every application of sensor boards for decoding messages that are specific to sensors only. |
| **APP EXT parser**  strain2  mtb4  rfe  psc | **Management class:**  SET\_ADDRESS  **Polling analog sensor class:**  SET\_BOARD\_ADX  GET\_FIRMWARE\_VERSION  **Polling motion control class:**  SET\_BOARD\_ID  GET\_FIRMWARE\_VERSION | Used in every application of new sensor boards (mtb4 and strain2) as an extension of the **APP SNS-specific class**. |
| **Service-FT parser**  strain2  strain | **Polling analog sensor class:**  SET\_CANDATARATE  GET\_FULLSCALE  SET\_TXMODE | Used in some sensor boards for management of the FT service. |
| **Service-FT former**  strain2  strain | **Periodic analog sensor class:**  UNCALIBFORCE\_VECTOR\_DEBUGMODE  UNCALIBTORQUE\_VECTOR\_DEBUGMODE  FORCE\_VECTOR  FORCE\_TORQUE | Used in some sensor boards for streaming data of the FT service. |
| **Config-FT parser**  strain2  strain | **Polling analog sensor class:**  SET\_MATRIX\_RC  SET\_CH\_DAC  SAVE2EE  GET\_MATRIX\_RC  GET\_CH\_DAC  GET\_CH\_ADC  SET\_MATRIX\_G  GET\_MATRIX\_G  SET\_CALIB\_TARE  GET\_CALIB\_TARE  SET\_CURR\_TARE  GET\_CURR\_TARE  SET\_FULLSCALE  SET\_SERIAL\_NO  GET\_SERIAL\_NO  GET\_EEPROM\_STATUS | Used in strain board for configuration of the FT service. |
| **Config-ext-FT parser**  strain2 | **Polling analog sensor class:**  AMPLIFIER\_RESET  AMPLIFIER\_RANGE\_OF\_GAIN\_GET  AMPLIFIER\_RANGE\_OF\_OFFSET\_GET  AMPLIFIER\_GAINOFFSET\_GET  AMPLIFIER\_GAINOFFSET\_SET  AMPLIFIER\_OFFSET\_AUTOCALIB  AMPLIFIER\_PGA308\_CFG1\_GET  AMPLIFIER\_PGA308\_CFG1\_SET  REGULATIONSET\_SET  REGULATIONSET\_GET | Used in strain2 board as an extension of the **Config-FT parser** . |
| **Service-IMU parser**  strain2  mtb4  rfe | **Polling analog sensor class:**  IMU\_CONFIG\_GET  IMU\_CONFIG\_SET  IMU\_TRANSMIT | Used in some sensor boards for management of the IMU service. |
| **Service-IMU former**  strain2  mtb4  rfe | **Periodic inertial sensor class:**  IMU\_TRIPLE  IMU\_QUATERNION  IMU\_STATUS | Used in some sensor boards for streaming data of the IMU service. |
| **Service-THERMO parser**  strain2  mtb4 | **Polling analog sensor class:**  THERMOMETER\_CONFIG\_GET  THERMOMETER\_CONFIG\_SET  THERMOMETER\_TRANSMIT | Used in some sensor boards for management of the THERMO service. |
| **Service-THERMO former**  strain2  mtb4 | **Periodic analog sensor class:**  THERMOMETER\_MEASURE | Used in some sensor boards for streaming data of the THERMO service. |
| **Service-SKIN parser**  mtb  mtb4 | **Polling analog sensor class:**  SKIN\_SET\_BRD\_CFG  SET\_TRIANG\_CFG  SET\_TXMODE | Used in some sensor boards for management of the SKIN service. |
| **Service-SKIN former**  mtb  mtb4 | **Periodic skin data class:**  TRG00  TRG01  TRG02  TRG03  TRG04  TRG05  TRG06  TRG07  TRG08  TRG09  TRG10  TRG11  TRG12  TRG13  TRG14  TRG15 | Used in some sensor boards for streaming data of the SKIN service. |
| **Service-ACGY parser**  mtb  mtb4 | **Polling analog sensor class:**  SKIN\_ACC\_GYRO\_SETUP  SET\_TXMODE | Used in mtb board for management of the accelerometer / gyroscope service. |
| **Service-ACGY former**  mtb  mtb4 | **Periodic skin data class:**  DIGITAL\_GYROSCOPE  DIGITAL\_ACCELEROMETER | Used in some sensor boards for streaming data of the accelerometer / gyroscope service. |
| **Service-HES parser**  mais | **Polling analog sensor class:**  SET\_RESOLUTION  SET\_CANDATARATE  SET\_TXMODE | Used in mais board for management of the HES service. |
| **Service-HES former**  mais | **Periodic skin data class:**  HES0TO6  HES7TO14 | Used in some sensor boards for streaming data of the HES service. |
| **Service-POS parser**  psc | **Polling analog sensor class:**  POS\_CONFIG\_GET  POS\_CONFIG\_SET  POS\_TRANSMIT | Used in the psc sensor board for management of the POS service. |
| **Service-POS former**  psc | **Periodic analog sensor class:**  POS\_MEASURE | Used in the psc sensor board for streaming data of the POS service. |
| **Service-BAT former**  BMS  BAT | **Periodic battery data class:**  BS\_PACK\_INFO  BS\_BMS\_STATUS\_INFO  BS\_BAT\_STATUS\_INFO  BS\_BMS\_ADVANCED\_STATUS\_INFO  BS\_DIAGN\_LOG\_INFO | Used in the BMS and BAT sensor board for streaming data of the Battery Service (BS). |

**Table 13** – The formers / parser supported by the CAN boards.

## The formers / parsers on each board

|  |  |
| --- | --- |
| Board | Parser / Former |
| **strain** | btl<strain>  **BTL common parser, BTL SNS parser**  app<strain>  **APP common parser, APP SNS parser**  **Service-FT parser, Service-FT former, Config-FT parser** |
| **mtb** | btl<mtb>  **BTL common parser, BTL SNS parser**  app<mtb>  **APP common parser, APP SNS parser**  **Service-SKIN parser, Service-SKIN former**  **Service-ACGY parser, Service-ACGY former** |
| **mais** | btl<mais>  **BTL common parser, BTL SNS parser**  app<mais>  **APP common parser, APP SNS parser**  **Service-HES parser, Service-HES former** |
| **strain2** | btl<strain2>  **BTL common parser, BTL EXT parser**  app<strain2>  **APP common parser, APP EXT parser**  **Service-FT parser, Service-FT former, Config-ext-FT parser**  **Service-IMU parser, Service-IMU former**  **Service-THERMO parser, Service-THERMO former** |
| **mtb4** | btl<mtb4>  **BTL common parser, BTL EXT parser**  app<mtb4>  **APP common parser, APP EXT parser**  **Service-SKIN parser, Service-SKIN former**  **Service-IMU parser, Service-IMU former**  **Service-THERMO parser, Service-THERMO former**  **Service-ACGY parser, Service-ACGY former** |
| **rfe** | btl<rfe>  **BTL common parser, BTL EXT parser**  app<rfe>  **APP common parser, APP EXT parser**  **Service-IMU parser, Service-IMU former** |
| **psc** | btl<psc>  **BTL common parser, BTL EXT parser**  app<psc>  **APP common parser, APP EXT parser**  **Service-POS parser, Service-POS former** |

**Table 14** – The formers / parser supported by the CAN boards.

Details of the services

The CAN messages which manage services such as FT, IMU, etc. deal with data formats and with terms that can be difficult to understand and explain in the description of the message format.

For this reason, this section of the document contains some more details of the services that can be useful to understand the format of the messages described in this document.

It is can be useful to read this section alongside the specification of the messages.

## The FT in strain / strain2 boards

The strain and strain2 boards offer the FT service by streaming force and torque values. They obtain these values by applying analog and digital transformation to the output of six strain-gauges that measure the deformation of the metal case of the board.

We refer to the parameters of the above transformation with the term: regulation set.

The strain board has only one regulation set. It is stored in internal EEPROM and it content is determined at production stage by means of the calibration procedure.

The more recent strain2 board instead, contains up to three regulation sets. They all are stored in EEPROM. As for strain board, the first is determined at production stage, and the remaining two have default neutral values that the user can tailor to his/her needs.

In the following, we describe the details of the regulation sets.

### The FT data flow

Following figure shows the processing flow inside the strain / strain2 board. Analog data from the strain-gauges is processed in an analog stage and then is digitised and manipulated in Q15 format through an affine transform.

The FT regulation stages in strain / strain2 boards

V0 = gain0 \* SG0 + offset0

ADC

SG0

**tare**

SG5

V5 = gain5 \* SG5 + offset5

ADC

**X**

**+**

**[M]**

**q15adc**

**q15ft**

Analog stage

Digital stage

The values expressed in Q15 format have range [-1.0, +1.0 - 2-15] which is de-normalized with its proper **fullscale**.

U16toQ15

**u16adc**

**Figure 16**: The FT regulation stages in strain / strain2 boards: analog and digital regulation. In strain boards, the gain of the analog stage is not modifiable.

If the HOST is interested in force and torque values, the so-called calibrated values, it must at first ask the value of fullscale with the messages GET\_FULL\_SCALES. Then, it must order transmission of calibrated data with the message SET\_TXMODE with strainMode::txCalibrated. At this point, the data in Q15 format (q15ft) is transported over CAN by the FORCE\_VECTOR and TORQUE\_VECTOR streaming messages and then is recomposed by the receiving host, which applied the proper fullscale and uses the force and torque values in SI units (N and Nm).

The streaming of FT calibrated values from strain / strain2 to the HOST

strain2

CAN bus

HOST

**u16ft**

**X**

**u16ft**

**FT**

**q15ft**

**fullscale**

U16toQ15

(expressed in N and Nm)

Q15toU16

u16f

u16t

**q15ft**

Q15toFLOAT

**normalisedFT**

The HOST must have previously asked the value of fullscale

FORCE\_VECTOR

TORQUE\_VECTOR

**Figure 17**: The HOST receives the CAN frames FORCE\_VECTOR and TORQUE\_VECTOR that carry calibrated data; it transforms the values into Q15 format and then de-normalizes with the application of the fullscale. After these transformations, we have the FT values expressed in N and in Nm.

The streaming of ADC raw values from strain / strain2 to the HOST

strain2

CAN bus

HOST

**u16adc**

**u16adc**

u16adc[0, 2]

u16adc[3,5]

The HOST just uses the values

FORCE\_VECTOR

TORQUE\_VECTOR

**Figure 18**: The HOST receives the CAN frames FORCE\_VECTOR and TORQUE\_VECTOR that carries un-calibrated ADC raw data and it just uses them.

