UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Câmpus de Ilha Solteira

THIAGO COSTA DE PAIVA

REMOTE DEVELOPMENT ENVIRONMENT WITH RECONFIGURABLE COMPONENTS IN THE ADVANCED TELECOM COMPUTING ARCHITECTURE CONTEXT

Ilha Solteira
2016

THIAGO COSTA DE PAIVA

REMOTE DEVELOPMENT ENVIRONMENT WITH RECONFIGURABLE COMPONENTS IN THE ADVANCED TELECOM COMPUTING ARCHITECTURE CONTEXT

Thesis presented to the Engineering School at Ilha Solteira, Sao Paulo State University, for partial compliance with the requirements in order to be awarded with the title of Master in Electrical Engineering.

Field of expertise: Automation.

Prof. Dr. Ailton Akira Shinoda
Supervisor

Ilha Solteira
2016

FICHA CATALOGRÁFICA
Desenvolvido pelo Serviço Técnico de Biblioteca e Documentação

P149r

Paiva, Thiago Costa de.
Remote development environment with reconfigurable components in the Advanced Telecom Computing Architecture context / Thiago Costa de Paiva. -- Ilha Solteira: [s.n.], 2016
124 f. : il.

Dissertação (mestrado) - Universidade Estadual Paulista. Faculdade de Engenharia de Ilha Solteira. Área de conhecimento: Automação, 2016

Orientador: Ailton Akira Shinoda
Inclui bibliografia

1. Remote development environment. 2. AdvancedTCA. 3. Reconfigurable components. 4. XVC. 5. IPMI.

UNIVERSIDADE ESTADUAL PAULISTA

Câmpus de Ilha Solteira

CERTIFICADO DE APROVAÇÃO

TÍTULO DA DISSERTAÇÃO: Remote Framework with Reconfigurable Components in ATCA Context" pois o texto está sendo redigido em inglês, sob minha supervisão

AUTOR: THIAGO COSTA DE PAIVA
ORIENTADOR: AILTON AKIRA SHINODA

Aprovado como parte das exigências para obtenção do Título de Mestre em ENGENHARIA ELÉTRICA, área: AUTOMAÇÃO pela Comissão Examinadora:

Prof. Dr. AILTON AKIRA SHINODA
Departamento de Engenharia Elétrica / Faculdade de Engenharia de Ilha Solteira

Prof. Dr. CLAUDIO KITANO
Departamento de Engenharia Elétrica / Faculdade de Engenharia de Ilha Solteira

Prof. Dr. CARLOS DIAS MACIEL
Departamento de Engenharia Elétrica e de Computação / Escola de Engenharia de São Carlos-USP

Ilha Solteira, 22 de setembro de 2016

Aos meus pais, Beatriz e Feliciano, e aos meus irmãos, Aline e Matheus, com quem sempre pude contar. À Jéssica, companheira de muitos anos e muitos sonhos.

ACKNOWLEDGMENTS

Firstly, I would like to thank Prof. Ailton Akira Shinoda for all the appropriate advice, encouragement and independence as supervisor of this work. I would like to thank also the thesis committee as a whole, including Prof. Claudio Kitano and Prof Carlos Dias Maciel, for all the insightful comments. I can not fail to mention Raphael Cobe, Rogerio Iope and Artur Baruchi for all the feedback during preliminary results and many other discussions that were not always directly related to the goals of this project but that I'm sure were important for this scientific work. I would like to thank Lucas Ramalho for his cooperation, dedication, and support during all the development we did together, this project would not be in the stage it is without his help. I would like to thank as well Prof. Mario Vaz who is not only a source of human and technical wisdom but also a friend. For the tireless reviews of the final text, while they could be doing much more pleasant things, I would like to thank: Liz Alexandrita, Lucas Pirola, André Cascadan, Vitor Finotti. I am grateful to the entire SPRACE/NCC and CMS/Fermilab groups, whose collaboration provided the necessary context and materials in this project, notably to Prof. Sergio Novaes, Tiehui Ted Liu and Jamieson Olsen. I must also mention colleagues of my current work at the Physics department in the University of Illinois, who do not have a direct relationship with this project but were pretty understanding to agree with my travel to Brazil for the finalization of the master's degree, particularly to Prof. Verena Marinez. I thank as well to the professors and students I met during the classes at UNESP Ilha Solteira for all the knowledge and friendship shared. Finally, I would like to extend my thanks to my family and friends who supported me during this so turbulent period. All this has been an incredible experience and a remarkable point in my life thus far, I can only hope for future challenges to be as exciting and engaging as this one.

"(...) que a importância de uma coisa não se mede com fita métrica nem com balança nem com barômetro etc. Que a importância de uma coisa há que ser medida pelo encantamento que a coisa produza em nós."

(Manoel de Barros)

RESUMO

A especificação Advanced Telecom Computing Architecture (AdvancedTCA) estabelece parâmetros para um bastidor eletrônico de altas perfomance e disponibilidade. Para tanto, grande enfoque é dado nas áreas de controle e monitoramento, o que é feito com o suporte da especificação Intelligent Platform Management Interface (IPMI). Uma das etapas deste trabalho deu-se justamente devido a implementação de um conjunto mínimo e funcional de recursos descritos pela IPMI de forma a adequar o projeto ao contexto em que estava inserido. Em contrapartida, este padrão acaba por restringir o acesso direto de elementos internos ao bastidor e, dessa forma, desenvolvimentos são dificultados quando componentes eletrônicos reprogramáveis são utilizados. Este projeto, então, teve como objetivo a composição de um ambiente de trabalho adequado a este contexto utilizando recursos tradicionais de redes de comunicações. O protocolo Xilinx Virtual Cable (XVC) mostrou-se extremamente conveniente para a situação e fez parte da solução implementada, possibilitando o uso remoto de técnicas de depuração. O ambiente de desenvolvimento final tem se mostrado estável e com boa performance, mesmo para conexões entre diferentes continentes, diminuindo custos com viagens e possibilitando a criação de estruturas compartilhadas.

Palavras-chave: Ambiente de desenvolvimento remoto. AdvancedTCA. Componentes reprogramáveis. XVC. IPMI.

ABSTRACT

The Advanced Telecom Computing Architecture (AdvancedTCA) specification defines parameters to provide a high performance and high availability environment. Therefore, there is a great focus on control and monitoring, which is achieved with the help of the Intelligent Platform Management Interface (IPMI) specification. One part of this project was devoted to the implementation of a minimal set of IPMI resources required by the context. On the other hand, the AdvacedTCA standard ends up restricting the access to devices inside of the shelf and thus developments on this scenario are hampered when reconfigurable electronic components are present. The goal of this project then was the conception of a suitable development environment using regular network resources. The Xilinx Virtual Cable (XVC) protocol came across as an appropriate tool to the context and is part of the implemented solution, allowing remote debugging techniques to be used. The final development environment is stable and with reasonable performance, even for connections between different continents, reducing travel expenses and facilitating the creation of shared structures.

Keywords: Remote development environment. AdvancedTCA. Reconfigurable components. XVC. IPMI.

LIST OF FIGURES

LIST OF TABLES

LIST OF FORMATS

ACRONYMS

ACK	Acknowledge
ADDR	address
AdvancedTCA	Advanced Telecom Computing Architecture
AMC	Advanced Mezzanine Card
ANSI	American National Standards Institute
ARM	Advanced RISC Machine
ASCII	American Standard Code for Information Interchange
ATLAS	A Toroidal LHC ApparatuS
BCD	Binary-Coded Decimal
BT	Block Transfer
C	Critical
CDF	Collider Detector at Fermilab
CERN	European Organization for Nuclear Research
CHK	Checksum
CMDID	Command Identification
CMS	Compact Muon Solenoid
DC	Direct Current
EDA	Electronic Design Automation
E-Keying	E-Keying Point-to-Point Connectivity
Fermilab	Fermi National Accelerator Laboratory
FIQ	Fast Interrupt Request
FMC	FPGA Mezzanine Module
FPGA	Field Programmable Gate Array
FRU	Field Replaceable Unit
FTK	Fast Tracker
FTP	File Transfer Protocol
GUID	Globally Unique Identifier
HDL	Hardware Description Language

HEP	High Energy Physics
HTTP	Hypertext Transfer Protocol
I^2C	Inter-Integrated Circuit
IC	Integrated Circuit
ICMB	Intelligent Chassis Management Bus
ID	Identifier
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
ILA	Integrated Logic Analyzer
IP	Internet Protocol
IPMB	Intelligent Platform Management Bus
IPMC	Intelligent Platform Management Controller
IPMI	Intelligent Platform Management Interface
IRQ	Interrupt Request
JTAG	Joint Test Action Group
KCS	Keyboard Controller Style
L1TT	Level 1 Tracking Trigger
LAN	Local Access Network
LAPP	Laboratoire d'Annecy-le-Vieux de Physique des Particules
LED	Light Emitting Diode
LHC	Large Hadron Collider
LUN	Logic Unit
MMC	Modular Management Controller
NC	Non-critical
NCC	Center for Scientific Computing or *Núcleo de Computação Científica* (in portuguese)
NETFN	Network Function
NR	Non-recoverable
OEM	Original Equipment Manufacturer
OS	Operating System
PCI	Peripheral Component Interconnect
PICMG	PCI Industrial Computer Manufacturers Group

RQ	requester
RS	responder
RTM	Rear Transition Module
RTOS	Real Time Operating System
RTX	Keil Real Time Operating System
SCL	Serial Clock
SDA	Serial Data
SDR	sensor data records
SEQNUM	Sequence Number
ShM	Shelf Manager
SMBUS	System Management Bus
SMIC	System Management Interface Chip
SOL	Serial on LAN
SPRACE	São Paulo Research and Analysis Center
TCK	Test Clock
TCP	Transmission Control Protocol
TDI	Test Data In
TDO	Test Data Out
TMS	Test Mode Select
TRST	Test Reset
UNESP	Sao Paulo State University or *Universidade Estadual Paulista "Júlio de Mesquita Filho"* (in portuguese)
USB	Universal Serial Bus
VITA	VMEbus International Trade Association
VME	Versa Module Europa
XADC	Xilinx Analog to Digital Converter
XVC	Xilinx Virtual Cable

IPMI COMMANDS

Compute Power Properties	NETFN 2Ch/2Dh CMDID 10h
Get FRU Inventory Area Info	NETFN 0Ah/0Bh CMDID 10h
Get Sensor Threshold	NETFN 04h/05h CMDID 27h
Get Device SDR Info	NETFN 04h/05h CMDID 20h
Get Device Locator Record ID	NETFN 2Ch/2Dh CMDID 0Dh
Get Sensor Reading	NETFN 04h/05h CMDID 2Dh
Get Power Level	NETFN 2Ch/2Dh CMDID 12h
Get Device SDR	NETFN 04h/05h CMDID 21h
Get PICMG Properties	NETFN 2Ch/2Dh CMDID 00h
Get Device ID	NETFN 06h/07h CMDID 01h
Platform Event	NETFN 04h/05h CMDID 01h
Read FRU Data	NETFN 0Ah/0Bh CMDID 11h
Reserve Device SDR Repository	NETFN 04h/05h CMDID 22h
Set Port State	NETFN 2Ch/2Dh CMDID 0Eh
Set Power Level	NETFN 2Ch/2Dh CMDID 11h
Set FRU Activation	NETFN 2Ch/2Dh CMDID 0Ch
Set Event Receiver	NETFN 04h/05h CMDID 00h

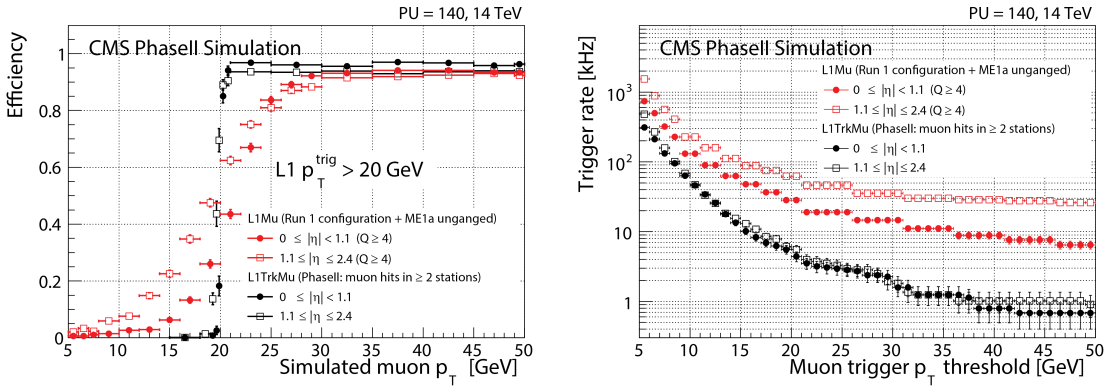
CONTENTS

1 INTRODUCTION

The Advanced Telecom Computing Architecture (AdvancedTCA) open standard was specified to provide an electronic infrastructure in the context of telecom, computing and data center systems, providing high availability and high performance conditions. It has been selected as the platform for many projects in the Large Hadron Collider (LHC) experiments at European Organization for Nuclear Research (CERN), the world's largest and most powerful particle collider, for trigger and data acquisition electronics (PEREK; MAKOWSKI, 2011; OKUMURA et al., 2013; LARSEN, 2010; 2012).

With the high luminosity upgrade of the LHC (ROSSI; BRÜNING, 2012), the Compact Muon Solenoid (CMS), one of the two general-purpose detectors, decided for a data acquisition trigger based on particle tracks detected using hardware and firmware, the Level 1 Tracking Trigger (L1TT). As illustrated in Figure 1, this approach will provide A highly efficient data filtering at high enough trigger rates, even for smaller particle momentum thresholds (CMS COLLABORATION et al., 2015).

Figure 1: Proposed L1TT simulated behavior. (a) the trigger filtering efficiency of muon detection as a function of the particle momentum, showing the momentum threshold for high efficiency. (b) the positive trigger output rate expected for different physics events as a function of the momentum threshold, to be compared with the expected events rate.



(a) (b)

Source: Adapted by the author from CMS collaboration et al. (2015).

