The strain2: CAN protocol

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](#_Toc530494769)

[2 The iCub CAN protocol 2](#_Toc530494770)

[2.1 The structure of the CAN frame 2](#_Toc530494771)

[2.1.1 The ID 3](#_Toc530494772)

[2.1.2 The PAYLOAD 3](#_Toc530494773)

[2.2 The iCub CAN nodes 3](#_Toc530494774)

[2.2.1 The types of nodes 3](#_Toc530494775)

[2.2.2 Processes executed by a CAN node: bootloader and application 4](#_Toc530494776)

[2.2.3 Messages accepted by a CAN node 5](#_Toc530494777)

[2.2.4 Messages decoded by a CAN node 6](#_Toc530494778)

[2.3 The iCub CAN protocol classes 6](#_Toc530494779)

[2.3.1 The unicast classes 6](#_Toc530494780)

[2.3.2 The broadcast classes 7](#_Toc530494781)

[2.4 The TYP of the broadcast classes 8](#_Toc530494782)

[2.4.1 Class periodic motor control 8](#_Toc530494783)

[2.4.2 Class periodic analog sensors 8](#_Toc530494784)

[2.4.3 Class periodic skin data 11](#_Toc530494785)

[2.4.4 Class periodic inertial data 11](#_Toc530494786)

[2.5 The CMD of the unicast classes 14](#_Toc530494787)

[2.5.1 Class polling motor control 14](#_Toc530494788)

[2.5.2 Class polling analog sensors 14](#_Toc530494789)

[2.5.3 Class bootloader – management 15](#_Toc530494790)

[3 Examples of typical use 19](#_Toc530494791)

[3.1 Typical CAN network 19](#_Toc530494792)

[3.2 Discovery of boards 19](#_Toc530494793)

[3.2.1 Basic discovery 19](#_Toc530494794)

[3.2.2 More detailed discovery 22](#_Toc530494795)

[3.3 Check of presence and of CAN protocol 22](#_Toc530494796)

[3.4 Management of nodes 22](#_Toc530494797)

[3.4.1 Change the additional information 22](#_Toc530494798)

[3.4.2 Change of CAN address 22](#_Toc530494799)

[3.5 FT service 23](#_Toc530494800)

[3.5.1 Configuration of FT service 23](#_Toc530494801)

[3.5.2 Start of the FT service 25](#_Toc530494802)

[3.5.3 Stop of the FT service 27](#_Toc530494803)

[3.6 IMU service 28](#_Toc530494804)

[3.6.1 Configuration of IMU service 28](#_Toc530494805)

[3.6.2 Start of IMU service 31](#_Toc530494806)

[3.6.3 Stop of IMU service 33](#_Toc530494807)

[3.7 THERMO service 34](#_Toc530494808)

[3.7.1 Configuration of THERMO service 34](#_Toc530494809)

[3.7.2 Start of THERMO service 37](#_Toc530494810)

[3.7.3 Stop of THERMO service 38](#_Toc530494811)

# 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 these messages are used 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. The frames are transmitted at 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 transport mode (unicast or broadcast). 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 transport 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 unicast or broadcast class. It may be:

* The destination address (DST) for unicast classes CLS = {000b, 010b, 111b}.
* The type of content (TYP) if the CLS is a broadcast class, 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 transport mode (unicast or broadcast):

* 000b: polling Motor Control [unicast]
* 001b: periodic Motor Control [broadcast]
* 010b: polling Analog Sensors [unicast]
* 011b: periodic Analog Sensors [broadcast]
* 100b: periodic Skin Data [broadcast]
* 101b: periodic Inertial Sensors [broadcast]
* 110b: for future use
* 111b: bootloader management [unicast]

CMD

ARGUMENTS

DATA

unicast:

broadcast

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 transport mode: either unicast or broadcast. 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 unicast transport mode, then the 4 bits must contain the destination address DST. In such a case, the first byte of the payload contains the command CMD and the remaining 7 bytes contain the required parameters.
* If CLS has a broadcast transport 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 associated transport mode.

* If ID.CLS is of broadcast type, then the payload contains only data. The number of used bytes and their format depends on the value of ID.TYP.
* If instead ID.CLS is of unicast type, the payload contains a command. The first byte is PAYLOAD.CMD and specifies which command to apply. The remaining 7 bytes are available for hosting the parameters of the command.

