ICARE Development Guide
-Draft-

Authors:
Laurent GANTEL
Sylvain LAFRASSE

30.03.2018- 17:19

Contents

1 Introduction **3**
1.1 ATLAS Overview . . . 3
1.1.1 Detectors Components . . . 3
1.1.2 Back-End System . . . 4
1.2 ATCA Platform . . . 5
1.2.1 Introduction . . . 6
1.2.2 IPMI Architecture . . . 7
1.2.3 IPMI Protocol . . . 7
1.3 IPMC Board . . . 7
1.3.1 Introduction . . . 7
1.3.2 Hardware Overview . . . 8
1.4 AMC Board . . . 8
1.4.1 Introduction . . . 9
1.4.2 Hardware Overview . . . 9

2 Development Tools **10**
2.1 ICARE Firmware . . . 10
2.1.1 Environment Setup . . . 10
2.1.2 Working Directories . . . 11
2.1.3 OpenOCD . . . 11
2.1.4 ICARE Source Code . . . 13
2.1.5 Package Development . . . 13
2.1.6 Firmware Upgrade . . . 15
2.2 MMC Firmware . . . 18
2.2.1 Atmel Studio . . . 19
2.2.2 Software Organization . . . 19
2.2.3 Firmware Upgrade . . . 21

3 ICARE Developments **31**
3.1 SPI Package . . . 31
3.1.1 SPI Multi-Master Feature . . . 31

3.1.2 DMA Feature . . . 34
3.2 CMC Package . . . 37
3.2.1 Testbench Feature [Deprecated] . . . 37
3.3 LArC Commands Package . . . 38
3.3.1 Command Parsing Feature . . . 38
3.3.2 Adding a New Command . . . 39
3.4 UART Listener Package . . . 42
3.4.1 Description . . . 42
3.4.2 Usage . . . 42

4 MMC Developments 44
4.1 User Sensors . . . 44
4.1.1 Sensors list . . . 44
4.1.2 Alerts and Sensors Thresholds . . . 45
4.1.3 LATOME Sensors Thresholds . . . 47
4.1.4 Sensors Data Conversion . . . 52
4.1.5 SDR Values: Experiments on the Shelf . . . 54
4.2 Task Management . . . 55
4.2.1 Introduction . . . 55
4.2.2 Task Control Block . . . 57
4.2.3 Scheduler . . . 57

A ICARE firmware compilation on CERN-SLC6 59

B Hard fault: Retrieve the faulty line 60

C Locally install OpenOCD on CERN-SLC6 61

D Status 63
D.1 ATCA Boards status . . . 63

E LArC Power Configuration 64

Chapter 1

Introduction

Contents

1.1 ATLAS Overview **3**
1.1.1 Detectors Components 3
1.1.2 Back-End System 4
1.2 ATCA Platform **5**
1.2.1 Introduction 6
1.2.2 IPMI Architecture 7
1.2.3 IPMI Protocol 7
1.3 IPMC Board **7**
1.3.1 Introduction 7
1.3.2 Hardware Overview 8
1.4 AMC Board **8**
1.4.1 Introduction 9
1.4.2 Hardware Overview 9

1.1 ATLAS Overview

1.1.1 Detectors Components

The ATLAS platform is composed of three detectors[1] as shown in *Figure 1.1*:

- The inner detector, including Pixel and the SCT and TRT trackers, measures the momentum of each charged particle

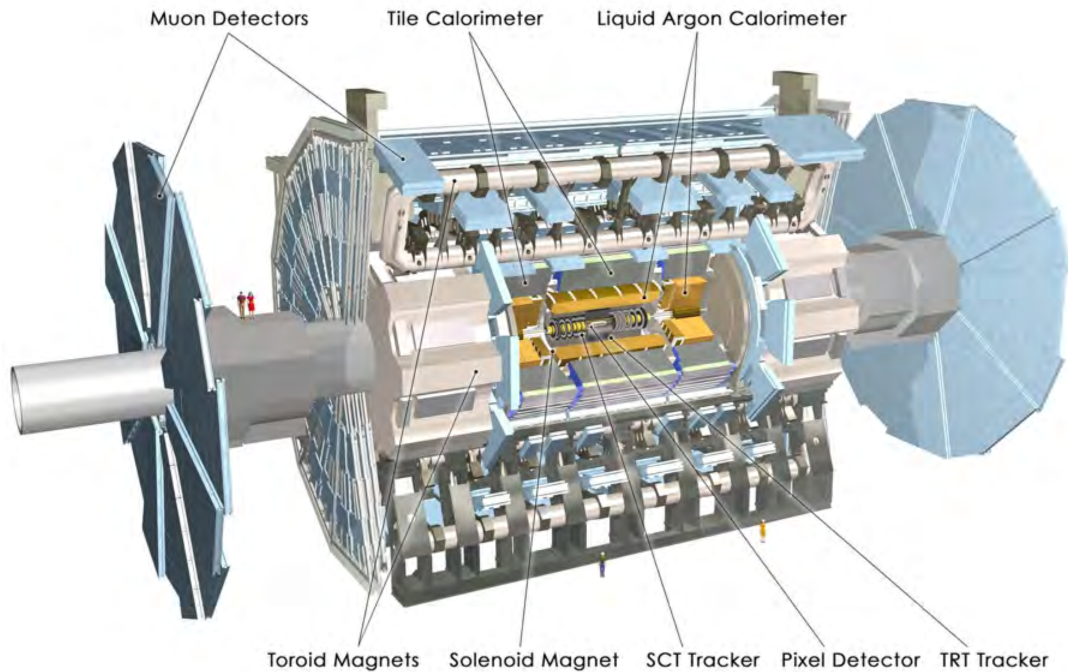


Figure 1.1: ATLAS detectors

- The calorimeters relies on liquid Argon or dedicated tiles to measure the energies carried by the particles
- The muon spectrometer identifies and measures the momenta of muons

These detectors are connected to the electronic Front-End. In our case we are interested in the calorimeter data and responsible for managing the Back-End system, which is located just after the Front-End.

1.1.2 Back-End System

The back-end system is composed of several blocks processing the data coming from the Front-End *(Figure 1.2)*. Data consists in Level1-Accept *(L1A)* signals indicating that a partition (ie. a region) of the detector detected an interesting level of energy.

The first element of the back-end is the Read-Out Driver (ROD). It gets data from the Front-End Board (FEB), which are controlled by the Trigger Timing Control (TTC) system, itself controlled by the Central Trigger Processor (CTP).

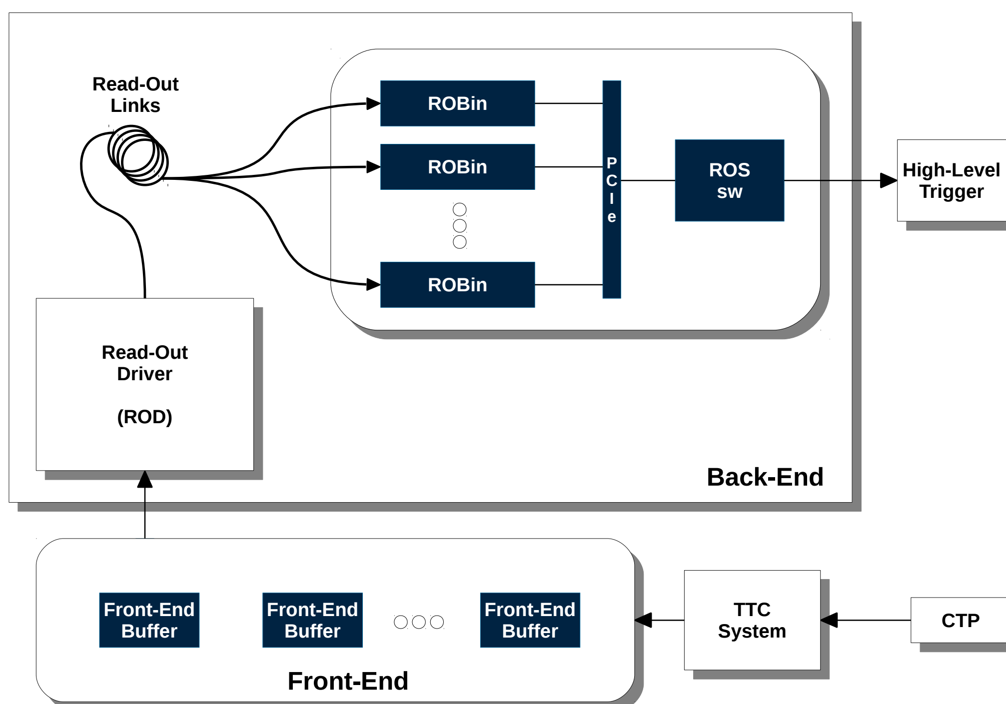


Figure 1.2: Back-End system

Data from the FEB are handled by the ROD. It contains:

- an input FPGA used to re-organize data for computing efficiency
- a Processing Unit (PU) composed of two TMS320C6414 DSP from Texas Instrument which compute the energy, the form factor and the timing of the energy pikes
- and an output FPGA which re-organizes the data and sends it to the Read-Out System (ROS) through dedicated Read-Out Link (ROL)

The ROS includes Read-Out Buffer INput (ROBIN) boards which are responsible for receiving and buffering the event data fragments from the RODs. These boards are connected on a PCIe bus and communicate with a host called the ROS software. On request from the High-Level Trigger (HLT), this piece of code transfers the interesting data to the latter which is a server cluster.

1.2 ATCA Platform

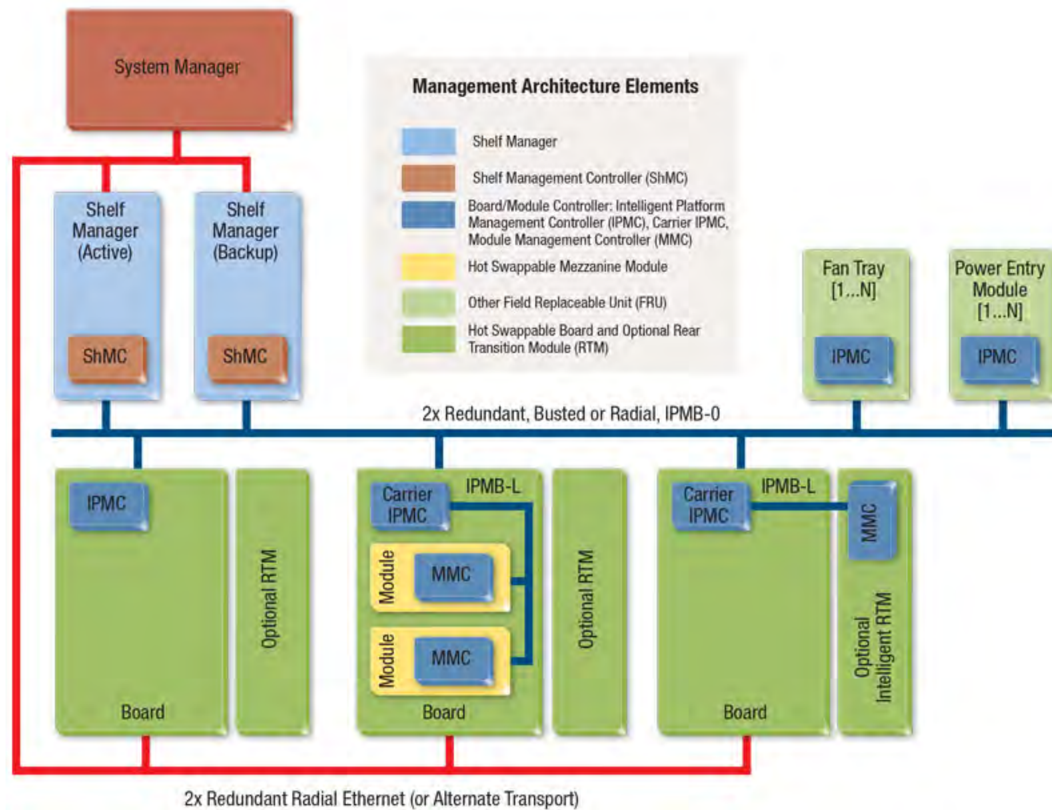


Figure 1.3: ATCA platform

1.2.1 Introduction

The management of the boards inside the Advanced Telecom Computer Architecture (ATCA) shelf is done by an Intelligent Platform Management Controller (IPMC) present on each board. This module provides the ability to manage the power, the cooling and the interconnections needed by the monitored devices. Its role is to monitor events and to log them into a central repository *(System Event Log (SEL))*.

An IPMC can monitor ATCA board or ATCA Carrier Board via System Management Bus (SMBus) which is an I^2C-like bus *(Figure 1.3)*. An ATCA Carrier Board includes one or several Advanced Mezzanine Cards (AMC). Each AMC board is itself monitored by a Module Management Controller (MMC) board.

The IPMC board is composed of two 32-bit Cortex-M4 micro-controllers, one implementing the IPMC protocol, the other being available for the users to access the board peripherals via the additional serial buses. Typically, peripherals are sensors or Ethernet controller.

1.2.2 IPMI Architecture

The Intelligent Platform Management Interface (IPMI) specification defines how to manage IPMC platforms built around an Intelligent Platform Management Bus (IPMB). The IPMC controls the SEL, the Sensor Data Records (SDR) repository and the Field Replaceable Unit (FRU) initialization information.

The SEL induces a hard-coded knowledge of the sensor. To avoid it, the SDR has been created to allow more flexibility in the event format. Actually, the SDR region contains information to link the SDR data with the corresponding FRU.

An initialization agent permits to write default settings on start-up, whereas FRU information should be stored into a non-volatile memory (EPROM). Access to this memory must be possible via JTAG or another way when the IPMB system is down.

1.2.3 IPMI Protocol

The IPMI protocol is based on a Request/Response interface which can be referred as Commands/Responses interface. The Requester and the Responder are defined by unique identifiers and exchange commands and data on the IPMB.

A command is composed of:

- Network Function Codes *(1 byte)*: 6 bits are used to encode the function, the 2 remaining bits are the Logical Unit Number (LUN) field which can be used to access a sub-module managed by the IPMC receiving the message. A function can be of several types, e.g. it can concern chassis, bridges, sensor events, applications, firmwares, storages, transport or custom functionality extensions.
- Optional data related to the command.

A response is composed of:

- Completion Codes: it can be viewed as a status flag. It corresponds to the first byte of the response. If the command is not supported or implemented, a special code 0xFF should be sent. In a normal case where the command is successfully executed, the code should be 0x00.
- Optional data related to the response

1.3 IPMC Board

1.3.1 Introduction

The purpose of the IPMC mezzanine is to handle the communication between the Shelf Manager and the ATCA board in which it is inserted. It implements the IPMI procotol and provides

the user with accesses to sensors serial interfaces such as I^2C or SPI. The typical architecture of an ATCA platform is described in *Figure 1.3*.

An ATCA board can be extended using an optional Rear Transition Module (RTM), or be used as a Carrier board, embedding AMC boards. The system is also composed of Power Entry modules to supply power on the ATCA boards and Fan trays in order to maintain a cool operating temperature.

The Shelf Manager centralizes the information provided by the different ATCA boards and allows the user to monitor them through a request interface *(Webpage or command line)*.

1.3.2 Hardware Overview

The IPMC board is mainly composed of two Micro-Controller Units (MCUs): the IPMC and the Input/Output InterFace (IOIF) *(Figure 1.4)*.

The IPMC executed code is strictly dedicated to manage the IPMB protocol and to communicate with the Shelf Manager.

On the IOIF, a part of the code running should be user-written, to control sensors available on the ATCA board. Addition of user code can be done implementing so-called modules. This chip is also used to update the firmwares of each MCU through Ethernet.