If instead, the HOST is interested in having raw (or un-calibrated data), it is enough that it orders the start of un-calibrated data sending command SET\_TXMODE with strainMode::txUncalibrated. The data in U16 format (u16adc) is transported over CAN by the FORCE\_VECTOR and TORQUE\_VECTOR streaming messages and then is used by the receiving HOST.

### Used data formats

The FT data uses the Q15 format for internal computation, but it also transforms the Q15 format to U16 for its transports and then to floating-point value. Here are some details.

The 1Q15 format is a way to represent decimal number in range [-1, +1-2-15] at a fixed resolution. The 1Q15 is a std::int16\_t where each unit is equal to 2-15 . The ADC value, which that has range [0, 64K), is mapped into a 1Q15 value and back with the following transformations:

* std::int16\_t U16toQ15(std::uint16\_t u) { return static\_cast<std::int16\_t>(u + 0x8000); }
* std::uint16\_t Q15toU16(std::int16\_t q) { return static\_cast<std::uint16\_t>(q - 0x8000); }

In addition, the 1Q15 is transformed into a float value normalised in range [-1.0, +1.0) with:

* float Q15toFLOAT(std::int16\_t q) { return static\_cast<float>(q) / 32768.0f; }

Finally, the float value can be scaled up to its range by direct multiplication with its fullscale.

### The FT settings of strain / strain2 boards

The strain board has a serial number and one regulation set. The former is a way to identify the specific board, so that one can load it with its proper regulation set. This latter contains the parameters of the digital stage (tare, matrix and associated fullscale) and the offset of the analog stage.

The FT settings in strain boards

regset-1

**[M]**

**fullscale**

**tare**

**offset**

EEPROM

regset-1

serialnumber

It is a string of up to 7 characters that identifies the board

**Figure 19**: FT settings in the strain board. There is one available regulation set. It is not possible to configure the gain in the analog stage.

The FT settings in strain2 boards

It tells which regset the board applies at bootstrap.

It can have one of these three values: regSet::first, regSet::second, regSet::third.

regset-x

**gain**

**[M]**

**fullscale**

**tare**

**offset**

EEPROM

regset-3

regset-2

regset-1

regset@boot

serialnumber

It is a string of up to 7 characters that identifies the board

**Figure 20**: FT settings in the strain2 board. There are three available regulation sets. At bootstrap, the board selects the regulation set specified by its value regset@boot.

The strain2 board instead has two more regulation sets, which also have a gain for the analog stage.

The gain allows adapting the acquisition to the very sensitivity of every strain gauge. The presence of three regulation sets allows a user to choose one regulation over another.

By default, the strain2 loads the regulation set specified by its variable regset@boot, which in production is imposed equal to 1. However, one can change this value by means of command REGULATIONSET\_SET and force the use of a different regulations set.

### The values of the regulation set

The values inside the first regulation set are imposed at production time for a particular board using the calibration procedure, described by a dedicated document.

In short, in the calibration procedure, an operator applies to the metal case of the board the force-torque values of interest; he/she acquires the corresponding un-calibrated values (u16adc), and finally runs an optimization algorithm that finds the values of tare, matrix and fullscale that best transforms the ADC values into the target force-torque values.

After this procedure, the calibration set contains values that are able to transform the variations of the electrical voltage produced by the strain gauges into the proper torque and force value effectively acting on the board.

The values of the other regulation sets at production have neutral values that the user can change to tailor his/her needs.

### Which regulation set is applied

At bootstrap, the strain / strain2 boards load the entire settings from EEPROM to RAM, pick up a regulation set and apply it to the analog and digital stages.

In strain boards, there is only one regulation set. Therefore, it is easy.

In strain2 boards, where there are three different regulation sets things are slightly more complex.

The rule is that the strain2 board at bootstrap selects the regulation set expressed by the permanent variable regset@boot, which can assume one of the values: regSet::first, regSet::second, or regSet::third.

The values of enum class regSet {theoneinuse = 0, first = 1, second = 2, third = 3} have values that explicitly refer to one of the three regulation sets, but it also has regSet:: theoneinuse which means whatever regulation set is currently used by the board.

This latter value is useful because it allows backwards compatibility in CAN messages for strain boards.

At runtime, we can select the use of a different regulation set with the command REGULATIONSET\_SET. The change can be in a temporary mode (at next bootstrap the board will load again the one in regset@boot) or in a permanent mode (hence, by modifying the value of regset@boot), according to the value of parameter VA chosen inside enum class regSetValidity {temporary = 0, permanent = 1}.

### Modifications to the values of the regulation set

The user can modify the values of the regulation set using the proper messages of the class polling analog sensors (CLS = 010b), such as SET\_MATRIX\_RC, SET\_FULLSCALES, SET\_CALIB\_TARE, etc.

These messages however, operate on the RAM copy of the settings. Thus, if one really wants to save the changes to EEPROM he/she must send the command SAVE2EE at the end of all changes.

### Differences in the analog stage between strain and strain2

The strain board allows only shifting the strain gauges voltages by an offset produced by a programmable DAC, whereas the strain2 sports one dedicated programmable amplifier that is able to apply an offset and a gain.

#### The offset in strain

The strain board supports only an offset voltage applied to the analog value of each strain gauge by means of the DAC peripherals of its MPU. That is the reason why the messages that manipulate the offset have the names SET\_CH\_DAC and GET\_CH\_DAC.

#### The programmable amplifiers PGA308 in strain2

The strain2 board mounts one programmable amplifier dedicated for each of its six strain gauges: the PGA308 by Texas Instruments (see reference [5]). We can operate on the registers of the PGA308 to tune its behavior at various amplification stages. The result of this operation is to change a gain and an offset of the amplifier according to formula **VOUT** = **gain** \* **VIN** + **offset**.

For the exact mapping between the values of the registers and the effect on gain and offset, please refer to the manual of the PGA308.

We have surely kept the old SET/GET\_CH\_DAC, which now operate on the offset inside the amplifier.

However, this command is not enough.

We have added two classes of CAN messages able to tune the amplifier:

A set of messages that are ready for a generic programmable amplifier because operate on the gain and offset. They are AMPLIFIER\_RESET, AMPLIFIER\_GAINOFFSET\_SET/GET, etc.

* A set of messages which set / get directly the values of the registers of the PGA in each channel: AMPLIFIER\_PGA308\_CFG1\_GET/SET;

The former messages are agnostic of the HW but may operate on a reduced range of the gain/offset, due to HW limitations or simplified software implementation. Hence, we added dedicated messages to query the supported range: AMPLIFIER\_RANGE\_OF\_OFFSET\_GET, AMPLIFIER\_RANGE\_OF\_GAIN\_GET.

The latter messages, despite the complexity of direct manipulation on the registers of the PGA308 offer the maximum flexibility. The GUI FirmwareUpdater uses these messages to impose values of gain and offset to the amplifiers. It resolves the mapping from gain-offset to register values at HOST level.

Here are the values of the registers for the most common gain values, with a midrange offset = 32K-1.

|  |  |  |
| --- | --- | --- |
| Gain | Offset | Register’s Values |
|  |  | GDAC  GI  I  GO  OS  ZDAC  AMPLIFIER\_PGA308\_CFG1\_SET.ARG = { } |
| 256 | 32767 | 0x8000  0x6  0  0x6  0x06  0x80cb  ARG = {0x00, 0x80, 0x66, 0x06, 0xcb, 0x80} |
| 128 | 32767 | 0x8000  0x5  0  0x6  0x0c  0x80cb  ARG = {0x00, 0x80, 0x56, 0x0c, 0xcb, 0x80} |
| 96 | 32767 | 0x4000  0x5  0  0x6  0x10  0x810f  ARG = {0x00, 0x40, 0x56, 0x10, 0x0f, 0x81} |
| 64 | 32767 | 0x8000  0x4  0  0x6  0x17  0x7f6d  ARG = {0x00, 0x80, 0x46, 0x17, 0x6d, 0x7f} |
| 48 | 32767 | 0x4000  0x4  0  0x6  0x1f  0x7fb1  ARG = {0x00, 0x40, 0x46, 0x1f, 0xb1, 0x7f} |
| 36 | 32767 | 0x1000  0x4  0  0x6  0x2a  0x8080  ARG = {0x00, 0x10, 0x46, 0x2a, 0x80, 0x80} |
| 32 | 32767 | 0x0000  0x4  0  0x6  0x2f  0x8039  ARG = {0x00, 0x00, 0x46, 0x2f, 0x39, 0x80} |
| 24 | 32767 | 0x4000  0x4  0  0x2  0x3e  0x7f62  ARG = {0x00, 0x40, 0x42, 0x3e, 0x62, 0x7f} |
| 20 | 32767 | 0x2000  0x4  0  0x2  0x4b  0x8015  ARG ={0x00, 0x20, 0x42, 0x4b, 0x15, 0x80} |
| 16 | 32767 | 0x0000  0x4  0  0x2  0x5e  0x8072  ARG = {0x00, 0x00, 0x42, 0x5e, 0x72, 0x80} |
| 10 | 32767 | 0xc000  0x0  0  0x2  0x64  0x6ef7  ARG = {0x00, 0xc0, 0x02, 0x64, 0xf6, 0x6e} |
| 8 | 32767 | 0x8000  0x0  0  0x2  0x64  0x6229  ARG = {0x00, 0x80, 0x02, 0x64, 0x29, 0x62} |
| 6 | 32767 | 0x4000  0x0  0  0x2  0x64  0x4cd4  ARG = {0x00, 0x40, 0x02, 0x64, 0xd4, 0x4c} |
| 4 | 32767 | 0x4000  0x0  0  0x0  0x64  0x2229  ARG = {0x00, 0x40, 0x00, 0x64, 0x29, 0x22} |

**Table 15** – The value of the registers of the PGA308 for some values of gain and offset = 32K-1 (midrange).

### Streaming rate

Typical acquisition time of the ADC stage in strain2 board is slightly more than 1 ms if we apply the digital regulation.

Hence, we recommend to impose a streaming period >= 2 ms.

The streaming period used in iCub is 10 ms.

## The IMU in mtb4 / strain2 / rfe boards

The mtb4 and strain2 boards offer the IMU service by streaming three-axis and four-axis values acquired by a proper IMU: the BOSCH BNO055.

### The used IMU: Bosch BNO055

For full details of the BOSCH BNO055 please refer to reference [].

Here are some configuration details.

We configure the BNO055 chip to operate fusion of its three internal MEMS (accelerometer, gyroscope and magnetometer) using the NDOF mode.

The available sensors are those described by the following type: enum class imuSensor { acc = 0, mag = 1, gyr = 2, eul = 3, qua = 4, lia = 5, grv = 6, status = 15, none = 16 }.