## The iCub CAN nodes

An iCub CAN node is a board with a given CAN address ADDR in range [0, 14] and with a given board type that is able to handle the iCub CAN frames in both reception and transmission.

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 2**: The iCub 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.

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

The iCub CAN node can run 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 behaviour at bootstrap of the CAN node

Countdown expired

bootloader

Received a CAN frame: stop countdown.

application

Starts 5 sec countdown

**Figure 3**: Behaviour at bootstrap of the CAN node.

### 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 broadcast classes CLS = { 001b, 011b, 100b, 101b }.
* all frames that belong to one of the unicast classes CLS = { 000b, 010b, 111b } and have an ID.DST equal to the node address ADR or equal to EVERYONE = 15.

CAN frames accepted for parsing

This is a broadcast frame

node @ addr = 1

This frame is sent to the CAN parser

This is a unicast 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 4**: 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 divided in unicast and broadcast classes, according to their transport type, which reflects the frame structure.

### The unicast classes

These classes are used to send command of the get<> or set<> type or also to communicate an order of some type. Typically, we use command belonging to these classes to initialise a service. The flow of these commands typically originates from the host to each boards.

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

|  |  |
| --- | --- |
| CLS | Description |
| 000b | Polling Motor Control  [transport: unicast]  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  [transport: unicast]  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  [transport: unicast]  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 2** – The unicast classes in the iCub CAN protocol.

### The broadcast classes

These classes are used to stream data from the boards towards other boards or more 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  [transport: broadcast]  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  [transport: broadcast]  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: broadcast]  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: broadcast]  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 3** – The broadcast classes in the iCub CAN protocol.

## The TYP of the broadcast 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  **Use**  It is used in debug mode to transport the raw ADC values of channels 0, 1 and 2. We use this message because in debug mode FORCE\_VECTOR will transports forces.  The format of each component is as follows.   * Raw data has a range [0, 64K) and represent the output of an ADC of the same range. It is considered saturated at low scale if is in range [0, 1000). It is considered saturated at high scale if it is in range [64000, 64K).   **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 it was detected 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 the channel x has experienced: NONE = 0, atLOWscale = 1, atHIGHscale = 2. |
| 0x9 | UNCALIBTORQUE\_VECTOR\_DEBUGMODE  **Use**  It is used in debug mode to transport the raw ADC values of channels 3, 4 and 5. We use this message because in debug mode TORQUE\_VECTOR will transports torques.  The format of each component is as follows.   * Raw data has a range [0, 64K) and represent the output of an ADC of the same range. It is considered saturated at low scale if is in range [0, 1000). It is considered saturated at high scale if it is in range [64000, 64K).   **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 it was detected 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 the channel x has experienced: NONE = 0, atLOWscale = 1, atHIGHscale = 2. |
| 0xA | FORCE\_VECTOR  **Use**  It contains force data as emitted by the strain/strain2 boards. It can be either raw (values taken directly from the ADC peripheral) or calibrated (passed through transformations).  The format of each component is as follows.   * Raw data has a range [0, 64K) and represent the output of an ADC of the same range. It is considered saturated at low scale if is in range [0, 1000). It is considered saturated at high scale if it is in range [64000, 64K). * Calibrated data has a range [0, 64) and contains a transformed 1Q15 number which is the result of the calibration inside the strain/strain2 board. To get the original 1Q15 number for the first component one need to do:   std::int16\_t q15X = static\_cast<std::int16\_t>(X – 0x8000);  After that, q15X will be represented in the range [-1, +1-2-15]. To achieve the force one must do:  FX = fullscale[0] \* static\_cast<double>(q15X) / 32768.0;  **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 if it was detected 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 the channel x has experienced: NONE = 0, atLOWscale = 1, atHIGHscale = 2. |
| 0xB | TORQUE\_VECTOR  **Use**  It contains torque data as emitted by the strain/strain2 boards. It can be either raw (values taken directly from the ADC peripheral) or calibrated (passed through transformations).  The format of each component is as follows.   * Raw data has a range [0, 64K) and represent the output of an ADC of the same range. It is considered saturated at low scale if is in range [0, 1000). It is considered saturated at high scale if it is in range [64000, 64K). * Calibrated data has a range [0, 64) and contains a transformed 1Q15 number which is the result of the calibration inside the strain/strain2 board. To get the original 1Q15 number for the first component one need to do:   std::int16\_t q15X = static\_cast<std::int16\_t>(X – 0x8000);  After that, q15X will be represented in the range [-1, +1-2-15]. To achieve the force one must do:  TX = fullscale[3] \* static\_cast<double>(q15X) / 32768.0;  **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 if X, Y, Z are calibrated values and if it was detected 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 the channel x has experienced: NONE = 0, atLOWscale = 1, atHIGHscale = 2. |
| 0xC | HES0TO6  **Use**  It contains data from the Hall effect sensors used in the mais board: channel 0 to 6  **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. |
| 0xD | HES7TO14  **Use**  It contains data from the Hall effect sensors used in the mais board: channel 7 to 14  **Format of DATA**  S7  S8  S9  S10  S11  S12  S13  S14   * sizeof(DATA) = 8. * Sx is data from sensor x expressed as a std::uint8\_t. |
| 0xE | THERMOMETER\_MEASURE  **Use**  It contains data from temperature sensor used in the strain2 and mtb4 boards. The used resolution is of 0.1 Celsius degrees.  **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. |