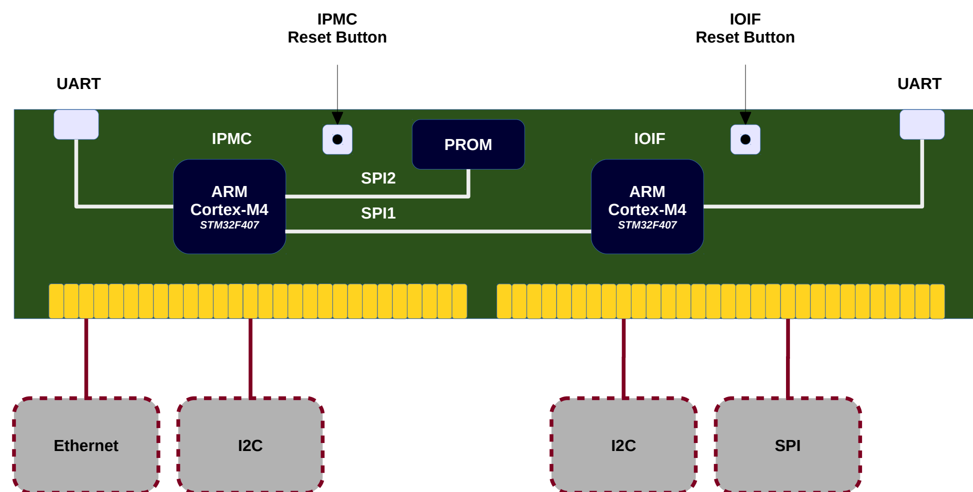


Figure 1.4: IPMC Mezzanine

1.4 AMC Board

1.4.1 Introduction

An AMC board can be plugged in a ATCA Carrier Board. This carrier board owns itself an IPMI controller, the IPMC.

The MMC is an IPMI controller designed to be integrated on AMC boards. In our case of the LAr Trigger prOcessing MEzzanine (LATOME) board *(Figure 1.5)*, the MMC is based on an **ATMEGA128** MCU from Atmel. On one side, the MMC is connected to the IPMB bus. On the other side, it is connected to another bus to access I^2C sensors present on the AMC board, such as temperature, voltage or current sensors.

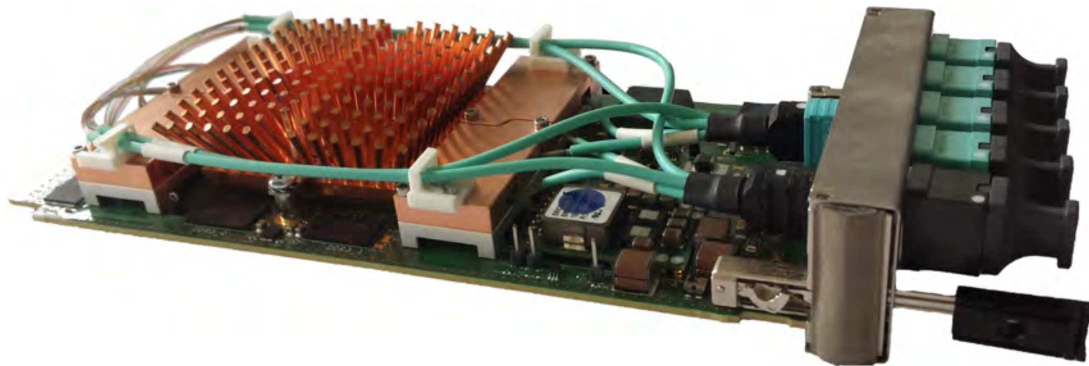


Figure 1.5: LATOME board

The development environment of our MMC is composed of the **Atmel Studio IDE** detailed in Chapter 2.2.1, the LATOME board designed to comply with the AMC format, and a "LATOMETT" docking board to supply the LATOME with all the required power levels. The MMC is then programmed via a custom JTAG cable connected to an ATMEL-ICE Debugger.

1.4.2 Hardware Overview

Hardware details about the LATOME board can be found in the *LATOME reference manual* [2].

Chapter 2

Development Tools

Contents

2.1 ICARE Firmware **10**

2.1.1 Environment Setup 10

2.1.2 Working Directories 11

2.1.3 OpenOCD 11

2.1.4 ICARE Source Code 13

2.1.5 Package Development 13

2.1.6 Firmware Upgrade 15

2.2 MMC Firmware **18**

2.2.1 Atmel Studio 19

2.2.2 Software Organization 19

2.2.3 Firmware Upgrade 21

2.1 ICARE Firmware

The Intelligent platform management Controller softwARE (ICARE) firmware is the application running on top of the IPMC mezzanine. It is compiled both for the IPMC and the IOIF MCUs.

2.1.1 Environment Setup

In addition to the information provided in the next subsections, the setup of the ICARE environment and its configuration are described in the document written by Guy Perrot [3]. It introduces the installation of the tools and drivers for Windows 8.1, and details the commands to get a fresh local copy of the firmware repository.

2.1.2 Working Directories

LAPP installation:

When working at **Laboratoire d'Annecy de Physique des Particules (LAPP)**, two folders need to be mounted in order to retrieve the source code and to be able to build the ICARE firmware:

- home1 or home3: this folder contains your user directory. It will be used to work on a local copy of the firmware sources
- software_dev : this directory contains the source code and the tools to build the firmware

On Linux, if not already done, these folders can be mounted using *sshfs*. Here are some aliases that can be pasted into your bash profile *(replace "'username"' by your own user name)*:

```
alias mount_home1='sshfs username@lappsl.in2p3.fr:/home1 /home1/'
alias mount_software='sshfs username@lappsl.in2p3.fr:/software_dev /software_dev'
```

To unmount them, define and use the following aliases:

```
alias unmount_home1='fusermount -u /home1'
alias unmount_software='fusermount -u /software_dev'
```

Local installation:

When working outside the LAPP network, the ICARE firmware can be installed locally using the auto-extractable archive that can be download on the ICARE website. The installation procedure can be found at the same location:

http://lappwiki.in2p3.fr/twiki/bin/view/AtlasLapp/Informatique

2.1.3 OpenOCD

In order to load the firmware on the IPMC board, the following tools are required:

- The ICARE environment previously installed
- The Olimex JTAG programmer with the OpenOCD software

The Olimex JTAG programmer can be used on SLC6, SLC7 or macOS 10.12+ with openocd-0.9.0 and later. On Windows 10, the Olimex drivers are working from the version **2.3** of the **Zadig** software. With previous version of Zadig *(e.g. 2.2)*, one has to use the NGX JTAG interface instead.

The OpenOCD tool is available on the Linux machines at LAPP in the directory mentioned below. The connection with the board can be made using the following commands.

1. First, load the ICARE environment. At this time, the latest release is 00-02-00:

```
$ source /software_dev/atlas/project/ICARE/releases/ICARE-00-02-00/admin/vor6/cmt/setup.sh
```

2. Then, go to the OpenOCD directory and connect to the board through the Olimex JTAG adapter:

```
$ cd /software_dev/atlas/project/ICARE/
$ cd ./contrib/openocd/Linux/openocd-0.9.0-201505190955/scripts/
$ openocd -f interface/olimex-arm-usb-ocd-h.cfg -f target/ipmcv2_1.cfg
```

It should display the following output:

```
[laurent@localhost scripts]$ openocd -f interface/olimex-arm-usb-ocd-h.cfg -f target/ipmcv2_1.
    cfg
GNU ARM Eclipse 64-bits Open On-Chip Debugger 0.9.0-00073-gdd34716-dirty (2015-05-19-09:57)
Licensed under GNU GPL v2
For bug reports, read
  http://openocd.org/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'jtag'
adapter speed: 1000 kHz
adapter_nsrst_delay: 100
cortex_m reset_config sysresetreq
Warn : Using DEPRECATED interface driver 'ft2232'
Info : Consider using the 'ftdi' interface driver, with configuration files in interface/ftdi
    /...
Info : max TCK change to: 30000 kHz
Info : clock speed 1000 kHz
Info : JTAG tap: ipmc.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0x4)
Info : JTAG tap: ipmc.bs tap/device found: 0x06413041 (mfg: 0x020, part: 0x6413, ver: 0x0)
Info : JTAG tap: cpld.cpld tap/device found: 0x020a20dd (mfg: 0x06e, part: 0x20a2, ver: 0x0)
Info : JTAG tap: ioif.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0x4)
Info : JTAG tap: ioif.bs tap/device found: 0x06413041 (mfg: 0x020, part: 0x6413, ver: 0x0)
Info : ipmc.cpu: hardware has 6 breakpoints, 4 watchpoints
Info : ioif.cpu: hardware has 6 breakpoints, 4 watchpoints
```

3. Once the connection is done, you can launch a telnet connection to interact with the MCUs thanks to OpenOCD or GDB commands:

```
$ telnet localhost 4444
```

The following output should be displayed:

```
Trying ::1...
telnet: connect to address ::1: Connection refused
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
>
```

Information about the two MCUs can be printed using the 'targets' command:

```
> targets
    TargetName         Type       Endian  TapName            State
--  ------------------ ---------- ------  ------------------ ------------
 0  ipmc.cpu           cortex_m   little  ipmc.cpu           running
 1* ioif.cpu           cortex_m   little  ioif.cpu           running
>
```

2.1.4 ICARE Source Code

When working at LAPP, the firmware source code can be fetched to a local directory using the following commands.

First, load the ICARE environment:

```
$ source /software_dev/atlas/project/ICARE/releases/\
ICARE-00-02-00/admin/vor6/cmt/setup.sh
```

Create a new local repository folder, for instance ICARE_workarea:

```
$ cd /home1/<username>/
$ mkdir ATLAS && cd ATLAS
$ cmt create_project ICARE_workarea
```

Finally in project.cmt, add the following line setting the ICARE version you want to use:

```
$ nano project.cmt
...
# Add the following line:
use releases ICARE-00-02-00
...
```

Several aliases can be created to ease the load of the ICARE environment:

```
# ICARE aliases
alias source_icare='source /software_dev/atlas/project/ICARE/releases/ICARE-00-02-00/admin/vor6/cmt/
    setup.sh'
alias cd2openocd='cd /software_dev/atlas/project/ICARE/contrib/openocd/Linux/openocd
    -0.9.0-201505190955/scripts/'
alias cd2icare_workarea='cd /home1/username/ATLAS/ICARE_workarea/'
```

2.1.5 Package Development

Package compilation:

By default, your local application will use the remote packages tagged in the SVN repository. If you want to configure a package with CMT or even make changes to the source code, you have to import it in your ICARE work area:

```
$ cd /home1/username/ATLAS/ICARE_workarea/
```

The version format for a package is vxrypz, with *x, y, z* being respectively the version, release and patch numbers. To get the SPI package along with the IMC, type the following command. It will import the spi and imc packages and will increment the revision version of each package *('-i r' option)*. It is also possible to increment the version *('-i v')* or the patch number *('-i p')*:

```
$ getpkg —i r spi imc
```

Once you get the package, you can go to the SPI package CMT directory:

```
$ cd spi/vor5/cmt
```

Use the command **"“make help"”** to get the list of the available commands associated with this packet. If any, the existing tests can be compiled with the command:

```
$ make && make test
```

The binary files can then be uploaded to the Flash using this command:

```
$ make flash_spi_test_1 # in the case of the SPI package
```

Finally, to circumvent manually removing / reinserting the carrier, it is possible to reset the MCUs using the JTAG interface.
Warning: JTAG programming only works with Carrier v3 or newer.

```
$ telnet localhost 4444
$ reset init
$ reset run
```

Sources commitment (SVN only):

Before committing, ensure your local repository URL points to the **trunk** branche:

```
$ svn info
Path: .
URL: file:///software_dev/atlas/repo/ICARE/spi/trunk
Repository Root: file:///software_dev/atlas/repo/ICARE
Repository UUID: 5d6f7848—0351—41c7—bcaa—bd82fec97149
Revision: 661
Node Kind: directory
Schedule: normal
Last Changed Author: gantel
Last Changed Rev: 657
Last Changed Date: 2017—02—08 18:16:25 +0100 (Wed, 08 Feb 2017)
```

Check if new files need to be added and the status of the modified files:

```
$ svn status
```

To see the modifications in a file, use the following command:

```
$ svn diff myFile
```

You can install and configure colordiff to get a fancier output. On Linux SLC6:

```
$ yum install colordiff
```

Then you can configure it as wanted:

```
$ cp /etc/colordiffrc ~/.colordiffrc
$ nano ~/.colordiffrc
```

Choose the colors you want to apply for each type of output. Then configure SVN to use colordiff:

```
$ nano ~/.subversion/config
```

This line allows to configure the editor:

```
editor-cmd = sublime_text
```

This line is to choose the program used to perform the diff:

```
diff-cmd = /usr/bin/colordiff
```

If you are satisfied with the modifications you have done to the package, add the new files and commit:

```
$ svn add myFile
$ svn commit -m "This is my new file" myFile
```

Finally, create a tag:

```
$ cd spi/vor5
$ svn update
$ svn cp . $SVNROOT/spi/tags/vor5 -m "Add Multi-Master mode using hardware NSS"
$ svntags -lv spi
# Should be the version that we just commited
```

2.1.6 Firmware Upgrade

The firmware upgrade utility is necessary to update the firmware through Ethernet and should be compiled with the application. **If needed** re-compile it but it is already integrated to the 00-02-00 repository:

```
$ cd fwUpgrade/vor1p4/cmt
$ make
$ source setup.sh
$ cd ../../../inet/vor1p2/cmt
$ make
$ make flash_inetsrvc
```

Go to your application directory (demo in this example):

```
$ cd ../../../demo/vor1/cmt
$ make
$ source setup.sh
```

If accessible, use the JTAG connection to get the IP address:

```
$ make info
#CMT---> Info: Document mcu_info

==========================
IPMC - OTP area information
==========================
OpenOCD server: lappc-at27

MCU ID:         0x193c
PCB version:    v2.2
Serial Number:  35
__________________________



==========================
IOIF - OTP area information
==========================
OpenOCD server: lappc-at27

MCU ID:         0x101f
PCB version:    v2.2
Serial Number:  35
MAC address:    00:22:8f:02:40:23
IP address:     134.158.98.183
__________________________
```

The IP address can also be retrieved sending a clia command to the shelf:

```
$ clia board sendcmd 2e 07 2e a1 00
Pigeon Point Shelf Manager Command Line Interpreter

Completion code: 0x0 (0)
Response data: 2E A1 00 00 22 8F 02 40 6E 80 8D CA CB FF FF FF 00 80 8D CA 01 01 01 64
```

The address is stored in the bytes 10 to 13. For instance 80 8D CA CB becomes 128.141.202.203.