This development is part of a collaboration between the São Paulo Research and Analysis Center (SPRACE) (SPRACE, 2015) instrumentation laboratory and the Center for Scientific Computing (NCC) of Sao Paulo State University (UNESP) with the CMS/L1TT

group at Fermi National Accelerator Laboratory (Fermilab). Fermilab is implementing an L1TT proposal based on the memory associative technique utilizing 3D custom integrated circuits, which in turn, uses custom AdvancedTCA boards from Pulsar IIb project. However, working with the reconfigurable components in that context were being hampered by the limited access to those resources. In this chapter, the context of the project is presented, including the AdvancedTCA standard, its relevant concepts and the Pulsar IIb design from Fermilab.

1.1 THE ADVANCEDTCA STANDARD

The PCI Industrial Computer Manufacturers Group (PICMG), consortium of industrial product vendors, aims to lead the standards development for the embedded computing market. They moved from the Peripheral Component Interconnect (PCI) to a more generic scope and what really remains is that they are a group with representatives of various sectors and companies trying to standardize the way that electronic projects can be integrated together.

One of their focus is high availability, a technology that uses a sophisticated and comprehensive management system to monitor the performance of whole structure, from hardware to software. High availability relies on a series of techniques such as reports, redundant resources and hot swap operation, enabling predictive maintenance, software upgrade in live systems and event logging. Thus, high levels of service availability (99.999% or greater) are attained through the integration of features for resiliency, redundancy, serviceability and manageability.

The AdvancedTCA standard is a set of PICMG specifications designed to provide an open and multi vendor architecture that incorporates high speed interconnect technologies, next generation of processors and improved reliability, manageability and serviceability. This family is denoted PICMG 3.x, where each value of x is a standard for a specific area, for example 3.0 is the AdvancedTCA Base Specification, 3.1 is related to Ethernet/Fiber Channel for AdvancedTCA and 3.7 is about AdvancedTCA Extensions.

These specifications include details like dimensions, equipment practice, connectors, power distribution and management architecture. There are plenty of information available for board, backplane, and chassis vendors to independently develop products that will be interoperable when integrated together, aimed to the reduction of costs and time of development.

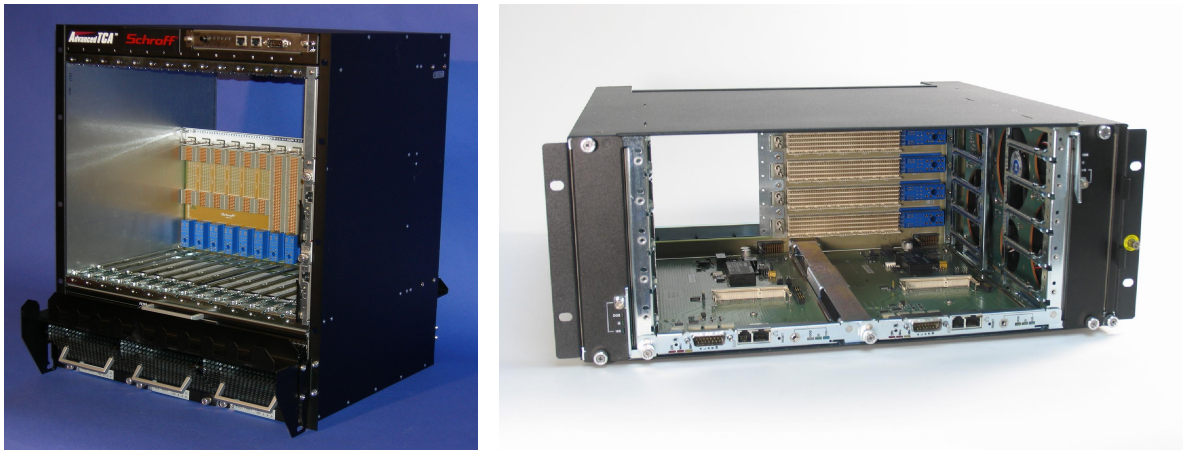
The key features of the AdvancedTCA standard include:

- high availability;
- hot-swappable and with redundant resources;
- comprehensive management and power distribution;
- high modularity and configurability;
- multiple switching fabric interfaces support;
- performance and size scalability for the system;

1.2 ADVANCEDTCA STANDARD CONCEPTS

The first item to be mentioned is the AdvancedTCA shelf, Figure 2, usually also known as crate, that holds multiple boards and providing them with many resources, such as mechanical structure, power supply, environment cooling, and an interconnection backplane (PICMG, 2008). A layer of management, the Shelf Manager (ShM), is there to turn the components on and off, negotiate power, take reading from sensors, make decisions, and trigger alarms and actuators (PICMG, 2008).

Figure 2: AdvancedTCA Shelves. (a) with vertical slots and bottom to top air flow. (b) with horizontal slots and lateral cooling system.



(a) (b)

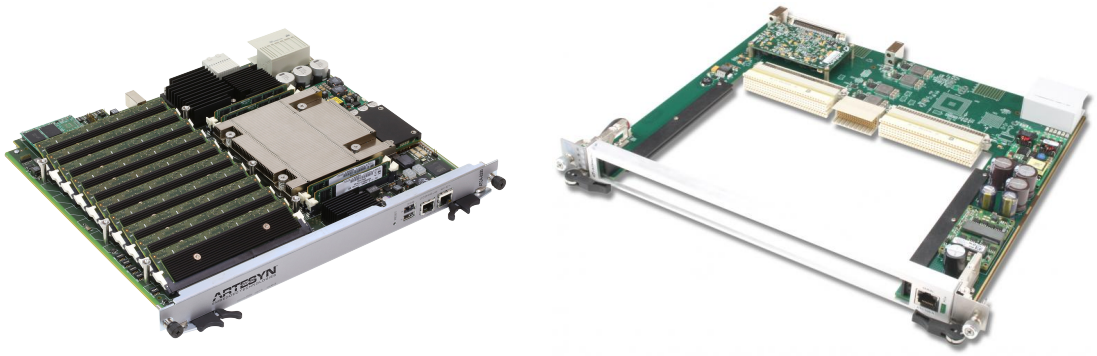
Source: Obtained from vendor website (PENTAIR, 2016).

Virtually every component in the AdvancedTCA shelf is a Field Replaceable Unit (FRU), which means that it may be replaced without powering down a device, namely the hot-swap mechanism. When a module is inserted into the shelf, just part of the power is

supplied and it stands temporarily with incomplete functions. Then, the ShM initiates the activation process involving power negotiation and information transferring. These steps make the device fully operational and, for the extraction procedure, the reverse process is performed. All these data exchanges rely on the Intelligent Platform Management Interface (IPMI) protocol.

One type of the boards mentioned above is a larger FRU board, as shown in the Figure 3, known as blade, carrier, or front boards, which is the main element where all the other will connect to. During the activation, these boards must report to the ShM its backplane interconnections before setting them up, allowing the ShM to detect any incoherence and to inform the board to do not activate those incompatible links, preventing damage.

Figure 3: AdvancedTCA blades, (a) is the ATCA-8330 model from Artesyn and (b) is the ATC106 from Vadatech.



(a) (b)

Source: Obtained from vendor websites (ARTESYN, 2016; VADATECH, 2016[b]).

There is another management layer embedded in the front boards and it is named Intelligent Platform Management Controller (IPMC). It has a main role in this project because the communication between it and the ShM is the focus of this implementation. The management channel is a Intelligent Platform Management Bus (IPMB) link, based on the Inter-Integrated Circuit (I^2C) protocol, that interconnects the ShM to the IPMCs inside one AdvancedTCA shelf (PICMG, 2008). This main IPMB channel is denominated IPMB-0. It is worth mentioning that the IPMI specification also defines links and control messages to platform management subsystems within the main board and, for that, there is a secondary management channel, the IPMB-L, connecting the IPMC to internal Modular Management Controllers (MMCs).

FPGA Mezzanine Modules (FMCs) may be attached to the carrier board for specific functions and it stands parallel to that. They are not hot-swappable because they do

not have the local controller, known as MMC. American National Standards Institute (ANSI) and VMEbus International Trade Association (VITA) specified the FMC standard that defines I/O mezzanine modules with connection to an Field Programmable Gate Array (FPGA) or other device with re-configurable I/O capability. It also specifies a low profile connector and compact board size for compatibility with several industry standard slot cards, blades, low profile motherboards, and mezzanine form factors. An FMC with an MMC is an Advanced Mezzanine Card (AMC), which is defined by its own standard (PICMG, 2006) also maintained by the PICMG consortium.

There is another type of board that can be inserted into the slot, and thus connected to the front board, using the back side of the shelf called Rear Transition Module (RTM) through a connector named as Zone-3, illustrated in Figure 4. Its main responsibility is to take the signals outside of the shelf and make possible to plug and unplug the main board without disturbing the cable arrangement (PICMG, 2008), relying also on the AMC standard for the managing aspect.

Figure 4: RTM for AdvancedTCA blades. (a) is the RTM436 from SANBlaze and (b) is the ART132 from Vadatech.

(a) (b)

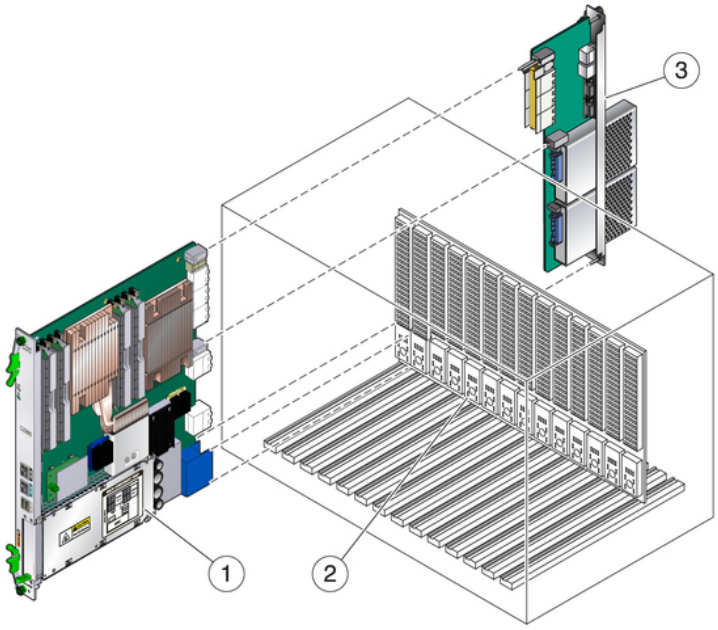
Source: Obtained from vendor websites (SANBLAZE, 2016; VADATECH, 2016[a]).

In the Figure 5, it is possible to see how each element is assembled according to the AdvancedTCA standard. It highlights also the positioning of the backplane inside the crate, providing connections between AdvancedTCA elements, for example blade to blade or blade to ShM links. Backplane offers control interfaces and link ports through Zone-1 and Zone-2 connectors, respectively. Clock signals, update ports and power distribution are also provided by the backplane.

1.3 THE PULSAR IIB PROJECT

The Pulsar project started in 2001, with the design and prototype of a general purpose Versa Module Europa (VME) interface for High Energy Physics (HEP) applications, to sup-

Figure 5: AdvancedTCA components assembling where (1) is the blade, (2) is the backplane and (3) is the RTM.



Source: CP3270 AdvancedTCA Blade Server User's Guide (ORACLE, 2010).

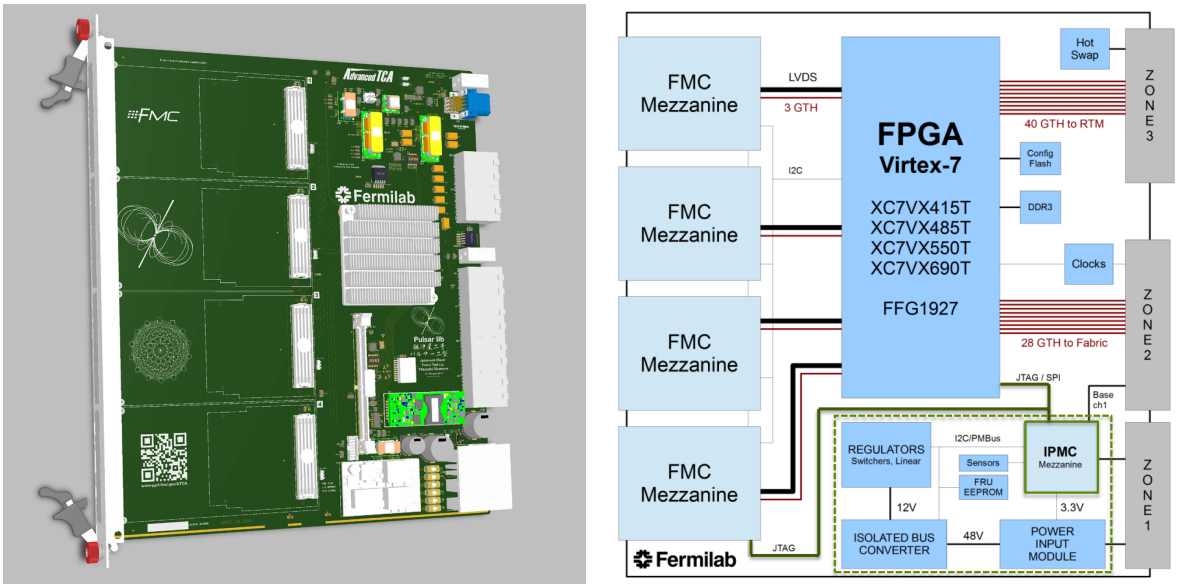
port the upgrade effort for the Collider Detector at Fermilab (CDF) at Tevatron (BHATTI et al., 2009). The design was generic enough to be used in many other applications, as for example a general purpose interface board, a standalone data and acquisition system or a software based trigger system. Pulsar design used a powerful motherboard to interface data with industrial standard link through the use of custom mezzanine cards.

When people from the initial project moved to the Fast Tracker (FTK) processor of the A Toroidal LHC ApparatuS (ATLAS) experiment, with much higher requirements, they decided to begin another project inspired by what made the Pulsar design successful: modularity, flexibility, scalability, high bandwidth communication, high performance and industrial standards. Based on design specifications and the need for future expansion capability, a full mesh AdvancedTCA backplane interconnection was found to be a natural fit (OLSEN; LIU; OKUMURA, 2013a; 2013b). They named this new project as Pulsar IIa. Then a second version of this design was made with even more powerful components, with a slight change in the name.