### Configurability via CAN

It is possible to configure the BNO055 with the message IMU\_CONFIG\_SET. This message does the following:

* It enables the streaming of the sensors in the field **SENSORS** independently from the specified functional mode.
* It sets the functional mode according to the value of the field **MODE**. The suggested values are 0x01 which enables the NDOF fusion mode or 0x00 which enables the non-fusion mode called AMG. See datasheet for details. More advanced configurability can be done according to the following tables.
* It allows to change orientation of the axis according to value of the field **ORIENT**. The value 0x00 ensures the standard orientation, the placement P1 specified by the datasheet. Other placements can be obtained according to the following tables.

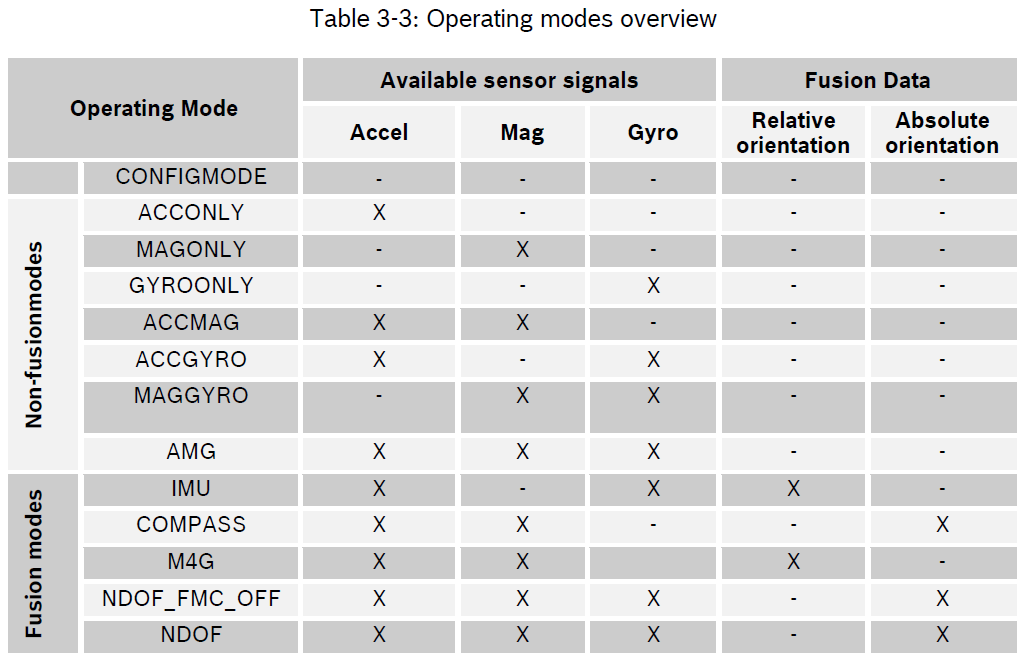
#### The operating modes

|  |  |
| --- | --- |
| Mode | Content of field MODE |
|  | MODE.T  MODE.PAR |
| NDOF | 0b  0000001b  MODE = {0b, 0000001b} = 0x01 |
| AMG | 0b  0000000b  MODE = {0b, 0000000b} = 0x00 |

**Table 16** – The resulting modes for the basic mode type with MODE.T = 0.

|  |  |
| --- | --- |
| Mode | Content of field MODE |
|  | MODE.T  MODE.PAR |
| ACCONLY | 1b  1  MODE = 0x81 |
| MAGONLY | 1b  2  MODE = 0x82 |
| GYRONLY | 1b  3  MODE = 0x83 |
| ACCMAG | 1b  4  MODE = 0x84 |
| ACCGYRO | 1b  5  MODE = 0x85 |
| MAGGYRO | 1b  6  MODE = 0x86 |
| AMG | 1b  7  MODE = 0x87 |
| IMU | 1b  8  MODE = 0x88 |
| COMPASS | 1b  9  MODE = 0x89 |
| M4G | 1b  10  MODE = 0x8A |
| NDOF\_FMC\_OFF | 1b  11  MODE = 0x8B |
| NDOF | 1b  12  MODE = 0x8C |

**Table 17** – The resulting modes for the advanced mode type with MODE.T = 1.



**Figure 21**: The modes shown inside the datasheet of the BNO055.

#### The orientation

|  |  |
| --- | --- |
| Placement | Content of field ORIENT |
|  | ORIENT.T  ORIENT.PAR |
| P1 | 0b  0  ORIENT = 0x00 |

**Table 18** – The used placement for the case of factory default with ORIENT.T = 0.

|  |  |
| --- | --- |
| Placement | Content of field ORIENT |
|  | ORIENT.T  ORIENT.PAR |
| P0 | 1b  0  ORIENT = 0x80 |
| P1 | 1b  1  ORIENT = 0x81 |
| P2 | 1b  2  ORIENT = 0x82 |
| P3 | 1b  3  ORIENT = 0x83 |
| P4 | 1b  4  ORIENT = 0x84 |
| P5 | 1b  5  ORIENT = 0x85 |
| P6 | 1b  6  ORIENT = 0x86 |
| P7 | 1b  7  ORIENT = 0x87 |

**Table 19** – The possible placements for the case of ORIENT.T = 1.

Diagram, engineering drawing

Description automatically generated

**Figure 22**: The placements shown inside the datasheet of the BNO055.

### Used data formats

The format of each component of the three-axis sensors and of the quaternion is a std::int16\_t, where each unity value has resolution value as the following:

- imuSensor::acc is in 0.01 [m/s2]

- imuSensor::mag in 0.0625 [microTesla]

- imuSensor::gyr in 0.0625 [deg/s]

- imuSensor::eul in 0.0625 [deg]

- imuSensor::qua in 0.00006103515625 [quaternion] (=1/16384)

- imuSensor::lia in 0.01 [m/s2]

- imuSensor::grv in 0.01 [m/s2]

The data of imuSensor::status hold information about the calibration status of the MEMS inside the IMU.

### Streaming rate

Typical acquisition time of all IMU data over I2C inside the strain2 / mtb4 / rfe is variable between 2 ms and 6 ms. However, the IMU produces data at most at 100 Hz (see table 3.14 in [] for details).

Hence, we recommend to impose a streaming period > 10 ms.

## The SKIN in mtb / mtb4 boards

The mtb and mtb4 boards offer the SKIN service by streaming pressure data acquired from skin patches.

### The skin patch

A skin patch is formed of up to 16 triangles, each with 12 points.

There are several kind of triangles: standard, palm, fingertip. All have 12 points but some points may be reserved for temperature compensation. Here is a scheme for the standard triangle.

The standard triangle

Dots 6 & 10 are used for thermic compensation; the remaining 10 dots measure the pressure

**Figure 23**: The standard triangle, seen from the side of the dots.

### Used data formats

The value in each point is represented with a std::uin8\_t.

### Streaming rate

The streaming data of a triangle is contained in two CAN frames of maximum length. Each board can stream up to 16 triangles, hence 32 CAN frames. The time required to transport 32 CAN frames is about 3 / 4 ms.

Typically, a SKIN service in iCub is formed of 7 mtb boards, hence we need 28 ms just to transport the streaming frames.

Because of that, the streaming period must be > 30 ms.

The typical rate is 50 ms.

## The accelerometer and gyroscope in mtb boards

The mtb boards stream acceleration and gyroscope data acquired from two MEMS by STMicroelectronics: the LIS331DLH 3-axis digital accelerometer and the L3G4200D 3-axis digital gyroscope.

### The used chips: LIS331DLH and L3G4200D by STMicroelectronics

As per title.

### Used data formats

The value of each component is represented with a std::int16\_t.

The accelometer has a fullscale equal to 2G, whereas the gyroscope has equal to 250 degree/sec.

To convert the accelerometer measure in m/s^2 the conversion factor is (fullscale in m/s^2)/(fullscale in raw) = (2\*g)/(2^15) ~= 5.9855e-04.

To convert the gyroscope measure in dps (deg/s) the conversion factor is (fullscale in dps)/(fullscale in raw) = (250)/(2^15) ~= 7.6274e-03.

### Streaming rate

Typically, in iCub the streaming period is 50 ms.

## The THERMO in mtb4 / strain2 boards

The mtb4 and strain2 boards offer the THERMO service by streaming temperature values acquired by the chip SI7051 by Silicon Labs.

### The used chip: SI7051 by Silicon Labs

As per title.

### Used data formats

The format of a temperature value is a std::int16\_t with 0.1 Celsius degrees.

### Streaming rate

The mtb4 / strain2 boards acquired the value in about 7 ms.

However, the streaming period is specified (so far) only in seconds. Typical value is 5 sec.

## The POS in psc board

The psc board offer the POS service by streaming angles values acquired by the Infineon TLV493D.

### The used chip: Infineon TLV493D

For full details of the Infineon TLV493D we remand to reference [].

The chip is used to acquire magnetic flux in three axis, measured with respect to a magnet that rotates on the X-Y axis. By getting the atan2() of the X and Y components we measure an angle with range [0, 360] degrees.

This angle, is compensated by the POS\_CONFIG\_SET message to offer: phase rotation to avoid the 360-0 discontinuity, reverse the sign of the angle, apply an offset to impose zero reading to a given position.

There are two such chips in the psc board, hence we can read two angles.

### Used data formats

The format of each angles is in DeciDegrees represented as a a std::int16\_t, where each unity value has 0.1 degrees resolution value.

### Streaming rate

Typical acquisition time of all data over I2C inside the psc is a few milliseconds.

Hence, we recommend to impose a streaming period > 10 ms.

Typical scenarios

The following examples show the use of CAN frames for some real use case. For full details of the syntax of the commands, please refer to the relevant section of this document.

## A CAN network

The typical CAN network in iCub contains one node at address ADDR = 0 which we call the host plus other nodes with addresses in range [1, 14]. The host is an ETH board such as EMS of MC4PLUS or a CFW2 drive, but one can use also a CAN-USB dongle attached to a PC. The other nodes are for instance a strain2 or a mtb4 or others. Typically, the host is responsible of performing discovery or check of presence, configure a service (IMU, THERMO, SKIN, FT, etc.), start it, stop it.

The typical CAN network for IMU / THERMO / SKIN service

HOST @ 0

HOST @ 0.

It is responsible of performing discovery or check of presence, configure a service (IMU, THERMO, SKIN, FT, etc), start it, stop it.

mtb4 @ 1

mtb4 @ 2

mtb4 @ 3

mtb4 @ 3

**Figure 24**: A CAN network capable of offering IMU, THERMO, SKIN services.

## Discovery of nodes

In here, a host at address 0 sends messages to discover the presence of boards in the CAN network. The host transmits the first message that discovers all the boards in broadcast mode even if it belongs to a unicast class. It can do that by using the broadcast address ID.DST = 0xf. All later messages are sent in unicast.

### Basic discovery

The host sends a DISCOVER message in broadcast. The boards in the CAN network reply with basic information.