Load the firmware in the memory, first the IPMC binary:

```
$ fwu -t IPMC -n 134.158.98.183 -f ../arm-gcc47-dbg/bmc_IPMC.bin
+-------------------------------------------------------------+
|                ___ ___   _   ___ ___                        |
|               |_ _/ __| /_\ | _ \ __|                       |
|                | | (__ / _ \|   / _|                        |
|               |___\___/_/ \_\_|_\___|                       |
|                                                             |
| Intelligent plateform management Controller softwARE |
|                                                             |
|               Firmware upgrade utility.                     |
|           Copyrigth (c) 2014-2015 LAPP/CNRS.                |
|                                                             |
+-------------------------------------------------------------+

Host       : 134.158.98.183
Port       : 5555
Target MCU : IPMC
Upgrade MCU: NO

Firmware Upgrade image : ../arm-gcc47-dbg/bmc_IPMC.bin (272.1KB)
```

```
Firmware image checksum: 9f60f71e
Downloading image 'bmc_IPMC.bin' ...OK
Download rate: 24.3KB/s 00:11.219
Programming CONFIGURATION flash memory ...OK
Programming 278652 bytes in CONFIGURATION flash memory successful.


$ fwu -t IPMC -n 134.158.98.183 -u
+--------------------------------------------------------------------+
|                  ___ ___    _    ___ ___                           |
|                 |_ _/ __|  /_\  | _ \ __|                          |
|                  | | (__  / _ \|   /  _|                           |
|                 |___\___/_/ \_\_|_\___|                            |
|                                                                    |
| Intelligent plateform management Controller softwARE              |
|                                                                    |
|                 Firmware upgrade utility.                          |
|            Copyrigth (c) 2014-2015 LAPP/CNRS.                      |
|                                                                    |
+--------------------------------------------------------------------+

Host       : 134.158.98.183
Port       : 5555
Target MCU : IPMC
Upgrade MCU: YES

Send update command ...OK
Performing upgrade stage (take approximately 20 seconds).
```

After waiting approximately 20-50 seconds depending on the size of the binary, repeat the operation for the IOIF binary:

```
$ fwu -t IOIF -n 134.158.98.183 -f ../arm-gcc47-dbg/bmc_IOIF.bin
+--------------------------------------------------------------------+
|                  ___ ___    _    ___ ___                           |
|                 |_ _/ __|  /_\  | _ \ __|                          |
|                  | | (__  / _ \|   /  _|                           |
|                 |___\___/_/ \_\_|_\___|                            |
|                                                                    |
| Intelligent plateform management Controller softwARE              |
|                                                                    |
|                 Firmware upgrade utility.                          |
|            Copyrigth (c) 2014-2015 LAPP/CNRS.                      |
|                                                                    |
+--------------------------------------------------------------------+

Host       : 134.158.98.183
Port       : 5555
Target MCU : IOIF
Upgrade MCU: NO

Firmware Upgrade image : ../arm-gcc47-dbg/bmc_IOIF.bin (226.9KB)
Firmware image checksum: 036fc318
Downloading image 'bmc_IOIF.bin' ...OK
Download rate: 23.7KB/s 00:09.573
Programming CONFIGURATION flash memory ...OK
Programming 232388 bytes in CONFIGURATION flash memory successful.
```

Update the boot memory:

```
$ fwu -t IOIF -n 134.158.98.183 -u
+------------------------------------------------------+
|                ___ ___   _   ___ ___                 |
|               |_ _/ __| /_\ | _ \ __|                |
|                | | (__ / _ \|   / _|                 |
|               |___\___/_/ \_\_|_\___|                |
|                                                      |
| Intelligent plateform management Controller softwARE |
|                                                      |
|               Firmware upgrade utility.              |
|          Copyrigth (c) 2014-2015 LAPP/CNRS.          |
|                                                      |
+------------------------------------------------------+

Host        : 134.158.98.183
Port        : 5555
Target MCU  : IOIF
Upgrade MCU: YES

Send update command ...OK
Performing upgrade stage (take approximately 20 seconds).
```

After the reboot of the IOIF which also takes 20-50 seconds, the IPMC will automatically reboot. Use the JTAG to check the new version currently running:

```
[gantel@lappc-f533 cmt]$ make version
#CMT---> Info: Document mcu_version

==================================
IPMC - Version
==================================
OpenOCD server: lappc-at27

Release ver.: ICARE-00-02-00
Compilator   : (GNU) gcc 4.7.0 [dbg]
Binary image: bmc_IPMC.bin
Build ver.   : Jul 21 2017, 16:40:33
----------------------------------



==================================
IOIF - Version
==================================
OpenOCD server: lappc-at27

Release ver.: ICARE-00-02-00
Compilator   : (GNU) gcc 4.7.0 [dbg]
Binary image: bmc_IOIF.bin
Build ver.   : Jul 21 2017, 16:40:33
----------------------------------
```

2.2 MMC Firmware

2.2.1 Atmel Studio

The Atmel Studio IDE version 7.0 is used to develop on the MMC board which manages power, temperature, and hot-swap of the ATCA carrier boards. It has been tested on Windows 10 (and on Windows 7 through VirtualBox 5.2.6 under macOS 10.13.2).

http://www.atmel.com/tools/ATMELSTUDIO.aspx

2.2.2 Software Organization

Repositories

The Git repository containing the MMC firmware can be found at this address:
https://gitlab.cern.ch/atlas-lar-ldpb-firmware/MMC

A **develop** branch has been added to the repository to host the stabilized features before merging with a release version. The project can be loaded using the project file **MMC/atmega128_mmc.atsln**. On-going developments must be pushed into *feature* or *hotfix* branches.
MMC original code coming from CERN is available from a SVN repository at:
https://svnweb.cern.ch/cern/wsvn/ph-ese/be/mmc_v2/

Two main branches are currently maintained:

- *develop*: contains the intermediate updates ⟵ This is the stable development branch.
- *master*: contains the last stable production version.

To load the .hex file in the LATOME MMC, an Atmel-ICE debugger is used with a custom JTAG ↔ AVR cable.

Documentation

In addition to this manual, some documentation is available from CERN for the MMC project. Check SVN repository at:

https://espace.cern.ch/ph-dep-ESE-BE-uTCAEvaluationProject/MMC_project/default.aspx

Additions to CERN code

UART support:

An UART has been added to the firmware code in order to allow an efficient debugging of the application. The UART communicates at 9600 bauds *(8N1)*. The system clock used to compute the baud-rate register value has been fixed to 4MHz instead of 16MHz in the starting code *Release2*.

In the main function, the pins on the MCU are initialized and interrupt activated to have this functionality. You can plug a USB ↔ Serial cable, and use a serial terminal to watch the debug prints *(TeraTerm or assimilated)*. The classic *printf()* method is available to output messages. But prints to UART are SLOW, having too much could cause timeouts in I^2C exchanges!

```
$ screen -L /dev/tty.usbserial-FTHBSX4H 9600,cs8,-ixon,-ixoff,-istrip

MMC Firmware - Feb 28 2018 13:44:55
--
> MMC init [OK]
00:00:03
            ________________
        VCC_GXB Voltage:    163 mV
        A10_VCC Voltage:    163 mV
        VCC_GXB Current:    200 mA
        A10_VCC Current:    200 mA
        TEMP_FPGA:     1 degC
00:00:06
            ________________
        VCC_GXB Voltage:    163 mV
        A10_VCC Voltage:    163 mV
        VCC_GXB Current:    200 mA
        A10_VCC Current:    200 mA
        TEMP_FPGA:     1 degC
00:00:07
```

Timer for sensor:

A common timer for LTC2495 sensors have been added. It is initialized in the MMC main code, and used in the functions for both software sensors (see .c files for those sensors).

I^2C buses:

A couple of Two Wire Interfaces (TWI) are used as I^2C buses:

- One for the communication with IPMC *(going outside of LATOME board)*
- One for the communication on the board with the sensors

Those I^2C buses are wired on LATOME test board *(A LATOME board without FPGA)* and a

Figure 2.1: Power supply in standby state

bus analyzer or a bus injector can be used to debug and analyze data.

Tools used:
http://www.totalphase.com/products/beagle-i2cspi/ (analyzer)
http://www.totalphase.com/products/aardvark-i2cspi/ (injector)

2.2.3 Firmware Upgrade

Update via JTAG

Hardware Setup:

The MMC firmware can be updated via JTAG using the Atmel IDE and the **LATOMETT** board:

1. The **LATOMETT** board must be supplied with 12V power in order to provide at least the 3.3V needed by the MMC *(ie. the Management Power)*. The supplied power could be higher than 12V to ensure that the regulated power is always correct *(Figure 2.1)*.

2. Insert the LATOME board inside the **LATOMETT** *(Figure 2.2)*

3. Plug the Atmel-ICE JTAG programmer into the MMC front panel *(Figure 2.3)*

4. Connect the USB output of the Atmel-ICE to your computer *(The other USB cable is used to connect the UART interface of the MMC for debugging)*

5. Switch the power supply on *(Figure 2.4)*

Figure 2.2: LATOMETT

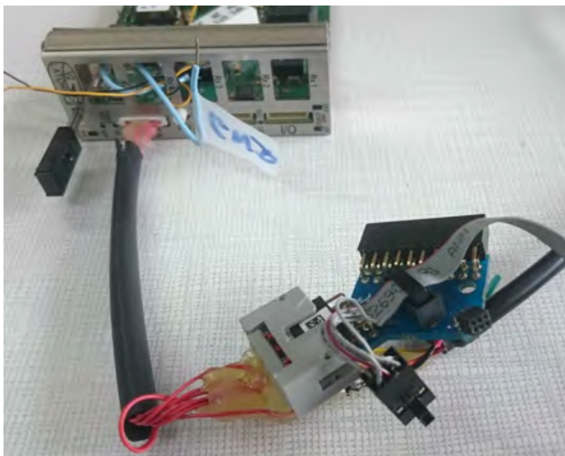


Figure 2.3: MMC JTAG front panel

Figure 2.4: Power supply ON

Figure 2.5: Atmel ICE JTAG programmer

At this point, the Atmel-ICE should show a green LED *(Figure 2.5)*, meaning that the MMC is correctly powered with the 3.3V derived from the 12V, and successfully detected by the JTAG programmer.

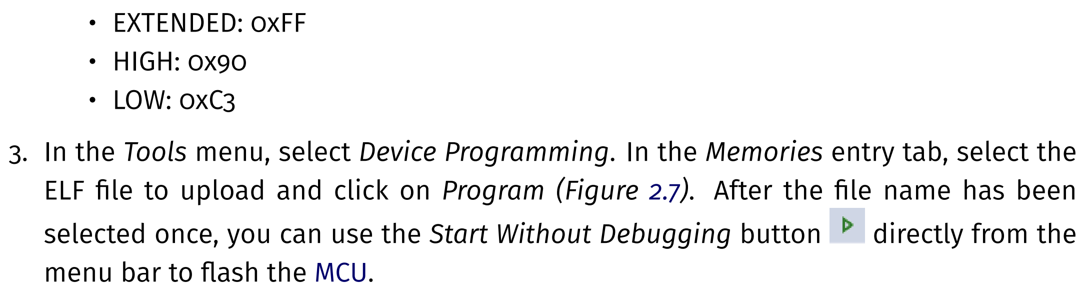
JTAG Programming:

The firmware can be flashed into the MMC using the Atmel IDE:

1. Open the project file **MMC/atmega128_mmc.atsln**
2. The fuses of the ATMega MCU of the MMC must be configured to get the correct clock configuration, specify the memory mapping and be able to use the UART output. In the *Tools* menu, select *Device Programming*, and click on *Apply* to connect to the AT-Mega128. Then select the *Fuses* item *(Figure 2.6)*.

   On the bottom of the window, the **Fuse Register** configuration should be set to:

   - EXTENDED: 0xFF
   - HIGH: 0x90
   - LOW: 0xC3

3. In the *Tools* menu, select *Device Programming*. In the *Memories* entry tab, select the ELF file to upload and click on *Program (Figure 2.7)*. After the file name has been selected once, you can use the *Start Without Debugging* button  directly from the menu bar to flash the MCU.

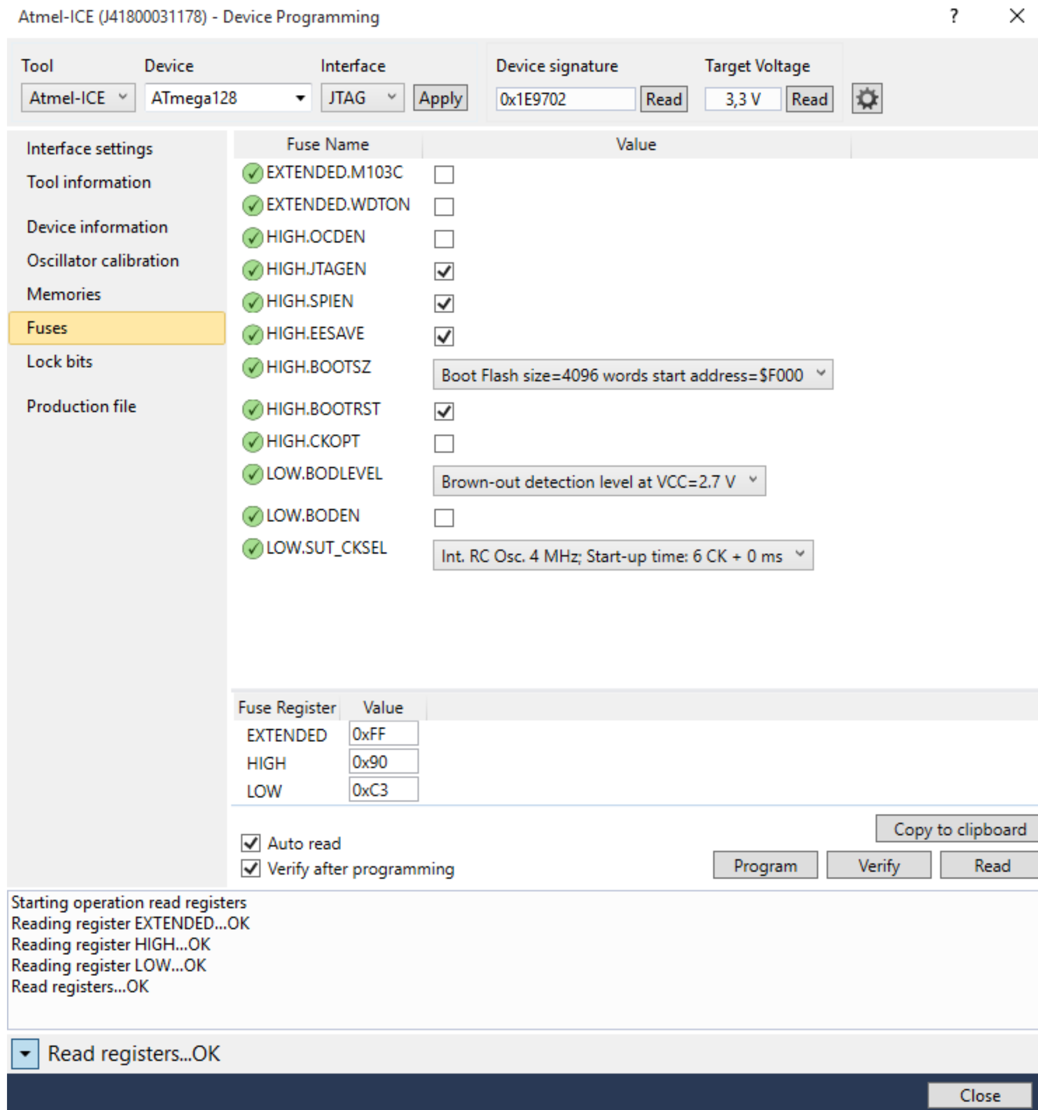


Figure 2.6: ATMega128 Fuses configuration

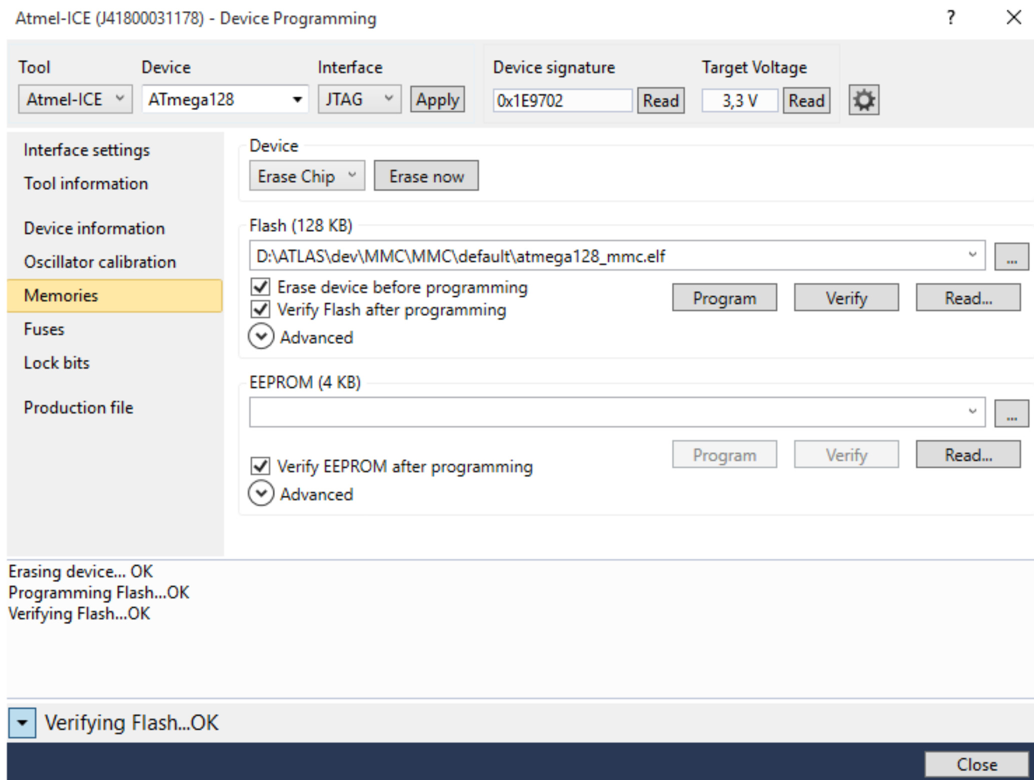


Figure 2.7: ATMega128 Programming

Firmware Update via HPM

MMC bootloader:

For the LATOME AMC to support the Hardware Platform Management (HPM) protocol, a compatible bootloader must be implemented in its MMC. The bootloader should check if an application can be run and start it, otherwise wait for HPM commands to receive and flash a new application binary into the MMC memory. Such a bootloader for the ATMega128 can be found in **HPM/atmega128_hpm.atsln**.

This Atmel IDE project has been configured to load the bootloader at memory address 0xF000 *(0x1E000 in the linker script option, as AVR MCU use 16-bits wide words instead of the 8-bits wide words used in the gcc port, thus addresses multiplied by 2)*. It must feat before memory address 0xFFFF *(Figure 2.8)*.

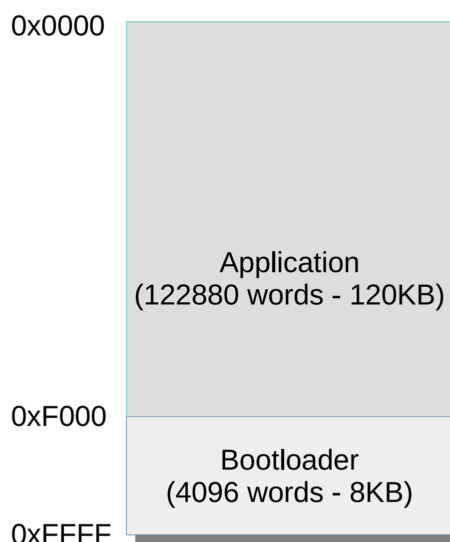


Figure 2.8: MMC memory Flash organization

The first byte of the **EEPROM** memory at address 0x00 is used to store the **bootloader flag**. If this flag equals 0xFF, the bootloader directly starts the application at boot. Otherwise, the GREEN and the RED LEDs are set on in order to indicate that the bootloader is running and waiting for HPM commands from the user.

In addition, the first byte of the **FRU** storage is used by the application to know if it is necessary to re-write the FRU information into the **EEPROM** memory. 0xFF means yes, other values means no.

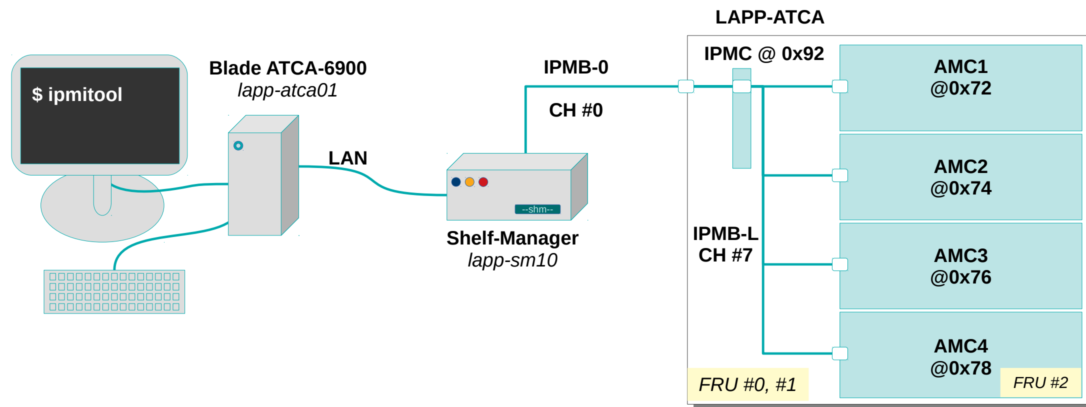


Figure 2.9: ipmitool command path

Warning: The first byte of the FRU is located at the absolute address 0x01 of the *EEPROM* memory, but is accessed via the *write_eeprom_byte()* and *read_eeprom_byte()* functions whose the address argument is systematically incremented by one. This means that *write_eeprom_byte(0x00, 0xFF)* writes 0xFF at address 0x01 of the EEPROM and so erases the first byte of the FRU storage.

RMCP commands via ipmitool:

The first version of the HPM specification *(HPM.1)* defines the firmware file format and IPMI commands protocol to update firmware in ATCA management controllers. This protocol is known as the **Remote Management Control Protocol (RMCP)**. It allows transport of **IPMI messages over LAN**.

RMCP is not directly supported by our IPMC, but by the Shelf-Manager itself. A software called **ipmitool** is available on Linux to send such RMCP messages.

In our case, **ipmitool** is available on the *(lapp-atca01)* embedded PC installed in the shelf. It allows to send IPMI commands to the Shelf-Manager, forwarding them to the ATCA board or to an AMC board as depicted in *Figure 2.9*.

To get management controller *(mc)* information, an IPMI request can be sent to the Shelf-Manager via the LAN using the following command:

```
$ ipmitool -H lapp-sm10 mc info
```

The same command can be sent to the ATCA board using the bridging option (*0x92* is the target address and *0* is the channel identifier for IPMB-0):

```
$ ipmitool -H lapp-sm10 -t 0x92 -b 0 mc info
```

Channels identifiers can be found using the *channel* command:

```
$ ipmitool -H lapp-sm10 channel info
```

To go further, commands can reach one of the AMC boards hosted by an ATCA board, using the double-bridging options. In *Figure 2.9*, the address of the AMC viewed by the IPMC *(ie. the ATCA board)* is *0x78* and the channel identifier is *7*:

```
$ ipmitool -H lapp-sm10 -T 0x92 -B 0 -t 0x78 -b 7 mc info
```

Firmware image generation:

The Atmel IDE generates an executable formatted in HEX. In order to be downloaded to an management controller via RMCP, it must be converted to binary format, and HPM commands must be inserted before the firmware image to indicate what the bootloader should precisely do.

The *HPMImgGenerator* software has been developed to perform this conversion, to create an HPM image called **img.hpm** from a given .hex file:

```
$ ./hpmimggenerator atmega128_mmc.hex

  MMC Information
*******************
Microcontroller version ([0] 8 bits / [1] 32 bits): 0
IANA Manufacturer ID: 0x00A12E
Product ID: 0x1235
Earliest major firmware rev. compatible : 1
Earliest minor firmware rev. compatible : 0
New major firmware : 4
New minor firmware : 1
```

Firmware upgrade procedure:

1. Get the target capabilities

```
$ ipmitool -H lapp-sm10 -T 0x92 -B 0 -t 0x78 -b 7 hpm targetcap

PICMG HPM.1 Upgrade Agent 1.0.9:

TARGET UPGRADE CAPABILITIES
-------------------------------
HPM.1 version............0
Component 0 presence....[y]
Component 1 presence....[n]
Component 2 presence....[n]
Component 3 presence....[n]
Component 4 presence....[n]
Component 5 presence....[n]
Component 6 presence....[n]
Component 7 presence....[n]
Upgrade undesirable.....[n]
```

```
Aut rollback override...[n]
IPMC degraded...........[y]
Defered activation......[y]
Service affected........[y]
Manual rollback.........[n]
Automatic rollback......[n]
Self test...............[n]
Upgrade timeout.........[30 sec]
Self test timeout.......[0 sec]
Rollback timeout........[0 sec]
Inaccessibility timeout.[60 sec]
```

2. Check the image compability

```
$ ipmitool -H lapp-sm10 -T 0x92 -B 0 -t 0x78 -b 7 hpm check img.hpm

PICMG HPM.1 Upgrade Agent 1.0.9:

Error getting component properties
compcode=0x83: Unknown (0x83)
Get CompRollbackVersion Failed for component Id 0

Validating firmware image integrity...OK
Performing preparation stage...OK

Comparing Target & Image File version
-------------------------------------------------------------------------
|ID  | Name         |                  Versions                        |
|    |              |     Active      |     Backup      |     File     |
-------------------------------------------------------------------------
|*^  0|LAPP MMC     |   1.00 00000000 | ---.-- -------- |  4.01 00000000 |
-------------------------------------------------------------------------
(*) Component requires Payload Cold Reset
(^) Indicates component would be upgraded
```

3. Upgrade the firmware and activate it

```
$ ipmitool -H lapp-sm10 -T 0x92 -B 0 -t 0x78 -b 7 hpm upgrade img.hpm activate

PICMG HPM.1 Upgrade Agent 1.0.9:

Error getting component properties
compcode=0x83: Unknown (0x83)
Get CompRollbackVersion Failed for component Id 0

Validating firmware image integrity...OK
Performing preparation stage...
Services may be affected during upgrade. Do you wish to continue? (y/n): y
OK

Performing upgrade stage:

------------------------------------------------------------------------------
|ID  | Name         |                  Versions                        | %   |
|    |              |     Active      |     Backup      |     File     |     |
|----|--------------|-----------------|-----------------|--------------|-----|
|*   0|LAPP MMC     |   1.00 00000000 | ---.-- -------- |  4.01 00000000 |100%|
|    |Upload Time: 01:11              | Image Size:   28288 bytes            |
------------------------------------------------------------------------------
(*) Component requires Payload Cold Reset
```

```
Performing activation stage:

Firmware upgrade procedure successful
```

4. Finally, reset the IPMC using the OEM command form the Shelf-Manager, in order to reload the updated FRU informations from the freshly flashed MMC.

Chapter 3

ICARE Developments

Contents

3.1 SPI Package **31**
3.1.1 SPI Multi-Master Feature 31
3.1.2 DMA Feature 34
3.2 CMC Package **37**
3.2.1 Testbench Feature [Deprecated] 37
3.3 LArC Commands Package **38**
3.3.1 Command Parsing Feature 38
3.3.2 Adding a New Command 39
3.4 UART Listener Package **42**
3.4.1 Description 42
3.4.2 Usage 42

3.1 SPI Package

3.1.1 SPI Multi-Master Feature

Objectives

The IPMC and IOIF MCUs communicate via a Serial Peripheral Interface (SPI) link. As message exchange should not be blocking, both MCUs are defined as Master on the bus.

Hence, when a MCU wants to send a message, it takes control over the bus and does not wait for an answer immediately. If no answer is received after a given timeout, the message is resent until a maximum number of retries has been reached. In this case, the receiver is

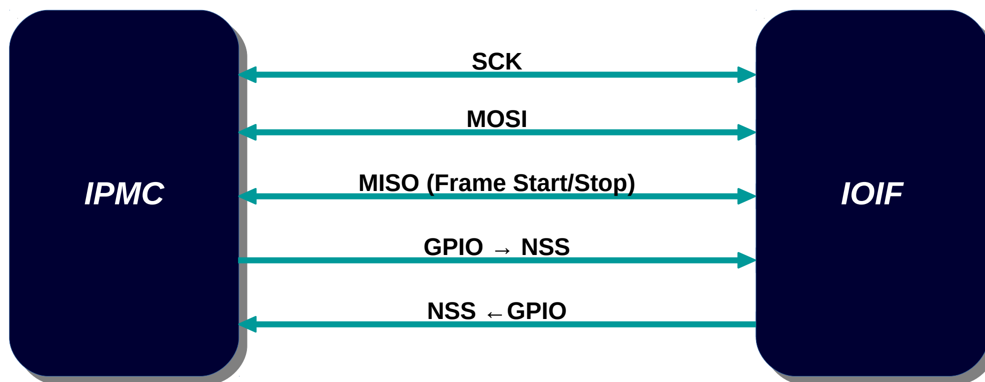


Figure 3.1: Hardware NSS feature - Multi-Master mode

then considered as disconnected.

The goal of this feature is to provide the IPMC and the IOIF MCUs with a way to automatically manage conflicting accesses on the SPI bus. To do so, relying on the Multi-Master topology, each MCU uses a GPIO pin configured as an output to control the hardware NSS pin of the other one, that is configured as an input pin *(See Figure 3.1)*:

- By default, both MCUs are in **Slave** mode.
- When one of the MCUs wants to communicate, it goes into **Master** mode to temporarily take control of the bus.
- If its **NSS** pin is **low**, it means that the other MCU is already trying to take control of the bus. In this case, an interrupt is generated, and the MCU goes into **MODe Fault *(MODF)* state** and return to **Slave** mode.
- Otherwise, when the communication is completed, the **Master** MCU releases the bus by setting the **Slave** MCU's **NSS** pin **high**.

In addition, as the communication is always uni-directional, we only use one wire to send and receive data *(MOSI)*. The *MISO* pin is thus used to notify the receiver that a new frame transmission has begun (*'frame start'* on rising edge, and *'frame stop'* on falling edge).

Software Architecture

The SPI library has been remodeled in order to support the Multi-Master mode. The use case representing the data transmission is depicted in *Figure 3.2*:

- The user can register two callbacks: one called when a requested transmission has been completed (**Register Tx Done Callback**), the other one called when a full data frame has been received (**Register Rx Done Callback**).

