|  |
| --- |
| **Cherenkov Telescope Array**  **Camera test LED system specification** |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Name | Date | Signature |
| Prepared by | Domenico della volpe | 2/28/2018 |  |
| Verified by |  |  |  |
| Authorized by |  |  |  |

Document Change Record

| Issue | Date | Changed Section | Description of Change |
| --- | --- | --- | --- |
| 1 | 2015-06-21 | All | First issue - preliminary |
| 1.1 | 2015-09-18 | 6.5.4 | Set CAN ADDR answer not corresponding to FW implementation |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**References**

|  |  |  |
| --- | --- | --- |
| Reference | Document name | Description |
| REF1 | EDMS 121623-v1.6 | Hexagonal SIPM Pre-amplifier & slow control boards specification |
|  |  |  |

**Acronyms**

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Table of contents

1 INTRODUCTION 5

1.1 Objectives 5

1.2 Overview 5

2 Camera cabling map validation 6

2.1 Test description 6

3 Camera Charge resolution Test 6

4 Camera Trigger Test 6

5 Equipment Requirements 7

6 LED system electronic description 9

6.1 Introduction 9

6.2 System Description 11

6.2.1 AC LEDs Control 11

6.2.2 DC LEDs Control 11

6.3 Boards description 11

6.3.1 LED driver board 11

6.3.2 LED carrier board 11

6.3.3 LED driver dispatch board 11

6.4 Hardware Interfaces 11

6.4.1 LED driver board interface 11

6.4.2 LED carrier board interface 12

6.4.3 LED driver dispatch board interface 12

6.5 CAN Protocol interface 13

6.5.1 CAN addressing 13

6.5.2 CAN messaging : data exchange 15

6.5.3 CAN messaging: Master requests/answers summary 16

6.5.4 CAN messaging: Master requests/answer details 17

Table of CAN requests & answers

[CAN addressing 13](#_Toc418758696)

[CAN messaging data 15](#_Toc418758697)

[Master requests/answers summary 16](#_Toc418758698)

[List of requests 17](#_Toc418758699)

[REQ:Abort 17](#_Toc418758700)

[ANS:Abort 17](#_Toc418758701)

[ANS:General error 17](#_Toc418758702)

[REQ: Set CAN ADDR 17](#_Toc418758703)

[ANS: Set CAN ADDR 18](#_Toc418758704)

[REQ: Set DAC level 18](#_Toc418758705)

[ANS: Set DAC level 19](#_Toc418758706)

[REQ: LED Control 19](#_Toc418758707)

[ANS: LED Control 19](#_Toc418758708)

# INTRODUCTION

## Objectives

The objective of the Camera Test LED System is to validate different features of the full assembled camera:

1. Camera Cabling Map Validation: the test will be done once the cabling with DigiCam is complete and before the camera chassis closure
2. Camera Charge Resolution Test: by having a flashing LED it is possible to reproduce for the camera and for many channels the charge resolution with and without NSB, emulated by a continuous LED.
3. Camera Trigger Test: By having the possibility of producing flashed patterns on the camera the trigger algorithm can be tested, Moreover by the use of a secondary LED continuous we can also monitor the performance with different level of NSB. Also flashing at High frequency and in presence of NSB can test the limitation in term of throughput.

## Overview

The test will need the PDP assembled and cabled and also the DigiCam installed on the rail on the camera chassis and cabled. Also the cabling of the power and the Can bus needs to be in place and working.

The test is performed for one sector of the camera at time (1 sector= 1/3 of the camera)

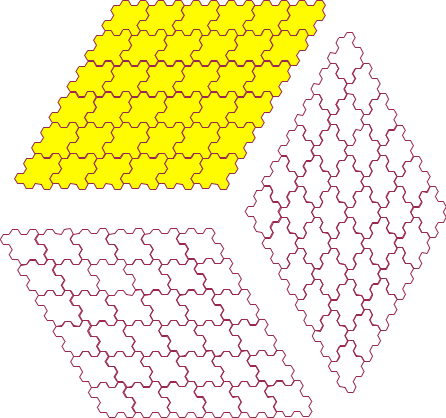


Figure 1: Left - Definition of a sector. Right - The part of the CTS equipped with light system

The best would be to use the final configuration, by controlling the PDP via the DigiCam, but as an alternative an external Can bus Controller can be used.

The light is injected with the Camera Test Setup (CTS), which is a light tight cover and will adapt to the PDP front mounted instead of the window.