**Table 4** – 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  **Use**  It contains gyroscope data as emitted by the mtb board (and for legacy purposes by the mtb4).  **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 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.   TBD: find out which measurement unit is used … |
| 0x1 | DIGITAL\_ACCELEROMETER  **Use**  It contains accelerometer data as emitted by the mtb board (and for legacy purposes by the mtb4).  **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 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.   TBD: find out which measurement unit is used … |
| 0x3 | IMU\_TRIPLE  **Use**  It contains 3-axis data emitted from an IMU.  We support the 3-axis 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::lia in 0.01 [m/s2] * imuSensor::grv in 0.01 [m/s2]   **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. |
| 0x4 | IMU\_QUATERNION  **Use**  It contains quaternion data emitted from an IMU.  Each component is expressed as a std::int16\_t with following resolution and measurement units:   * imuSensor::qua in 0.00006103515625 [quaternion] (=1/16384)   **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. |
| 0x5 | IMU\_STATUS.  **Use**  It contains acquisition status of the IMU.  **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 matching 3-axis data amongst them and with status messages. * SGY contains the calibration status of the gyroscope expressed by enum class Calib { none = 0, poor = 1, medium = 2, good = 3 }. * SAC contains the calibration status of the accelerometer expressed by enum class Calib { none = 0, poor = 1, medium = 2, good = 3 }. * SMA contains the calibration status of the magnetometer expressed by enum class Calib { none = 0, poor = 1, medium = 2, good = 3 }. * ACQDURATION contain the time required for the acquisition expressed as a std::uint132\_t in little endian in micro-sec units.   NOTE. 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. |