The Pulsar IIb is a general purpose FPGA-based processor board designed for full mesh AdvancedTCA backplanes. This hardware is intended to support the L1TT research and development of projects for LHC. According to Figure 6, Pulsar IIb is an AdvancedTCA front board with a large FPGA, capable of almost 1 Tbps I/O rates. It has up to 80

gigabit transceivers connecting the FPGA to RTM, FMC, backplane, and other elements in the same AdvancedTCA shelf.

Figure 6: Pulsar IIb blade designed by Fermilab, with four FMC slots, a Xilinx Virtex-7 FPGA and Zone connectors to the backplane and RTM. (a) shows a three dimensional computerized image while (b) offers a glance of its architecture and connections between components.



(a) (b)

Source: Adapted from Olsen, Liu and Okumura (2013).

1.4 PROBLEMATIZATION

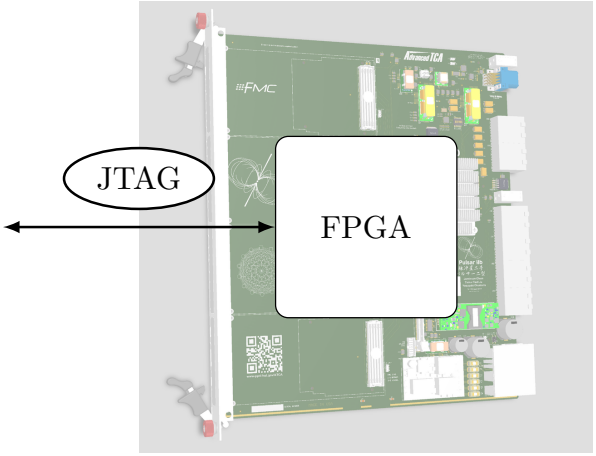
Previous explanation about the context of this design are not enough to make clear which points are intended to be worked on in this project. Thus, in this section, some of the specific problems raised when all of those technologies are put together will be discussed, in a way to underline the limitations and how they had been overcoming.

1.4.1 Development with reconfigurable devices

Main devices in some designs, such as processors and FPGAs, are normally Joint Test Action Group (JTAG) enabled, that is, they use a very specific four-wire protocol to provide access to their debug, emulation and programming functions. The Pulsar IIb project, presented in section 1.3, is not an exception about this and will rely on this kind of connection to interface its main component, as illustrated in Figure 7.

However, JTAG interfaces are not available by default in computers commonly used

Figure 7: JTAG interface to the main component of the Pulsar IIb project, which is an FPGA.



Source: Elaborated by the author.

by developers. The usual approach during the development cycle is to utilize any ready to use connection and then convert the data into JTAG information, as diagrammed in the Figure 8, allowing interactive download of configuration data to a system during prototyping, program data into the system during production and some other debug procedures.

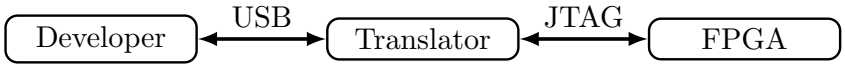
Figure 8: Diagram representing the connection between developer and FPGA during the development stage, from any standard interface to a JTAG connection.

Source: Elaborated by the author.

Nowadays, one of the most used schemes involves the conversion between Universal Serial Bus (USB) and JTAG connections, as shown in Figure 9, and is named as USB-JTAG link. This solution is assembled directly in a cable that interfaces a USB port on a host computer to a header connected to an FPGA mounted on a printed circuit board, which means that one cable is needed for each JTAG connection interfaced. Many options are available in the market, including the ones provided by the FPGA vendors.

Although the just mentioned arrangement is reasonable for majority of electronic projects in this context, that is not completely true for AdvancedTCA designs because of the devices density and also due to hard-to-access inner components. In Figure 10, USB/JTAG cables are seen coming off the crate for each Pulsar IIb blade. In this case,

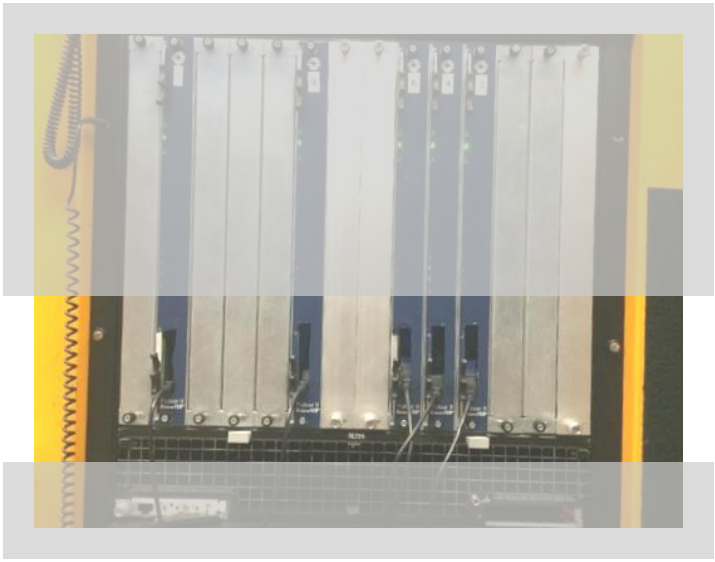
Figure 9: Diagram representing the connection between developer and FPGA during the development stage, from USB interface to a JTAG connection.



Source: Elaborated by the author.

the situation was slight worsened because no JTAG interface was exposed in the front panel, forcing the link to be made directly to connectors inside of the shelf.

Figure 10: USB-JTAG cables connected directly inside the crate, one for each Pulsar IIb blade.

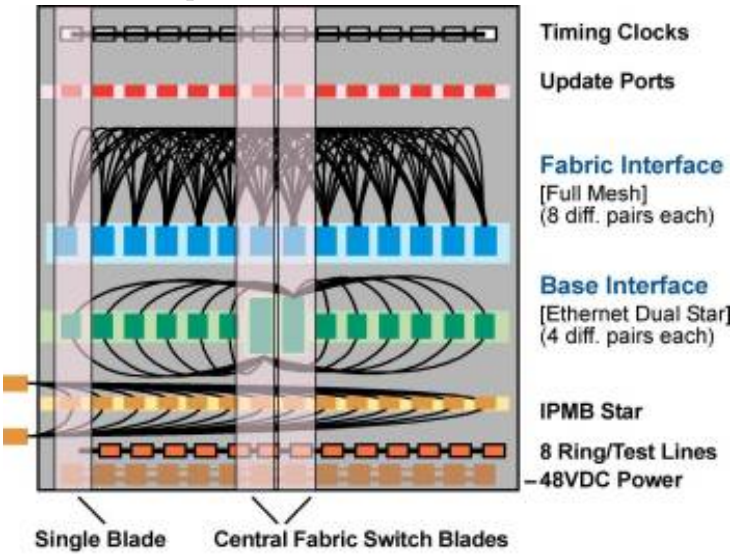


Source: Elaborated by the author.

AdvancedTCA standard provides many network resources to the blades through the backplane as shown in Figure 11, which are exposed by an AdvancedTCA network switch. Since the Pulsar IIb IPMC is connected to the Base Interface, as required by the specification, and is part of the JTAG chain, as designed by Fermilab, one possible alternative would be to establish a JTAG link using regular network protocols.

This architecture opens space for the Xilinx Virtual Cable (XVC) protocol, specified by Xilinx and are supported by their recent development tools. As the Pulsar IIb project relies only on FPGAs of the just mentioned vendor, the IPMC would provide a service defined by that protocol, acting as a translator between the regular network packages and the JTAG signals, as diagrammed in Figure 12. With this solution, not only the access to the reconfigurable devices inside the crate are guaranteed but also it allows new approaches for really remote connections.

Figure 11: Interconnection between AdvancedTCA blades through multiple backplane interfaces. AdvancedTCA switches are able to route packages between boards and also expose interfaces to external world.



Source: Obtained from Gilbert and Rasovsky (2004).

Figure 12: Diagram representing the connection between developer and FPGA during the development stage, using a XVC service to translate data between a regular network and a JTAG connections.

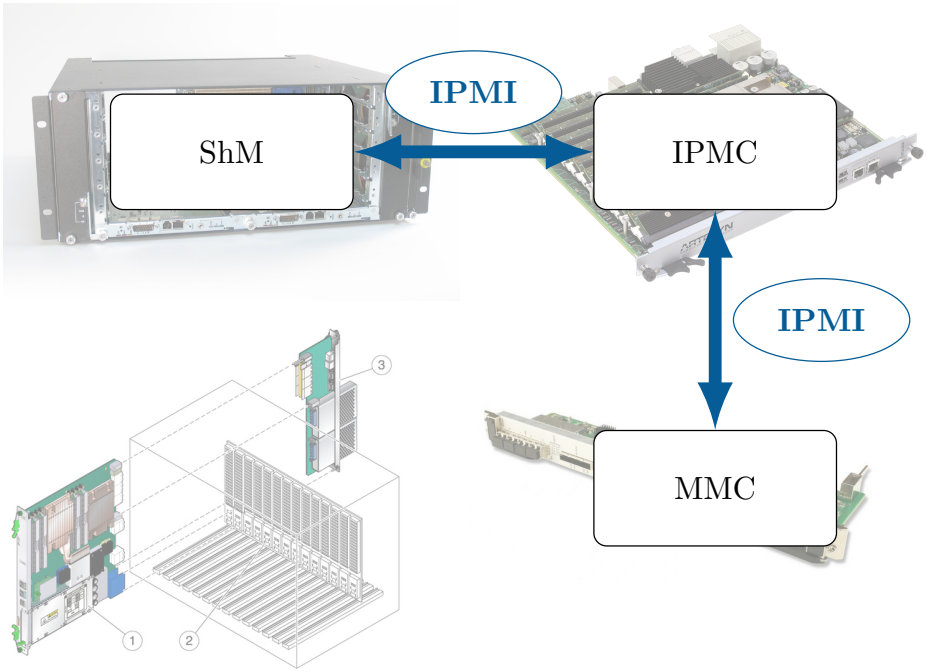
Source: Elaborated by the author.

Under this scope, a safe environment is required once developers will not be sitting near the boards during the development. Next section will discuss about the resource management in the AdvancedTCA context.

1.4.2 AdvancedTCA management

As explained in the section 1.2, the AdvancedTCA standard defines three management layers, relying on the IPMI specification. The ShM is in charge to manage all the elements that are inside of the crate. The IPMC is in charge of the related blade resources and talk directly to the ShM. Each intelligent device on a blade is required to have an MMC to manage inner resources and to respond to a local IPMC. Figure 13 put those elements in context and also shows the communication between then using the IPMI protocol. Messages from ShM to the MMC are possible in a two-step approach through the IPMC.

Figure 13: AdvancedTCA elements and its controllers: shelf with ShM, blade with IPMC, and RTM with MMC. Communication between them happens through IPMI protocol.



Source: Elaborated by the author.

1.4.3 Pulsar IIb IPMC card

The Pulsar IIb IPMC, Figure 14, is a small card designed by Fermilabto serve as a controller for the Pulsar IIb blade. However, it also support many other non-IPMI

functions, such as the File Transfer Protocol (FTP) server, the Telnet service and the sensor management. Thus, the IPMC emerged as a multi-functional project and it was decided by using an Real Time Operating System (RTOS) to manage the resources. The Keil Real Time Operating System (RTX) was the RTOS of choice, alongside with the TCPnet network solution, both discussed in the following sections.

Figure 14: Picture of the Pulsar IIb IPMC card designed by Fermilab and the central element of this project.

Source: Provided by Fermilab.

1.4.3.1 The RTX Real Time Operating System

The Pulsar IIb IPMC is in charge of multiple functions, managed by its Operating System (OS) RTX (KEIL, 2016[a]), as adopted by Fermilab. Some of its resources shown in the Figure 15 are explained below due to their importance to this project:

Advanced RISC Machine (ARM) Cortex-M: RTX was developed for this architecture, which is the one used in the Pulsar IIb IPMC microcontroller, the LPC1788.

Hardware Interruption Requests: direct access to the interruptions generated by peripherals, named Fast Interrupt Request (FIQ) and Interrupt Request (IRQ), what provide a simpler code development.

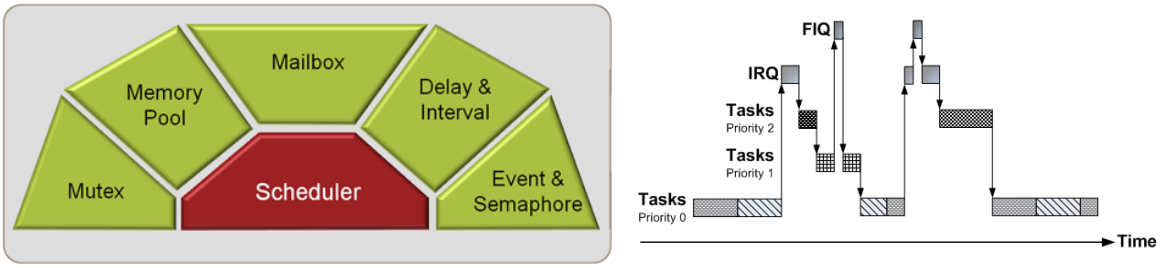
Tasks: basic unit of execution, usually focused on activities.

Scheduler: task execution according to priority or time management.

Event Handler: generation and process of software events direct managed by the OS.

RTX deal with activities switching, assuming only one processor and thus just one process can run each time. The priority is configurable during task creation and same priority tasks will be activated in a round-robin fashion, which means that each one will receive the same amount of processing time to run in a circular order. However, high priority tasks will always run if there is only lower priority tasks waiting for. Hardware interrupt requests, IRQ and FIQ, will always have the highest priority possible by design, so they should be used only to trigger other tasks.

Figure 15: RTX is an RTOS developed by Keil. (a) shows its main features. (b) shows how priorities are handled while tasks switching.



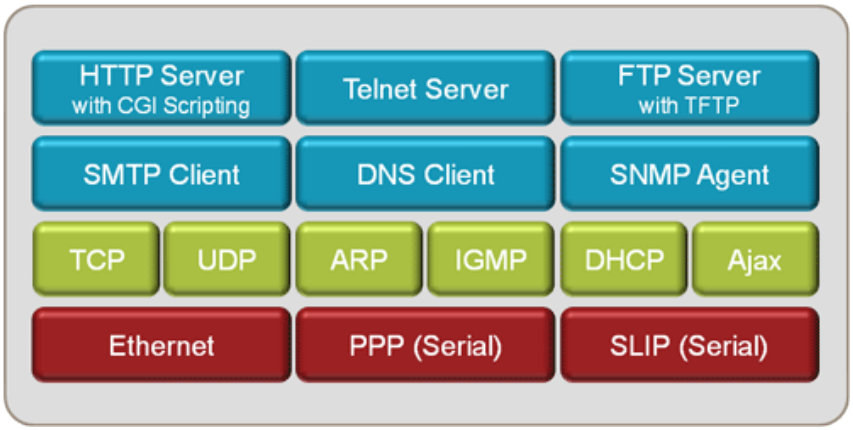
(a) (b)

Source: Keil's RTX support material (KEIL, 2016[a]).