Not all the complete system is equipped. The zone covered are shown in color in Figure 1

The active part is segmented in 11 boards each covering 48 pixels.

Each pixel faces two LEDs that can be operated independently. One will be used for continuous illumination while the other will be used in pulsed mode.

The central zone is important to test trigger data sharing and moving across different sectors.

The CTS is controlled via Can bus through a Labview application. The data are readout all together via DigiCam Ethernet optical fiber or can be read for each FADC board of DigiCam (à la Demoboard) via the Ethernet connector present on each board.

In the first case dedicated software to collect data and write them to disk is needed. In the second case the board can be readout with the same LabView application controlling the CTS

# Camera cabling map validation

## Test description

The test is done by switching ON a single LED in continuous mode and verifying that the correct channel in DigiCam is responding. The level of light does not matter for the test, it just need to be sufficient to be over the minimal threshold (>50 p.e.). It would be good not to saturate the electronic and then to stay below (300-400 p.e.) but this is not a requirement.

The software should be able to make the check on line without any data analysis. A repost should be produced with the current mapping. In case of errors the test is repeated and the cabling validates.

The test should be as fast as possible but there are no particular timing requirements here. Just to be able to switch ON/OFF the LED a read the pixels.

# Camera Charge resolution Test

TBD

# Camera Trigger Test

TBD

# Equipment Requirements

The system has to cover 1/3 of the camera (1 sector) plus 96 more pixels to cover completely the 144 pixels in the center of the camera.

The system is segmented in 11 boards each with 48 channels.

Each channel corresponds to a camera pixel, and it is equipped with 2 LED.

* 1 LED (~400 nm) will work with continuous light with a programmable level (bias voltage). There are no requirements on the granularity needed to set the bias, as this LED is used for the cabling check and to emulate the NSB, which in principle are the same for all pixels
* 1 flashing LED (~ 400 nm), which will emulate flashes. It requires to be driven by a pulse with a configurable height (nr of photons) and period (up to 10 kHz).   
  The pulse duration can be fixed but should produce a light pulse in the order of 2-3 ns.   
  The best would be to have the capability of driving each LED independently, but to keep the complexity and the cost reasonable even a patch of 3 neighboring pixels is ok.  
  To have more flexibility the flashing LED can also be driven by an external pulse generator.

The LEDs will be put side by side. The Flashing LED will be positioned in the geometrical center of the pixel, while the continuous LED will be position on a side.

The boards will use microcontroller for its performance and will be driven by Can Bus. The control electronics is hosted in a separate board that is plugged on the one hosting the LEDs via a connector, this is to keep margin to evolve/ change the LEDs but saving the most complicated electronics.

The system will have both an on-board power supply but will also have the possibility to an external bias voltage for the LED to avoid that in future, in case the led need to be changes we need to redesign the board to change the DC-DC converter.

The system should provide an external trigger for the digital readout.

The system will be split into two boards, one hosting the driving system and another one when only the LED and resistor will be soldered.

Figure 2 exhibits the mechanical elements on which the LED carrier boards are mounted.

