CAN protocol for the strain2

This document describes the iCub CAN protocol with focus on the strain2 board.

Approval History

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Version | Author | | Date | Approved | | Date |
| 1.0 | M. Accame | IIT - HSP |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Comments |
| 1.0 | 14 Nov 18 | M. Accame | First WIP version. |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[1 Introduction 1](#_Toc530740123)

[2 The iCub CAN protocol 2](#_Toc530740124)

[2.1 The structure of the CAN frame 2](#_Toc530740125)

[2.1.1 The ID 3](#_Toc530740126)

[2.1.2 The PAYLOAD 3](#_Toc530740127)

[2.2 The iCub network 3](#_Toc530740128)

[2.2.1 The iCub host 4](#_Toc530740129)

[2.2.2 The iCub CAN nodes 5](#_Toc530740130)

[2.2.3 The used communication modes 7](#_Toc530740131)

[2.2.4 Frames accepted and parsed 10](#_Toc530740132)

[2.3 The iCub CAN protocol classes 11](#_Toc530740133)

[2.3.1 The command classes 11](#_Toc530740134)

[2.3.2 The streaming classes 12](#_Toc530740135)

[2.4 The TYP of the streaming classes 13](#_Toc530740136)

[2.4.1 Class periodic motor control 13](#_Toc530740137)

[2.4.2 Class periodic analog sensors 13](#_Toc530740138)

[2.4.3 Class periodic skin data 18](#_Toc530740139)

[2.4.4 Class periodic inertial data 18](#_Toc530740140)

[2.5 The CMD of the command classes 22](#_Toc530740141)

[2.5.1 Class polling motor control 22](#_Toc530740142)

[2.5.2 Class polling analog sensors 22](#_Toc530740143)

[2.5.3 Class bootloader (or management) 23](#_Toc530740144)

[3 Examples of typical use 32](#_Toc530740145)

[3.1 Typical CAN network 32](#_Toc530740146)

[3.2 Discovery of boards 32](#_Toc530740147)

[3.2.1 Basic discovery 32](#_Toc530740148)

[3.2.2 More detailed discovery 34](#_Toc530740149)

[3.3 Management of nodes 34](#_Toc530740150)

[3.3.1 Change the additional information 34](#_Toc530740151)

[3.3.2 Change the CAN address 35](#_Toc530740152)

[3.3.3 Firmware update 35](#_Toc530740153)

[3.4 FT service 35](#_Toc530740154)

[3.4.1 Configuration of FT service 35](#_Toc530740155)

[3.4.2 Start of the FT service 37](#_Toc530740156)

[3.4.3 Stop of the FT service 39](#_Toc530740157)

[3.5 IMU service 40](#_Toc530740158)

[3.5.1 Configuration of IMU service 40](#_Toc530740159)

[3.5.2 Start of IMU service 41](#_Toc530740160)

[3.5.3 Stop of IMU service 44](#_Toc530740161)

[3.6 THERMO service 45](#_Toc530740162)

[3.6.1 Configuration of THERMO service 45](#_Toc530740163)

[3.6.2 Start of THERMO service 47](#_Toc530740164)

[3.6.3 Stop of THERMO service 48](#_Toc530740165)

# Introduction

This document describes the iCub CAN protocol used by the strain2 board.

The first part deals with the general format of the CAN frame in the iCub protocol and details on the messages. The second part contains examples of how to use these messages for the case of the strain2 board.

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}

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: for future use
* 111b: bootloader management [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. See following Table for details.
* 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).

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 CANreal GUI manages raw frames.