1.4.3.2 *Keil Network Suite solution – TCPnet*

Fermilab acquired the Keil Networking Suite, called TCPnet (KEIL, 2016[b]) module, which may run over RTX. It is a full network stack implementation, with support for Transmission Control Protocol (TCP) and Internet Protocol (IP) resources, specifically written for Cortex-M processor-based microcontroller and includes also common networking applications, as shown in Figure 16, such as Hypertext Transfer Protocol (HTTP), telnet, FTP servers, among others. For this project, the important resources are the TCP/IP protocol itself and the TCP socket interface.

Figure 16: TCPnet Module provided by Keil is a full network stack implementation, with solutions ranging from services, clients, sockets and interfaces.



Source: Keil's TCPnet support material (KEIL, 2016[b]).

1.4.3.3 *IPMI status*

According to the AdvancedTCA standard, every front board is required to support the IPMI protocol for the communication between its controller and the ShM. Despite of

the fact of the Pulsar IIb project is in the AdvancedTCA context, there were no support for IPMI and a complete implementation of that would not be feasible during the time available for this project, which was also dependent on the infrastructure provided by Fermilab. Thus, some options for the IPMI protocol were evaluated:

- Pigeon Point sells a design reference with the IPMI functions for IPMC, but it is not only expensive, but also the electronic adaptation would take more time than was available and there is already a group using this solution at CERN (MENDEZ et al., 2016).
- CoreIPM is a free and open source project for the IPMI protocol. It shares code between the ShM, IPMC and MMC implementations, which makes it complex but a fair start point. This option was mainly excluded because of license incompatibilities with the proprietary solutions (RTX and TCPnet) approach followed by Fermilab.
- Laboratoire d'Annecy-le-Vieux de Physique des Particules (LAPP) designed an IPMC project aligned with the form factor required by the Pulsar IIb blade (ATLAS LAPP, 2016) but it was not available at Fermilab. This project is also based in the CoreIPM, thus to use only the code for the Fermilab IPMC would not be possible for the same reason of the previous item.

As the goal of this project is only a prove of concept for an remote development environment using the AdvancedTCA standard, not the implementation of the IPMI protocol itself, and as the above mentioned solutions were discarded, the only option was to build a simple IPMI implementation.

1.5 GOALS

Based on the previous discussion, it was decided for a IPMI implementation with only the bare minimum resources needed. For that, the following points were pursued:

- Pulsar IIb front board should be recognized by ShM after insertion in the crate.
- ShM should be able to track the Pulsar IIb front board sensors to ensure a safe operation.

On the other hand, the XVC protocol and the related necessary structure should be implemented, allowing in-circuit programming and debugging of the Xilinx devices of

Pulsar IIb blades through the network resources offered by the AdvancedTCA standard, such as connections through AdvancedTCA switch and backplane.

1.6 STRUCTURE OF THIS TEXT

In this chapter, the general aspects about this development were discussed. Chapters 2 and 3 contains the theory background and the materials and methods, respectively, complete the substrate of the implementation, which is described in the chapter 4. Results, and discussion about them, are shown in the chapter 5, while in the chapter 6 some conclusions are announced together with plans for future work.

2 THEORY BACKGROUND

In this chapter some of the basic concepts about IPMI and XVC, as well as the related technologies, will be visited to prepare the appropriate knowledge base for further developments.

2.1 THE I^{2}C PROTOCOL

This section elaborates on the I^{2}C protocol that is the base of the IPMI communication inside the shelf. Thereafter, a bottom to top method is outlined, where the IPMI protocol is in the other end.

Several electronics systems, in industrial or telecommunication area, need similar requirements like management or remote control, wide resource types, circuit switching, and others. In scope of IPMB protocol, the I^{2}C interface describes the Physical Layer for a simple media access control. In the case of multi-master links, which is the one for IPMI standard, I^{2}C synchronizes transmitters and receivers and also makes the throughput control. I^{2}C Bus specification (NXP SEMICONDUCTORS, 2014) provides a bidirectional 2-wire physical interface to enhance the efficiency without lose simplicity for communication between Integrated Circuits (ICs). Some relevant aspects of I^{2}C bus are:

- The connection needs only a wire for Serial Data (SDA) line and other for Serial Clock (SCL) line;
- Each device has a unique address to communicate in the bus;
- Use the simple client-server (master-slave) model;
- Include collision detection for data corruption prevention in the case of multi-master simultaneous data transferring;
- The bus is used in a 8-bits serial mode, half-duplex in 100 kbps in Standard Mode, up to 400 kbps in Fast Mode, or up to 3.4 Mbps in High-Speed Mode;
- The number of devices per bus is limited by the maximum fan out of the bus.

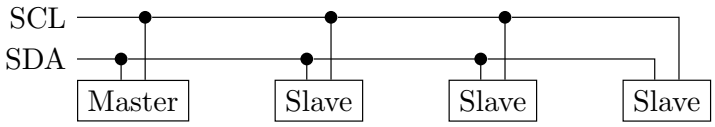
Taking into account the terminology in Table 1, the I²C specification is a description of a simple communication between two devices, which share the same 2-line bus, as illustrated in Figure 17. The master generates a clock pulse on SCL wire for each bit transmitted on the SDA wire that are sampled on the rising edge. For better signal conditioning, the data changes must occur when the falling edge of SCL signal happens.

Table 1: Adopted terminology and brief description for I²C elements.

Term	Description
Transmitter	The device which sends data to the bus
Receiver	The device which receives data from the bus
Master	The device that initiates and after terminates a transfer, and generates clock signals
Slave	The device addressed by a master
Multi-Master	More than a master can control the bus data transfers
Arbitration	Process that checks data corruption in case of a possible collision on bus
Synchronization	Process that synchronize the clock signals of two or more devices

Source: Adapted from NXP Semiconductors (2014).

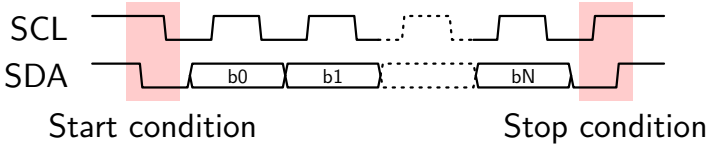
Figure 17: Diagram of I²C devices interconnection with shared SCL and SDA signals.



Source: Elaborated by the author.

To initiate a data transfer from device A to device B, the master (device A) should provide a falling edge of SDA while SCL is set high. This first step, named `start condition`, indicates to all devices that the bus will be busy until the final step, named `stop condition`, which in turn happens when the master provides a rising edge of SDA while SCL is set high. This timing diagram is shown in Figure 18.

Figure 18: Start and stop condition for I²C communication.

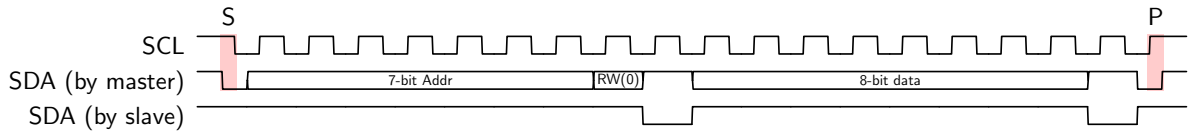


Source: Elaborated by the author.

The format of the data frames sent by the master, Figure 19, begins with an unique slave 7-bit I²C address after the `start condition`. The bit that complement the first byte is the `data direction` bit which means a write command when is set to low level. To send the next 8-bit word, the master must receive an Acknowledge (ACK) condition provided by the slave, which means an extra clock pulse where the slave must hold down

the SDA line. The slave device will receive data from the master until the `stop condition` happens, which means that this block of words should be written to the slave memory in a sequential fashion.

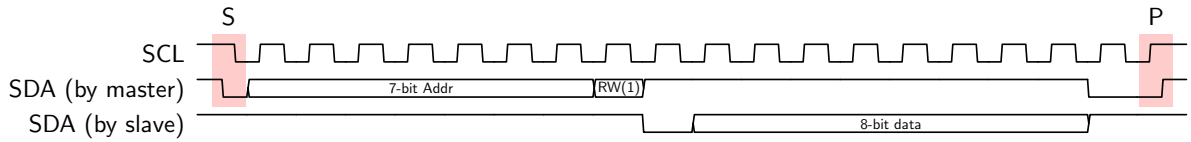
Figure 19: Write operation in an I^2C communication for just one byte. Direction bit is zero.



Source: Elaborated by the author.

For read commands, shown in Figure 20, the `data direction` bit must be set to low level and after the ACK is received from the slave, the master must release the SDA line and acknowledge each word received in a similar fashion of the write operation. Again, the slave will send the data until the `stop condition` is raised and it means that a block of words is completed in the master side.

Figure 20: Read operation in I^2C communication for just one byte. Direction bit is 1.



Source: Elaborated by the author.

2.2 THE IPMI PROTOCOL

The IPMI is a very extensive protocol for hardware management, controlling the conditions of the AdvancedTCA boards throughout the operation period. For this project, the minimum implementation found involves basically two situations: first, the board must be recognized by the ShM when it is plugged; and second, the ShM must be able to monitor the blade state.

The activation and deactivation stages are part of the Hot Swap procedure, which provides a safe method to insert boards into the shelf, or remove them from that, without shutting the AdvancedTCA system down. ShM make sure, for instance, that the payload power is enough for all the entities inside the shelf before incorporating another one. It is during the initial process that the ShM becomes aware of the blade and its attributes.

Once the information about the blade is registered, ShM will continue to monitor the boards during the operation, checking sensors and taking actions to ensure a safe

running. For examples, ShM will increase the fan speed to reduce the temperature of the components inside the shelf.

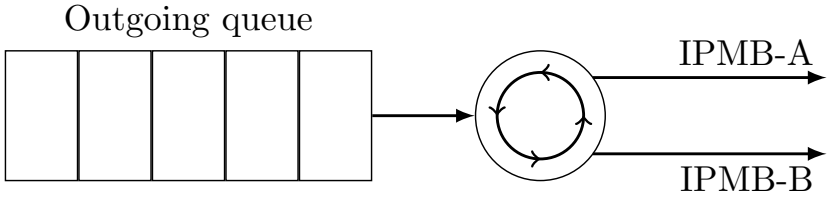
It is interesting to add that the IPMI is a very wide-ranging specification (INTEL et al., 2013a) and defines several interfaces, such as Intelligent Chassis Management Bus (ICMB), Keyboard Controller Style (KCS), System Management Interface Chip (SMIC), Block Transfer (BT), System Management Bus (SMBUS), Local Access Network (LAN), Serial/Modem and Serial on LAN (SOL). This project is focused on the IPMB interface because it is the one used in the AdvancedTCA shelf context, which will be elaborated in the next sections, along with the Hot Swap and monitoring phases.

2.2.1 IPMB communications protocol specification

The IPMB communications specification defines a byte level transport for IPMI transfers over I^2C (INTEL et al., 1999). The AdvancedTCA standard improves the reliability of the IPMB with the addition of a second channel. Those are named IPMB-A and IPMB-B and together form the IPMB-0. Queues are defined in both sides of the transmission lines to make the access transparent for the IPMI mechanisms.

As it is illustrated in the Figure 21, messages are held in an outgoing queue and the devices alternate transmission between channels in a round robin fashion, provided by a circular buffer.

Figure 21: IPMB transmission queue where messages are kept before transferring, with round robin access to the medium.

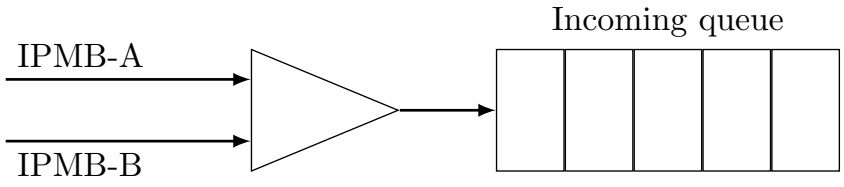


Source: Elaborated by the author.

For the reception side, devices should be ready to get data from both channels, which are chained in an incoming queue, as shown in Figure 22. After that, each message is retrieved from the queue and then processed, minimizing the chance of information being overwritten because of delays in the process.

The IPMI protocol uses a request/response model. Every valid request must have a response in a separate message. The physical bus management acknowledge made by I^2C

Figure 22: IPMB reception queue where messages are stored before being processed.

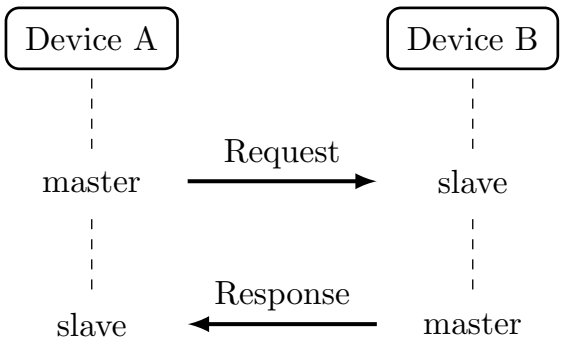


Source: Elaborated by the author.

do not imply that the incoming data is safe for further use and Checksum (CHK) using 2's complement method to insure data integrity. A completion code in the answer informs the status of the result sent, where non-zero values indicates a not successful situation. In addition, generic completion codes can be used as part of any answer, such as for normal execution `00h`, or command specific which cover Original Equipment Manufacturer (OEM) commands.

Messages between intelligent devices are always sent as master, meaning that the bus operates in a multi-master mode and also that the I^2C read command is never used, that is, the I^2C direction bit is always zero and is taken as part of the address (ADDR) for IPMI transactions. Requests and responses are sent as I^2C master devices and received as slave, as illustrated in Figure 23.

Figure 23: During one IPMB communication, the sender always master the channel for both cases, request or response.