Sequence diagram for basic DISCOVERY

HOST @ 0

FRAME-DISCOVER

strain2 @ 1

FRAME-R2

FRAME-R1

The host sends the DISCOVER message

The strain2 tells it is present

The hosts stores info about the presence of a strain2 @ addr = 1 and a mtb4 @ addr = 2.

The mtb4 tells it is present.

mtb4 bootloader @ 2

**Figure 25**: Example of CAN frames used for the basic discovery. In the CAN bus there are two boards: a strain2 running the application and an mtb4 running the bootloader.

FRAME-DISCOVER

ID

This frame does not need any arguments.

{CLS = BOOTLOADER-MANAGEMENT = 111b} {SRC = 0000b} {DST = 1111b} = 0x70F

CMD

[]

DISCOVER = 0xFF

sizeof(ARG) = 0

**Figure 26**: This frame asks to every device (ID.DST = 0xF ) info about their presence.

FRAME-R1: reply at the FRAME-DISCOVER (case of application)

ID

This frame has the same CMD value as FRAME- DISCOVER, but is it recognizable as an answer because the size of ARG is 4. In ARG is there is the type of board (strain2 has 0x0C) and its application version (major = 1, minor = 2, build = 3).

{CLS = BOOTLOADER-MANAGEMENT = 111b} {SRC = 0001b} {DST = 0000b} = 0x710

CMD

ARG

[ 0x0C ] [ 0x01 ] [ 0x02 ] [ 0x03 ]

DISCOVER = 0xFF

sizeof(ARG) = 4

**Figure 27**: The strain2 board replies with a frame that contains the type of board it is and the version of its application.

FRAME-R2: reply at the FRAME-DISCOVER (case of bootloader)

ID

This frame has the same CMD value as FRAME- DISCOVER, but is it recognizable as an answer because the size of ARG is 3. Also, from this size we can tell that the bootloader and not the application is is running. In ARG is there is the type of board (mtb4 has 0x0B) and its bootloader version (major = 4, minor = 5).

{CLS = BOOTLOADER-MANAGEMENT = 111b} {SRC = 0010b} {DST = 0000b} = 0x720

CMD

ARG

[ 0x0B ] [ 0x04 ] [ 0x05 ]

DISCOVER = 0xFF

sizeof(ARG) = 3

**Figure 28**: The mtb4 board replies with a frame that contains the type of board it is and the version of its bootloader.

### More detailed discovery

After the host has a list of CAN nodes with {node address, node type, process in execution, process version}, it can ask for more information such as the additional info or details on its CAN protocol version.

With message with CMD = GET\_ADDITIONAL\_INFO of class bootloader-management it is possible to retrieve a user-defined short string which is contained in the EEPROM of the node.

With message with CMD = GET\_FIRMWARE\_VERSION of class polling-analog it is possible to check if a node is present, if it runs the application and if its CAN protocol is coherent with the messages the host is expecting to run its services.

Sequence diagram for more detailed DISCOVERY: additional info

HOST @ 0

FRAME- GET\_ADDITIONAL\_INFO

strain2 @ 1

FRAME-R1

The host sends the GET\_ADDITIONAL\_INFO message

The strain2 send its additional info, which it has stored inside its EEPROM as a short string

The hosts stores the received string.

FRAME-R8

**Figure 29**: Example of CAN frames used for the detailed discovery. The HOST send a GET\_ADDITIONAL\_INFO and the strain2 replies with its string “Hi, I am a strain2”.

Sequence diagram for more detailed DISCOVERY: application version

HOST @ 0

FRAME- GET\_FIRMARE\_VERSION

strain2 @ 1

FRAME-R9

The host sends the GET\_FIRMARE\_VERSION message

The strain2 send its application and protocol versions

The hosts stores the received versions of application and of CAN protocol.

**Figure 30**: Example of CAN frames used for the detailed discovery. The HOST send a GET\_FIRMWARE\_VERSION and the strain2 replies with its application and protocol versions.

## Management of nodes

See following sections

### Change the additional information

With message with CMD = SET\_ADDITIONAL\_INFO of class bootloader-management it is possible to impose a user-defined short string to be contained in the EEPROM of the node.

Sequence diagram for change of additional info

HOST @ 0

FRAME-SET\_ADDITIONAL\_INFO

strain2 @ 1

The host sends a burst of SET\_ADDITIONAL\_INFO messages which contain: “I am a strain2!

The strain2 stores the new additional info in its EEPROM

**Figure 31**: Example of CAN frames used for the change of additional info. The HOST send a burst of SET\_ADDITIONAL\_INFO frames and the strain2 stores its new string “**I am a strain2!**”.

FRAME- SET\_ADDITIONAL\_INFO-0

ID

The first part of the new string: “I am”.

{CLS = MANAGEMENT = 111b} {SRC = 0000b} {DST = 0001b} = 0x701

CMD

[0x00] [‘**I**’] [‘’] [‘**a**’] [‘**m**‘]

SET\_ADDITIONAL\_INFO = 0x0D

sizeof(ARG) = 5

**Figure 32**: This frames changes a portion of the temporary buffer of additional info.

FRAME- SET\_ADDITIONAL\_INFO-1

ID

The second part of the new string: “ a s”.

{CLS = MANAGEMENT = 111b} {SRC = 0000b} {DST = 0001b} = 0x701

CMD

[0x01] [‘’] [‘**a**’] [‘’] [‘**s**‘]

SET\_ADDITIONAL\_INFO = 0x0D

sizeof(ARG) = 5

**Figure 33**: This frames changes a portion of the temporary buffer of additional info.

FRAME- SET\_ADDITIONAL\_INFO-2

ID

The third part of the new string: “trai”.

{CLS = MANAGEMENT = 111b} {SRC = 0000b} {DST = 0001b} = 0x701

CMD

[0x02] [‘**t**’] [‘**r**’] [‘**a**’] [‘**i**‘]

SET\_ADDITIONAL\_INFO = 0x0D

sizeof(ARG) = 5

**Figure 34**: This frames changes a portion of the temporary buffer of additional info.

FRAME- SET\_ADDITIONAL\_INFO-3

ID

The fourth part of the new string: “n2! ”.

{CLS = MANAGEMENT = 111b} {SRC = 0000b} {DST = 0001b} = 0x701

CMD

[0x03] [‘**n**’] [‘**2**’] [‘**!**’] [0x00]

SET\_ADDITIONAL\_INFO = 0x0D

sizeof(ARG) = 5

**Figure 35**: This frames changes a portion of the temporary buffer of additional info.

FRAME- SET\_ADDITIONAL\_INFO-4/5/6/7

ID

The last 4 frames just contain zeros

{CLS = MANAGEMENT = 111b} {SRC = 0000b} {DST = 0001b} = 0x701

CMD

[0x04/5/6/7] [0x00] [0x00] [0x00] [0x00]

SET\_ADDITIONAL\_INFO = 0x0D

sizeof(ARG) = 5

**Figure 36**: The remaining four frames must contain all zeros. When all 8 frames are received the board copies the temporary buffer into permanent memory.

### Change the CAN address

With message with CMD = SET\_BOARD\_ADX of class polling-analog it is possible to change the CAN address of the node.

The same operation can be done with CMD = SET\_ADDRESS of class bootloader-management.

Sequence diagram for change of address

HOST @ 0

FRAME-SET\_BOARD\_ADX

mtb4 @ 3

The host sends the SET\_BOARD\_ADX message to impose ADR = 8

The mtb4 stores the new address to be = 8.

**Figure 37**: Example of CAN frames used for the change of address. The HOST send a SET\_BOARD\_ADX to the mtb4, which stores its new address.

FRAME- SET\_BOARD\_ADX

ID

The new address is = 8

{CLS = ANALOG-POLLING = 010b} {SRC = 0000b} {DST = 0011b} = 0x203

CMD

[0x08]

SET\_BOARD\_ADX = 0x32

sizeof(ARG) = 1

**Figure 38**: This frame commands the node at address 3 to assume a new address = 8.

### Change the CAN address in a random way

If a CAN network has two nodes with the same address it is impossible to recover using a message with CMD = SET\_BOARD\_ADX of class polling-analog, because both the nodes will change the address to the same value.

However, a message with CMD = SET\_ADDRESS of class management is able to change the CAN address of the two nodes in a random way and allows to recover.

Here is the situation.

Sequence diagram for random change of address

HOST @ 0

FRAME- SET\_ADDRESS

strain2 @ 1 mtb4 @ 2 mtb4 @ 8 mtb4 @ 8

The host sends the SET\_ADDRESS message to ADR = 8 to impose a random address which is not = 1 or 2

The two mtb4 boards with ADR = 7 will assign a new random address which is not 1 or 2.

**Figure 39**: Example of CAN frames used for the random change of address. The HOST send a SET\_ADDRESS to the ADR = 8 and the two boards will assign a new random address. If the two new addresses are still equal, the host keeps sending a similar SET\_ADDRESS message.

FRAME- SET\_ADDRESS

ID

0xFF tells to use a random value. The address must not be ADR = 1 or ADR = 2 because the invalid mask 0x0006 has the bits in position 1 and 2 with a 1 value (0x0006 = 0000000000000110b

{CLS = BOOTLOADER-MANAGEMENT = 111b} {SRC = 0000b} {DST = 1000b} = 0x708

CMD

[0xFF] [0x06] [0x00]

SET\_ADDRESS = 0x32

sizeof(ARG) = 3

**Figure 40**: This frame commands the nodes at address 8 to assume a new random address with exclusion of addresses 1 and 2.

### The firmware update procedure

With messages of class bootloader / management, it is possible to perform firmware update of the application.

The used messages have CMD equal to: BOARD = 0x00, ADDRESS = 0x01, START = 0x02, DATA = 0x03, END = 0x04. Here is how the procedure works.

The host orders the start of the procedure with two consecutive commands BOARD sent in unicast. If the application receives the first, it forces execution of the bootloader. When the bootloader receives one or two of such messages, it prepares itself for receiving further messages.

Then, the host uses bursts of commands ADDRESS and DATA to deliver the code that the bootloader writes in its code space.

Finally, the host terminates the procedure with messages END and START and it launches the application.

The firmware update procedure

BOARD

delay

YES

BOARD

END

START

Send a CODE section

End of sections

NO

Send a CODE section

n -= (sizeof(PAYLOAD(DATA))-1)

n = ARG(ADDRESS).N

ADDRESS

DATA

n < 0

NO

YES

**Figure 41**: The firmware update procedure.

## The FT service

In here, we configure, start, and stop a FT service. A host at address 0 sends messages to a given strain2 board with address 1.

In here, we assume that the board has already passed through a proper calibration procedure.

### Configuration of FT service

The FT service needs to know the full-scale values imposed at calibration. We can ask the full-scale values, one for each of the six channels, by using the following messages.