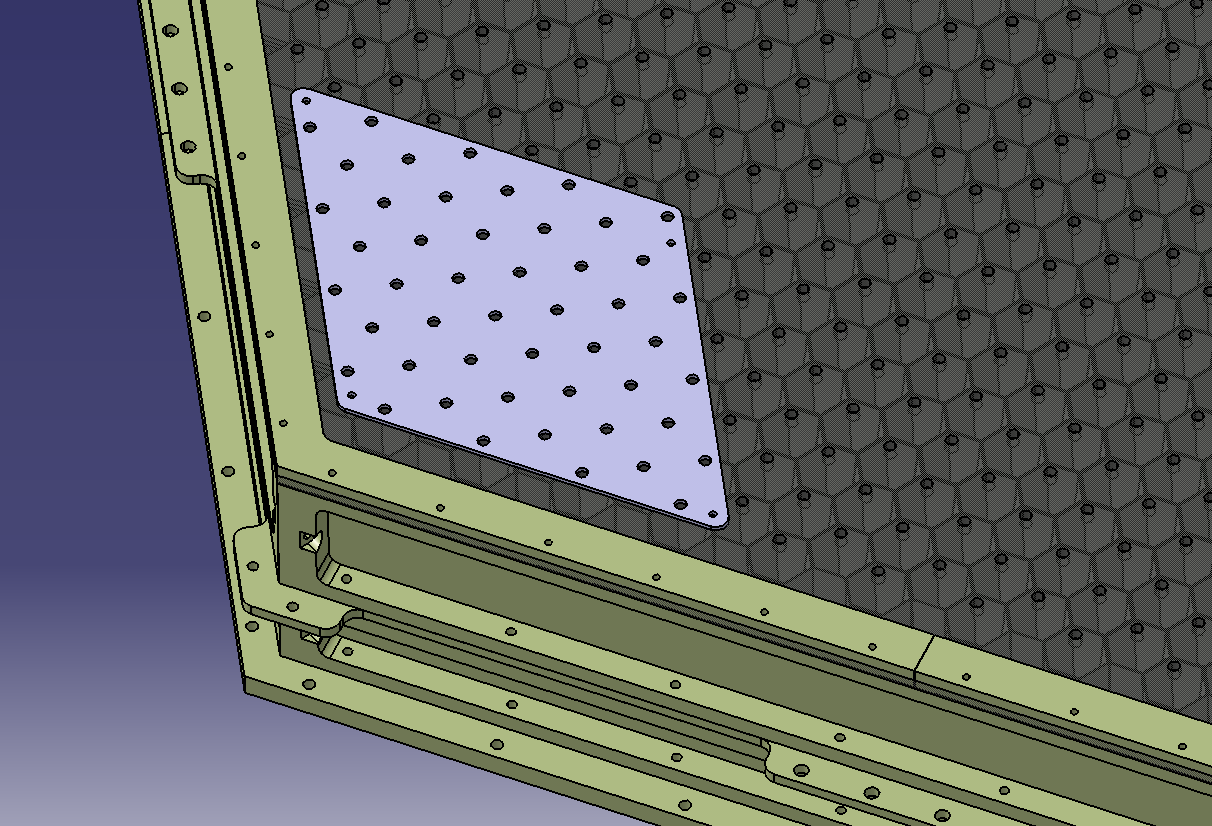


Figure 2: Mechanical element on which the LED carrier board is fixed

# LED system electronic description

## Introduction

The LED system is divided in two boards (Figure 3):

* the LED Carrier Board which contains 48 AC pulsing LEDs & 48 DC continuous LEDs, the associated voltage DC/DC converters & the LEDs regulators
* the Driver board which contains 4 microcontrollers, the DAC for the LEDs regulators, the CAN interface, some logic for the triggering function and the DC/DC converters used for digital & analog ICs

This system is designed to be able to generate short light pulses (AC mode) and provide continuous light too (DC mode). Each mode uses 48 LEDs (so 96 LEDs per carrier board) in order to test 48 pixels (1 LED AC + 1 LED DC by pixel tested) of the CTA camera.

Figure 3: Bloc diagram of the LED driver and the LED carrier board

## System Description

### AC LEDs Control

In this mode the LEDs intensity is controlled by group of 3 LEDs i.e. 16 linear regulators regulates the 48 LED of the AC part of the LED carrier board. The regulators are controlled by the 2 x 8-channels DAC located on the AC-Control part of the LED driver board

Each LED can be switched on or off individually by the µCs of the AC-Control part of the LED driver board. The LED can also be synchronized by an external trigger signal available on the LED driver board

### DC LEDs Control

In this mode the LEDs intensity is controlled globally i.e. 1 linear regulator regulates the 48 LED of the DC part of the LED carrier board. The regulator is controlled by only one channel of the 8-channels DAC located on the DC-Control part of the LED driver board

Each LED can be switched on or off individually by the µCs of the DC-Control part of the LED driver board.

## Boards description

### LED driver board

The board is composed of:

* 4 microcontrollers: two for the AC LEDs and two for the DC. A microcontroller is dedicated to control 24 LEDs and 8 channels of its associated DAC. Each of them have an independent hardware address set by 4 resistors so the same firmware can identify its affectation i.e. either AC µC 0 or 1, either DC µC 0 or 1
* 4 DAC of 10 bits: two for the AC LEDs part and two for DC part although in this case only one is necessary and used. TBD range/resolution values
* For each microcontroller, there is one more LED that can be used. In this case, this LED is able to blink more or less rapidly depending of microcontroller state. TBD state & period values
* Components
* TBD