Source: Elaborated by the author.

After the first byte, also used by the I^2C protocol, all the others are part of the IPMI payload only. As an I^2C message just carries the destination ADDR, the request/response model forces the source ADDR of the message to be included in the payload. Still under the addressing scope, the IPMI specification defines the concept of a Logic Unit (LUN),

making possible to have more than one device in the same ADDR.

Requesters (RQs) are supposed to keep the message for retransmission in case of time out, but that is not true for responders (RSs). The request should be removed from the outgoing RQ queue when the related response arrives or the maximum number of tries is surpassed. The Sequence Number (SEQNUM) is defined by the IPMI specification as a way to differentiate between them and should be renewed for each new request but not for retransmissions. As the responses are not re-sent, the process relies on the request retransmission.

Resource groups called Network Function (NETFN) are defined as in the Table 2 and have different sets of commands. Each Command Identification (CMDID) represents a specific instruction in that area and is the same for a pair of request and response, wich can be recognized using:

- node addresses (physical and logical);
- network function;
- command identification; and
- sequence number.

Table 2: Name, reserved values and brief description for the main Network Function groups defined by the IPMI specification.

Function name	Request	Response	Description
Chassis	00h	01h	Chassis (board) control or status
Bridge	02h	03h	Message for bridging to next bus
Sensors and Events	04h	05h	Configuration or transmission of events or sensors
Application	06h	07h	Related to the node unique functionality
Firmware	08h	09h	Related to Firmware Transfer
Storage	0Ah	0Bh	Messages for non-volatile storage devices
Transport	0Ch	0Dh	Media-Specific Configuration of Serial and LAN Ethernet interfaces
Reserved	0Eh – 2Bh		30 reserved NETFN codes
PICMG / AdvancedTCA	2Ch	2Dh	AdvancedTCA context related to IPMI commands
OEM	2Eh	2Fh	Non-IPMI group of commands
Controller specific	30h – 3Fh		Vendor Specific IPMI commands

Source: Adapted from PICMG (2008) and Intel et al. (2013a)

A request consists of a connection header with an even network function and the command, which may require other information, as shown in Format 1. The RS address is the I^2C ADDR itself. The length of each field are: ADDR is 8 bits long (or 7 bits plus the I^2C direction bit that is always zero); NETFN and SEQNUM are 6 bits long and support 64 different values; LUN is 2 bits, which results in 4 logical units for each I^2C ADDR.

Format 1: Specified format for a frame containing an IPMB request.

	7	6	5	4	3	2	1	0	
1 byte	RSADDR								Address of destination device (responder).
	NETFN						RSLUN		NETFN identifier. Destination LUN address (responder).
1 byte	CHK 1								First checksum related to the previous bytes.
1 byte	RQADDR								Address of source device (requester).
	SEQNUM						RQLUN		Sequence number. Source LUN address (requester).
1 byte	CMDID								Command identifier.
Variable length	Up to 25 bytes.								Request information body.
1 byte	CHK 2								Second checksum, related bytes after first checksum.

Source: Adapted from Intel et al. (1999) and Intel et al. (2013a)

The description above is analogous for responses. However, the connection header of a response must be assembled in the opposite direction, with the requester information as destination. The sequence number will be the same of the request but the network function should be changed to the related odd value. Also, the completion code and associated data must be added to the message informing the requester the status and results about the related request. The response frame is illustrated in Format 2.

It is interesting to notice that the way the CHKs are placed, completion codes will never report failed integrity checks. Until the first CHK arrives, there is no information about the sender and if the error is flagged by the second CHK, it is not possible to trust on the source ADDR field. That means those messages will be discarded and the target will wait for a retransmition.

Interface from IPMC to subsidiary FRUs can include intelligent controllers and communicate with them over an IPMB-compatible interface named IPMB-L. In the following sections many operations will be described that makes use of the structure described above. It is worth mentioning that only the body of the message, request and response data, will be detailed as the format of the header will not change.

Format 2: Specified format for a frame containing an IPMB response.

	7	6	5	4	3	2	1	0	
1 byte	RQADDR								Address of destination device (requester).
	NETFN						RQLUN		NETFN identifier. Destination LUN address (requester).
1 byte	CHK 1								First checksum related to the previous bytes.
1 byte	RSADDR								Address of source device (responder).
	SEQNUM						RSLUN		Sequence number. Source LUN address (responder).
1 byte	CMDID								Command identifier.
1 byte									Completion code.
Variable length	Up to 24 bytes.								Response information body.
1 byte	CHK 2								Second checksum, related bytes after first checksum.

Source: Adapted from Intel et al. (1999) and Intel et al. (2013a)

2.2.2 FRU State Machine

Once the basic structure for IPMI communication is founded, the greatest interest of this project was to make the ShM aware of the blade, as discussed in the section 1.4. As defined before, FRU are the devices able to perform hot-swap operations at the AdvancedTCA shelf. In this scope, the operator is able turn a carrier board on, off or even remove it from the shelf in a safe manner. The control provided to the operator is related to some elements:

Hot-swap handle: The hot-swap handle allows activation or deactivation requests to be sent to the ShM by the operator. It is a three state switch: open, close-inactive, close-active.

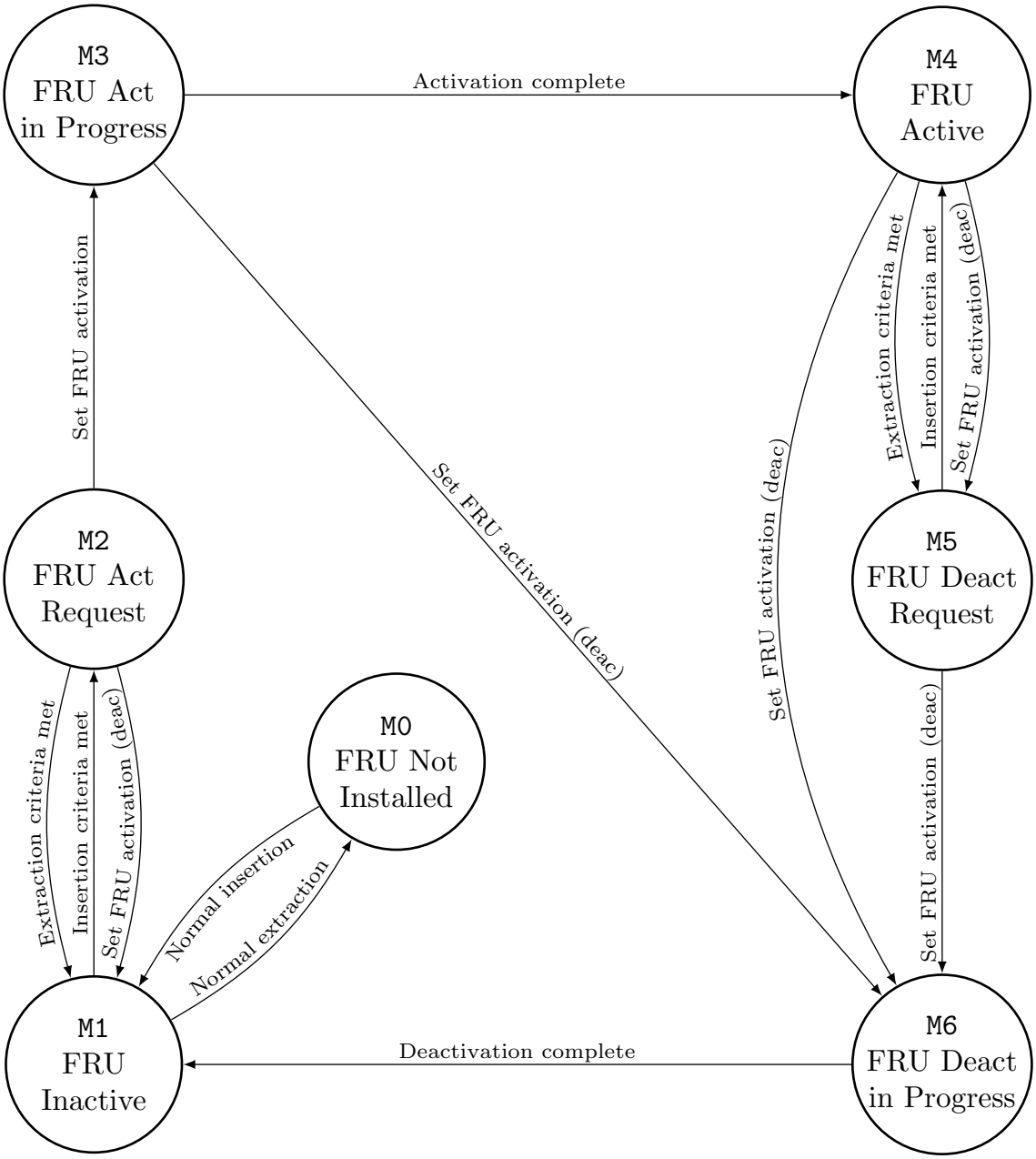
IPMC: when the FRU is not active and the switch handle is open, the carrier board receives power enough to activate only the IPMC, allowing message exchanges for basic procedures, such as identification.

Blue Light Emitting Diode (LED): through its blink behavior, the operator is able to follow the activation and deactivation process steps, with no need of direct access to the IPMI messages exchanged or a external equipment.

FPGA: after the activation requisition from the operator, the FPGA must be turned off until ShM decides if is safe the FRU become active.

The hot-swap mechanism is managed through a state machine, illustrated in Figure 24. When the board is disconnected, it is in the state `M0`. After plugged, the FRU will assume the innactive state, `M1`. If the activation process is triggered, the state machine will change to `M2` and then to `M3`, where it will wait for instructions from ShM. If the activation is not allowed, then the state machine goes directly to `M6`. If the activation is allowed, than FRU will stay as `M4` while a deactivation process is not triggered, what would take it to `M5`. It must be highlighted that if the deactivation is refutted, FRU will be sent back to `M4`. From `M6`, only `M1` can be reached.

Figure 24: FRU State Machine defined by the IPMI specification, which is used to drive the hot-swap mechanism.



Source: Adapted from PICMG (2008) and Perek and Makowski (2011).

Together with a short description of the hot-swap states, in the Table 3 is found the behavior of the Blue LED defined by the AdvancedTCA standard. If it is steady, board is powering up or ready for extraction. If the LED is blinking, hot-swap is in process. And if it is off, the FRU is active (PICMG, 2008).

Table 3: Relationship between the FRU state machine and the carrier board situation during the hot-swap process.

State	State Name	Additional Info	Blue LED Behavior
M0	FRU Not Installed	Board on hands	Off
M1	FRU Inactive	Handle switch open	On
M2	FRU Act Request	Handle switch closed	Long blink
M3	FRU Act in Progress	ShM validate activation	Off
M4	FRU Active	Power Negotiation Granted	Off
M5	FRU Deact in Req	Handle switch open	Short blink
M6	FRU Deact in Progress	ShM validate deactivation	Short blink

Source: Adapted from PICMG (2008) and Perek and Makowski (2011).

Every FRU state transition on IPMC have to be reported to ShM using the NETFN 04h/05h CMDID 01h (Platform Event) request (INTEL et al., 2013a). This command informs ShM if the hot-swap handle is open or closed, the current and previous states, and also the cause of transition. According to which state the IPMC is in a specific moment, there are different sets of information that the ShM should exchange through IPMI transactions.

In the activation process, the communication starts with transition from M0 to M1 state. From the moment that this Platform Event request is reported, ShM knows the IPMC address and starts to ask for information from the related carrier board using the NETFN 06h/07h CMDID 01h (Get Device ID) (INTEL et al., 2013a) and NETFN 2Ch/2Dh CMDID 00h (Get PICMG Properties) (PICMG, 2008) requests. Responses for them will show device identification, firmware version and capabilities. NETFN 04h/05h CMDID 00h (Set Event Receiver) configure the destination of the events generated by the IPMC (INTEL et al., 2013a).

Just when the hot-swap handle is closed, the FRU activation is requested. In this case, IPMC reports to ShM the M1 to M2 transition using the Platform Event request. After this, ShM starts to get the front board sensors information, called sensor data records (SDR), using the NETFN 04h/05h CMDID 21h (Get Device SDR) request (INTEL et al., 2013a). However the M2 to M3 transition will only happen when the ShM validate the activation process with the NETFN 2Ch/2Dh CMDID 0Ch (Set FRU Activation) (PICMG,

2008) request.

The M3 state, FRU act in progress, must perform a power negotiation between ShM and IPMC. With the knowledge of the power payload available for each shelf physical slot and the number of FRUs and its states, ShM can analyse if the current activation proced should perform. A NETFN 2Ch/2Dh CMDID 12h (Get Power Level) request is issued to IPMC, which in turn must inform the steady and desired power draw levels. After validation, ShM set the power draw level that IPMC should use at that moment using NETFN 2Ch/2Dh CMDID 11h (Set Power Level) (PICMG, 2008) request.

The M3 state is used by ShM to collect the FRU Data Repository from the related front board and IPMC must answer the NETFN 0Ah/0Bh CMDID 11h (Read FRU Data) (INTEL et al., 2013a) request. This repository brings, among other information, the base and fabric point-to-point interfaces, also called E-Keying Point-to-Point Connectivity (E-Keying). At this moment, the ShM tries to ensure compatibility with the backplane disabling channels that are not available through NETFN 2Ch/2Dh CMDID 0Eh (Set Port State) (PICMG, 2008) request.

At this point the IPMC can perform the transition from M3 to M4 at any time and the FRU will be active with full capacity. Consistently, ShM must be informed through a Platform Event request. The whole sequence is illustrated in Figure 25.