Sequence Diagram for retrieval of full scale of channel 2

HOST @ 0

FRAME-1

strain2 @ 1

FRAME-2

The host sends request for the full-scale of channel 2

The strain2 just replies with the full scale value

**Figure 42**: Example of CAN frames used to retrieve the value of full-scale of channel 2 for the FT service.

FRAME-1: it asks the full-scale of channel #2

ID

This frame has an ARG formed by one byte only which is split into two nibbles.

The LSN (lower significant nibble) contains the number of the channel from 0 to 6. In our case, its value is 2. The MSN (most significant nibble) contains the calibration set to query amongst the three possible ones. Valid values are = 0, 1, 2, 3. A value of 0 means the calibration set currently in use by the board.

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[0000b, 0010b]

GET\_FULL\_SCALE = 0x18

sizeof(ARG) = 1

**Figure 43**: The first frame asks the full scale of channel #2.

FRAME-2: reply at the command which asks the full-scale of channel #2

ID

This frame has the same CMD value as FRAME-1, but is it recognizable as an answer because the size of ARG is 3. The content of ARGUMENTS is: CMD[0] is the same as in FRAME-1 and contains the queried calibration set and full-scale; CMD[1, 2] contains the full-scale value as a std::uint16\_t stored in BIG-ENDIAN.

{CLS = POLLING-ANALOG = 010b} {SRC = 0001b} {DST = 0000b} = 0x210

CMD

ARG

[ 0000b, 0001b ] [ MSB ] [ LSB ]

GET\_FULL\_SCALE = 0x18

sizeof(ARG) = 3

**Figure 44**: The second frame sends back the value of full-scale.

After we have the six full-scales, we need to set in advance the data rate of the FT sensor.

Sequence diagram for configuration of the TX rate of FT data

HOST @ 0

FRAME-3

strain2 @ 1

The host configures the strain2 @ addr = 1 to TX at 10 ms rate

The strain2 applies the TX rate to the FT service.

**Figure 45**: Example of CAN frames used to configure the TX rate of FT service.

FRAME-3: it sets the TX rate of the FT service at 10 ms

ID

Explanation of the content of ARG.

* ARG[0] contains the TX datarate expressed in milliseconds.

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[ 0x0A ]

SET\_CANDATARATE = 0x08

sizeof(ARG) = 1

**Figure 46**: This frame sets the TX rate of the FT service at 10 ms.

### Start of the FT service

The start of transmission happens with the sending of a single command. We request calibrated data.

Sequence diagram for start of transmission of calibrated FT data

HOST @ 0

strain2 @ 1

10 ms

FRAME-4

It starts broadcasting the FT calibrated values at the rate of 10 ms as previously configured

FRAME-A

It commands the start of the service with calibrated data.

FRAME-B

FRAME-A

FRAME-B

**Figure 47**: Example of a CAN frames used to start a FT service. Frame-4 is sent in unicast, whereas the group of frames –A and –B are sent in broadcast and periodically.

FRAME-4: command that starts the transmission of calibrated FT data

ID

This frame commands to start transmission of the FT value with a mode specified by ARG[0]. In our case we have ARG[0] = StrainMode::txCalibrated = 0x00, thus the order is to transmit calibrated data.

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[ 0x00 ]

SET\_TXMODE = 0x07

sizeof(ARG) = 1

**Figure 48**: The frame that starts the transmission.

FRAME-A: values of Force

ID

Explanation of the content of DATA: see relevant table

{CLS = PERIODIC-ANALOG = 011b} {SRC = 0001b} {TYP = FORCE\_VECTOR= 0xA} = 0x31A

DATA

[LSB-X] [MSB-X] [LSB-Y] [MSB-Y] [LSB-Z] [MSB-Z] [ADC-SATURATION-INFO]

sizeof(DATA) = 6 or 7

**Figure 49**: These frames contain the calibrated force values.

FRAME-B: values of Torque

ID

Explanation of the content of DATA: see relevant table

{CLS = PERIODIC-ANALOG = 011b} {SRC = 0001b} {TYP = TORQUE\_VECTOR= 0xB} = 0x31B

DATA

[LSB-X] [MSB-X] [LSB-Y] [MSB-Y] [LSB-Z] [MSB-Z] [ADC-SATURATION-INFO]

sizeof(DATA) = 6 or 7

**Figure 50**: These frames contain the calibrated torque values.

### Stop of the FT service

It is required to send a command similar to the one used to start it, but with a different argument.

Sequence diagram for stopping teh FT service

HOST @ 0

strain2 @ 1

FRAME-5

It stops broadcasting whatever FT data it was sending

FRAME-A

It commands the stop of the service.

FRAME-B

**Figure 51**: Example of a CAN frame used to stop an FT service. Frame-5 is sent in unicast.

FRAME-4: command that stops the transmission of calibrated FT data

ID

This frame commands to start transmission of the FT value with a mode specified by ARG[0]. In our case we have ARG[0] = StrainMode::txNone = 0x01, thus the order is to stop transmission.

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[ 0x01 ]

SET\_TXMODE = 0x07

sizeof(ARG) = 1

**Figure 52**: The frame that stops the transmission.

## The IMU service

In here, we configure, start, and stop an IMU service. A host at address 0 sends messages to a given strain2 board with address 1.

### Configuration of IMU service

In here, we configure an IMU service with accelerometer measures, gyroscope measures and the status of the IMU.

Sequence diagram for configuration of an IMU service

HOST @ 0

FRAME-1

strain2 @ 1

FRAME-3

FRAME-2

The host configures the strain2 @ addr = 1 for IMU acquisition of accelerometer and gyroscope plus the IMU status

The strain2 applies the settings of the IMU service

It replies to the request and sends back the settings of IMU service.

It asks back the configuration of the IMU for check.

**Figure 53**: Example of CAN frames used to configure an IMU service. Frames from -1 to -3 are sent in unicast.

FRAME-1: command that sets the IMU configuration

ID

Explanation of the content of ARG: see relevant table.

* ARG[0] is LSB and ARG[1] is MSB of sensormask. The used flags are imuSensor::acc = 0, imuSensor::gyr = 2 and imuSensor::status = 15.
* ARG[2] is mode. The used value is 0x01 which means the the IMU uses standard data fusion.
* ARG[3] is the orientation. The used value is 0x00 which means that the IMU uses its standard orientation.
* ARG[4], ARG[5], ARG[6] are for future use. It is OK if = 0.

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[ 00000101b ] [ 10000000b ] [ 0x01 ] [ 0x00 ] [ 0x00 ] [ 0x00 ] [ 0x00 ]

IMU\_CONFIG\_SET = 0x34

sizeof(ARG) = 7

**Figure 54**: The first frame sets the IMU for the accelerometer, gyroscope and its status.

FRAME-2: command that asks the IMU configuration

ID

This frame does not need any arguments.

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

[]

IMU\_CONFIG\_GET = 0x33

sizeof(ARG) = 0

**Figure 55**: The second frame asks the configuration for a check.

FRAME-3: reply at the command which asks the IMU configuration

ID

This frame has the same CMD value as FRAME-2, but is it recognizable as an answer because the size of ARGUMENTS is 8. The content of ARGUMENTS is the same as for IMU\_CONFIG\_SET

{CLS = POLLING-ANALOG = 010b} {SRC = 0001b} {DST = 0000b} = 0x210

CMD

ARG

[ 00000101b ] [ 10000000b ] [ 0x01 ] [ 0x00 ] [ 0x00 ] [ 0x00 ] [ 0x00 ]

IMU\_CONFIG\_GET = 0x33

sizeof(ARG) = 8

**Figure 56**: The third frame send the configuration back.

### Start of IMU service

In here, we start the transmission of the data from accelerometer, gyroscope and the status of the IMU.

Sequence diagram for start of an IMU service

HOST @ 0

strain2 @ 1

50 ms

FRAME-4

It starts broadcasting the IMU values which was configured for, at a rate of 50 ms

FRAME-A

It commands the start of the service with a rate of 50 ms.

FRAME-B

FRAME-C

FRAME-A

FRAME-B

FRAME-C

**Figure 57**: Example of a CAN frames used to start an IMU service. Frame-4 is sent in unicast, whereas the group of frames -A, -B, and -C are sent in broadcast and periodically.

FRAME-4: command that starts the transmission @ 20 Hz

ID

This frame commands to start transmission of the configured IMU values with a frequency specified in milliseconds by ARG[0]. In our case is ARG[0] = 0x32, thus 50 ms.

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[ 0x32 ]

IMU\_TRANSMIT = 0x35

sizeof(ARG) = 1

**Figure 58**: The frame that starts the transmission.

FRAME-A: values from an accelerometer

ID

Explanation of the content of DATA: see relevant table.

* DATA[0] is a sequence number which helps matching other inertial values acquired at the same time. Its value SEQ cycles from 0 to 255.
* DATA[1] is a descriptor of the type of inertial content. In here we have 0x00 = imuSensor::acc, hence we have values from an accelerometer.
* DATA[2] and DATA[3] represent the x component as an std::int16\_t value mapped in little endian.
* DATA[4] and DATA[5] represent the y component as an std::int16\_t value mapped in little endian.
* DATA[6] and DATA[7] represent the z component as an std::int16\_t value mapped in little endian.

{CLS = PERIODIC-INERTIAL = 101b} {SRC = 0001b} {TYP = IMU\_TRIPLE = 0x3} = 0x513

DATA

[ SEQ ] [ 0x00 ] [LSB-X] [MSB-X] [LSB-Y] [MSB-Y [LSB-Z] [MSB-Z]

sizeof(DATA) = 8

**Figure 59**: These frames contain the accelerometer values and are repeated every 50 ms.

FRAME-B: values from a gyroscope

ID

Explanation of the content of DATA: see relevant table.

* DATA[0] is a sequence number which helps matching other inertial values acquired at the same time. Its value SEQ cycles from 0 to 255.
* DATA[1] is a descriptor of the type of inertial content. In here we have 0x02 = imuSensor::gyr, hence we have values from a gyroscope.
* DATA[2] and DATA[3] represent the x component as an std::int16\_t value mapped in little endian.
* DATA[4] and DATA[5] represent the y component as an std::int16\_t value mapped in little endian.
* DATA[6] and DATA[7] represent the z component as an std::int16\_t value mapped in little endian.

{CLS = PERIODIC-INERTIAL = 101b} {SRC = 0001b} {TYP = IMU\_TRIPLE = 0x3} = 0x513

DATA

[ SEQ ] [ 0x02 ] [LSB-X] [MSB-X] [LSB-Y] [MSB-Y [LSB-Z] [MSB-Z]

sizeof(DATA) = 8

**Figure 60**: These frames contain the gyroscope values and are repeated every 50 ms.

FRAME-C: the status of the IMU

ID

Explanation of the content of DATA: see the relevant table

{CLS = PERIODIC-INERTIAL = 101b} {SRC = 0001b} {TYP = IMU\_STATUS = 0x5} = 0x515

DATA

[ SEQ ] [ GYR-CALIB ] [ ACC-CALIB ] [ MAG-CALIB] [ - ] [ - ] [ - ] [ - ]

sizeof(DATA) = 8

**Figure 61**: These frames contain the status of the IMU and are repeated every 50 ms.

### Stop of IMU service

In here, we stop the transmission of IMU data.