### LED carrier board

TBD

### LED driver dispatch board

TBD

## Hardware Interfaces

### LED driver board interface

TBD

* 2 Inputs :
  + Lemo : to provide the pulse generator signal for AC LEDs
  + 24 V IN connector : to provide an alimentation of 24 V
* Input/Output : CAN connector : to receive and send frame on CAN Bus
* 4 Output :
  + AC LEDs : enables the interface between the LED Driver Board and the LED Carrier Board for the 48 AC LED, their control bus and their 16 linear regulators
  + Power Supply: enables to aliment the LED Carrier Board.
  + +/-5 V alimentation: enables to aliment LEDs on the LED Carrier Board.
  + DC LEDs : enables the interface between the LED Driver Board and the LED Carrier Board for the 48 DC LEDs, their control bus and their unique linear regulator

### LED carrier board interface

TBD

### LED driver dispatch board interface

TBD

## CAN Protocol interface

### CAN addressing

#### CAN addressing

The CAN Standard Identifier is coded on 11-bits. The number of nodes is 11 x 4µC=48 (maximum=9 x 3 x 4µC=108 for a full camera eventual test), coded on 7-bits:

- 108 nodes

- 1 master

- 1 uninitialized module (unknown address and/or bootload from firmware)

The network ID for a camera is coded on 4-bits and corresponds to the MSBs of CAN standard ID: **110xb** with x=0 (dominant) for a master request to a dedicated module, and 1 (recessive) for a slave answer

2 different communication schemes can be envisaged:

* Broadcast request for all modules. Typically a “temperature get” request, or a bias voltage overall set. In any case the answer or acknowledge from the nodes should be scheduled in order to anticipate contention despite the fact CAN protocol handles it automatically anyway with the CAN ID priorities (dominant/recessive). In this case each node answers sequentially with a delay multiplied by its node address. This delay corresponds to the frame answer added to a security latency duration. It is also possible for the master to broadcast a request without any answer from the slaves (silent mode)
* Single request for 1 module only. The answer from the node is always required but contention is not an issue

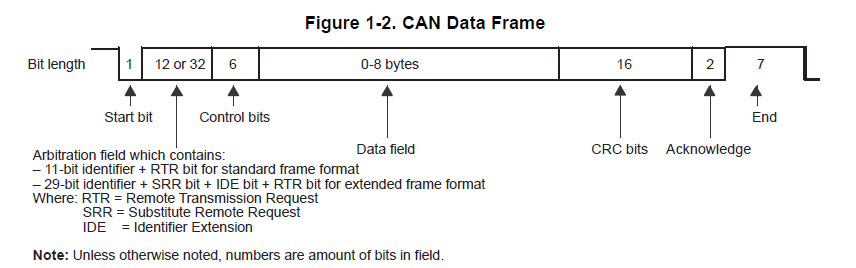
The priority is given to master requests i.e. it must have the lowest value on the camera network ID : **1100b**. In case of master request, the following IDs are available:

* CAN ID = **110 0**000 0000b = 0x600 for master request to all modules **with** answer (broadcast)
* CAN ID = **110 0**000 0001b = 0x601 for master request to all modules **without** answer (broadcast)
* CAN ID = **110 0**000 0010b = 0x602 for master request to module 1
* CAN ID = **110 0**110 1101b = 0x66D for master request to module 108
* CAN ID = **110 0**111 1111b = 0x67F for master request to uninitialized node

The priority is then given to the slaves answers are sorted by module address and the camera network ID **1101b**:

* CAN ID = **110 1**000 0010b = 0x682 for module 1 answer (highest answer priority)
* CAN ID = **110 1**110 1101b = 0x6ED for module 108 answer
* CAN ID = **110 1**111 1111b = 0x6FF for uninitialized node answer (lowest answer priority)

When sending a message, the node (master or module) sends its CAN ID and up to 8 data bytes per frame as shown on Figure 4.