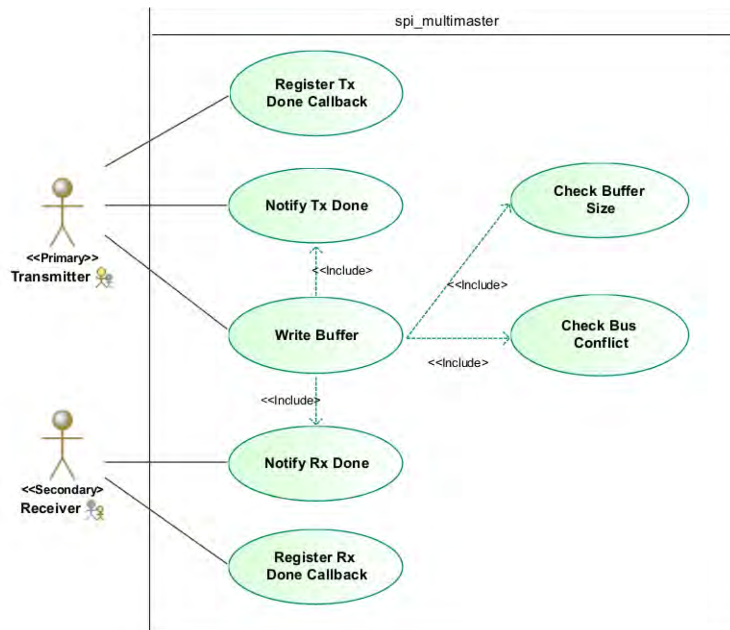


Figure 3.2: SPI Multi-Master mode - Data Transmission Use-Case

- The library manages two internal buffers: one used to transmit data, the other for reception. The TX buffer is copied from the given user buffer in the **Write Buffer** case. The RX buffer is directly used by the library to store incoming data packets. It is provided to the user in the **Rx Done Callback** case. The user should then copy the content of this buffer on its own, as it is released after exiting the callback.
- Before sending data, the buffer size and the MODe Fault status are checked (**Check Buffer Size** and **Check Bus Conflict**).

The class diagram representing the library API is depicted in the *Figure* 3.3, showing the user's public interface one hand, and the private implementation details on the other hand.

An error callback is also defined, to handle communication failures: it provides to the user with the status of the communication, to decipher whether a message should be retransmitted or not.

Tests Suite

A test is defined as a scenario involving a transmitter and a receiver. For each test function, the two cases are described and a feature or event is tested. Tests are executed sequentially. Because some tests can cause an MCU to be reseted, it is thus necessary to re-synchronize the MCUs sequences.

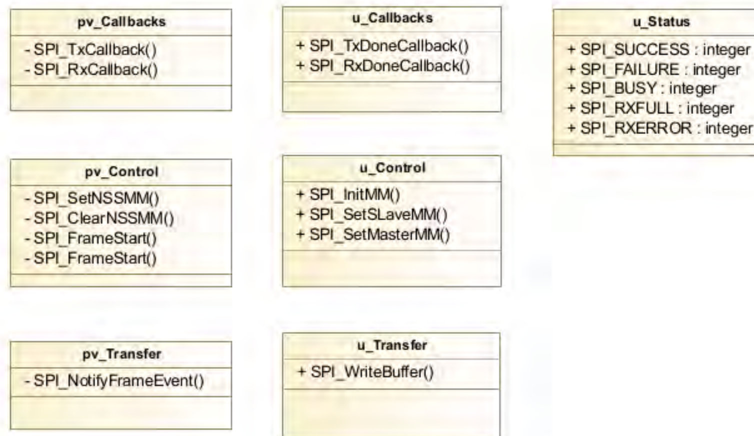


Figure 3.3: SPI Multi-Master mode - Class Diagram - Library API

The synchronization is done as follow. At the beginning of each test, the IPMC sends its test number and then waits for the IOIF. The IOIF waits the test number of the IPMC, then sends its own:

- If (own_nb > other_nb) $\longrightarrow$ Wait for another number
- If (own_nb < other_nb) $\longrightarrow$ Skip the test
- If (own_nb == other_nb) $\longrightarrow$ Run the test

Figure 3.4 depicts the state machines run by each MCU to synchronize the test sequence.

3.1.2 DMA Feature

Objectives

The SPI peripheral can be configured to use an internal Direct Memory Access (DMA) engine. DMA allows to directly transfer data from memory to peripheral registers and vice-versa, without involving the central processor unit. As shown in *Figure 3.5*, each DMA engine can access a given set of peripherals via the AHB bus matrix.

This feature enhances the transfer bandwidth, as the communication is no longuer dependant of the MCU usage.

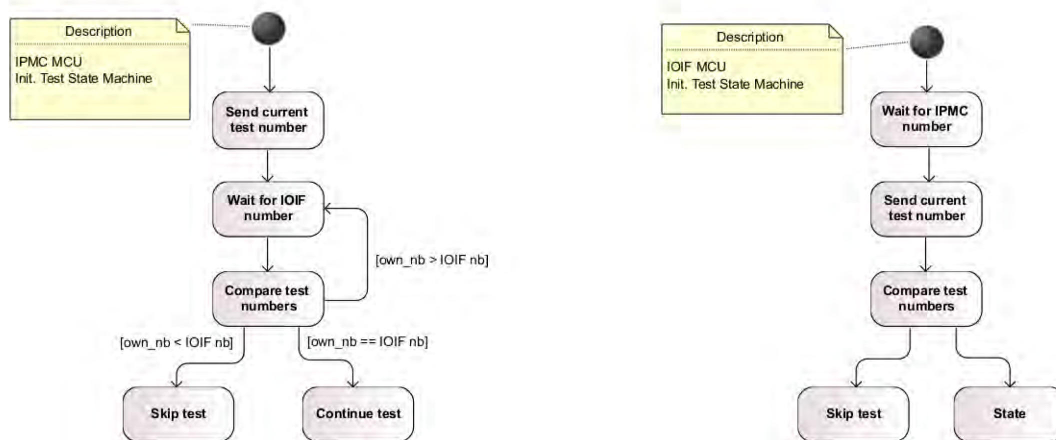
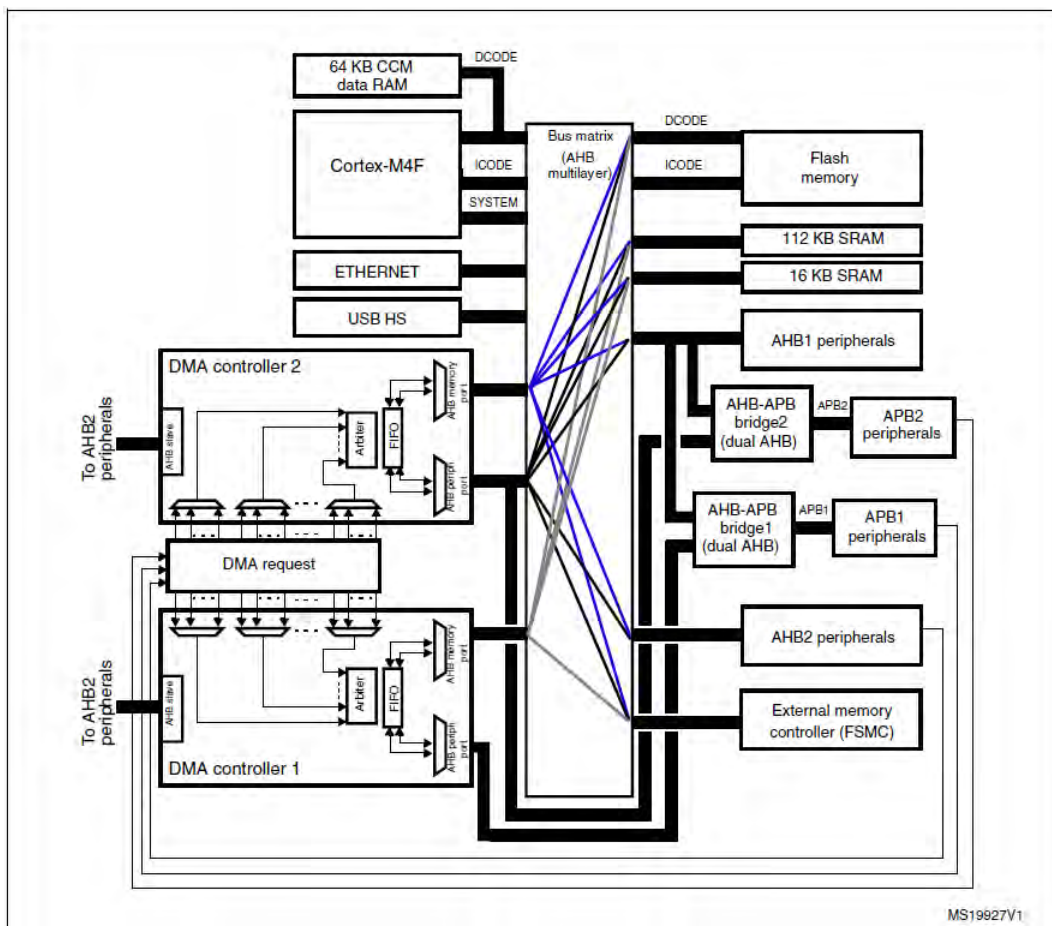


Figure 3.4: SPI Multi-Master mode - Test Synchronization FSM

Software Architecture

For the SPI peripheral, two DMA streams can be set to manage the transfer:

- The **stream 2** of **DMA2**, to silently move incoming data from the SPI RX register to a buffer located in memory. Once the transfer is completed, an interruption is raised.
- The **stream 3** of **DMA2**, to directly transfer data from memory to the SPI TX register. Once the transfer is completed, another interruption is raised.



1. The DMA1 controller AHB peripheral port is not connected to the bus matrix like in the case of the DMA2 controller, thus only DMA2 streams are able to perform memory-to-memory transfers.

Figure 3.5: STM32F4 DMA Engines [4]

3.2 CMC Package

3.2.1 Testbench Feature [Deprecated]

Note: The CMC testbench relies on the IPMB but has not been updated to comply with the latest version of this libray, thus is not usable in its current state.

The Carrier Manager Controller (CMC) package is a module responsible for ensuring the communication between the IPMC and the AMC boards. By extension, its role is to ensure the communication between these AMC boards and the Shelf-Manager. CMC manages the different state transitions of the AMC boards inserted in any one of the carrier slots.

Software Architecture

The communication between the IPMC and the Shelf-Manager is done via the global IPMB redundant buses: *IPMB-A* and *IPMB-B*.
The communication between the IPMC and AMCs is done through a local IPMB bus: *IPMB-L*.

When receiving messages from the global IPMB bus at destination of one of the AMCs, the message is encapsulated in order to later retrieve the original request and to transfer the answer back to the Shelf-Manager.

Objectives

The CMC testbench aims to test the communication between the Shelf-Manager and the AMC boards.

The IPMC MCU is running the CMC module and thus, emulate precisely the behavior of the Carrier Board running the ICARE firmware.

The IOIF MCU is responsible for allowing the reproduction of the user interactions with the Carrier and the AMCs, as well as simulating the Shelf-Manager behavior.

The user interactions include actions such as inserting the AMC or toggling the handle switch, whereas the Shelf-Manager actions includes among other things, activating the FRU or reading sensors values.

Hardware Architecture

AMC Control Lines as shown in *Figure 3.6*, are detailed in *Table 3.1*.

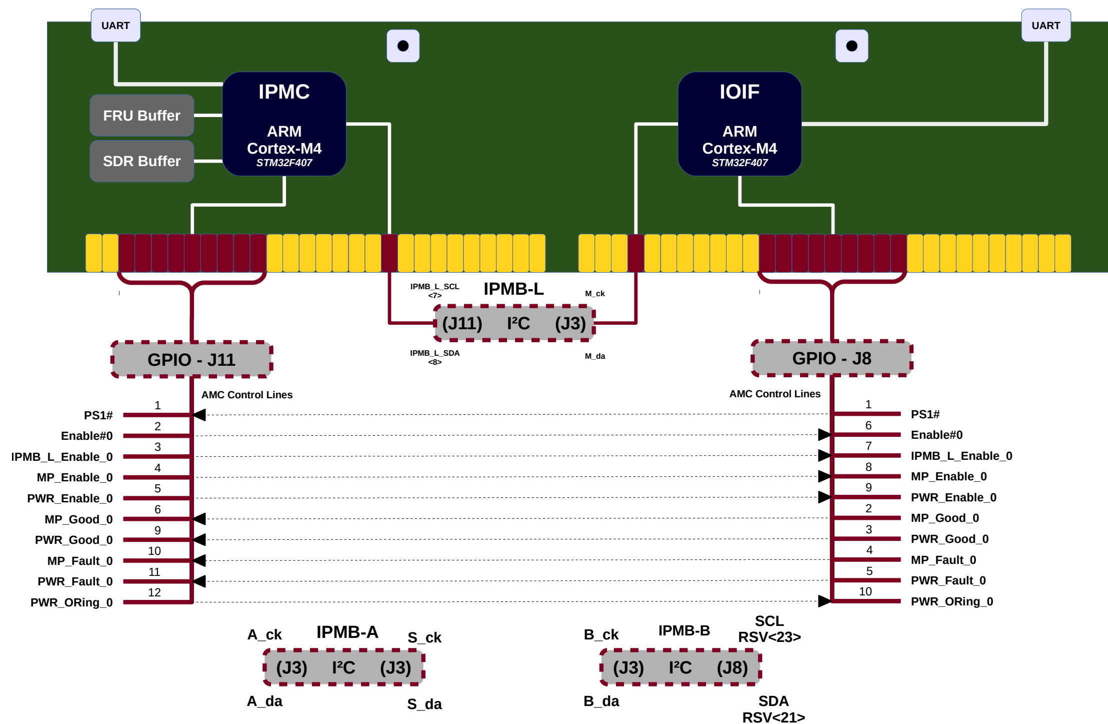


Figure 3.6: CMC Testbench Block Diagram

3.3 LArC Commands Package

3.3.1 Command Parsing Feature

This library defines a set of commands that can be executed by the IOIF. These commands allow the user to enable or disable some flags and to perform some actions.

Commands can be sent for instance, over the UART or the Ethernet connections. Those connections, and thus the reception of the commands, should be handled externally by a dedicated module.