Sequence diagram for stop of an IMU service

HOST @ 0

strain2 @ 1

FRAME-5

It stops broadcasting whatever IMU data it was sending

FRAME-A

It commands the stop of the service.

FRAME-B

FRAME-C

**Figure 62**: Example of a CAN frames used to stop an IMU service. Frame-5 is sent in unicast.

FRAME-5: command that stops the transmission of IMU data

ID

This frame commands to start transmission of the configured IMU values with a frequency specified in milliseconds by ARG[0]. In our case is ARG[0] = 0x00, thus no transmission at all.

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[ 0x00 ]

IMU\_TRANSMIT = 0x35

sizeof(ARG) = 1

**Figure 63**: The frame that stops the IMU transmission.

## The THERMO service

In here, we configure, start, and stop a thermometer service. A host at address 0 sends messages to a given strain2 board with address 1.

### Configuration of THERMO service

In here, we configure a thermometer service with acquisition of a single temperature inside the board.

Sequence diagram for configuration of a thermometer service

HOST @ 0

FRAME-1

strain2 @ 1

FRAME-3

FRAME-2

The host configures the strain2 @ addr = 1 for acquiring from one thermometer

The strain2 applies the settings of the Thermo service

It replies to the request and sends back the settings of Thermo service.

It asks back the configuration of the Thermo for check.

**Figure 64**: Example of CAN frames used to configure a Thermo service. Frames from -1 to -3 are sent in unicast.

FRAME-1: command that sets the THERMO configuration

ID

Explanation of the content of ARG.

* ARG[0] contains the thermomask. The used flags so far are only: Thermo::one = 0.

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[ 00000001b ]

THERMO\_CONFIG\_SET = 0x39

sizeof(ARG) = 1

**Figure 65**: The first frame sets the chosen thermometer.

FRAME-2: command that asks the THERMO configuration

ID

This frame does not need any arguments.

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

[]

THERMO\_CONFIG\_GET = 0x38

sizeof(ARG) = 0

**Figure 66**: The second frame asks the configuration for a check.

FRAME-3: reply at the command that asks the IMU configuration

ID

This frame has the same CMD value as FRAME-2, but is it recognizable as an answer because the size of ARGUMENTS is 1. The content of ARGUMENTS is the same as for THERMO\_CONFIG\_SET

{CLS = POLLING-ANALOG = 010b} {SRC = 0001b} {DST = 0000b} = 0x210

CMD

ARG

[ 00000001b ]

THERMO\_CONFIG\_GET = 0x38

sizeof(ARG) = 1

**Figure 67**: The third frame send the configuration back.

### Start of THERMO service

In here, we start the transmission of the data from the thermometer.

Sequence Diagram for start of a THERMO service

HOST @ 0

strain2 @ 1

2 sec

FRAME-4

It starts broadcasting the THERMO values which was configured for, at a rate of 2 sec

FRAME-A

It commands the start of the service with a rate of 2 sec.

FRAME-A

**Figure 68**: Example of a CAN frames used to start a THERMO service. Frame-4 is sent in unicast, whereas the frames -A are sent in broadcast and periodically.

FRAME-4: command that starts the transmission @ 0.5 Hz

ID

This frame commands to start transmission of the configured THERMO values with a frequency specified in seconds by ARG[0]. In our case is ARG[0] = 0x02, thus 2 seconds.

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[ 0x02 ]

THERMO\_TRANSMIT = 0x3A

sizeof(ARG) = 1

**Figure 69**: The frame that starts the transmission.

FRAME-A: values from a thermometer

ID

Explanation of the content of DATA.

* DATA[0] contains the mask telling which thermometer(s) we are dealing with. The used flags so far are only: Thermo::one = 0.
* DATA[1, 2] contains the value of the temperature expressed in 0.1 Celsius degrees by a std::int16\_t ordered in little endian.
* DATA[3, 4] are for future use.

{CLS = PERIODIC-ANALOG = 011b} {SRC = 0001b} {TYP = THERMO\_MEASURE = 0xE} = 0x31E

DATA

[MASK] [LSB-T] [MSB-T] [FFU] [FFU]

sizeof(DATA) = 5

**Figure 70**: These frames contain the temperature values and are repeated every 2 seconds.

### Stop of THERMO service

In here, we stop the transmission of THERMO data.

Sequence Diagram for stop of a THERMO service

HOST @ 0

strain2 @ 1

FRAME-5

It stops broadcasting whatever THERMO data it was sending

FRAME-A

It commands the stop of the service.

**Figure 71**: Example of a CAN frames used to stop a THERMO service. Frame-5 is sent in unicast.

FRAME-5: command that stops the transmission of THERMO data

ID

This frame commands to start transmission of the configured THERMO values with a frequency specified in seconds by ARG[0]. In our case is ARG[0] = 0x00, thus no transmission at all.

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[ 0x00 ]

THERMO\_TRANSMIT = 0x3A

sizeof(ARG) = 1

**Figure 72**: The frame that stops the THERMO transmission.

## The SKIN service

In here, we configure, start, and stop a skin service. A host at address 0 sends messages to a given mtb4 board with address 1.

### Configuration of the SKIN service

In here, we configure a skin service to stream the values of two triangles only at a 50 ms rate.

Sequence diagram for configuration of a skin service

HOST @ 0

FRAME-1

mtb4 @ 1

FRAME-3

FRAME-2

The host send to the mtb4 @ addr = 1 the configuration common to every triangle

The mtb4 applies the general settings for skin acquisition

It disables all triangles.

It now sends command to disable all triangles.

And command to enable only two triangles

It enables the two triangles.

**Figure 73**: Example of CAN frames used to configure two triangles on an mtb4 board.

FRAME-1: command that configures the board for skin acquisition

ID

Explanation of the content of ARG.

* ARG[0] contains the compensation type: SkinComp::withTemperatureCompensation = 0.
* ARG[1] contains the tx rate: 50 ms.
* ARG[2] contains the no-load value

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[ 0x00 ] [ 0x32 ] [ 0xF5 ]

SKIN\_SET\_BRD\_CFG = 0x4D

sizeof(ARG) = 3

**Figure 74**: The first frame sets the configuration of skin service that are common to every triangle.

FRAME-2: command that disable all triangles

ID

Explanation of the content of ARG.

* ARG[0] contains the first triangle upon which apply the command: 0.
* ARG[1] contains last triangle upon which apply the command: 15.
* ARG[2] contains shift value: 2
* ARG[3] contains the enable flag: 0, hence we disable all the selected triangles
* ARG[4, 5] contain the CDC offset: 0x2200 in little endian

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[ 0x00 ] [ 0x0F ] [ 0x02 ] [ 0x00 ] [ 0x2200 ]

SET\_TRIANG\_CFG = 0x50

sizeof(ARG) = 6

**Figure 75**: The second frame disables all the triangles.

FRAME-3: command that enables the first two triangles

ID

Explanation of the content of ARG.

* ARG[0] contains the first triangle upon which apply the command: 0.
* ARG[1] contains last triangle upon which apply the command: 1.
* ARG[2] contains shift value: 2
* ARG[3] contains the enable flag: 1, hence we disable the selected triangles
* ARG[4, 5] contain the CDC offset: 0x2200 in little endian

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[ 0x00 ] [ 0x01 ] [ 0x02 ] [ 0x01 ] [ 0x2200 ]

SET\_TRIANG\_CFG = 0x50

sizeof(ARG) = 6

**Figure 76**: The third frame configures only two triangles.

### Start of the SKIN service

The start of transmission happens with the sending of a single command.

Sequence diagram for start of transmission of skin data

HOST @ 0

mtb4 @ 1

50 ms

FRAME-4

It starts broadcasting the skin values at the rate of 50 ms as previously configured

FRAME-A

It commands the start of the service.

FRAME-B

FRAME-C

FRAME-D

**Figure 77**: Example of a CAN frames used to start a skin service. Frame-4 is sent in unicast, whereas the group of frames -A, -B, -C, and -D are sent in broadcast and periodically.

FRAME-4: command that starts the transmission of skin data

ID

This frame commands to start transmission of the skin values with ARG[0] = SkinMode::txData = 0

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[ 0x00 ]

SET\_TXMODE = 0x07

sizeof(ARG) = 1

**Figure 78**: The frame that starts the transmission.

FRAME-A: the first triangle, points from 0 to 6

ID

Explanation of the content of DATA: see relevant table

{CLS = PERIODIC-SKIN = 100b} {SRC = 0001b} {TYP = TRG00= 0x0} = 0x410

DATA

[0x40] [P0] [P1] [P2] [P3] [P4] [P5] [P6]

sizeof(DATA) = 8

**Figure 79**: These frames contain the values of the first seven points of the first triangle.

FRAME-B: the first triangle, points from 7 to 11

ID

Explanation of the content of DATA: see relevant table

{CLS = PERIODIC-SKIN = 100b} {SRC = 0001b} {TYP = TRG00= 0x0} = 0x410

DATA

[0xC0] [P7] [P8] [P9] [P10] [P11] [STATUS0] [STATUS1]

sizeof(DATA) = 8

**Figure 80**: These frames contain the values of the remaining five points of the first triangle.

FRAME-C: the second triangle, points from 0 to 6

ID

Explanation of the content of DATA: see relevant table

{CLS = PERIODIC-SKIN = 100b} {SRC = 0001b} {TYP = TRG01= 0x1} = 0x411

DATA

[0x40] [P0] [P1] [P2] [P3] [P4] [P5] [P6]

sizeof(DATA) = 8

**Figure 81**: These frames contain the values of the first seven points of the second triangle.

FRAME-B: the second triangle, points from 7 to 11

ID

Explanation of the content of DATA: see relevant table

{CLS = PERIODIC-SKIN = 100b} {SRC = 0001b} {TYP = TRG01= 0x1} = 0x411

DATA

[0xC0] [P7] [P8] [P9] [P10] [P11] [STATUS0] [STATUS1]

sizeof(DATA) = 8

**Figure 82**: These frames contain the values of the remaining five points of the second triangle.

### Stop of the SKIN service

In here, we stop the skin service. It is required to send a command similar to the one used to start it, but with a different argument.

Sequence diagram for stopping the skin service

HOST @ 0

mtb4 @ 1

FRAME-5

It stops broadcasting whatever skin data it was sending

FRAME-A

It commands the stop of the service.

FRAME-B

**Figure 83**: Example of a CAN frame used to stop a skin service. Frame-5 is sent in unicast.

FRAME-5: command that stops the transmission of skin data

ID

This frame commands to stop transmission of the skin value with ARG[0] = SkinMode::txNone = 1.