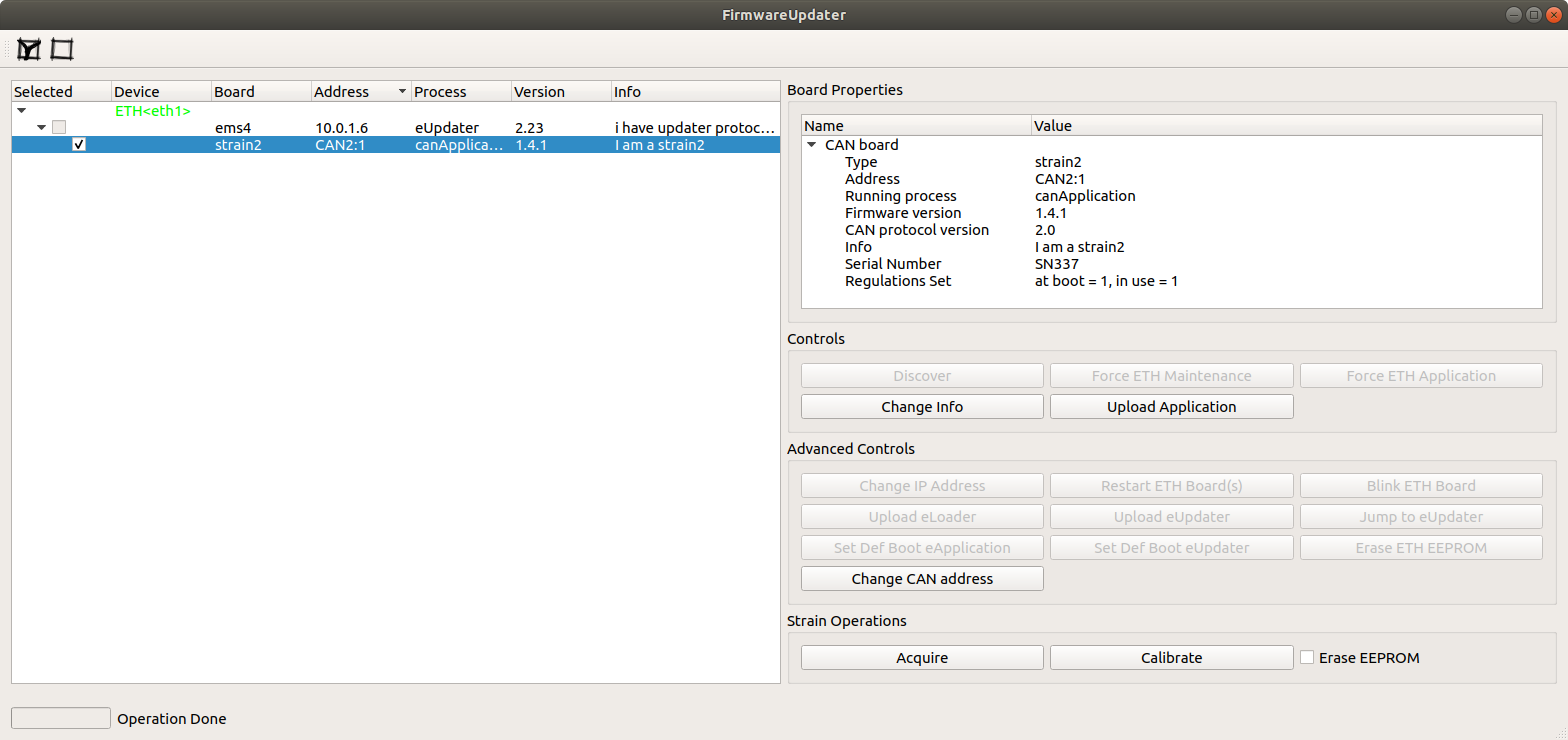
#### The host is an ETH-board inside iCub.

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 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 a board with a given CAN address ADDR in range [1, 14] and with a given board type that is able to handle a subset of iCub CAN frames. In particular, it can decode some command frames it receives from the HOST and if needed to send back replies. I can also transmit streaming frames.

The CAN node runs two processes: the bootloader and the application.

#### Structure of a CAN node

A CAN node has a unique CAN address, a proper board type and runs a process with parsing capabilities which depend on the type of process and on its board type.

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.

#### 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 types of nodes

See following table.