**Table 5** – The TYPs of the periodic inertial class.

## The CMD of the unicast classes

Here 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 – management

In this class there are the CMD values described in the following table. They are relatively few with respect to the full 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 | Name | Description |
| 0x00 | BOARD | It is used to start the firmware update procedure  **Parsed**  By bootloader and application of every board type.  **Arguments**  ARG.size = 1  ARG[0] = eraseeeprom, = { 0, 1}  **On reception**  *Application*  It just resets to execute the bootloader.  *Bootloader*  It stores the eraseeeprom info, enters the firmware update procedure.  It replies with the same MSG = 0x00 and with  ARG.size = 0 |
| 0x01 | ADDRESS | It is used inside the firmware update procedure to tell how many bytes will arrive and their starting address.  **Parsed**  By bootloader only of every board type.  **Arguments**  ARG.size = 5  ARG[0] = size of incoming program data  ARG[1, 2, 3, 4] = address in FLASH of incoming program data. In little endian.  **On reception**  *Bootloader*  It memorises size and address.  It does not reply. |
| 0x02 | START | It is used inside the firmware update procedure to terminate the writing to FLASH of all cached data and if previously specified by command BOARD it erases the EEPROM.  **Parsed**  By bootloader only of every board type.  **Arguments**  ARG.size = 0  **On reception**  *Bootloader*  It ensures that all received program data is burned in FLASH, and if requested it erases the EEPROM.  It replies with a message of the same CMD = 0x02 and with:  ARG.size = 1,  ARG[0] = 1 / 0. Value of 1 is OK, value of 0 is KO. |
| 0x03 | DATA | It is used inside the firmware update procedure to send to bootloader the data to burn in FLASH.  **Parsed**  By bootloader only of every board type.  **Arguments**  ARG.size = variable  ARG[0, etc] = it contains ARG.size bytes to burn in FLASH.  **On reception**  *Bootloader*  It copies in RAM the ARG.size bytes inside ARG[] for their later burn into FLASH at the address equal to the one specified in the latest ADDRESS message and incremented by all the DATA messages received after. The effective burn into FLASH depends on the RAM capacity and on the type of FLASH.  It replies with a message of the same CMD = 0x03 and with:  ARG.size = 1,  ARG[0] = 1 / 0, where 1 is OK and 0 is KO. |
| 0x04 | END | It is used inside the firmware update procedure to inform the bootloader that the programming is finished and that it can restart.  **Parsed by**  By bootloader only of every board type.  **Arguments**  ARG.size = 0  **On reception**  *Bootloader*  It sends back a reply and then it restarts.  It replies with a message of the same CMD and with ARG.size = 1, where ARG[0] = 1 / 0. Value of 1 is OK, value of 0 is KO. |
| 0x0C | GET\_INFO | It is used to get additional information which is stored in storage of the node. The additional information is a string of up to 32 characters.  **Parsed by**  By bootloader and application only of every board type.  **Arguments**  ARG.size = 0  **On reception**  *Bootloader + Application*  It sends back as many messages are required (numofframes) to hold all the string of up to 32 characters info32[]. Each message can contain up to 4 characters and it has:  The same CMD = 0x05,  ARG.size = 5,  ARG[0] = n. Its value ranges from 0 up to numofframes-1. Initial value is n = 0, and after each transmission n is incremented.  ARG[1] = info32[n\*4+0]. A character of the string  ARG[2] = info32[n\*4+1]. A character of the string  ARG[3] = info32[n\*4+2]. A character of the string  ARG[4] = info32[n\*4+3]. A character of the string |
| 0x0D | SET\_INFO | It is used to set additional information to be stored in storage of the node. . The additional information is a string of up to 32 characters  **Parsed by**  By bootloader and application only of every board type.  **Arguments**  ARG.size = 5  ARG[0] = n. Its value ranges from 0 up to 7. It tells which subset of the 32 bytes to write.  ARG[1] = character of the string in position n\*4  ARG[2] = character of the string in position n\*4+1  ARG[3] = character of the string in position n\*4+2  ARG[4] = character of the string in position n\*4+3  **On reception**  *Bootloader + Application*  It copies ARG[1, 2, 3, 4] into the relevant position of the additional info string and then it saves it in permanent storage.  It does not reply. |
| 0x32 | SET\_ADDRESS | It is used to change the CAN address. It is a new message that enables both explicit address assignment also the random assignment mode. This latter is useful when two or more node accidentally have the same address.  **Parsed by**  By bootloader and application of latest board only: mtb4, strain2, rfe.  **Arguments**  ARG.size = 3  ARG[0] = newaddress. If its value is 255, then assign the address randomly.  ARG[1, 2] = randominvalidmask expressed in little endian mode. If random mode is enabled by ARG[0] = 255, the randominvalidmask contains the addresses which must be avoided expressed as bit positions.  **On reception**  *Bootloader + Application*  It assign a new address as expressed by ARG[0]. In the random assignment mode, the forbidden addresses are contained in the mask. For instance, if we want to avoid addresses 0, 1, 2 and 14 we use randominvalidmask = 0100000000000111b = 0x4007 which is store into ARG[1] = 0x07 and ARG[2] = 0x40.  It does not reply. |
| 0xFF | DISCOVER | Also named: BROADCAST.  It is used to discover the nodes in the CAN network. It is typically transmitted with ADDR = EVERYONE.  **Parsed by**  By bootloader and application of every board type.  **Arguments**  ARG.size = 0  **On reception**  *Bootloader*  It replies with a message of the same CMD and with ARG.size = 3, where:  ARG[0] = BOARDTYPE  ARG[1] = major number of bootloader version  ARG[2] = minor number of bootloader version  *Application*  It replies with a message of the same CMD and with ARG.size = 4, where:  ARG[0] = BOARDTYPE  ARG[1] = major number of application version  ARG[2] = minor number of application version  ARG[3] = build number of application version |

**Table 6** – 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. In here, there will be both unicast and broadcast CAN frames belonging to different classes. The detailed syntax of the commands is shown in the dedicated 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 USB to CAN gateway attached to a PC. The other node 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 @ address ADDR = 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 5**: 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. The boards in the CAN network reply with basic information.