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[ 0x01 ]

SET\_TXMODE = 0x07

sizeof(ARG) = 1

**Figure 84**: The frame that stops the transmission.

## Advanced configuration of the strain2 for FT service

In here, we show how we can configure the strain2 board for its advanced use: use of a different regulation set, tuning of analog front end, etc.

### Use of a different regulation set at bootstrap

It is possible to choose one of three regulation sets. Let us suppose that the strain2 at bootstrap loads regulation set regSet::first. In here, we verify this value and we impose regSet::second as the new value of the regulation set at bootstrap.

Sequence Diagram for retrieval of the regulation set at bootstrap

HOST @ 0

FRAME-1

strain2 @ 1

FRAME-2

The host sends request for the the regset@boot

The strain2 just replies with the value

**Figure 85**: Example of CAN frames used to retrieve value of regulation set at bootstrap.

FRAME-1: it asks the regulation set at bootstrap

ID

The least significant nibble is = 1 = regSetvalidity::permanent

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[0x0, 0x1]

REGULATIONSET\_GET = 0x3E

sizeof(ARG) = 1

**Figure 86**: The first frame asks the regulation set at bootstrap.

FRAME-1: it sends back the value of the regulation set at bootstrap

ID

The least significant nibble is = 1 = regSetvalidity::permanent. The most significant nibble is 1 = regSet::first

{CLS = POLLING-ANALOG = 010b} {SRC = 0001b} {DST = 0000b} = 0x210

CMD

ARG

[0x1, 0x1]

REGULATIONSET\_GET = 0x3E

sizeof(ARG) = 1

**Figure 87**: The second frame sends back the value of the regulation set at bootstrap.

Then, we can assign the new value of the regulation set at bootstrap to be = regSet::second.

Sequence diagram for changing the value of regulation set at bootstrap

HOST @ 0

FRAME-3

strain2 @ 1

The host sends a new value for regset @ boot

The strain2 applies the new value regset @ boot in RAM.

**Figure 88**: Example of CAN frames used to change the value of regulation set at bootstrap.

FRAME-3: it sets the value of regulation set at bootstrap

ID

The least significant nibble is = 1 = regSetvalidity::permanent. The most significant nibble is 2 = regSet::second

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[ 0x2, 0x1 ]

REGULATIONSET\_SET = 0x3D

sizeof(ARG) = 1

**Figure 89**: This frame sets the regulation set at bootstrap to the value regSet::second.

Finally, one must save the changes to EEPROM. See next section for details of messages.

Sequence diagram for saving all the regulation sets in EEPROM

HOST @ 0

strain2 @ 1

The host sends a request to save to EEPROM the regulation sets.

The strain2 saves the regulation sets to EEPROM and sends an ACK

FRAME-save2eeprom

FRAME-save2eeprom-ack

**Figure 90**: Example of CAN frames used to save the regulation sets in EEPROM.

After the save to EEPROM operation, at the next bootstrap, the strain2 board will load the second regulation set. However, at this stage the strain2 is still using the first regulation set.

We can force to use the second regulation set with a command that operates on the temporary regulation set. See one of the next examples.

### Saving to EEPROM the changes done on a regulation set

After any command that changes the regulation set, all the changes surely are effective but are saved only in the RAM copy. If we want to permanently store them in EEPROM we need to send the proper message.

Sequence diagram for saving all the regulation sets in EEPROM

HOST @ 0

strain2 @ 1

The host sends a request to save to EEPROM the regulation sets.

The strain2 saves the regulation sets to EEPROM and sends an ACK

FRAME-save2eeprom

FRAME-save2eeprom-ack

**Figure 91**: Example of CAN frames used to save the regulation sets in EEPROM.

FRAME-save2eeprom: it synchronizes the full FT configuration from RAM to EEPROM

ID

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

[]

SAVE2EE = 0x09

sizeof(ARG) = 0

**Figure 92**: This frame synchronizes the full FT configuration from RAM to EEPROM.

FRAME-save2eeprom-ack: it sends result of save to EEPROM request

ID

{CLS = POLLING-ANALOG = 010b} {SRC = 0001b} {DST = 0000b} = 0x210

CMD

ARG

[0x01]

SAVE2EE = 0x09

sizeof(ARG) = 1

0x01 is OK

**Figure 93**: This frame sends ACK to save to EEPROM command.

### Use of a different temporary regulation set

It is possible to choose a different regulation set only in temporary way. The new regulation set will be in use only until a new change of the temporary regulation set and at latest until the next bootstrap.

We can force the strain2 to use the third regulation in the following way.

Sequence diagram for changing the value of regulation set in use

HOST @ 0

FRAME-4

strain2 @ 1

The host sends a new value for regset in use

The strain2 applies the new regulation set to its analog and digital stages

**Figure 94**: Example of CAN frames used to change the value of the regulation set in use.

FRAME-4: it sets the value of regulation set in use

ID

The least significant nibble is = 0 = regSetvalidity::temporary. The most significant nibble is 2 = regSet::third

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[ 0x3, 0x0 ]

REGULATIONSET\_SET = 0x3D

sizeof(ARG) = 1

**Figure 95**: This frame sets the regulation set in use to be regSet::third.

At this stage, the strain2 is using the third regulation set.

We can verify in the following way.

Sequence Diagram for retrieval of the regulation set in use

HOST @ 0

FRAME-5

strain2 @ 1

FRAME-6

The host sends request for the regset in use

The strain2 just replies with the value

**Figure 96**: Example of CAN frames used to retrieve value of regulation set in use.

FRAME-5: it asks the regulation set in use

ID

The least significant nibble is = 0 = regSetvalidity::temporary

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[0x0, 0x0]

REGULATIONSET\_GET = 0x3E

sizeof(ARG) = 1

**Figure 97**: The first frame asks the regulation set at bootstrap.

FRAME-6: it sends back the value of the regulation set in use

ID

The least significant nibble is = 0 = regSetvalidity::temporary. The most significant nibble is 3 = regSet::third

{CLS = POLLING-ANALOG = 010b} {SRC = 0001b} {DST = 0000b} = 0x210

CMD

ARG

[0x3, 0x0]

REGULATIONSET\_GET = 0x3E

sizeof(ARG) = 1

**Figure 98**: The second frame sends back the value of the regulation set in use.

### Tuning of the analog front end of a given regulation set

In here, we show how to tune the analog front end of the third regulation set. We shall apply a gain = 48 to every channel apart one, Channel::two, where we apply gain = 16. The offset will be the mid-range one: 32K-1.

For these operations, we use the CAN messages that operates directly on the registers of the PGA308 amplifiers. In addition, we suppose that the regulation set in use is the regSet::third.

Sequence Diagram for changing the values of the PGA308 registers

HOST @ 0

FRAME-1

strain2 @ 1

FRAME-2

The host wants to apply the same settings to every amplifier

Now operates on one amplifier only

The strain2 applies the values to every amplifier

It now applies value to one amplifier only

**Figure 99**: Example of CAN frames used to change the values of gain and offset of the amplifiers with two different values.

FRAME-1: it sets the registers of every amplifier

ID

The least significant nibble of first byte is = 0xf = Channel::all and allows application of the message to every channel

The most significant nibble of first byte is = 0x3 = regSet::three and ask application of the message to the third regulation set

All the remaining 6 bytes contain the register’s values to force gain = 48 and offset = 32767. See relevant table

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[0x3, 0xf] [0x00] [0x40] [0x46] [0x1f] [0xb1] [0x7f]

AMPLIFIER\_PGA308\_CFG1\_SET = 0x2B

sizeof(ARG) = 7

**Figure 100**: The first frame set a gain = 48 and offset = 32767 to every amplifier of the third regulation set.

FRAME-2: it sets the registers of only one amplifier

ID

The least significant nibble of first byte is = 0x2 = Channel::two and allows application of the message to channel 2

The most significant nibble of first byte is = 0x0 = regSet::theoneinuse and ask application of the message to the regulation set in use, which is the third

All the remaining 6 bytes contain the register’s values to force gain = 16 and offset = 32767. See relevant table

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[0x0, 0x2] [0x00] [0x00] [0x42] [0x5e] [0x72] [0x80]

AMPLIFIER\_PGA308\_CFG1\_SET = 0x2B

sizeof(ARG) = 7

**Figure 101**: The first frame set a gain = 16 and offset = 32767 to the amplifier number two (the third) of the regulation set in use, which we know to be the third.

At this stage, all the changes to the analog front end are effective but are saved only in the RAM copy. If we want to permanently store them in EEPROM we need to send the proper message.

Sequence diagram for saving all the regulation sets in EEPROM

HOST @ 0

strain2 @ 1

The host sends a request to save to EEPROM the regulation sets.

The strain2 saves the regulation sets to EEPROM and sends an ACK

FRAME-save2eeprom

FRAME-save2eeprom-ack

**Figure 102**: Example of CAN frames used to save the regulation sets in EEPROM.

### Tune the offset of the analog front end to move the ADC acquisition at midrange

After the analog front-end is assigned with gain and default offset values for its amplifiers, the ADC acquisition in the idle position may require being centred ad midrange.

Here is the way to do that.

Sequence diagram for autotune of the offset of amplifier to midrange acquisition

HOST @ 0

strain2 @ 1

The host sends the autotune command

The strain2 runs an algorithm which sets the correct offset value to match a target ADC value

FRAME-autotune

FRAME-autotune-result

**Figure 103**: Example of CAN frames used to run the autotune procedure.

FRAME-autotune: starts autotune on all amplifiers of the reg set in use

ID

The least significant nibble of first byte is = 0xf = Channel::all and allows application of the message to every channel

The most significant nibble of first byte is = 0x0 = regSet::theoneinuse and ask application of the message to the regulation set in use

All the remaining 6 bytes contain the parameters of the algorithm. The target ADC value is 0x8000 = 32K (midrange), the tolerance is 0x0100 = 256, and the last byte is 0x0a = 10 and tells to average 10 acquisition ADC sample to filter noise away.

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[0x0, 0xf] [0x00] [0x00] [0x80] [0x00] [0x01] [0x0a]

AMPLIFIER\_OFFSET\_AUTOCALIB = 0x22

sizeof(ARG) = 7

**Figure 104**: The first frame starts the autotune of the offset of every amplifier in the regulation set in use. The ADC value which the algorithm must produce is = 32K.

FRAME-autotune-result: runs autotune on all amplifiers of the reg set in use

ID

The least significant nibble of first byte is = 0xf = Channel::all and allows application of the message to every channel

The most significant nibble of first byte is = 0x0 = regSet::theoneinuse and ask application of the message to the regulation set in use

All the remaining 6 bytes contain the results of the algorithm. The second byte tells that there were no noisy ADC channels, the third byte tells that the algorithm was successful to all the six channels, the fourth byte tells that the final ADC measure matched the target for the six channels, and finally the last two bytes give the mean absolute error = 8.