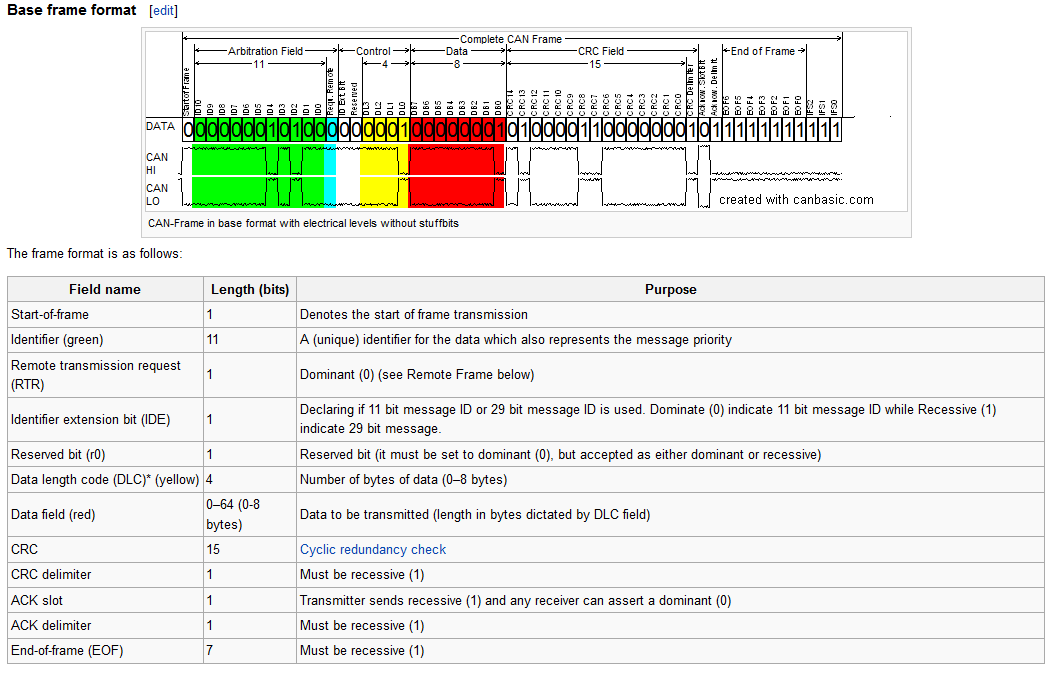


Figure 4 : CAN data frame

The details of all the requests and answer are described in the next paragraph.

The master is always the initiator of a communication with the nodes except in automatic answer mode for temperature readout.

### CAN messaging: data exchange

#### CAN messaging data

Either for a master request or a slave answer the data field is composed of 8 bytes and is described below:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Data field | | | | | | | |
| Byte 0 | Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | Byte 7 |
| Rq/Ans ID +  frame number | PARAMETERS | | | | | | |

The byte #0 of the 8-bytes data field always corresponds to the request or the answer ID and the frame number in case of a multi-frame request:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Data byte 0 | | | | | | | |
| Bit-7 | Bit-6 | Bit-5 | Bit-4 | Bit-3 | Bit-2 | Bit-1 | Bit-0 |
| Frame number (0-7) | | | Request/Answer ID (0-31) | | | | |

The 7 remaining bytes of the message are the request/answer parameters

**Important remark**: **the node address (CAN ID LSB = 0x7F) corresponds to an uninitialized slave**, i.e. booted from the “safety” firmware (exit from manufacturing or safety bootloader automatically reloaded after a firmware download error). In this case, **the only request accepted by the slave will be the “Address set” Request** **and with an individual node access** i.e. the request is rejected when coming from a broadcast request.

### CAN messaging: Master requests/answers summary

#### Master requests/answers summary

|  |  |  |  |
| --- | --- | --- | --- |
| **Request** | **ID** | **Frame NB** | **Parameters/frame** |
| Abort | 0x00 | 1 | None |
| Set CAN Address | 0x01 | 1 | 2 bytes |
| Set DAC level | 0x02 | 2 | 3 bytes |
| Set LED | 0x03 | 1 | 4 bytes |
| Get LED & DAC values | 0x04 | 1 | 1 byte |
| Get version | 0x1E | 1 | 0 byte |

|  |  |  |  |
| --- | --- | --- | --- |
| **Answer** | **ID** | **Frame NB** | **Parameters/frame** |
| Abort | None | None | None |
| General answer | 0x00 | 1 | 1 byte |
| Set CAN Address | 0x01 | 1 | 1 byte |
| Set DAC level | 0x02 | 1 | 1 byte |
| Set LED | 0x03 | 1 | 1 byte |
| Get LED & DAC values | 0x04 | 3 | 6 bytes |
| Get version | 0x1E | 1 | 3 bytes |

### CAN messaging: Master requests/answer details

#### List of requests

##### REQ: Abort

* + Description: abort a previous request (broadcast or individual)
  + Command ID: **0x00**
  + Frame number: 1
  + Parameters: None

##### ANS: Abort

* + There is no answer from slave on a master “abort” request

##### ANS: General error

* + Description: answer from an unrecognized or error request
  + Answer ID: **0x00**
  + Frame number: 1
  + Parameters: 1 byte