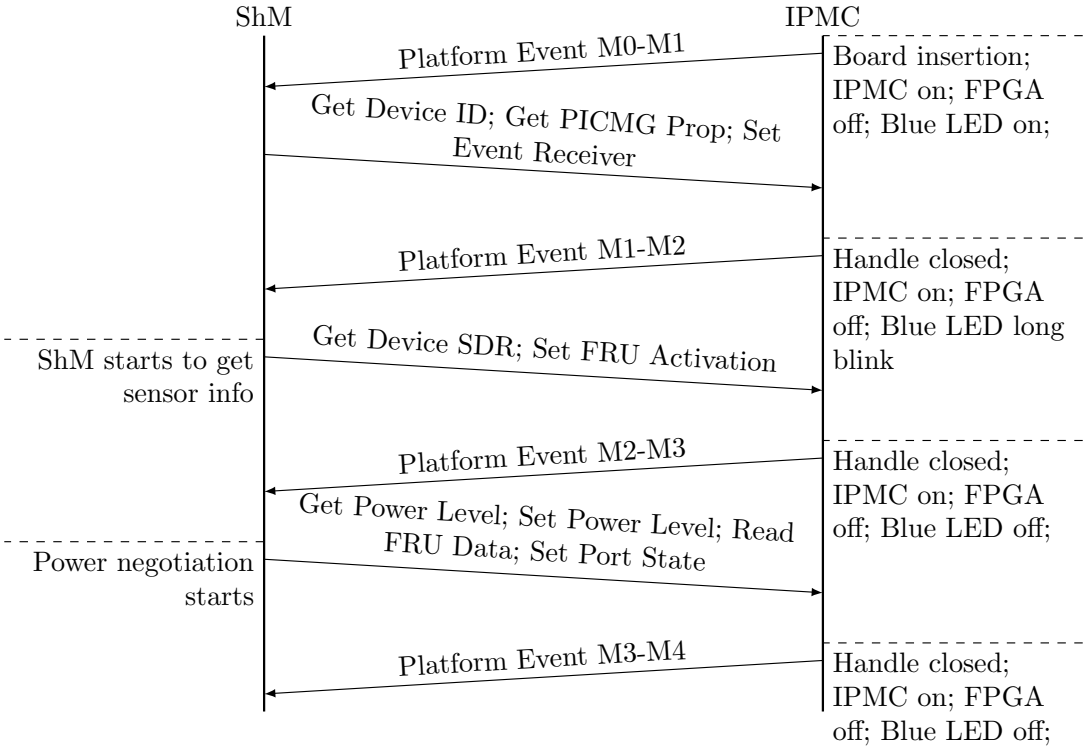
During the activation process, ShM gets some specific board information. SDRs with the sensor records and FRU Data with E-Keying provide ShM with resources to make decisions about compatibility and control.

The FRU deactivation process only begins when the operator opens the hot-swap handle or when ShM sends a Set FRU Activation with deactivation setted (PICMG, 2008). In this way, transitions and related Platform Event requests reporting happens in sequence from M4 to M5, M5 to M6 and M6 to M1, where FRU finally reaches the inactive state, as seen in Figure 26.

2.2.2.1 FRU Information Storage Definition specification

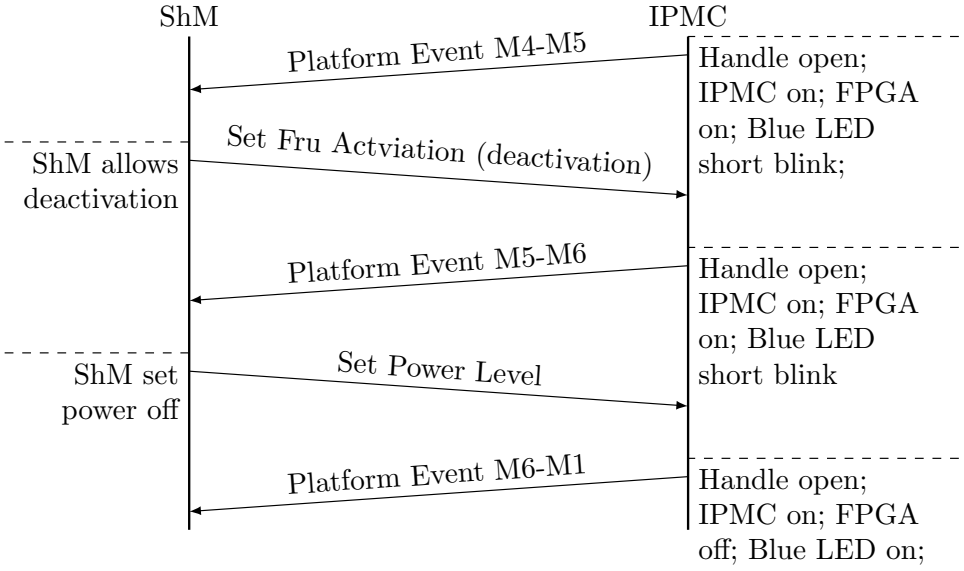
The description about the resources of a system that uses IPMI protocol is the primary function of FRU Information Devices, which provides inventory data about the boards where the device is located on. The FRU Storage Definition Specification (INTEL et al., 2013b) describes the format and the uses of that, splitting the data into six different areas, detailed bellow. It is important to add that checksum is repeatedly used along the

Figure 25: FRU activation diagram with focus on IPMI requests exchanged between ShM and IPMC.



Source: Elaborated by the author.

Figure 26: FRU deactivation diagram with focus on IPMI requests exchanged between ShM and IPMC.



Source: Elaborated by the author.

sub-structures to ensure integrity of data.

The Common Header is always placed first in the structure because it is the starting point to access the data information and is considered as the reference for the offsets it holds for the others, as illustrated in Format 3, where some names are anticipated. When one of those offsets is set to zero, the respective area is considered absent. It is important to point out that it is the only mandatory area.

Format 3: Common Header describing the structure of all FRU Information areas.

	7	6	5	4	3	2	1	0	
	0h				1h				Reserved. Format version.
1 byte	Multiples of 8 bytes.								Internal Use Area offset. `00h` if not used.
1 byte	Multiples of 8 bytes.								Chassis Info Area offset. `00h` if not used.
1 byte	Multiples of 8 bytes.								Board Info Area offset. `00h` if not used.
1 byte	Multiples of 8 bytes.								Product Info Area offset. `00h` if not used.
1 byte	Multiples of 8 bytes.								MultiRecord Info Area offset. `00h` if not used.
1 byte	00h								Padding
1 byte	Zero checksum.								Common Header Area Checksum.

Source: Adapted from Intel et al. (2013b).

`Type/Length` byte format plays an important role in the subsequent areas because they contain a number of variable length fields. Each of these fields is preceded by that special byte, which indicates the length and the type of the encoding used, as shown in Format 4. The type code can be chosen between binary or unspecified (`00b`), Binary-Coded Decimal (BCD) Plus (`01b`), packed 6-bit American Standard Code for Information Interchange (ASCII) (`10b`) and 8-bit characters (`11b`), as it is in specification (INTEL et al., 2013b).

Format 4: `Type/Length` byte to describe FRU data allowing variable length fields.

	7	6	5	4	3	2	1	0	
1 byte	0 – 3		0 – 63						Type code. Number of data bytes.

Source: Adapted from Intel et al. (2013b).

It is also important to point that the `Type/Length` byte makes possible a more flexible way to work with data structures when chained. The application only needs to track the fields sequentially until reach the terminator `Type/Length` which the reserved value `C1h`. This approach is found when a region do not have pre-defined length or order, such as custom fields.

The first offset of the Common Header is for the Internal Use Area. This region is reserved area for private non-volatile storage for the internal devices in the entity that holds that information device. The second offset is related to Chassis Info Area, which in

turn holds information about the chassis and must exist in only one place, meaning that it is found in a device that is part of a board associated with chassis (PEREK; MAKOWSKI, 2011; INTEL et al., 2013b), and it is outside of the scope of this project.

The third offset reaches the Board Info Area with information of any replaceable entities, boards or sub-assemblies where the device is located on, and that are not considered as a standalone product. This format brings information about language, date and time, board manufacturer, product name, serial number, part number, FRU file identifier and other manufacturer additional information, as presented in the Format 5 (PEREK; MAKOWSKI, 2011; INTEL et al., 2013b).

Format 5: Board Info Area describes any replaceable entities, obards or sub-assemblies where the FRU Information Device is located on.

	7 | 6 | 5 | 4 | 3 | 2 | 1 | 0	
	`0h` | `1h`	Reserved. Format version.
1 byte	Multiples of 8 bytes.	Board Info Area Length.
1 byte	`00h` for English.	Language code.
3 bytes	Number of minutes from 1996-01-01T00:00.	Manufacture Date/Time
1 byte	`Type/Length` byte	Board Manufacturer descriptor.
Variable length	Manufacturer specific.	Board Manufacturer description.
1 byte	`Type/Length` byte	Board Product Name descriptor.
Variable length	Manufacturer specific.	Board Product Name description.
1 byte	`Type/Length` byte	Board Serial Number descriptor.
Variable length	Manufacturer specific.	Board Serial Number description.
1 byte	`Type/Length` byte	Board Part Number descriptor.
Variable length	Manufacturer specific.	Board Part Number description.
1 byte	`Type/Length` byte	Board FRU File Identifier descriptor.
Variable length	Manufacturer specific.	Board FRU File Identifier description.
Variable length	Manufacturer specific. Pairs of `Type/Length` byte and data.	Custom Manufacturer Information.
1 byte	`C1h`	Last field identifier.
Variable length	Not used.	Remaining space.
1 byte	Zero checksum	Board Info Area Checksum.

Source: Adapted from Intel et al. (2013b).

The Product Info Area, forth offset, complements the Board Info region describing entities that are considered standalone products but it is not the case of using that for

this project.

MultiRecord Info Area is pointed by the fifth and last offset of the Common Header. It consists of a sequence of records with predefined headers as in the Format 6 and complemented by the Table 4. The Type field specify the information in the record and a bit in the second byte identify if that record is the last one. Then, in a similar approach of the `Type/Length` method, the application must read record by record until the end of list bit is enabled (INTEL et al., 2013b).

Format 6: MultiRecord Area record header consists of a flexible approach using records described by predefined headers.

	7	6	5	4	3	2	1	0	
1 byte									Record type
	`0b` or `1b`		`000b`			`2h`			End of list (`1b` to stop). Reserved. Format version.
1 byte	Length in bytes.								Data size
1 byte	Zero checksum								Record checksum.
1 byte	Zero checksum								Header checksum.

Source: Adapted from Intel et al. (2013b).

There are some record types defined in the FRU data specification, like Power Supply Information, Direct Current (DC) output and DC load (INTEL et al., 2013b). However the most important for this project is the extension provided by the OEM record type. Making use of this, the AdvancedTCA standard extended the inventory management capabilities with types more suitable for the expected resources (PICMG, 2008).

Through the record type E-Keying, Format 7, AdvancedTCA standard (PICMG, 2008) ensures that interfaces of any front board is compatible and interoperable with different shelves, preventing not only damage but also mis-operation. Three kinds of point to point interfaces are described by the E-Keying information:

Base: used to support 10/100/1000BASE-T Ethernet connections among boards in a Shelf in a star topology trought hub boards.

Fabric: supports a number of topologies including full mesh, where each board has a direct connection to every other board in the backplane. A channel consists of 8 differential signal pairs, protocol agnostic.

Update Channel: it is comprised of 10 differential signal pairs in a point to point connection between two boards.

Table 4: Types of records that might be used in the description of the MultiRecord Area.

id	Record type
00h	Power Supply Information
01h	DC Output
02h	DC Load
03h	Management Access Record
04h	Base Compatibility Record
05h	Extended Compatibility Record
06h	reserved for ASF Fixed SMBus Device record.
07h	reserved for ASF Legacy-Device Alerts
08h	reserved for ASF Remote Control
09h	Extended DC Output
0Ah	Extended DC Load
0Bh – 0xBFh	Reserved
C0h – 0xFFh	OEM Record Types

Source: Adapted from Intel et al. (2013b).

With that, ShM can enable only compatible ports according to the configuration of the backplane and the other boards connected. Backplane description is part of the shelf FRU data and will not be detailed here.

In the E-Keying record has type OEM, as it is an extension for the basic IPMI definitions, and identification 14h, as the AdvancedTCA standard defines other records that are not mentioned in this document. The Globally Unique Identifier (GUID) is a 128 bits long value assumed as not repeated for more then a thousand years if generated in a compliant manner (PICMG, 2008). Each link descriptor in the list describes one type of point to point protocol supported by the referenced channels.

For the AdvancedTCA protocol, the FRU information is accessed through IPMC, what protects the data and makes the storage method transparent to the ShM. The main command for that is the Read FRU Data, where the body content is detailed in the Format 8 for the request and for the response, which interactively get the information in pieces due to the frame length limitation of the protocol (INTEL et al., 2013a).

2.2.2.2 Sensor Data Record

Each sensor or object inside the hardware platform management system is described by an SDR, which provides the only method detailing the existence and access methods of FRU information. The AdvancedTCA standard defines that each FRU device must have an associated SDR to allow information to be accessed. IPMCs are considered dynamic

Format 7: Layout specified for the AdvancedTCA E-Keying record, used to describe the link interfaces provided by the FRU element.

Header	7	6	5	4	3	2	1	0	
1 byte	OEM record type								`C0h`
1 byte	`0b` or `1b`	`000b`			`2h`				End of list (`1b` to stop). Reserved. Format version.
1 byte	0 – 255								Record length
1 byte	Zero checksum								Record checksum.
1 byte	Zero checksum								Header checksum.
Body	7	6	5	4	3	2	1	0	
3 bytes	PICMG identification (`00315Ah`)								Manufacturer identifier.
1 byte	`14h`								E-Keying PICMG record id.
1 byte	`00h`								Record format version.
1 byte	0 – 255								OEM GUID count.
Variable length	16 bytes for each GUID								OEM GUID list.
Variable length	4 bytes for each link descriptor								Link descriptor list.

Source: Adapted from Intel et al. (2013b).

Format 8: Layout of the Read FRU Data command used by the ShM to access the FRU information.

Request	7	6	5	4	3	2	1	0	
1 byte	FRU device identifier								0 – 255.
2 bytes	FRU inventory offset to read								Up to 64K.
1 byte	Count to read								1 – 255.
Response	7	6	5	4	3	2	1	0	
1 byte	Completion code								Generic or `81h` if busy.
1 byte	Count returned								1 – 255.
2 bytes	Requested data								Up to 64K. From offset position on.

Source: Adapted from Intel et al. (2013b).

sensor devices and are required to maintain device SDRs for all entities they manage and will be exposed to the ShM (PICMG, 2008; INTEL et al., 2013a).

Commands to access that information must be inplemented. The command Get Device SDR, Format 9, allows the SDR information to be returned. It consists of:

header that is the same for all records with record identifier, type and length; among types, there are descriptions for sensors, events, devices, entity association, OEM reserved, and others.

key fields that together are unique among instances of a given record type;

body with the specific information.