|  |  |  |
| --- | --- | --- |
| Code | Name | Description |
| 0x00 | dsp | DSP motor controlling board  **OBSOLETE** |
| 0x01 | pic | PIC board  **OBSOLETE** |
| 0x02 | 2dc | DSP motor controlling board  **OBSOLETE** |
| 0x03 | mc4 | 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 | Motor control board for 2 brushless motors  It is used in CAN-based iCub. |
| 0x05 | mtb | mtb  It manages skin data and basic inertial sensors. |
| 0x06 | strain | strain  It manages FT data. |
| 0x07 | mais | MAIS board  It is used in the forearms of iCub to acquire positions of the hand. |
| 0x08 | foc | 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 | 6SG  **OBSOLETE** |
| 0x0A | jog | JOG  A test board used to verify hardware in the robot, typically AEA etc. |
| 0x0B | mtb4 | MTB4  An enhancement of the mtb board. It interfaces skin patches, it has an IMU and temperature sensors. |
| 0x0C | strain2 | STRAIN2  An enhancement of the strain board. It manages FT data, it has an IMU and temperature sensors. |
| 0x0D | rfe | Robot Face Expression Board  It handles LEDs for face expression in iCub. |

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

#### The parsing capabilities of a node

The following of the document will use the icons in the table to express if a given board can parse a message.

|  |  |
| --- | --- |
| Icon | Description |
| bootloader | Basic parser  The bootloader of every board type can decode the message. |
| application | Basic parser  The application of every board type can decode the message. |
| strain2 | Specialised parser  The strain2 application can parse the message. |
| bl<strain2> | Specialise parser  The strain2 bootloader can parse the message. |

**Table 2** – The icons describing the parsing capabilities.

### The used communication modes

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

#### Transport modes

Here they are.

Unicast

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

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 an round point.

Streaming

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). It is represented with a diamond shaped point.

#### Communication paradigms

Here they are.

SET<>

node @ 1

HOST @ 0

Its sets a value as expressed by PAYLOAD.CMD and PAYLOAD.ARG

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

**Figure 10**: The SET<> paradigm. Used to write a value.

GET<>

node @ 1

HOST @ 0

Its sends a request expressed by PAYLOAD.CMD

It sends back the value in PAYLOAD.ARG

**Figure 11**: The GET<> paradigm. Used to read a value.

ORDER<>

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 a OK/KO.

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

STREAM<>

node @ 1

HOST @ 0

It send data stored in PAYLOAD.DATA with periodic frames

**Figure 13**: The STREAM<> paradigm. It is used by a node to send acquired data in a periodic and spontaneous way.

### Frames accepted and parsed

A CAN node can accept a frame because it is sent in a transport mode which is compatible to its CAN address (unicast to its address, broadcast or streaming) and send it to its parser. But it may be unable to decode it because its process or board type do not manage it.

#### 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 message classes CLS = {000b, 010b, 111b} and have an 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 the node

This is a streaming frame

node @ addr = 1

This frame is sent to the CAN parser

This is a broadcast frame with ID.DST = 15

This frame is sent to the CAN parser

This is a unicast frame with ID.DST = 1

This frame is ignored

This frame is sent to the CAN parser

This is a unicast frame with ID.DST = 2

**Figure 14**: A node accepts three type 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.

#### Messages decoded by a CAN node

Every CAN node is able to manage (decode and reply to) all messages of the class bootloader-management (111b) plus some other messages of other classes. The exact number and type of those other messages depends on:

* if the node is running the bootloader or the application process,
* the board type of the node.

In general, the bootloader of every board type is capable of the same parsing capabilities and supports what we call the basic bootloader parser. This parser handles discovery, address change, firmware update procedure.

Instead, the applications of every board type share the capability to parse a subset of instructions: the basic application parser. This parser handles discovery, address change, command which change process to bootloader so that this latter can starts firmware update.

The application of each board type will also execute dedicated parsers, which are relevant to a specified task. For instance, board which have an IMU such as the strain2 and the mtb4 support messages that handle the IMU, which other boards do not support.

A detailed table with all messages supported by each board type will follow later in the document.

## The iCub CAN protocol classes

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

### The command classes

These classes are used for SET<> or GET<> or ORDER<> communication paradigms. Typically, we use messages belonging to these classes to initialise a service. The flow of these commands typically originates from the host and terminates on each boards.

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 | Bootloader management  [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 bus, for setting and asking user-defined descriptive information of the node, and for performing firmware update. |

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

### The streaming classes

These classes are used to stream data from the boards towards other boards or more typically to the host. They use the STREAM<> communication paradigm.

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.  There is no class polling skin data because of … shortage of classes, hence we use analog sensor for configuration. |
| 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.  There is no class polling inertial data because of … shortage of classes, hence we use analog sensor for configuration. |

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

## The TYP of the streaming classes

Here are for each class.