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data field | | | | | | |
| Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | Byte 7 |
| Error code | NA | NA | NA | NA | NA | NA |

* + - **Byte 1 : Error code :** 
      * 1: unknown command
      * 2: wrong parameters number
      * 3: wrong frame number sequence (NB must be incremented by 1)
      * 4: frame sequence timeout (multi-frame request)
      * 5: command not yet supported
      * 6: wrong expected command (multi-frame request)
      * 7: rejected command on uninitialized node (see “Important remark” on p.15)
      * 8: incorrect value type for a “get LED & DAC values” request
      * 9: protection error (parameter set request upon a still protected parameter)

REQ: Set CAN ADDR

* + Description: set CAN Address for microcontrollers, unprotect parameter too
  + Command ID: **0x01**
  + Frame number: 1
  + Parameters: 2 bytes

With a board not initialized, all microcontrollers have the same CAN address 0x67F in request and 0x6FF in answer. When the address of a board is set, the address value specified is written at the same time on the four microcontrollers of the board and then their firmware attribute at each other the CAN address + hardware address of each microcontrollers (0 to 3)

* For each board the CAN address value for the request set Can ADDR must be a multiple of 4 as 0x602, 0x606, 0x60A, etc. The other values such as 0x603, 0x604, 0x605, etc. must be rejected as request parameter.
* Once initialized, each µC of board can be accessed individually through its own address ie. 0x602 for µC HW addr. 0, 0x603 for µC HW addr. 1, 0x604 for µC HW addr. 2 and 0x604 for µC HW addr. 1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data field | | | | | | |
| Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | Byte 7 |
| Variable Type | Value | NA | NA | NA | NA | NA |

* + - **Byte 1: Variable Type :** 
      * 0: Unprotect parameter (8-bits value)
      * 1: CAN Slave address (8-bits value, protected, EEPROM stored), rejected in broadcast mode
    - **Byte 2 : 8-bits Value:**
      * **Unprotect parameter :** disable the protection of the following parameters when the value is set to its corresponding number:
        + **7: CAN Slave address:** disable protection for SCB slave address set. This bit is automatically cleared after a SCB Slave address set request
      * **CAN Slave address:** CAN address of the module from 1 (0x01) to 108 (0x6C) modulo 4 (see remark above)

ANS: Set CAN ADDR

* + Description: request to return error about set CAN address.
  + Frame number: 4
  + Parameters: 1 byte

Even if the set CAN ADDR command consists on one single frame to a specific CAN address, each of the four microcontrollers on board answers to it with one frame.

The CAN ID on each of the four frames corresponds to the CAN address of the node that sent the frame.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data field – Frame #0 | | | | | | |
| Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | Byte 7 |
| Error code | NA | NA | NA | NA | NA | NA |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data field – Frame #1 | | | | | | |
| Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | Byte 7 |
| Error code | NA | NA | NA | NA | NA | NA |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data field – Frame #2 | | | | | | |
| Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | Byte 7 |
| Error code | NA | NA | NA | NA | NA | NA |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data field – Frame #3 | | | | | | |
| Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | Byte 7 |
| Error code | NA | NA | NA | NA | NA | NA |

* + - **Byte 1: Error code :**
      * 0 : acknowledge (no error)
      * 1 : out of range slave address error
      * 2 : EEPROM error on Address
      * 3 : broadcast command rejected
      * 4 : wrong variable type request
      * 5 : wrong protect parameter request
      * 6 : forbidden CAN address (i.e. not modulo 4 address, see remark above)

REQ: Set DAC level

* + Description: set DAC level 10 bits
  + Command ID: **0x02**
  + Frame number: 1
  + Parameters: 3 bytes

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data field | | | | | | |
| Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | Byte 7 |
| Channel Num | DAC level (MSB) | DAC level (LSB) | NA | NA | NA | NA |

* + - **Byte 1 : Channel value:**
      * [0-7] : set the value for only the specified channel
      * 8 = set the value for all channels
    - **Byte 2 [MSB] & Byte 3 [LSB] : Value of DAC level**
      * Range: TBD
      * Coding:
        + 10-bits: 0x000 to 0x3FF with Byte1[1..0] & Byte2

Specificities:

* AC part, HW address 0 & 1:
  + all channels (0 à 7) are available
* DC part:
  + HW address 2: only the channel 0 is available, the others channels (1 to 7) are rejected
  + HW address 3: all channels (0 à 7) are rejected