SEQUENCE DIAGRAM FOR BASIC DISCOVERY

host @ addr = 0

FRAME-DISCOVER

strain2 application @ addr = 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 @ addr = 2

**Figure 6**: Example of CAN frames used for the basic discovery. In the CAN bus there are two boards: a strain2 which runs its application process and an mtb4 running its bootloader process.

FRAME-DISCOVER

ID

This frame does not need any arguments.

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

CMD

ARGUMENTS

[]

DISCOVER = 0xFF

sizeof(ARGUMENTS) = 0

**Figure 7**: This frame asks to very device in the channel info about their presence. It uses ID.DST = 0xf to deliver the frame to every device.

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

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

CMD

ARGUMENTS

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

DISCOVER = 0xFF

sizeof(ARGUMENTS) = 8

**Figure 8**: The strain2 board replies with a frame that contains the type of board it is and the version of its application.

FRAME-2: command which asks the IMU configuration

ID

This frame does not need any arguments.

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

CMD

ARGUMENTS

[]

IMU\_CONFIG\_GET = 0x33

sizeof(ARGUMENTS) = 0

**Figure 9**: The second frame asks the configuration for a check.

### 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.

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

## Check of presence and of CAN protocol

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\_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 of 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.

## 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 @ addr = 0

FRAME-1

strain2 @ addr = 1

FRAME-2

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

The strain2 just replies with the full scale value

**Figure 10**: Example of CAN frames used to retrieve the value of full-scale of channel 2 for the FT service.

FRAME-1: command which asks the full-scale #2

ID

This frame has an ARGUMENTS 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

ARGUMENTS

[0000b, 0010b]

GET\_FULL\_SCALE = 0x18

sizeof(ARGUMENTS) = 1

**Figure 11**: 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 ARGUMENTS 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

ARGUMENTS

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

GET\_FULL\_SCALE = 0x18

sizeof(ARGUMENTS) = 3

**Figure 12**: The second frame sends back the value of full-scale.

After the six full-scales are retrieved, the FT sensor need to be assigned with its data rate in advanced.

Sequence diagram for configuration of the TX rate of FT data

Host @ addr = 0

FRAME-3

strain2 @ addr = 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 13**: Example of CAN frames used to configure the TX rate of FT service.

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

ID

Explanation of the content of ARGUMENTS.

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

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

CMD

ARGUMENTS

[ 0x0A ]

SET\_CANDATARATE = 0x08

sizeof(ARGUMENTS) = 1

**Figure 14**: 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 @ addr = 0

strain2 @ addr = 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 15**: 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]. Valid values are: { txCalibData = 0, txNothing = 1, txUncalibData = 3, txBothData = 4 }. 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

ARGUMENTS

[ 0x00 ]

SET\_TXMODE = 0x07

sizeof(ARGUMENTS) = 1

**Figure 16**: The frame that starts the transmission.

FRAME-A: values of Force

ID

Explanation of the content of DATA.

* DATA[0, 1] contains the x component of force expressed as a std::int16\_t in BIG-ENDIAN mode.
* DATA[2, 3] contains the y component of force expressed as a std::int16\_t in BIG-ENDIAN mode.
* DATA[4, 5] contains the z component of force expressed as a std::int16\_t in BIG-ENDIAN mode.

If the acquisition of any of the 6 ADC channels is in saturation, then we have sizeof(DATA) equal 7 and DATA[7] contains information about the saturation of the channels 0, 1, and 2. DATA[7] is split into 4 bit-pairs: PAIR(0, 1) = 1 to mean that there is saturation; PAIR(2, 3), PAIR(4, 5) and PAIR(6, 7) keeps saturation info on channel 0, 1, and 2 respectively. Saturation info have three possible value: { NONE = 0, ATLOWSCALE = 1, ATHIGHSCALE = 2 }.

{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 17**: These frames contain the calibrated force values.

FRAME-B: values of Torque

ID

Explanation of the content of DATA.

* DATA[0, 1] contains the x component of torque expressed as a std::int16\_t in BIG-ENDIAN mode.
* DATA[2, 3] contains the y component of torque expressed as a std::int16\_t in BIG-ENDIAN mode.
* DATA[4, 5] contains the z component of torque expressed as a std::int16\_t in BIG-ENDIAN mode.