{CLS = POLLING-ANALOG = 010b} {SRC = 0001b} {DST = 0000b} = 0x210

CMD

ARG

[0x0, 0xf] [0x00] [0x3f] [0x3f] [0x00] [0x08] [0x00]

AMPLIFIER\_OFFSET\_AUTOCALIB = 0x22

sizeof(ARG) = 7

**Figure 105**: This frame contains the result of the autotune procedure.

At the end of this procedure, the ADC values obtained from acquisition will be all equal to 32K.

If one wants to save the settings to EEPROM, we remind the save to EEPROM procedure.

Sequence diagram for saving all the regulation sets in EEPROM

HOST @ 0

strain2 @ 1

The host sends a request to save to EEPROM the regulation sets.

The strain2 saves the regulation sets to EEPROM and sends an ACK

FRAME-save2eeprom

FRAME-save2eeprom-ack

**Figure 106**: Example of CAN frames used to save the regulation sets in EEPROM.

## The POS service

In here, we configure, start, and stop a POS service. A host at address 0 sends messages to a given psc board with address 1.

### Configuration of POS service

In here, we configure a POS service with acquisition of two angles expressed in deci-degrees inside the board.

Sequence diagram for configuration of a position service

HOST @ 0

FRAME-1

psc @ 1

FRAME-3

FRAME-2

The host configures the psc @ addr = 1 for acquiring two angles

The psc applies the settings of the service

It replies to the request and sends back the settings of service.

It asks back the configuration for check.

**Figure 107**: Example of CAN frames used to configure a POS service. Frames from -1 to -3 are sent in unicast.

FRAME-1: command that sets the POS configuration

ID

Explanation of the content of ARG.

* ARG[0] tells that we are using DeciDeg.
* ARG[1, 2, 3] contains the config of the first angle with label 0x4, and zero = 0x0000 = 0.0 deg. The nibble Q is 0b1000 and tells: E = 1 enabled, D = 0 the same direction, R = 0 no rotation.
* ARG[4, 5, 6] contains the config of the second angle with label 0x5, and zero = 0xFED2 = -302 (= -30.2 deg) The nibble R is 0b1101 and tells: E = 1 enabled, D = 1 inverted direction, R = 1 180 deg rotation.

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[ 0x00 ] {[0xQ4] [0x00] [0x00]}{[0xR5] [0xD2] [0xFE] } [0x00] [0x00]

POS\_CONFIG\_SET = 0x52

sizeof(ARG) = 7

**Figure 108**: The first frame configures two angles. The two angles have labels = 4 and 5. The first angle does not need any compensation. The second needs: phase rotation of 180 degrees, direction inversion, an offset of -30.20 degrees.

FRAME-2: command that asks the POS configuration

ID

This frame does not need any arguments.

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

[]

POS\_CONFIG\_GET = 0x51

sizeof(ARG) = 0

**Figure 109**: The second frame asks the configuration for a check.

FRAME-3: reply at the command that asks the POS configuration

ID

This frame has the same CMD value as FRAME-2, but is it recognizable as an answer because the size of ARGUMENTS is 8. The content of ARG is the same as for POS\_CONFIG\_SET

{CLS = POLLING-ANALOG = 010b} {SRC = 0001b} {DST = 0000b} = 0x210

CMD

ARG

[ 0x00 ] {[0xQ4] [0x00] [0x00]}{[0xR5] [0xD2] [0xFE] } [0x00] [0x00]

POS\_CONFIG\_GET = 0x51

sizeof(ARG) = 8

**Figure 110**: The third frame send the configuration back.

### Start of POS service

In here, we start the transmission of the angles.

Sequence Diagram for start of a POS service

HOST @ 0

psc @ 1

50 ms

FRAME-4

It starts broadcasting the POS values which was configured for, at a rate of 50 ms

FRAME-A

It commands the start of the service with a rate of 50 ms.

FRAME-A

**Figure 111**: Example of a CAN frames used to start a POS service. Frame-4 is sent in unicast, whereas the frames -A are sent in broadcast and periodically.

FRAME-4: command that starts the transmission @ 50 ms rate

ID

This frame commands to start transmission of the configured POS values with a frequency specified in ms by ARG[0]. In our case is ARG[0] = 0x32, thus 50 ms.

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[ 0x32 ]

POS\_TRANSMIT = 0x53

sizeof(ARG) = 1

**Figure 112**: The frame that starts the transmission.

FRAME-A: angle values

ID

Explanation of the content of DATA.

* DATA[0] contains the TYPE of POS data. We have DeciDeg, hence 0x00.
* DATA[1] contains two nibble: the most significant in = 0x4 and is the label of the first value. The least significant is 0x2 and tells that we have two values.
* DATA[2, 3] contains the value of the first angle (with label = 4) expressed in int16\_t deci-degrees.
* DATA[4, 5] contains the value of the second angle (with label = 5) expressed in int16\_t deci-degrees.
* Other bytes are zero.

{CLS = PERIODIC-ANALOG = 011b} {SRC = 0001b} {TYP = POS\_MEASURE = 0x1} = 0x311

DATA

[0x00] [0x42] [LSB-P0] [MSB-P0] [LSB-P0] [MSB-P0] [0x00] [0x00]

sizeof(DATA) = 8

**Figure 113**: These frames contain the angles values and are repeated every 50 ms.

### Stop of POS service

In here, we stop the transmission of POS data.

Sequence Diagram for stop of a POS service

HOST @ 0

psc @ 1

FRAME-5

It stops broadcasting whatever POS data it was sending

FRAME-A

It commands the stop of the service.

**Figure 114**: Example of a CAN frames used to stop a POS service. Frame-5 is sent in unicast.

FRAME-5: command that stops the transmission of POS data

ID

This frame commands to start transmission of the configured POS values with a frequency specified in seconds by ARG[0]. In our case is ARG[0] = 0x00, thus no transmission at all.

{CLS = POLLING-ANALOG = 010b} {SRC = 0000b} {DST = 0001b} = 0x201

CMD

ARG

[ 0x00 ]

POS\_TRANSMIT = 0x53

sizeof(ARG) = 1

**Figure 115**: The frame that stops the POS transmission.

## The BAT service

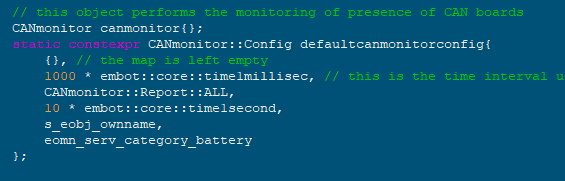
In here, we describe the ***Battery*** service. This service cannot be currently configured by xml file but a standard (default) configuration is defined in the code of the ***embot\_app\_eth\_theBATservice***. The compatible boards are BAT and BMS\_18239, which start the transmission of their periodic messages at a given rate after they receive the **ENABLE** command through **ICUBCANPROTO\_POL\_AS\_CMD\_\_SET\_TXMODE** polling message from the EMS board when the ***yarprobotinterface*** opens correctly except for the BMS (with Wingst code 15586) that starts the period messages transition as soon as it powers on.

Currently the BAT’s transmission period is not configurable. (Default is 10 millisec)

Similarly, the CAN transmission of BAT and BMS can be stopped by the EMS board that sends on the same polling message the **DISABLE** command. The configurability by xml file is WIP.

### Configuration of BAT service

That is for future implementation. Currently the default configuration for the ***canmonitor*** we are setting for the BAT and BMS that share the ***service embot\_app\_eth\_theBATservice.cpp*** is the following:



Where we are basically streaming each CAN message with a frequency of 10 Hz and each second the EMS board does a touch on the CAN boards to check if they are still transmitting. Moreover, the report message when a transmission is continuously lost is made each 10 seconds (first time a CAN board is lost or retrieved after being lost the report is instantly sent, then every 10 seconds).

### Start of BAT service

Currently, both BMS\_18239 and BAT boards, starts silently. Their transmission is then started by the EMS board when they receive the transmission enable command. The old version of the BMS, i.e. BMS\_15586, cannot do that but it is always streaming on the CAN bus with a frequency of 1Hz.

Sequence Diagram for streaming of BMS/BAT periodic CAN frames

HOST @ 0

BMS/BAT @ 2

**100** ms. This frequency is defined for BMS\_18239 and for the BAT. Moreover, for the BMS\_18239 is possible to re-configure the transmission rate through a specific message. Then the BMS\_15586 always sends messages list at **1** Hz without being previously enabled.

It starts broadcasting after receiving enable command through the CAN frame SEND\_TXMODE from EMS with a frequency of 10 Hz (this frequency is defined for the BMS\_18239 and for the BAT.)

List of CAN frames

List of CAN frames

SET\_TXMODE

**Figure 116**: Example of the list of CAN frames used to give info in a BAT service. Each transmission interval (by default 100ms) BAT and BMS\_18239 send periodically the whole list of all the configured CAN frames. Differently, BMS\_15586 sends messages with a frequency of 1Hz. Moreover, BMS\_18239 can be configured to send the list of CAN frames at a custom frequency, which can be configured through a specific message.

### Logging of error and diagnostic messages

Sequence Diagram for BMS\_DIAGN\_LOG\_INFO CAN frame in BMS\_18239

HOST @ 0

BMS\_18239 @ 2

Each new entry, if present, is sent one after the other until all the errors happened in the last run and requested by the EMS are transmitted.

It starts sending the list of errors happened in the last run as soon as the command **getDiagnosticLogEntry** has been received from EMS.

New error entry

Last error entry

**getDiagnosticLogEntry**

**Figure 117**: Example of the **getDiagnosticLogEntry** used by the EMS, at the restart, for asking for the list of errors and diagnostic messages, if present, got in the last run on the BMS\_18239. After the EMS starts the request the BMS\_18239 sends all the errors entries saved on the board until the whole list of errors in the last run gets cleaned.

### Stop of BAT service

The emission of BMS\_15586 messages cannot be stopped. Currently only the new BMS\_18239 and the BAT boards can stop the transmission of their messages after receiving, from the EMS, the transmission disable command. After that, they remain silent.

Conclusions

In here we have presented the iCub CAN protocol for sensor boards.

References

[1] CANLOADER MANUAL, RBCS, 2008

[2] MTB user manual, iCub Facility, 2013

[3] USAGE MANUAL: SIX AXIS TORQUE FORCE SENSOR BOARD (STRAIN), RBCS, 2008

[4] Multiple Analog Interface for Hall Effect Sensors (MAIS), RBCS, 2008

[5] PGA308: User's Guide, Texas Instruments, Literature Number: SBOU069B June 2009–Revised January 2012.

[6] BNO055: Intelligent 9-axis absolute orientation sensor, Bosch, June 2016, revision 1.4