##### ANS: Set DAC level

* + Description: answer error about set DAC level request
  + Command ID: **0x02**
  + Frame number: 1
  + Parameters: 1 byte

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data field | | | | | | |
| Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | Byte 7 |
| Error code | NA | NA | NA | NA | NA | NA |

* + - **Byte 1: Error code :**
      * 0 : acknowledge (no error)
      * 1 : out of range or forbidden (DC) DAC channel
      * 2 : out of range value

##### REQ: Set LED

* + Description: request to turn On/Off LED
  + Command ID: **0x03**
  + Frame number: 1
  + Parameters: 4 bytes

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data field | | | | | | |
| Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | Byte 7 |
| LED  (23 to 16) | LED  (15 to 8) | LED  (7 to 0) | Global ON | NA | NA | NA |

* + - **Byte 1 to 3 :** 
      * Bit = 0 : corresponding led OFF
      * Bit = 1 : corresponding led ON
    - **Byte 4 : global ON**
      * 0: DC/DC converter OFF
      * 1: DC/DC converter ON

NB: the Global ON = 1 command is rejected on µC HW address 1 & 3. HW address 0 µC controls the DC/DC converter for the AC part, HW address 2 µC controls the DC/DC converter for the DC part.

##### ANS: Set LED

* + Description: request to return error about turn On/Off LED
  + Command ID: **0x03**
  + Frame number: 1
  + Parameters: 1 byte

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data field | | | | | | |
| Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | Byte 7 |
| Error code | NA | NA | NA | NA | NA | NA |

* + - **Byte 1: Error code :**
      * 0 : acknowledge (no error)
      * 1 : global ON rejected on µC HW address 1 & 3

##### REQ: Get LED & DAC values

* + Description: request to get LED status and DAC level values
  + Command ID: **0x04**
  + Frame number: 1
  + Parameters: 1 byte

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data field | | | | | | |
| Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | Byte 7 |
| Variable type | NA | NA | NA | NA | NA | NA |

* + - **Byte 1 :** 
      * 0: LED status
      * 1: DAC level for all channels

ANS: Get LED & DAC values

* + Description: answer to return LED status and DAC level values
  + Command ID: **0x04**
  + Frame number: 1 or 3
  + Parameters: 3 or 6 bytes/frame

For the **variable type = 0** set from a master request, the answer is the following type:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data field – Frame #0 | | | | | | |
| Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | Byte 7 |
| LED  (23 to 16) | LED  (15 to 8) | LED  (7 to 0) | Global ON | NA | NA | NA |

* + - **Byte 1 to 3 :** 
      * Bit = 0 : corresponding led OFF
      * Bit = 1 : corresponding led ON
    - **Byte 4 : global ON**
      * 0: DC/DC converter OFF (always 0 for µC addr. 1 & 3)
      * 1: DC/DC converter ON

For the **variable type = 1** set from a master request, the answer is the following type

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Data field – Frame #0 | | | | | | | |
| Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | | Byte 7 |
| DAC level (MSB) | DAC level (LSB) | DAC level (MSB) | DAC level (LSB) | DAC level (MSB) | DAC level (LSB) | | NA |
| Channel#0 | | Channel#1 | | Channel#2 | |  | |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Data field – Frame #1 | | | | | | | |
| Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | | Byte 7 |
| DAC level (MSB) | DAC level (LSB) | DAC level (MSB) | DAC level (LSB) | DAC level (MSB) | DAC level (LSB) | | NA |
| Channel#3 | | Channel#4 | | Channel#5 | |  | |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Data field – Frame #2 | | | | | | | |
| Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | | Byte 7 |
| DAC level (MSB) | DAC level (LSB) | DAC level (MSB) | DAC level (LSB) | 0 | 0 | | NA |
| Channel#6 | | Channel#7 | | NA | |  | |

**NB: in case of error due to an incorrect variable type, a general error answer will be returned with error code = 8**

##### REQ: Get version

* + Description: request for status & firmware version of the microcontroller module
  + Command ID: **0x1E**
  + Frame number: 1
  + Parameters: 0 byte

ANS: Get version

* + Description: answer from master “status get” request
  + Command number: **0x1E**
  + Frame number: 1
  + Parameters: 3 bytes

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Data field – Frame #0 | | | | | | | | | | | |
| Byte 1 | Byte 2 | | Byte 3 | | Byte 4 | | Byte 5 | | Byte 6 | | Byte 7 |
| MSB | LSB | | hw\_version | | NA | | NA | | NA | | NA |
| Firmware version | |  | |  | |  | |  | |  | |

* + Byte 1&2 : returns the firmware version on 16-bits :

|  |  |
| --- | --- |
| Firmware version (16-bits) | |
| Bit-15..Bit-4 | Bit-3..bit-0 |
| Major revision (12-bits) | Minor revision (4-bits) |

The byte 3 : hw\_version is only available for µC addr 0. The others will set the byte 3 to 0.