If the acquisition of any of the 6 ADC channels is in saturation, then we have sizeof(DATA) equal 7 and DATA[7] contains information about the saturation of the channels 3, 4, and 5. DATA[7] is split into 4 bit-pairs: PAIR(0, 1) = 1 to mean that there is saturation; PAIR(2, 3), PAIR(4, 5) and PAIR(6, 7) keeps saturation info on channel 3, 4, and 5 respectively. Saturation info have three possible value: { NONE = 0, ATLOWSCALE = 1, ATHIGHSCALE = 2 }.

{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 18**: 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 @ addr = 0

strain2 @ addr = 1

FRAME-5

It stops broadcasting whatever FT data it was sending

FRAME-A

It commands the stop of the service.

FRAME-B

**Figure 19**: 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]. Valid values are: { txCalibData = 0, txNothing = 1, txUncalibData = 3, txBothData = 4 }. In our case we have ARG[0] = 0x01, thus the order is to stop transmission.

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

CMD

ARGUMENTS

[ 0x01 ]

SET\_TXMODE = 0x07

sizeof(ARGUMENTS) = 1

**Figure 20**: 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 @ addr = 0

FRAME-1

strain2 @ addr = 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 21**: 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 ARGUMENTS.

* 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

ARGUMENTS

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

IMU\_CONFIG\_SET = 0x34

sizeof(ARGUMENTS) = 7

**Figure 22**: 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

ARGUMENTS

[]

IMU\_CONFIG\_GET = 0x33

sizeof(ARGUMENTS) = 0

**Figure 23**: 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

ARGUMENTS

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

IMU\_CONFIG\_GET = 0x33

sizeof(ARGUMENTS) = 8

**Figure 24**: 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 STRART OF AN IMU SERVICE

Host @ addr = 0

strain2 @ addr = 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 25**: 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

ARGUMENTS

[ 0x32 ]

IMU\_TRANSMIT = 0x35

sizeof(ARGUMENTS) = 1

**Figure 26**: The frame that starts the transmission.

FRAME-A: values from an accelerometer

ID

Explanation of the content of DATA.

* 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 27**: 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.

* 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 28**: 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.

* 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 quality indicator of the calibration of the gyroscope. Values are [0, 3], where 3 is OK.
* DATA[2] is a quality indicator of the calibration of the accelerometer. Values are [0, 3], where 3 is OK.
* DATA[3] is a quality indicator of the calibration of the magnetometer. Values are [0, 3], where 3 is OK.
* DATA[4] to DATA[7] represent the duration of the acquisition time in microseconds represented as a std::uint32\_t and memory mapped in little endian.

{CLS = PERIODIC-INERTIAL = 101b} {SRC = 0000b} {TYP = IMU\_STATUS = 0101b}

DATA

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

sizeof(DATA) = 8

**Figure 29**: 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 @ addr = 0

strain2 @ addr = 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 30**: 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

ARGUMENTS

[ 0x00 ]

IMU\_TRANSMIT = 0x35

sizeof(ARGUMENTS) = 1

**Figure 31**: 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 @ addr = 0

FRAME-1

strain2 @ addr = 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 32**: 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 ARGUMENTS.

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

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

CMD

ARGUMENTS

[ 00000001b ]

THERMO\_CONFIG\_SET = 0x39

sizeof(ARGUMENTS) = 1

**Figure 33**: 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

ARGUMENTS

[]

THERMO\_CONFIG\_GET = 0x38

sizeof(ARGUMENTS) = 0

**Figure 34**: 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

ARGUMENTS

[ 00000001b ]

THERMO\_CONFIG\_GET = 0x38

sizeof(ARGUMENTS) = 1

**Figure 35**: 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 STRART OF AN IMU SERVICE

Host @ addr = 0

strain2 @ addr = 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 36**: 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

ARGUMENTS

[ 0x02 ]

THERMO\_TRANSMIT = 0x3A

sizeof(ARGUMENTS) = 1

**Figure 37**: 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 38**: 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 @ addr = 0

strain2 @ addr = 1

FRAME-5

It stops broadcasting whatever THERMO data it was sending

FRAME-A

It commands the stop of the service.

**Figure 39**: 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

ARGUMENTS

[ 0x00 ]

THERMO\_TRANSMIT = 0x3A

sizeof(ARGUMENTS) = 1

**Figure 40**: The frame that stops the THERMO transmission.