The library exports a list of commands *(See Listing 3.1)* and three methods to manipulate the flags:

- LArCCmd_enable(cmd) which enables a command flag
- LArCCmd_disable(cmd) which disables a command flag
- LArCCmd_enabled(cmd) which returns the status of a command flag

Pin	I/O	Description
PS1#	In	Indicates that the AMC module is inserted in the carrier
Enable#0	Out	Indicates to the AMC module that it is fully inserted in the carrier
IPMB_L_Enable_0	Out	Connect AMC to the IPMB-L bus
MP_Enable_0	Out	Activate the Management Power (MP) - (3.3V to AMC)
PWR_Enable_0	Out	Activate the module Payload Power (PP) (+12V to AMC)
MP_Good_0	In	Asserted when the MP voltage is within the required levels
PWR_Good_0	In	Asserted when the PP voltage is within the required levels
MP_Fault_0	In	Asserted when the MP current reaches the limit
PWR_Fault_0	In	Asserted when the PP current reaches the limit
PWR_Oring_0	Out	Optional – for 12V redundancy

Table 3.1: AMC Control Lines

3.3.2 Adding a New Command

A new command can be added through the following steps:

1. In **larcCommands.h**, define a new command identifier:

Listing 3.1: "LArC Commands List"

```
/**
 * @brief LArC Commands list
 * @note FINAL_LARCCMD_NB_CMD SHOULD ALWAYS BE listed at the end of the enum and
 * other values SHOULD NOT BE explicitely defined expect the first one which
 * SHOULD BE equal to 0.
 */
typedef enum {
  LARCCMD_FPGA_ENABLE = 0,  /**> Enable/Disable FPGA read/write */
  LARCCMD_FPGA_HANDSHAKE,   /**> Enable/Disable handshake line */
  LARCCMD_FPGA_LOOPTEST,    /**> Enable/Disable loopback testing LVDS to IPMC */
  LARCCMD_DBG_SENSORS,      /**> Enable/Disable sensors printing */
  LARCCMD_FPGA_WRITE,       /**> Enable/Disable sending sensors to FPGA */
  LARCCMD_NEW_COMMAND,
  FINAL_LARCCMD_NB_CMD      /**> The number of elements contained in the enum */
} LArCCommands_e;
```

2. In **larcCommandslib.c**, define your flag if necessary:

```
/* Private variables ---------------------------------------------------------*/

/** @brief Default FPGA read/write enabled (type 'F' to disable) */
static bool FPGA_ENABLE = true;

...

/** @brief Default (true) disable sending sensors to FPGA (type 'w' to enable) */
static bool FPGA_Write = true;

/** @brief My new flag */
static bool New_Flag = false;
```

3. Then define the hookup functions for this flag:

```
/* Private functions ---------------------------------------------------------*/

/** @brief FPGA Enable enable hookup function */
static void FPGA_ENABLE_enable(void) { FPGA_ENABLE = true; }
/** @brief FPGA Enable disable hookup function */
static void FPGA_ENABLE_disable(void) { FPGA_ENABLE = false; }
/** @brief FPGA Enable enabled hookup function */
static bool FPGA_ENABLE_enabled(void) { return FPGA_ENABLE; }

...

/** @brief New Flag enable hookup function */
static void New_Flag_enable(void)  New_Flag = true;
/** @brief New Flag disable hookup function */
static void New_Flag_disable(void)  New_Flag = false;
/** @brief New Flag enabled hookup function */
static bool New_Flag_enabled(void)  return New_Flag;

/** @brief Commands hooks */
static LArCCmdHookFunctions_t LArCCmdArray[FINAL_LARCCMD_NB_CMD] = {
  // FPGA Enable command
  {.fnEnable = FPGA_ENABLE_enable, .fnDisable = FPGA_ENABLE_disable, .fnEnabled =
      FPGA_ENABLE_enabled},
  // FPGA Handshake command
  {.fnEnable = FPGA_handshake_enable, .fnDisable = FPGA_handshake_disable, .fnEnabled =
      FPGA_handshake_enabled},
  // FPGA Loop Test command
  {.fnEnable = FPGA_looptest_enable, .fnDisable = FPGA_looptest_disable, .fnEnabled =
      FPGA_looptest_enabled},
  // Debug Sensors command
  {.fnEnable = DBG_Sensors_enable, .fnDisable = DBG_Sensors_disable, .fnEnabled =
      DBG_Sensors_enabled},
  // FPGA Write command
  {.fnEnable = FPGA_Write_enable, .fnDisable = FPGA_Write_disable, .fnEnabled =
      FPGA_Write_enabled},
  // New Flag command
  .fnEnable = New_Flag_enable, .fnDisable = New_Flag_disable, .fnEnabled = New_Flag_enabled,
};
```

4. Finally, add the flag processing into the parsing method:

```
bool LArCCmd_parse_console_input(int c) {
  extern void IMC_DumpStats(void);
  bool bRtn = true;

  switch(c) {
    case 'o':
      GPIOClear(GPIO_USER_IO_19); //select alt boot flash
      break;

      ...

    case 'n':
      LArCCmd_enable(LARCCMD_NEW_COMMAND); // Enable FPGA write
      break;
    case 'N':
      LArCCmd_disable(LARCCMD_NEW_COMMAND); // Disable FPGA Write
      break;
    default:
      bRtn = false;
```

```
        printf("Unknown command entered '%c'\n", c);
        break;
    }

    return bRtn;
}
```

3.4 UART Listener Package

3.4.1 Description

This library allows any module to get notified about characters coming from the UART port. Each time a character is received, the library executes the callback methods registered by the different modules interested by those data (*Figure 3.7*).

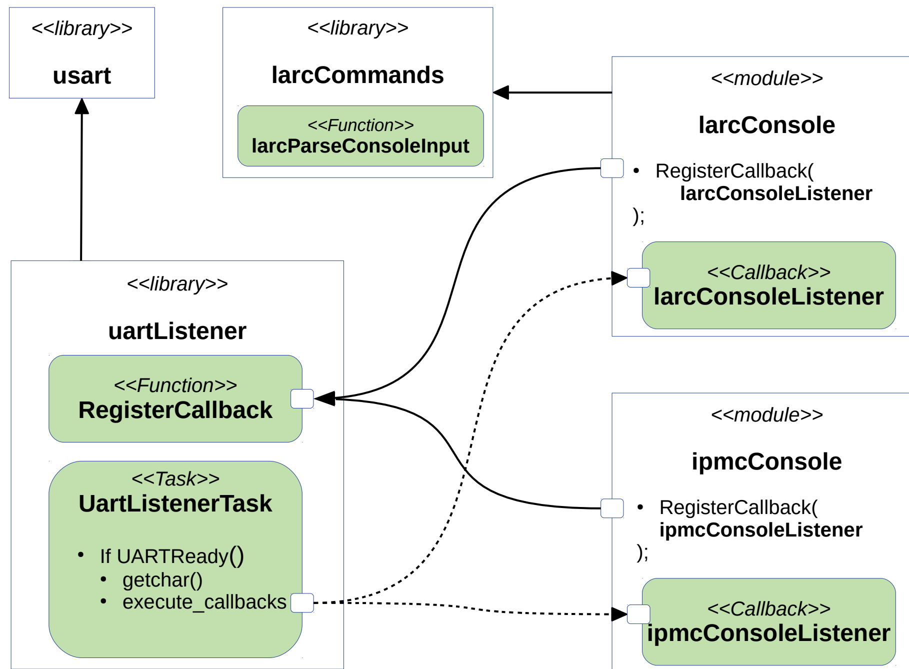


Figure 3.7: UART Listener diagram

3.4.2 Usage

If a module wants to get the content of the UART commands entered by the user, it can register a callback to the UartListener library. This can be done for instance in the module's initialization function (*Listing 3.2*).

Listing 3.2: "UART Listener callback registration"

```
/**
 * @brief Module initialization
 */
static bool larcUartConsole_Init(void) {
  // Initialize the UartListener library if not already done
  if (!UartListener_isInitialized()) {
    UartListener_Init();
```

```
  }

  // Register the listener for the UART console
  return UartListener_RegisterCallback(larcConsoleListener, NULL);
}
```

larcConsoleListener is a user-defined function conforming to the protoype defined in *Listring* 3.3.

Listing 3.3: "UART Listener registration callback prototype"

```
/**
 * @brief Callback function type
 *
 * @param character The character received on the UART
 * @param pvArg A private argument for the user
 */
typedef void(* UartListener_Callback_t)(int character, void * pvArg);
```

To stop receiving information from the UART Listener, just unregister the callback, for example in the module's cleaning function as shown in *Listing* 3.4.

Listing 3.4: "UART Listener callback unregistration"

```
/**
 * @brief Module cleanup
 */
static bool larcUartConsole_Clean(void) {
  // Unregister the listener for the UART console
  return UartListener_UnregisterCallback(larcConsoleListener);

  // Cleanup the UartListener library if it is initialized and that there is no more callback
       registered
  if (UartListener_isInitialized() && UartListener_isListEmpty()) {
    UartListener_Clean();
  }
}
```

Chapter 4

MMC Developments

Contents

4.1 User Sensors . . . 44
4.1.1 Sensors list . . . 44
4.1.2 Alerts and Sensors Thresholds . . . 45
4.1.3 LATOME Sensors Thresholds . . . 47
4.1.4 Sensors Data Conversion . . . 52
4.1.5 SDR Values: Experiments on the Shelf . . . 54
4.2 Task Management . . . 55
4.2.1 Introduction . . . 55
4.2.2 Task Control Block . . . 57
4.2.3 Scheduler . . . 57

4.1 User Sensors

Our MMC user-code contains the FRU/SDR information for the LATOME board. It also maps the different sensors monitoring the board.

4.1.1 Sensors list

Sensors.h contains the list of sensors (and sub-sensors) of the LATOME board, with their names, I^2C addresses (and sub-addresses), conversion values and specific thresholds.

The *Sensor* folder contains the files for each different sensors (initialization, reading part). Those files have been made with the tool given by the MMC software, and implemented

in the project. There are 2 different hardware sensors, managed by 3 different software components:

- LM95234: Temperature sensor
- LTC2495: Current sensor
- LTC2495: Voltage sensor

LTC2495 is used either as a current or a voltage sensor. It requires an I^2C request to initialize the conversion (160ms with normal configuration). So a specific configuration has been made to handle this issue using a structure with buffered values in LTC2495_common.h for both current and voltage use-case. This allows sending a frame to initialize the conversion, to after read the same channel. This occurs when trying to update the value, and it updates them one after another. In all other cases buffered values are returned.

4.1.2 Alerts and Sensors Thresholds

Sensor's values are monitored regarding their upper and lower thresholds. The ATCA specification defines three thresholds for both upper and lower values. Those thresholds are used in an hysteresis way, that defines the assertion and de-assertion of the different alerts as illustrated in *Figure 4.1*.

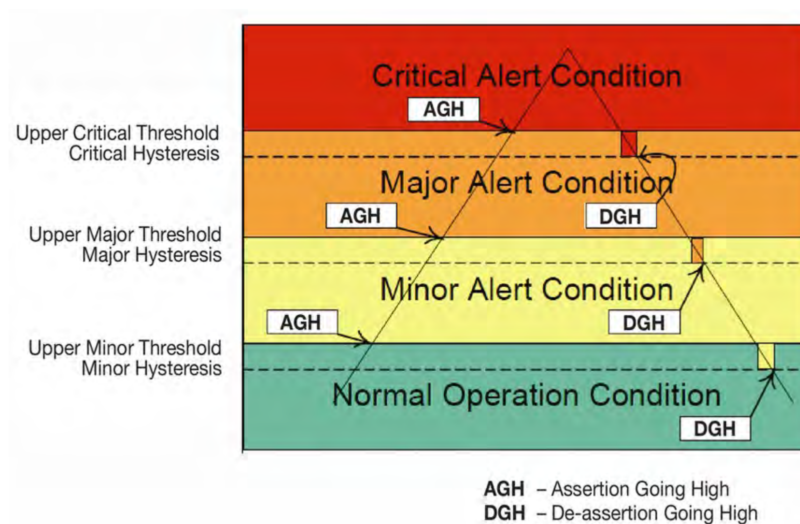


Figure 4.1: Alert Hysteresis System [5]

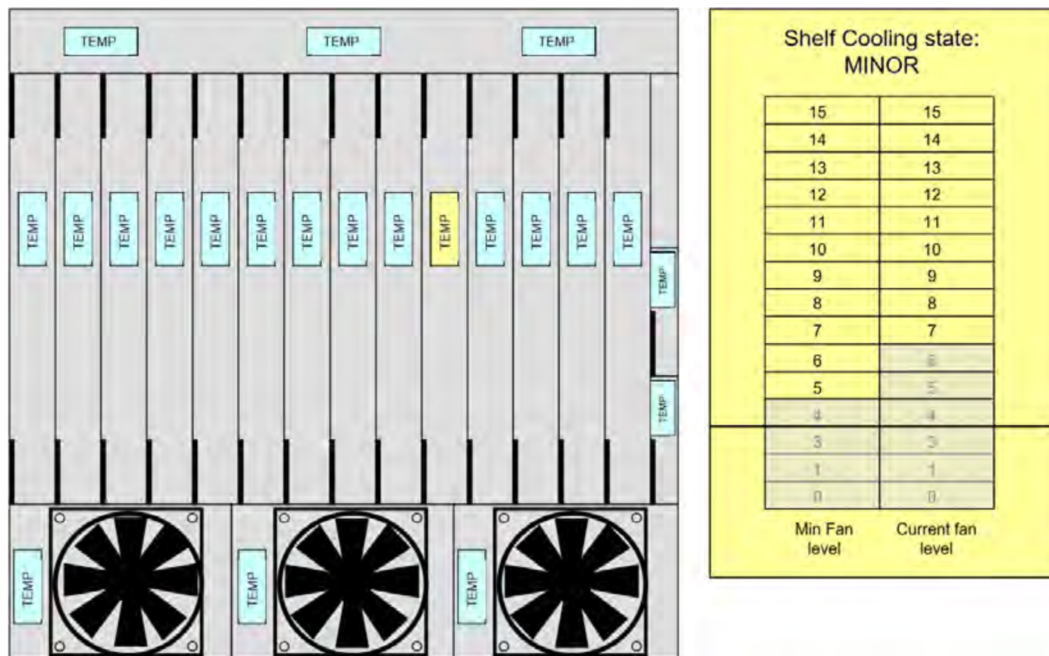


Figure 4.2: Shelf Manager Minor Alert [6]

-> IPMI **non-critical** or **Minor alert**: a warning meaning things are somewhat out of normal range, but not really a "problem" yet:

- upper-non-critical *(Upper-NC)*
- lower-non-critical *(Lower-NC)*

In this case, the Shelf Manager only progressively increases the fan level regarding the temperature value *(Figure 4.2)*.

-> IPMI **critical** or **Major alert**: things are still in valid operating range, but are getting close to the edge; unit still operating within vendor-specified tolerances:

- upper-critical *(Upper-C)*
- lower-critical *(Lower-C)*

In this case, the Shelf Manager sets the fan level at its maximum *(15)*, then decreases it progressively regarding the temperature value *(Figure 4.3)*.

-> IPMI **non-recoverable** or **Critical alert**: unit no longer operating within vendor-specified tolerances:

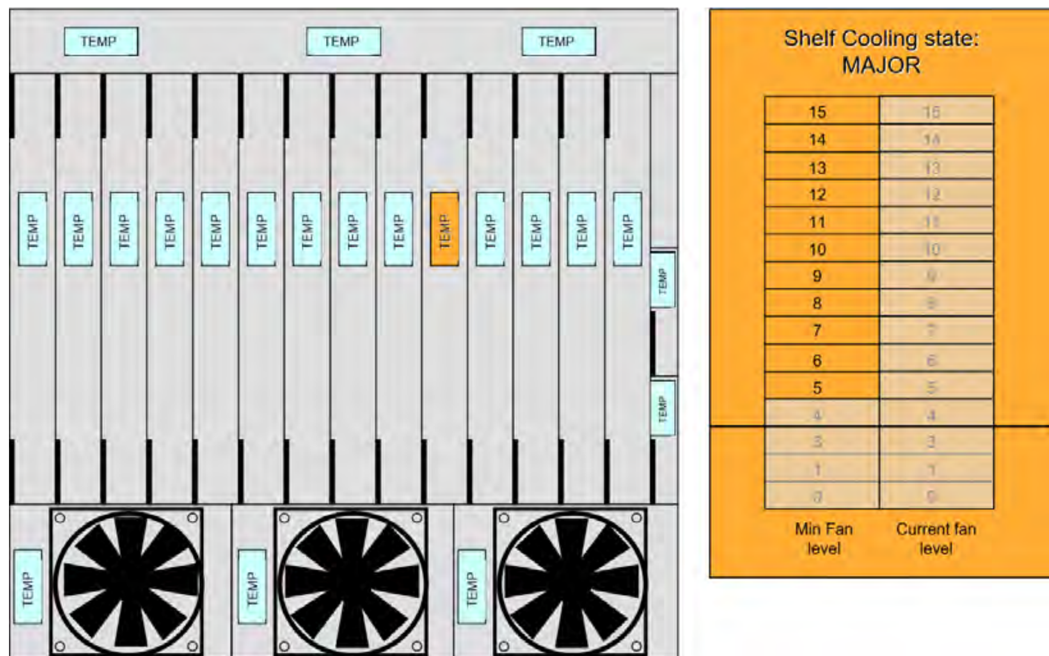


Figure 4.3: Shelf Manager Major Alert [6]

- upper-non-recoverable *(Upper-NR)*
- lower-non-recoverable *(Lower-NR)*

In this case, the Shelf Manager sets the fan level at its maximum level, ie. 15 (*Figure* 4.4), and can power-down the board if it has been configured to do it *(PEF global action)*.