Format 9: Layout of the Get Device SDR command, issued by the ShM to access the SDR information of a FRU element.

Request	7	6	5	4	3	2	1	0	
2 bytes	Reservation identifier								Required for partial reads with non-zero offset.
2 bytes	Identifier of the record to get.								`0000h` returns the first record.
1 byte	Offset into record								
1 byte	Bytes to read								`FFh` for full record.
Response	7	6	5	4	3	2	1	0	
1 byte	Completion code								Generic or `80h` if record changed during transaction.
2 bytes	Record identifier for next record								`0000h` returns the first record.
Variable length	Data bytes								Requested bytes from record.

Source: Adapted from Intel et al. (2013a)

It is worth mentioning that the `Type/Length` format for the SDR records is slightly different from the one defined for the storage devices. As shown in Format 10, the fifth bit is reserved, remaining only 4 bits for the length description. Also, for the first two bits, `00h` defines a unicode string instead of a binary field.

Format 10: Slight change in the behavior of the `Type/Length` byte in the context of SDR records.

	7	6	5	4	3	2	1	0	
1 byte	0 – 3		`0b`	0 – 31					Type code. Reserved. Number of data bytes.

Source: Adapted from Intel et al. (2013a)

2.2.3 Monitoring

Through the informations available in SDRs, the ShM is able to get the sensor readings using NETFN `04h/05h` CMDID `2Dh` (Get Sensor Reading) and translate them to useful information, such as temperature or voltage (INTEL et al., 2013a). All the measured sensor readings collected are compared with pre-defined thresholds obtained with the SDR information. If a threshold is surpassed, ShM should act to ensure the healthy operation of the AdvancedTCA system. It is possible to define thresholds for both, high and low aspects:

Non-critical (NC): puts ShM under alert. If possible, ShM will try to revert the situation, for example increasing speed of the fans to cool down the temperature.

Critical (C): after this point, ShM will try harder to control the situation, such as setting fans to the maximum if temperature reach this situation.

Non-recoverable (NR): this limit means that the situation is out of control and the device should be turned off.

Basically, for a safe operation, measured values must be between the high and low NC thresholds.

During the monitoring stage, ShM makes periodic use also of the already mentioned command Get Device ID to verify if the board is still present, if the board was not changed without notification, and to ensure that the communication channel is not interrupted.

With the health of the whole AdvancedTCA shelf assured by the IPMI specification, it is safe to perform other kind of services using the AdvancedTCA equipment as a safe and properly monitored work environment. Next section will present the base of the XVC protocol.

2.3 THE XVC PROTOCOL

The Standard Test Access Port and Boundary-Scan Architecture (MAUNDER; TULLOSS, 1990) interface is defined by the Institute of Electrical and Electronics Engineers (IEEE) Standard 1149.1-2013 and is usually known as JTAG connection because it was developed by the association with the same name. It implements standards for on-chip instrumentation in Electronic Design Automation (EDA) as a complementary tool to

digital simulation and offers debug capabilities over a serial communication interface without requiring direct access to the inner structures.

The following signals support the operation of the boundary scan process:

Test Clock (TCK): shared sync signal for the internal state machine operations, always driven by the master of the bus;

Test Mode Select (TMS): shared signal for the next state, sampled on the rising edge of the clock, always driven by the master of the bus;

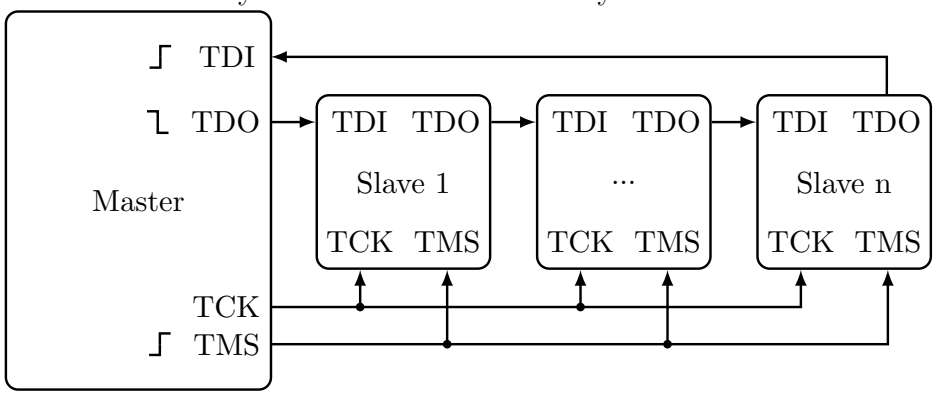
Test Data In (TDI): data shifted into the device, sampled at the rising edge of the clock when the internal state machine is in the correct state;

Test Data Out (TDO): data shifted out of the device, valid on the falling edge of the clock when the internal state machine is in the correct state;

Test Reset (TRST): optional pin which to reset the internal state machine.

Master devices always drive the TCK and the TMS signals, which many slave devices can be monitoring. In the other hand, the TDO and TDI signals are chained by each slave device until return to the master device, closing the cycle as illustrated in Figure 27.

Figure 27: JTAG connections between components of a JTAG chain, where the TDO signal pass through every slave device before returning to the master. TCK and tms signals are driven by the master and shared by the slave devices.



Source: Elaborated by the author.

As computers do not have JTAG interfaces to master those chains, thus JTAG connections are usually established through USB dongles, which in turn are short-range cables and offer only local access. However, the XVC protocol (XILINX, 2016[a]) is used to emulate JTAG connections between the workstation and Xilinx devices over an internet

link, in particular composed by TCP and IP protocols. For the context of this project, the Pulsar IIb IPMC will receive internet packets, extract the payload and convert them to JTAG data, acting as a master to the JTAG chain.

A common case of use is to have reconfigurable components inside data centers, what makes difficult to directly access them for programming and debug procedures through traditional techniques. However, those boards usually support regular network resources and it is natural to think about using that to access the devices. Thus, the developer can program and debug a firmware through remote access.

XVC protocol consists of only few commands, two of them to configure the connection and another which is a literal translation of low level JTAG vectors obtained in the payload of packet messages to JTAG signals, service that can be effortlessly processed in a microprocessor. It presents the same behavior of a native communication by programming cable, so the change is straight forward in the developer side. Also, this protocol is supported by Xilinx tools, which enable their own XVC solutions for dealing with their devices (XILINX, 2016[b]).

Once the TCP/IP link is established, the XVC request from the client is embedded with the following format:

```
<name>:<body>
```

where `<name>` is the name of the command as a characters string always followed by the colon character as a separator and `<body>` is the content of the command, which may consist of binary or text data. Is important to point that for each request, a response is required. There are only three commands available in the XVC protocol, as it is shown below.

The `getinfo` command is used to inform the version of the protocol and the maximum length for the JTAG vectors supported. The request format is

```
getinfo:
```

with empty `<body>` and the response format is

```
<XVC Server Version>,<maximum length>\n
```

with the fields specified before separated by a comma and ended with the newline character.

The `settck` command attempts to set the period of the JTAG clock rate in nanoseconds:

```
settck:<period in ns>
```

and the returned value is the actual specified period:

```
<actual period in ns>
```

where both are written as a four-byte binary value in little-endian.

The `shift` command carry the actual array of bits that will be placed in to the logic devices over the JTAG link following the format

```
shift:<number of bits><TMS vector><TDI vector>
```

where `<number of bits>` is a four-byte little-endian binary value and defines the number of bits that need to be shifted on TMS and TDI signals limited by the maximum length defined on the `getinfo` command, although `<TMS vector>` and `<TDI vector>` are byte driven. `<number of bits>` is also used for the return of the sampled TDO bits:

```
<TDO vector>
```

and it is not needed in the answer.

With all the gathered information about the AdvancedTCA specification, the Pulsar IIb project and the IPMI and XVC protocols, the implementation of a safe and remote work environment is possible and will be presented in the next chapter.

3 MATERIALS AND METHODS

As this development is related to the Pulsar IIb design, many of the choices were previously made by the Fermilab team, and the reasons behind that, are out of the scope of this project, such as the conception of the IPMC and blade boards or the proprietaries RTOS and network solution used. However, in this section, not only the available resources will be presented and how to use them will be discussed, but the tools and procedures involved will be described.

3.1 THE PULSAR IIB PROJECT

Instrumentation on the Pulsar IIb blade has more than 27 sensor channels, which include temperature, voltage, and regulator output current. In the Table 5, information about those sensors and their planned threshold are found, which are informed to ShM and used for monitoring.

As shown at Figure 28, the Pulsar IIb IPMC has a microcontroller and several interfaces that makes possible support for some remote control functions. Those interfaces are described bellow:

I^2C Interfaces: Pulsar IIb IPMC measures and controls the cooling, power sourcing and hot-swapping of the board system using IPMI protocol over IPMB, which in turn uses the I^2C to access the physical lines. For that, three I^2C peripherals are provided by the microcontroller, that is, I^2C-0, I^2C-1 and I^2C-2.

Ethernet Connection: This interface is connected with Zone 1 of AdvancedTCA Shelf. It is possible the remote access through an AdvancedTCA Switch, like Telnet using a TCP/IP connections from a workstation. It allows for instance, to acquire data from voltage and temperature sensors remotely, besides mezzanines power measure. The IPMC IP address is hard-coded and depend directly on which physical slot the Pulsar IIb is connected in the AdvancedTCA shelf.

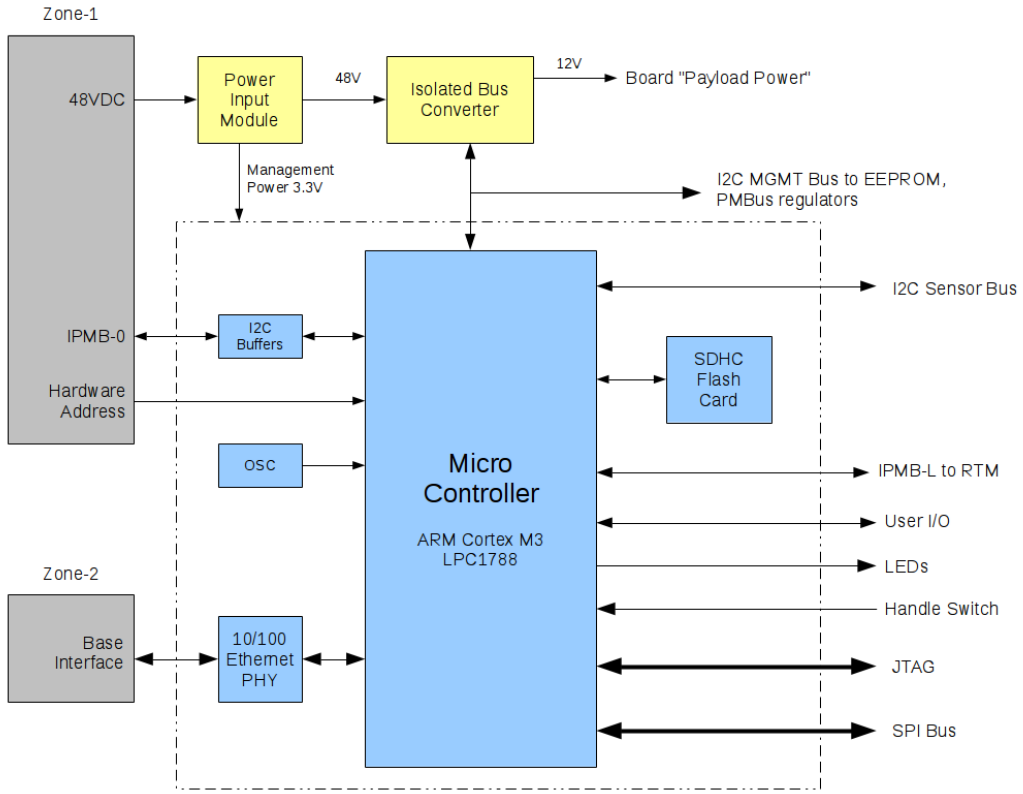
Master JTAG Interface: Interface that can program and debug the FPGA. This communication is designed to be bit-banged through regular I/O pins of the microcontroller.

Table 5: Sensor devices placed in the Pulsar IIb carrier board with related thresholds used for description in the the SDR.

Device	**Param**	**Units**	**Low NR**	**Low C**	**Low NC**	**High NC**	**High C**	**High NR**
EBDW025A0B41Z	Temp	°C	—	—	—	50	75	85
12V bus converter	Vin	V	36	40	44	56	64	72
	Vout	V	—	—	11.75	12.25	—	—
	Iout	A	—	—	—	20	22	25
UDT020A0X3-SRZ	Vin	V	—	—	3.00	14.40	—	—
Voltage regulator	Vout	V	—	—	1.10	1.30	—	—
MGTAVTT	Iout	A	—	—	—	15	17	20
UDT020A0X3-SRZ	Vin	V	—	—	3.00	14.40	—	—
voltage regulator	Vout	V	—	—	3.20	3.40	—	—
VCC3V3	Iout	A	—	—	—	15	17	20
MDT040A0X3-SRPHZ	Vin	V	—	—	4.50	14.40	—	—
Voltage regulator	Vout	V	—	—	0.90	1.10	—	—
VCC1V0	Iout	A	—	—	—	30	35	40
PDT006A0X3-SRZ	Vin	V	—	—	3.00	14.40	—	—
Voltage regulator	Vout	V	—	—	1.70	1.90	—	—
VCC1V8	Iout	A	—	—	—	4	5	6
MDT040A0X3-SRPHZ	Vin	V	—	—	4.50	14.40	—	—
Voltage regulator	Vout	V	—	—	0.90	1.10	—	—
MGTAVCC	Iout	A	—	—	—	30	35	40
PIM400KZ	Temp	°C	—	—	—	50	75	85
Power Input Module	HoldUP	V	—	—	50	95	—	—
	Iout	A	—	—	—	4	5	10
	Vin_A	V	36	40	44	56	64	72
	Vin_B	V	36	40	44	56	64	72
LTC2990	Air Temp	°C	—	—	—	35	45	55
Temperature Sensor	FPGA temp	°C	—	—	—	50	75	85
	3.3V	V	—	—	3.20	3.40	—	—

Source: Provided by Fermilab

Figure 28: Architecture of the IPMC designed by Fermilab for the pulsar carrier board. For this project, the most relevant interfaces to the internal components are JTAG and I^2C; and to the external devices are the Base Interface and IPMB-0.