### Class periodic motor control

[TBD]

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

### Class periodic analog sensors

Here they are.

|  |  |
| --- | --- |
| 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 = TxMode::all (its value is 4) and stopped with ARG.MODE = TxMode::none (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) |

|  |  |
| --- | --- |
| 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 = TxMode::all (= 4) ) and stopped with ARG.MODE = TxMode::none (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) |

|  |  |
| --- | --- |
| 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 = TxMode:: calibrated (= 0) the board will transmit the X Y Z force components; else if ARG.MODE = TxMode::uncalibrated (= 3) the board will transmit the ADC values of channels 0, 1, and 2. ). Transmission is stopped with ARG.MODE = TxMode::none (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, 64) 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 = TxMode::uncalibrated (= 3) the board will transmit the ADC values of channels 3, 4, and 5. Transmission is stopped with ARG.MODE = TxMode::none (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, 64) 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; |

|  |  |
| --- | --- |
| 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 = TxMode::hall (= 0) and stopped with ARG.MODE = TxMode::none (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 = TxMode::hall (= 0) and stopped with ARG.MODE = TxMode::none (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. |

|  |  |
| --- | --- |
| TYP | Description |
| **0xD** | **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 * FFU is for future use. It will probably contain temperature from another thermometer. |

**Table 5** – The TYPs of the periodic analog sensors class.

### Class periodic skin data

[TBD]

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

### Class periodic inertial data

Here are is description of the periodic inertial class.

|  |  |
| --- | --- |
| 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. |

|  |  |
| --- | --- |
| 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  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  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  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  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::uint132\_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. |

**Table 6** – The TYPs of the periodic inertial class.

## The CMD of the command classes

Here they are.

### Class polling motor control

[TBD]

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

### Class polling analog sensors

[TBD]

The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

### Class bootloader (or management)

In this class there are the CMD values described in the following table. They are relatively few with respect to the 256 possible ones.

The name of the class is originally bootloader because this class contained mostly commands used mainly for the firmware updater procedure that the bootloader executes. However, there are commands that are used also for other purposes such as discovery and change of address.

|  |  |
| --- | --- |
| 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.  **Parsed by**  bootloader  **Format of PAYLOAD**  0x01  N  ADDR   * sizeof(ARG) = 5. * N contains the number of bytes of program data that the command DATA is going to transmit. * ADDR contain the starting address in code space of those bytes. It is a std::uint32\_t mapped in little endian.   **Actions on reception**  bootloader  It memorises N and ADDR 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 replies to the sender with the same CMD and OK/KO information.  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. |

|  |  |
| --- | --- |
| 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 messages, with M = [1, 8], 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 from 0 to 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 messages, with M = [1, 8], 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 from 0 to M-1. * The Cx bytes contain the value to copy into info32[4\*SEQ+x].   **Actions on reception**    bootloader  application  It copies into info32[] the values contained into the payload.  **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 node 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). |

|  |  |
| --- | --- |
| 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 = 0xf = 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. |

**Table 7** – The CMDs of the bootloader-management class.

# Examples of typical use

The following examples show the use of CAN frames for some real use case. The detailed syntax of the commands is shown in the relevant tables.

## Typical 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 15**: A CAN network capable of offering IMU, THERMO, SKIN services.

## Discovery of boards

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 application @ 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 16**: 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}

CMD

ARG

[]

DISCOVER = 0xFF

sizeof(ARG) = 0

**Figure 17**: This frame asks to every device (ID.DST = 0xf ) info about their presence.

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

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}

CMD

ARG

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

DISCOVER = 0xFF

sizeof(ARG) = 4

**Figure 18**: 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 running)

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 = 0001b} {DST = 0000b}

CMD

ARG

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

DISCOVER = 0xFF

sizeof(ARG) = 3

**Figure 19**: 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.

[TBD] The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

## 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.

[TBD] The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

### 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 = SETCANADDRESS of class bootloader-management.

[TBD] The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

### Firmware update

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

The used messages have CMD equal to: BOARD = 0x00, ADDRESS = 0x01, START = 0x02, DATA = 0x03, END = 0x04. See relevant table for more details.