When an alert occurs, the shelf manager can also negotiate new power-load values if this case is handled by the board. However, in all cases the upper level (for instance a human supervisor) is notified, and it is up to the supervisor to clear the alert, take the right decisions and operate in the most convenient way.

4.1.3 LATOME Sensors Thresholds

The list of the sensors available on the LATOME board is shown in *Table* 4.1. They are sorted into three categories:

1. High-Priority *(HP)* sensors: all used in the production version, and always enabled.
2. Low-Priority *(LP)* sensors: can also be used in the production version - but to limit memory usage, they are only enabled if required in the configuration at compile-time.

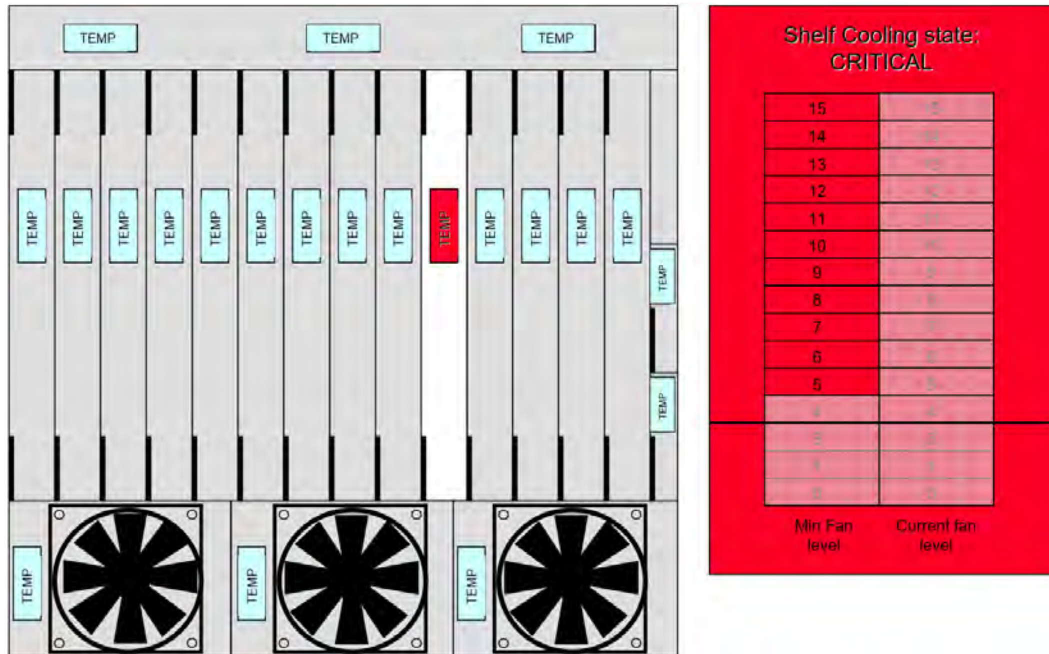


Figure 4.4: Shelf Manager Critical Alert [6]

3. Test *(TEST)* sensors: added on demand at compile-time when memory usage and IPMB congestion is not a constraint (ie. when debugging few LATOME boards in the shelf).

Those sensors and their respective thresholds are described in a header file in the MMC firmware repository *(./MMC/user/sdr/)*. There is one header file per LATOME version *(V1 or V2)* and per production version *(HP, LP or TEST)*. Those files can be generated using the provided **IPMI_ThresholdsGenerator** Python tool.

This Python script lies in *Tools/Sensor_Configs/IPMI_ThresholdsGenerator* in the Git repository. Each sensor's name and threshold values are listed in *IPMI_SensorList.py (See Figure 4.5)*. The main program is located in *src/IPMI_ThresholdsGenerator.py*. Generated files can be found in the *'generated'* directory.

The temperature flags are detailed in *Table* 4.2.

LATOME V1 Thresholds

Table 4.3 depicts the use of the different voltage channels of the LTC2495 and their respective threshold values.

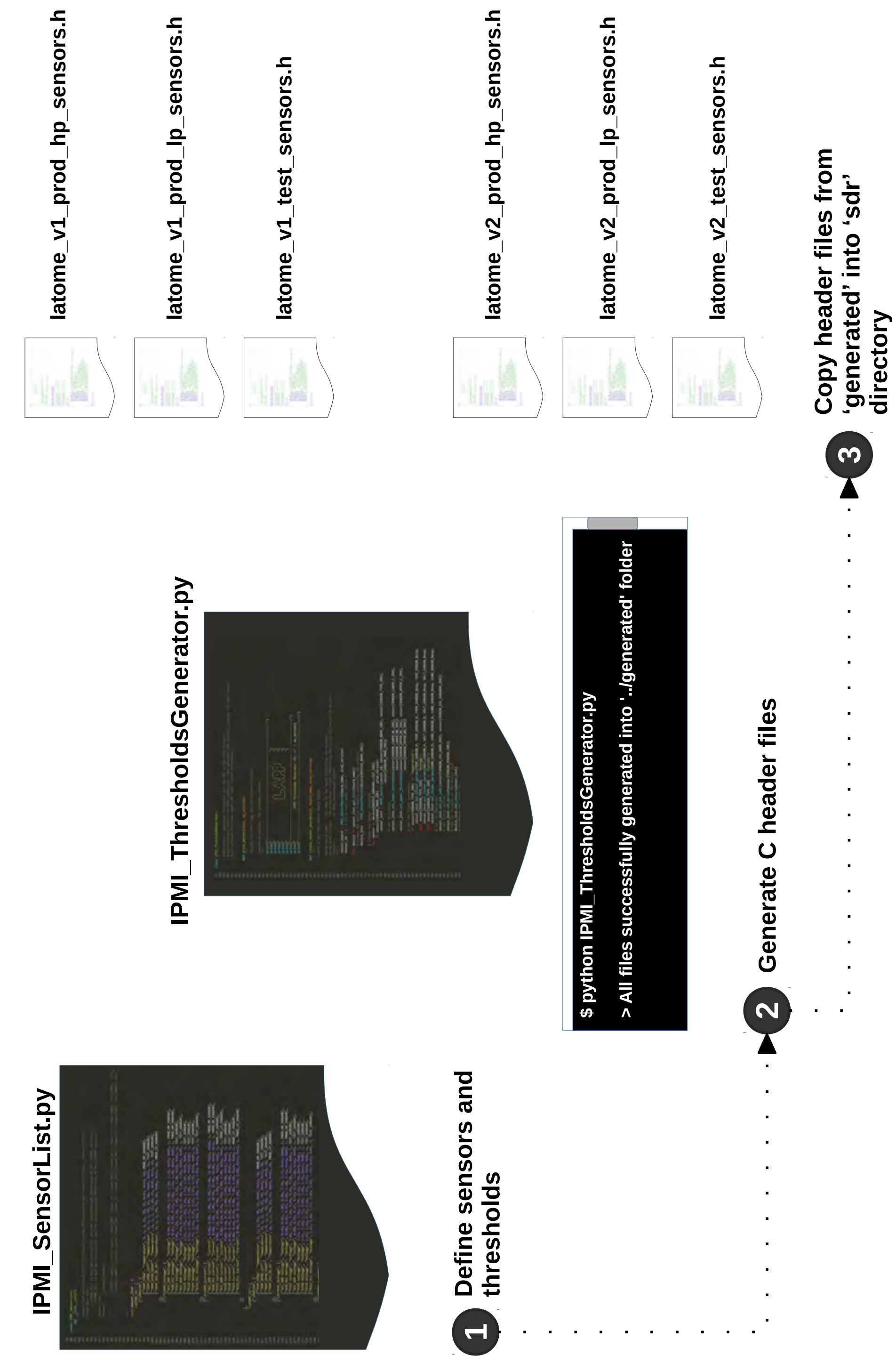


Figure 4.5: IPMI Thresholds Generator

Sensor	Priority	Description
Temperature (°C)		
TEMP_FPGA	HP	FPGA temperature
TEMP_UPOD_LEFT	TEST	Left μPOD temperature
TEMP_UPOD_RIGHT	TEST	Right μPOD temperature
TEMP_VCC_DC_DC	LP	DC/DC Converter temperature
Voltage (V)		
VCC_GXB Voltage	HP	VCC GXB voltage
A10_VCC Voltage	HP	Arria10 VCC voltage
P3V3 Voltage	TEST	3.3V
VCC_RAM Voltage	TEST	VCC RAM voltage
P1V5 Voltage	TEST	1.5V
P1V8 Voltage	TEST	1.8V
P2V5 Voltage	TEST	2.5V
VCCA_PLL Voltage	TEST	VCCA PLL voltage
Current (A)		
VCC_GXB Current	HP	VCC GXB current
A10_VCC Current	HP	Arria10 VCC current
P3V3 Current	LP	3.3V current
VCC_RAM Current	LP	Arria10 VCC RAM current
P1V5 Current	TEST	1.5V current
P1V8 Current	TEST	1.8V current
P2V5 Current	LP	2.5V current
VCCA_PLL Current	LP	VCCA PLL current

Table 4.1: LATOME Sensors List

Temperature Name	Register	Lower			Upper		
		NR	C	NC	NC	C	NR
TEMP_FPGA	Temp 1 MSB unsigned	0	1	2	75	80	85
TEMP_UPOD_LEFT	Temp 2 MSB unsigned	0	1	2	50	55	60
TEMP_UPOD_RIGHT	Temp 3 MSB unsigned	0	1	2	55	60	65
TEMP_VCC_DC_DC	Temp 4 MSB unsigned	0	1	2	50	55	60

Table 4.2: LATOME V1 Temperature thresholds (in °C)

The COM pin is connected to 1.8V and not $\frac{3.3}{2}$ *(the schematic is outdated on this point)*. *Table* 4.4 depicts the use of the different current channels (differential voltage) of the LTC2495 and their respective minimum and maximum values:

Voltage Name	Channel	Lower			Nominal	Upper		
		NR	C	NC		NC	C	NR
VCC_GXB Voltage	CH1	0.90	0.95	1.00	1.03	1.20	1.30	1.35
A10_VCC Voltage	CH3	0.85	0.90	0.92	0.95	1.10	1.15	1.20
P3V3 Voltage	CH5	3.05	3.15	3.20	3.30	3.40	3.45	3.50
VCC_RAM Voltage	CH7	0.85	0.90	0.92	0.95	0.98	1.00	1.05
P1V5 Voltage	CH9	1.32	1.37	1.42	1.50	1.58	1.63	1.68
P1V8 Voltage	CH11	1.61	1.66	1.71	1.80	1.89	1.94	1.99
P2V5 Voltage	CH13	2.27	2.32	2.37	2.50	2.62	2.67	2.72
VCCA_PLL Voltage	CH15	1.61	1.66	1.71	1.80	1.89	1.94	1.99

Table 4.3: LATOME V1 Voltage thresholds (in V)

Current Name	Channels	Lower			Upper		
		NR	C	NC	NC	C	NR
VCC_GXB Current	CH0 - CH1	0	0.5	1	12	15	18
A10_VCC Current	CH2 - CH3	0	1	3	20	25	28
P3V3 Current	CH4 - CH5	0	0.1	0.5	2	3	4
VCC_RAM Current	CH6 - CH7	0	0.1	0.5	2	3	4
P1V5 Current	CH8 - CH9	0	0.1	0.5	2	3	4
P1V8 Current	CH10 - CH11	0	0.1	0.5	2	3	4
P2V5 Current	CH12 - CH13	0	0.1	0.5	5	7	8
A10_VCCA_PLL Current	CH14 - CH15	0	0.1	0.5	5	7	8

Table 4.4: LATOME V1 Current thresholds (in A)

LATOME V2 Thresholds

Table 4.5 depicts the use of the different voltage channels of the LTC2495 and their respective threshold values.

The COM pin is connected to 1.8V and not $\frac{3.3}{2}$ *(the schematic is outdated on this point)*. *Table* 4.6 depicts the use of the different current channels (differential voltage) of the LTC2495 and their respective minimum and maximum values:

The temperature flags are detailed in *Table* 4.7.

Voltage Name	Channel	Lower			Nominal	Upper		
		NR	C	NC		NC	C	NR
VCC_GXB Voltage	CH1	0.90	0.95	1.00	1.03	1.20	1.30	1.35
A10_VCC Voltage	CH3	0.80	0.85	0.87	0.90	1.05	1.10	1.15
P3V3 Voltage	CH5	3.05	3.15	3.20	3.30	3.40	3.45	3.50
VCC_RAM Voltage	CH7	0.80	0.85	0.87	0.90	0.93	0.95	1.00
P1V5 Voltage	CH9	1.32	1.37	1.42	1.50	1.58	1.63	1.68
P1V8 Voltage	CH11	1.61	1.66	1.71	1.80	1.89	1.94	1.99
P2V5 Voltage	CH13	2.27	2.32	2.37	2.50	2.62	2.67	2.72
VCCA_PLL Voltage	CH15	1.61	1.66	1.71	1.80	1.89	1.94	1.99

Table 4.5: LATOME V2 Voltage thresholds (in V)

Current Name	Channels	Lower			Upper		
		NR	C	NC	NC	C	NR
VCC_GXB Current	CH0 - CH1	0	0.5	1	12	15	18
A10_VCC Current	CH2 - CH3	0	1	3	20	25	28
P3V3 Current	CH4 - CH5	0	0.1	0.5	2	3	4
VCC_RAM Current	CH6 - CH7	0	0.1	0.5	2	3	4
P1V5 Current	CH8 - CH9	0	0.1	0.5	2	3	4
P1V8 Current	CH10 - CH11	0	0.1	0.5	2	3	4
P2V5 Current	CH12 - CH13	0	0.1	0.5	5	7	8
VCCA_PLL Current	CH14 - CH15	0	0.1	0.5	5	7	8

Table 4.6: LATOME V2 Current thresholds (in A)

Temperature Name	Register	Lower			Upper		
		NR	C	NC	NC	C	NR
TEMP_FPGA	Temp 1 MSB unsigned	0	1	2	75	80	85
TEMP_UPOD_LEFT	Temp 2 MSB unsigned	0	1	2	50	55	60
TEMP_UPOD_RIGHT	Temp 3 MSB unsigned	0	1	2	55	60	65
TEMP_VCC_DC_DC	Temp 4 MSB unsigned	0	1	2	50	55	60

Table 4.7: LATOME V2 Temperature thresholds (in °C)

4.1.4 Sensors Data Conversion

Data related to the board's sensors are stored into the SDR. For example, a temperature sensor will have an entry in this database. In this entry, we will find information about the

sensor, ie. its name, the data format and also the value of the sensor. This value is encoded on 8-bits. As a sensor resolution is generally greater than 8-bits (ie. 11-bits, 12-bits or even 16-bits), a formula is defined in the specification to retrieve the real value of the sensor from the reduced 8-bits value:

$$y = (M * x + B * 10^{Bexp}) * 10^{Rexp}$$

Our goal is to select a range and the function parameters which would permit us to loose the minimum of information during the conversion from N-bits to 8-bits, and to come out with a consistent value that fits the application requirements (ie. the events that we want to monitore must be expressed within the 8-bits reduced sensor range).