NB: 0xFAFE is reserved word for the bootloader (i.e. revision 4015.14)

* 1. Microcontroller software

### Software analysis

The Figure 5 presents the software architecture of the microcontroller



Figure 5 : simplified UML software architecture

For simplification, and despite the fact the µC code will not be programmed either in C++ nor in C with real objects structure, the object classes presented in Figure 5 rather corresponds to “C modules” since there are instanced only once. In other words, a module corresponds to a .c/.h file and its public members and public methods to extern variables and procedures.

The main procedure is not shown but will be described below.

Neither private members nor private methods except the interrupt service routines (isr) are shown. Please refer to the detailed description of the objects and to c code for a detailed description.

### Detailed description of the objects:

* **Main procedure:**
  + Summary:
    - Call all the init() methods of all objects.
    - Call the CI\_run() command in the background task (infinite loop)
* **DAC\_DACdriver:**
  + Public methods:
    - DAC\_init(): initialization of the module
    - DAC\_getValue(): get the DAC value of the channel
    - DAC\_write(): start the SPI communication with the DAC either for a dedicated channel and its associated value, either for a broadcast to all channels with the same value. This function doesn’t use any interrupt (polling)
  + Isr: None
* **TIM\_Timer:**
  + Public methods:
    - TIM\_init(): initialization of the module. A pointer to a function must be pass in order to connect the call back function executed in the isr
    - TIM\_ack(): acknowledge **must** be called from user in order to avoid overrun
    - TIM\_setPeriod(): stop the current timer, set the new timer period and restart it
    - TIM\_getPeriod(): returns the current timer period
    - TIM\_stop(): stops the timer
    - TIM\_start(): starts the timer
  + Isr:
    - Called at each period of the timer: verify any overrun and if ok, executes the call-back passed during init() function. This call-back should execute the LED blinking procedure.
* **CAN\_CANdriver:**
  + Public methods:
    - CAN\_init(): initialization of the module, parameter = CAN address of the node
    - CAN\_setID(): set the CAN address of the node
    - CAN\_read(): blocking method up to a new CAN frame reception. The function returns the data of the messages and freed the CAN peripheral buffer. This function is not using any interrupt (polling)
    - CAN\_write(): This method sends data over the bus and the transmission can be delayed (parameter) according to the address of the slave in case of an answer to a broadcast request. The method is blocked upon completion of the write or in case of an abort request from the master. This function is not using any interrupt (polling)
    - CANwriteIsr():This method directly sends data over the bus. It can be called within an interrupt service routine and is mainly used when warning are sent over the bus with a slave initiative
  + Isr:
    - isrRx(): Called at the end of a valid frame reception
* **EEP\_Eeprom:**
  + Public methods:
    - EEP\_init(): initialization of the module
    - EEP\_setByte(): set 1 byte in the EEPROM. This function is not using any interrupt (polling)
    - EEP\_getByte(): write 1 byte in the EEPROM. This function is not using any interrupt (polling)
    - EEP\_setWord(): set 1 word in the EEPROM. This function is not using any interrupt (polling)
    - EEP\_getWord(): write 1 word in the EEPROM. This function is not using any interrupt (polling)
  + Isr: None
* **CI\_CommandInterpreter:**
  + Public methods:
    - CI\_init(): initialization of the module. During this init() phase, the CAN address is read in EEPROM in order to verify if the parameter value differs from 0xFFFF which is the EEPROM erased default value. If it differs, the CAN ID is set to the EEPROM value + µC HW address, else if the EEPROM address is 0xFFFF then the CAN ID is set to default address + µC HW address.
    - CI\_run(): run the state machine executing the command interpreter of the CAN messages. The message is decoded when received, then if the command is recognized it is executed internally and an answer message is sent back to the master. This function must be used in the background task of the µC
  + Isr: None
* **Initialization phase:**
  + The following actions are performed during initialization of the module:
    - Initialize all the modules