[TBD] The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

## 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 order to be effective, the FT service requires the execution of previous calibration procedure. After that, the six FT values sent by the board will keep the effective torques and forces. In here, we assume that this procedure is already done and its results are stored in the EEPROM of the board.

### Configuration of FT service

The FT service needs to know the full-scale values imposes at calibration. We need to ask for the full-scale value 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 20**: 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 #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}

CMD

ARG

[0000b, 0010b]

GET\_FULL\_SCALE = 0x18

sizeof(ARG) = 1

**Figure 21**: The first frame asks the full scale of channel #2.

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

CMD

ARG

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

GET\_FULL\_SCALE = 0x18

sizeof(ARG) = 3

**Figure 22**: 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 23**: 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}

CMD

ARG

[ 0x0A ]

SET\_CANDATARATE = 0x08

sizeof(ARG) = 1

**Figure 24**: 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 25**: 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] = 0x00, thus the order is to transmit calibrated data.

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

CMD

ARG

[ 0x00 ]

SET\_TXMODE = 0x07

sizeof(ARG) = 1

**Figure 26**: 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 = 0000b} {TYP = FORCE\_VECTOR= 0xA}

DATA

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

sizeof(DATA) = 6 or 7

**Figure 27**: 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 = 0000b} {TYP = TORQUE\_VECTOR= 0xB}

DATA

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

sizeof(DATA) = 6 or 7

**Figure 28**: 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 29**: 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] = 0x01, thus the order is to stop transmission.

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

CMD

ARG

[ 0x01 ]

SET\_TXMODE = 0x07

sizeof(ARG) = 1

**Figure 30**: The frame that stops the transmission.

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

FRAME-1: command which 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 fusion. The used value is imuFusion::enabled = 1.
* ARG[3], ARG[4], ARG[5], ARG[6] are for future use and their value is dontcare. It is OK if = 0.

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

CMD

ARG

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

IMU\_CONFIG\_SET = 0x34

sizeof(ARG) = 7

**Figure 32**: 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}

CMD

ARG

[]

IMU\_CONFIG\_GET = 0x33

sizeof(ARGUMENTS) = 0

**Figure 33**: 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}

CMD

ARG

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

IMU\_CONFIG\_GET = 0x33

sizeof(ARG) = 8

**Figure 34**: 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 35**: 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}

CMD

ARG

[ 0x32 ]

IMU\_TRANSMIT = 0x35

sizeof(ARG) = 1

**Figure 36**: 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 = 0000b} {TYP = IMU\_TRIPLE = 0011b}

DATA

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

sizeof(DATA) = 8

**Figure 37**: 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 = 0000b} {TYP = IMU\_TRIPLE = 0011b}

DATA

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

sizeof(DATA) = 8

**Figure 38**: 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 = 0000b} {TYP = IMU\_STATUS = 0101b}

DATA

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

sizeof(DATA) = 8

**Figure 39**: 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 40**: 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}

CMD

ARG

[ 0x00 ]

IMU\_TRANSMIT = 0x35

sizeof(ARG) = 1

**Figure 41**: The frame that stops the IMU transmission.

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

FRAME-1: command which 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}

CMD

ARG

[ 00000001b ]

THERMO\_CONFIG\_SET = 0x39

sizeof(ARG) = 1

**Figure 43**: The first frame sets the chosen thermometer.

FRAME-2: command which asks the THERMO configuration

ID

This frame does not need any arguments.

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

CMD

ARG

[]

THERMO\_CONFIG\_GET = 0x38

sizeof(ARG) = 0

**Figure 44**: 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 1. The content of ARGUMENTS is the same as for THERMO\_CONFIG\_SET

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

CMD

ARG

[ 00000001b ]

THERMO\_CONFIG\_GET = 0x38

sizeof(ARG) = 1

**Figure 45**: 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 46**: 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}

CMD

ARG

[ 0x02 ]

THERMO\_TRANSMIT = 0x3A

sizeof(ARG) = 1

**Figure 47**: 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 = 0000b} {TYP = THERMO\_MEASURE = 0xE}

DATA

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

sizeof(DATA) = 5

**Figure 48**: 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 49**: 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}

CMD

ARG

[ 0x00 ]

THERMO\_TRANSMIT = 0x3A

sizeof(ARG) = 1

**Figure 50**: The frame that stops the THERMO transmission.