1. Temperature sensor
   Device: LM95234

   In order to convert the 11-bits value of the temperature sensor to an 8-bits value, we need to define a reduced range and to specify the format and formula parameters allowing to retrieve the real value of the sensor. In our case we only keep the 8 MSB bits:

   - Range: 0 °C → 255 °C
   - Step: +1.0 °C
   - Format: unsigned 8-bits
   - From the 11-bits unsigned conversion result, we keep the bits [12:5]
   - Formula parameters:
     - M = 1
     - B = 0
     - Bexp = 0
     - Rexp = 0
2. Voltage sensor
   Device: LTC2495

   In order to convert the 16-bits value of the voltage sensor to an 8-bits value, we need to define a reduced range and to specify the format and formula parameters allowing to retrieve the real value of the sensor.
   In this case, as the conversion result is expressed in 2's complement, we have to keep the MSB bit representing the sign of the voltage value:

   - Range: 0,150 V → 3.450 V
   - Step: +0.013 V
   - Format: 2's complement 8-bits
   - From the 24-bits conversion result, we keep the bits [22:15]

- Formula parameters:
  - M = 13
  - B = 15
  - Bexp = 1
  - Rexp = -3

3. Current sensor
   Device: LTC2495

   In order to convert the 16-bits value of the current sensor to an 8-bits value, we need to define a reduced range and to specify the format and formula parameters allowing to retrieve the real value of the sensor.
   In this case, as the conversion result is expressed in 2's complement and **we supposed that the value should be positive**. We have to convert it into an unsigned value and to keep the significant bits that contains the sensor value:

   - Range: 0 A $\longrightarrow$ 51.0 A
   - Step: +0.200 A
   - Format: 2's complement 8-bits
   - From the 16-bits register values, we keep the bits [10:3] ([15:8] in the conversion result)
   - Formula parameters:
     - M = 200
     - B = 0
     - Bexp = 0
     - Rexp = -3

4. 12V Payload
   The 12V payload is measured by the internal ADC of the ATMega128 on the port PF0. The formula parameters to compute the payload value is the following:

   - $M = \frac{V_{ref}}{resolution} * \frac{(R_{top}+R_{bottom})}{R_{top}} = \frac{3{,}3}{256} * \frac{(49{,}9+150)}{49{,}9} = 52 * 10^{-3} (= 0x34)$
   - B = 0
   - Bexp = 0
   - Rexp = -3

   Vref is the reference voltage of the board. 256 is the ADC resolution (10-bits truncated to 8-bits – we keep only the MSBs). The PF0 port is connected to the 12V via a voltage divider in which the top resistor equals 150 ohm, and the bottom one 49.9 ohm.

4.1.5 SDR Values: Experiments on the Shelf

The previously defined SDR parameters have been tested on the LATOME board (V1) plugged in the ATCA shelf. The SDR values computed by the shelf are compared with the expected

Sensor	LATOME V1 Full Expected value	LATOME V1 Full SDR value	LATOME V2 Bare Expected value	LATOME V2 Bare SDR value
Temperature (°C)				
TEMP_FPGA	45.0	49.0	27.0	27.0
TEMP_UPOD_LEFT	40.0	40.0	27.0	26.0
TEMP_UPOD_RIGHT	40.0	43.0	27.0	27.0
TEMP_VCC_DC_DC	35.0	38.0	27.0	31.0
Voltage (V)				
VCC_GXB Voltage	1.030	1.060	1.030	1.047
A10_VCC Voltage	0.950	0.969	0.900	0.891
P3V3 Voltage	3.300	3.309	3.300	3.309
VCC_RAM Voltage	0.950	0.956	0.900	0.904
P1V5 Voltage	1.500	1.502	1.500	1.502
P1V8 Voltage	1.800	1.801	1.800	1.801
P2V5 Voltage	2.500	2.477	2.500	2.490
VCCA_PLL Voltage	1.800	1.788	1.800	1.801
Current (A)				
VCC_GXB Current	1.000	1.500	0.600	0.500
A10_VCC Current	5.000	6.100	0.000	0.100
P3V3 Current	0.000	0.100	0.000	0.100
VCC_RAM Current	0.000	0.100	0.000	0.100
P1V5 Current	0.000	0.100	0.000	0.100
P1V8 Current	0.000	0.100	0.000	0.100
P2V5 Current	1.500	1.500	0.300	0.300
VCCA_PLL Current	1.200	1.200	0.000	0.100

Table 4.8: Expected and measured SDR values

ones in *Table* 4.8.

4.2 Task Management

4.2.1 Introduction

Task management feature has been added to allow the user to define priorities between the different sensors and to fairly share the processor time between every sensor's read process and the other user's tasks, such as display (print sensors values or elapsed time) and monitoring (eg. payload power and DC/DC monitoring).

Before launching the tasks, the MMC goes through an initialization phase where it sets up all the I/O, interrupts and different structures needed to manage the FRU and SDR repositories.

It also initializes a simple scheduler which will execute the tasks, as well as the different sensors reading and their associated tasks, in addition to the timer used in the main loop of the application. The initialization process is detailed in *Figure* 4.6.

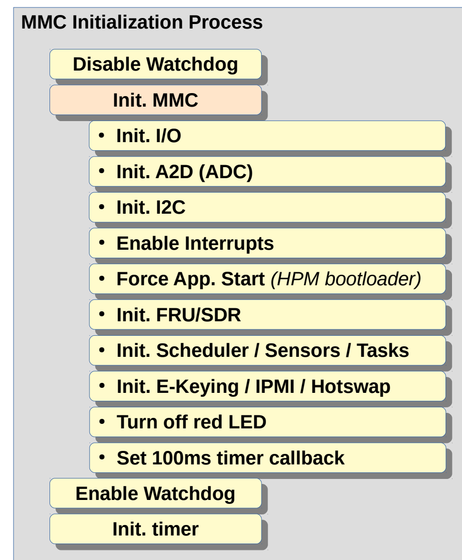


Figure 4.6: MMC initialization

In the main loop of the application, the watchdog is frequently reseted to avoid timeouts and this part of the loop is done as soon as possible *(Figure 4.7)*.

The other part is executed every 100 milliseconds *(ms)*. It starts by checking if IPMI responses has been received. If not and that the timeout for a given request is expired, the application can choose to send this request again. Then it checks the hotswap status, ie. the state of handle switch. Then the scheduler is activated to run the ready tasks. Finally, as the sensors values have been updated by their tasks, it checks if particular events should be sent. For instance, a major alert can be sent if a temperature sensor goes higher than the pre-defined threshold.

Different tasks have been defined:

- one per sensor, to read its last value *('LM95234 Task' to read a temperature sensor, 'LTC2495 Voltage Task' for a voltage sensor and 'LTC2495 Current Task' for a current sensor)*
- *Sensor Monitoring Task* to display the sensors values
- *Clock Task* to display the elapsed time since last reboot (uptime)
- *UART Listener Task* to get UART commands from user

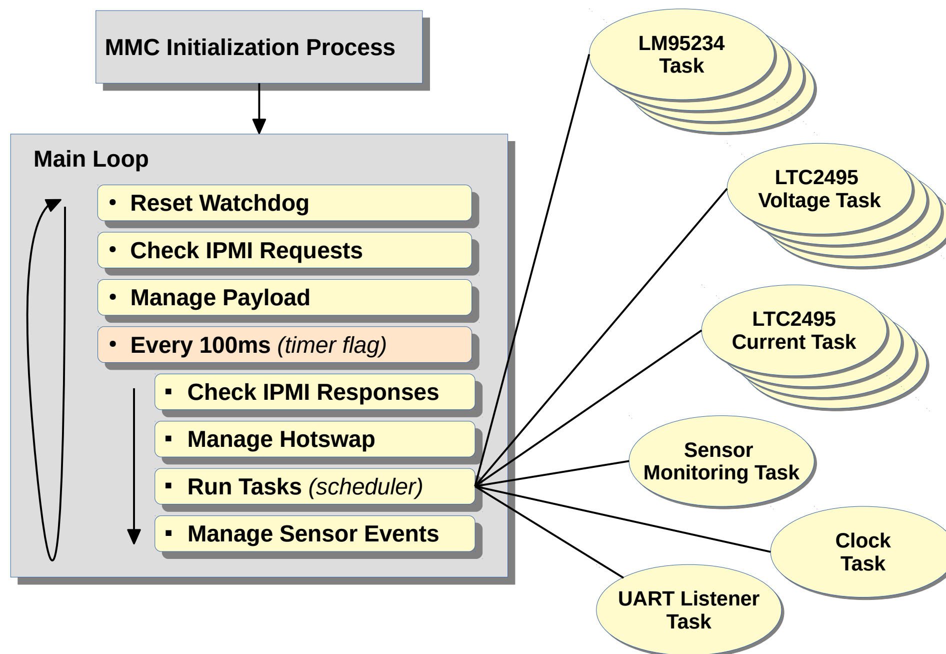


Figure 4.7: MMC main loop

4.2.2 Task Control Block

A task is defined by a Task Control Block (TCB), as detailed in *Figure 4.8*.

This structure contains among other things, a pointer to the function to be processed, the period of the task and user data to be transmitted as the context of the task.

4.2.3 Scheduler

A simple scheduler oversees the execution of all the tasks. It manages the TCBs, and permits to insert or remove tasks in the application.

Tasks are executed when they are ready, comparing their period with the current scheduler tick value. Ready tasks are executed in a sequential order. The scheduler simply iterates over the TCBs list, starting from handle number 0.

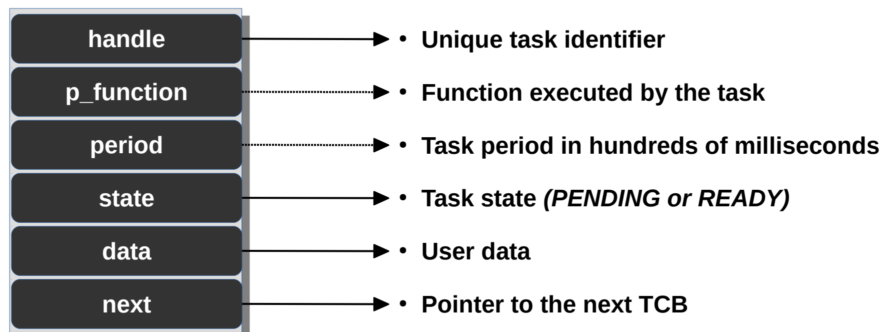


Figure 4.8: Task TCB

Appendix A

ICARE firmware compilation on CERN-SLC6

When compiling the ICARE framework *($make build)*, install 32-bit libraries on CERN-SLC6 to overcome the following issues:

- Error: /lib/ld-linux.so.2: bad ELF interpreter

```
$ yum install glibc.i686
```

- Error: libz.so.1: cannot open shared object file

```
$ yum install zlib.i686
```

  Two other libraries could be necessary to compile some package (for instance the SPI package):

```
$ yum install libX11.i686
$ yum install libtermcap.i686
```

Appendix B

Hard fault: Retrieve the faulty line

Retrieve the faulting line in case of hard fault, using the Link Register (LR) value:

```
$ arm-none-eabi-addr2line -e inetsrvc_IOIF.elf 0x0800a481
```

Appendix C

Locally install OpenOCD on CERN-SLC6

With the latest version of ICARE, OpenOCD is already available in the **contrib** directory. However, if you want to install it aside, you can follow these instructions:

```
$ wget https://downloads.sourceforge.net/project/openocd/openocd/0.9.0/
    openocd-0.9.0.tar.bz2
$ cd Downloads
$ tar xjvf openocd-0.9.0.tar.bz2
$ cd openocd-0.9.0
$ ./configure --enable-legacy-ft2232_libftdi
$ make
$ sudo make install
```

Then, in /usr/local/share/openocd/scripts/target/, copy the file ipmcv2_1.cfg.

Add a udev rules to allow any user to launch OpenOCD, for instance creating the file ´/etc/udev/rules.d/95-olimex.rules:..

```
SUBSYSTEM=="usb", ATTR{idProduct}=="002b", ATTR{idVendor}=="15ba", MODE="
    0666"
```

... replacing the product ID and the vendor ID with those of your device *(Use the command ĺsusb´to get these IDs*.

Finally, run the OpenOCD server:

```
$ openocd -f interface/olimex-arm-usb-ocd-h.cfg -f target/ipmcv2_1.cfg
```

When programming the IPMC *(make flash ...)*, if the following error occurs...

```
arm-none-eabi-gdb: error while loading shared libraries: libtermcap.so.2:
    cannot open shared object file: No such file or directory
```

... check that the libncurses library is installed and create a symbolic link to replace the libtermcap library which is deprecated:

```
$ ln -s /lib/libncurses.so.5.7 /lib/libtermcap.so.2
```

Appendix D

Status

D.1 ATCA Boards status

Currently, three ATCA boards are installed in the EMF setup:

Slot Number	Carrier ID	Carrier Version	IPMC ID	Prog. JTAG	Prog. ETH	IP address
4	3-5	3	110	Yes	Yes	128.141.202.203
9	2.1-3	2.1	133	No	Yes	128.141.202.204
12	3-3	3	108	Yes	Yes	128.141.202.208

Table D.1: ATCA board setup information

Appendix E

LArC Power Configuration

By default, the configuration file indicating the power requirements for the LAr Carrier (LArC) board is located in *LArC/share/data/LArC.m4*. In order to select another configuration file, the following commands should be issued. For instance, to load LArC_400w.m4:

```
$ source setup.sh —tag_add=400w
```

To check if the tags has been correctly loaded:

```
$ cmt show tags
...
400w (from CMTEXTRATAGS)
...
```

To change an existing configuration and load a new one, for instance LArC_200w.m4:

```
$ source cleanup.sh
$ source setup.sh —tag_add=200w
```

Bibliography

[1] ATLAS Experiment, 2016. 3

[2] LATOME reference manual, 2017. 9

[3] IPMC environment & configuration, 2016. 10

[4] RM0090 Reference Manual, 2016. 36

[5] PICMG 3.0 R3.0 AdvancedTCA Base Specification, 2008. 45

[6] Schroff GmbH. Shelf Management for optimized cooling in ATCA shelves, 2008. 46, 47, 48

Glossary

AMC Advanced Mezzanine Card. 6, 8, 9, 26–28, 37
ATCA Advanced Telecom Computer Architecture. 6–9, 19, 27, 28, 45, 54

CMC Carrier Manager Controller. 37
CTP Central Trigger Processor. 4

DMA Direct Memory Access. 34, 35

FEB Front-End Board. 4
FRU Field Replaceable Unit. 7, 26, 27, 56

HLT High-Level Trigger. 5
HPM Hardware Platform Management. 26–28

ICARE Intelligent platform management Controller softwARE. 10, 11, 13, 37
IOIF Input/Output InterFace. 8, 10, 31, 32, 37
IPMB Intelligent Platform Management Bus. 7–9, 37, 48
IPMC Intelligent Platform Management Controller. 6–11, 27, 28, 31, 32, 37
IPMI Intelligent Platform Management Interface. 7, 9, 27, 56

LAPP Laboratoire d'Annecy de Physique des Particules. 11, 13
LArC LAr Carrier. 64
LATOME LAr Trigger prOcessing MEzzanine. 9, 19–21, 26, 44, 47, 48, 50–52, 54
LR Link Register. 60
LUN Logical Unit Number. 7

MCU Micro-Controller Unit. 8–10, 12, 14, 23, 26, 31–34, 37
MMC Module Management Controller. 6, 9, 19–21, 23, 26, 30, 44, 48, 56

PU Processing Unit. 5

RMCP Remote Management Control Protocol. 27, 28
ROBIN Read-Out Buffer INput. 5
ROD Read-Out Driver. 4

ROL Read-Out Link. 5
ROS Read-Out System. 5
RTM Rear Transition Module. 8

SDR Sensor Data Records. 7, 44, 52, 54–56
SEL System Event Log. 6, 7
SMBus System Management Bus. 6
SPI Serial Peripheral Interface. 31, 32, 34, 35

TCB Task Control Block. 57, 58
TTC Trigger Timing Control. 4
TWI Two Wire Interface. 20