Source: Provided by Fermilab.

According to the AdvancedTCA standard (PICMG, 2008), the front board shall have an IPMC with support the IPMI protocol over IPMB, which are organized in the following was in the Pulsar IIb project:

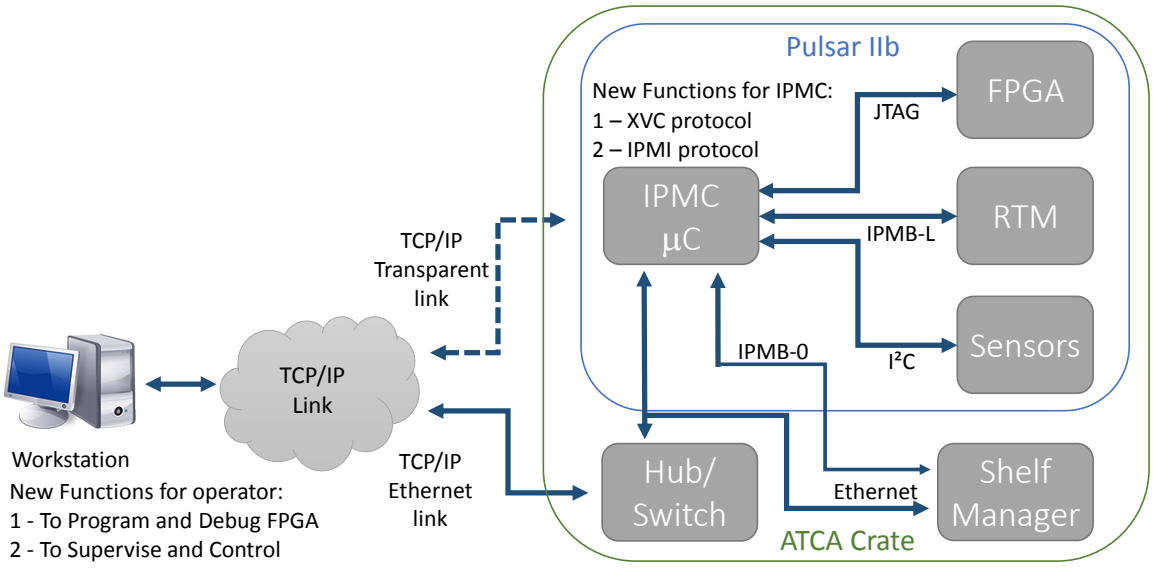
IPMB-0 with redundant connection to ShM, using I^2C-0 channel for IPMB-A and I^2C-1 for IPMB-B;

IPMB-L tied to I^2C-2 for connection with the MMC located in the RTM;

As the AdvancedTCA standard requires that only intelligent devices should access the IPMB-0 and the IPMB-L channels (PICMG, 2008), non-intelligent elements such sensors and memory are placed in a specific bus that uses regular I/O pins for communication over I^2C.

The AdvancedTCA shelf used by this project has its architecture summarized in the Figure 29. The workstation connect to an AdvancedTCA hub/switch device over regular network resources, which provides communication to the other elements in the shelf, such as the front boards and the shelf manager.

Figure 29: AdvancedTCA shelf architecture available for the Pulsar IIb project at Fermilab. A switch exposes the IPMC and the ShM to the external world through internet link. Communication between IPMC and ShM by IPMB-0. Related to the inner components, connections to FPGA, RTM and sensors are established by JTAG, IPMB-L and I^2C, respectively.



Source: Elaborated by the author.

3.2 RELEVANT TOOLS

This project was only possible with the use of many tools in the different stages of the development. This section describes the most relevant of them.

3.2.1 Programming Languages

All the implementation of this project was made using C language, which is a widely used programming language for embedded projects because of its capability to access the low level functions of the system with little overhead, offering yet a structured programming context under an abstraction layer over the machine basic instructions set.

On the other end, Python is an interpreted programming language designed to be highly extensible and thus offers many capabilities provided by external modules, such as TCP/IP sockets and binary data processing. With these scripts, it also makes possible the creation of automated procedures, allowing comprehensive verification. Python interpreters are available for all the most popular computer operating systems and, therefore, this language comes across as a very appropriate tool for development environments, such as the one found in this project.

3.2.2 Keil μVision

μVision is a proprietary Integrated Development Environment (IDE) provided by Keil required to work with their solutions, RTX and TCPnet, mentioned in the sections 1.4.3.1 and 1.4.3.2. It is a single environment for test, verification and optimization of the application code that has some facilities to work with embedded projects and provides resources such as source files management, run-time environment, debugging, configuration wizards, C/C++ code editing and syntax checking, templates, pre-built software packages, and documentation access.

3.2.3 ShM tools

Once the operator is logged in the ShM, the system provides some ancillary tools to deal with the devices connected to the IPMB channels. The `clia` program, standing for Command Line Interface Agent, allows the direct management and monitoring of the shelf resources through IPMI transactions, which are wrapped in a more user friendly approach

(PIGEON POINT, 2015). Some of the commands are closely related to this project and are listed below:

board: shows information about boards.

deactivate: deactivate a specific FRU.

fru: gathers FRU Information.

frudata: provides raw access to the FRU Information.

fruinfo: translates the FRU Information into a user friendly output.

getipmbstate: shows the current state of the IPMB-0.

ipmc: shows information about IPMCs.

sensor: shows the description of one or a group of sensors.

sensordata: shows the current reading of one or a group of sensors.

sel: presents a history log of registered events.

threshold: presents the threshold for sensors available in the blade.

The `ipmb_traced` is another useful software provided in the ShM, which is able to log the communication to the devices attached in the IPMB-0 channel. This tool allows later analysis of the exchanged packets.

3.2.4 The Xilinx tools

Xilinx Vivado Design Suite IDE for syntesis and analysis of Hardware Description Language (HDL) designs, including a in-built logic simulator. This suite is divided in two layers:

- the user interface with several development resources such as project management, configuration assistance and code editing.
- the Hardware Server which is in charge to provide access to Xilinx reconfigurable components, usually over USB or XVC.

Worth mentioning that Hardware Server has a standalone installation and can be used in a dedicated computer as a gateway only to the hardware in a remote stand.

3.2.5 Wireshark

Wireshark is a free and open source packet analyzer most commonly used to sniff network communications, providing information about what is inside of an exchanged packet. It is of straight application in the XVC context, where the widespread internet protocol suite is used.

However, data generated by the `ipmb_traced` software, described in the previous section, is not valuable unless if processed and Wireshark appears again as a suitable tool for that. Less known is the very basic, nevertheless helpful, support for IPMI transactions. Although Wireshark is not able to describe every single field of the several IPMI commands, it is still able to provide some interesting information like destination, length, time and raw data in hexadecimal format for each packet. More than that, this tool is aware of some of the structural aspects of the protocol such as IPMB-0 being composed by IPMB-A and IPMB-B.

Wireshark is also able to sort and filter the packets, which is very useful to create small sets of data, which is very useful for troubleshooting, protocol development and communication assessment in both cases.

3.3 DEVELOPMENT PROCEDURES

Although this project is divided in two stages, the IPMI implementation and the XVC implementation, all the basic development procedures are shared. Both were programmed using the C language and rely on the basic debug techniques provided by the μVision IDE, such as breakpoints, variables watching and in-circuit debugging – going step by step through the code running in the target. Another common point is that both were suppose to work flawlessly with vendors' tools, which impose the strategy of monitoring the communication lines while the communication is happening.

However the protocols have very different conceptions. In the IPMI standard there is one or more commands for each feature needed while in the XVC approach, all the JTAG commands are forwarded to the FPGA in a transparent way. In the next sections the implementation strategy adopted in each case will be explained.

3.3.1 IPMI methodology

With the support of the `ipmb_traced` tool, communications between the ShM and AdvancedTCA devices found in the market were used as reference for procedures like hot-swap and monitoring. This was important not only to guide the implementation process but as a reference for the communication started by the IPMC and also to better understand the relationship between the exchanged messages and the IPMI specification.

After the realization of the low level I^2C communication using microcontroller peripherals, messages from the ShM could be received and analyzed directly in the IPMC memory using debug tools. For each new identified request, related studies were made in the IPMI specification to contextualize and build a suitable response.

Those approaches were repeated until all the monitoring requests and the hot-swap mechanism were fully supported, allowing the ShM to manage the crate power and cooling resources using information gathered from the Pulsar IIb IPMC.

For the hot-swap procedure in the context of the Pulsar IIb project, only two electronic boards are able to perform such process: the carrier board and the RTM, however just the first is focused in this project.

When a board is inserted in the crate, the management power is first enabled to control the devices with limited functionalities (PEREK; MAKOWSKI, 2011). After that, the ShM initiates the activation process that negotiates the supply of power, sets the connections and reads information about the module. These steps are necessary for turning the new device operational.

Conversely, when a board is extracted from the crate, the revers process is performed. The IPMC uses the FRU state machine to trigger control messages to the ShM. In this way, the activation and deactivation of each Pulsar IIb board present in the crate will just be validated when the ShM ensures it is safe to do so, using context information such as payload power and temperature.

This design is taking into account the SDR, the FRU data formatting, the IPMI frame format and commands provided in the Pulsar IIb IPMC. Taking into account which FRU state the IPMC is in a given moment, ShM has a set of information that must be transferred through IPMI transactions. In this way, the hot-swap operation of eachPulsar IIb board present in the crate will just be validated when the ShM ensures it is safe to do so, using context information such as payload power and temperature of the components.

3.3.2 XVC methodology

As discussed in the section 2.3, the XVC protocol requires just three commands. One of them behaves like a wrapper for the JTAG messages while the others are just used to establish the connection between the XVC service and the Hardware Server. Unlike the IPMI aproach described in the previous section, there was no communication reference for the XVC to base the development of the service.

The first step was the implementation of a very basic link between the computer and the IPMC using a standalone test stand and a raw TCP socket provided by the TCPnet library, which was tested using a telnet connection. Once the link was built, one XVC client emulator was implemented using Python to provide the three commands defined by the XVC specification in a very controlled way. With this, the commands `getinfo` and `settck` were implemented in a straight forward way because they didn't depend on Xilinx devices.

The `shift` command implementation was performed in smaller steps. First, it was assured that the fields were interpreted according to the specification, comparing the sent and the returned data in the emulator. After that, the information was routed outside of the microcontroller using the JTAG pins where the input and output data signals were short-circuited and again compared in the emulator.

Once the basic functionality of the XVC service were ready, tests with the actual devices took place, plugging the IPMC card to the Pulsar IIb blade instead of in the standalone test stand and connecting it to the Vivado software. Sniffing the connection with Wireshark showed that despite the fact that the two connection related commands were working fine, the same `shift` command was being repeated and the conclusion was that the answer was wrong. Further studies shown that the problem happened because that the data from the FPGA was clocked one cycle earlier than expected.

Once this point was solved, the JTAG chain was recognized and further improvements were done in the code and in the TCP socket configuration. With a solid communication, multiple boards were moved to an AdvancedTCA shelf behind a gateway computer and final tests involving the Xilinx Hardware Server were performed.

4 IMPLEMENTATION

This chapter discusses the implementation of both services made with C language for microcontrolers. Despite the fact that only a reduced part of the IPMI specification was designed, it still have a considerable amount of code lines and only the most relevant features will be described. The XVC implementation is much simpler and a complete overview of the code is presented.

4.1 IPMI IMPLEMENTATION

This section describes the implementation of the communication between ShM and Pulsar IIb IPMC and some of the features provided in this project, such as I^2C driver, message handler, hot-swap state machine, data parsers and commands.

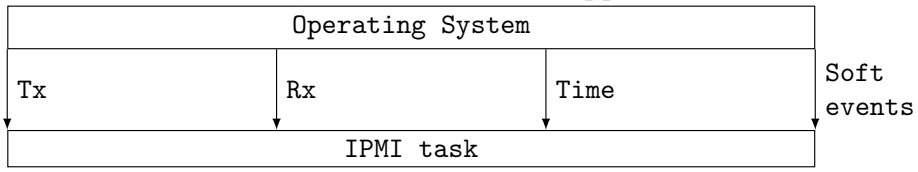
4.1.1 IPMI task

One of the activities performed by the OS is the task that is related to the IPMI resources. This task was designed as event driven, that is, the process is only triggered if there is a reason, meaning that it will be inactive most part of the time. According to Figure 30, events capable to trigger the IPMI task are:

- received messages (any I^2C channel);
- messages to send (any I^2C channel);
- timed events;
- soft events.

By design, each call from the OS to the IPMI task will solve only one event and after that the system will be free to work on other tasks. If more events are waiting, they will be attended in the next round from the OS.

Figure 30: IPMI task is only executed when: IPMB packets are received; IPMB packets need to be sent; periodic processes should run; IPMI soft events happened.



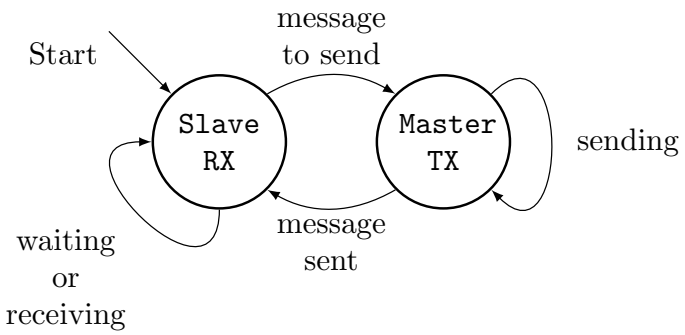
Source: Elaborated by the author.

4.1.2 I²C Driver

During an IPMI communication, all devices of a I²C bus must be able to master the transmission lines to send messages. In the driver implementation, it is important to keep in mind that each bus controller must operate in a multi-master state machine. Also, as required by the AdvancedTCA standard, the IPMB interfaces of the Pulsar IIb carrier board are dedicated, that is, there are only intelligent devices connected to these buses, thus there is no need for a master condition for receiving.

For this implementation, each I²C channel in the IPMC has a state machine as illustrated in Figure 31. The device will hold position in the slave state where it is able to receive messages, only moving to the master state when data must be sent and returning to the passive mode as soon the outgoing transmission finish.

Figure 31: State machine for multi-master mode transfers in an I²C channel. Driver only leaves the slave condition to send messages and return to it as soon as the transmission is finished.



Source: Elaborated by the author.

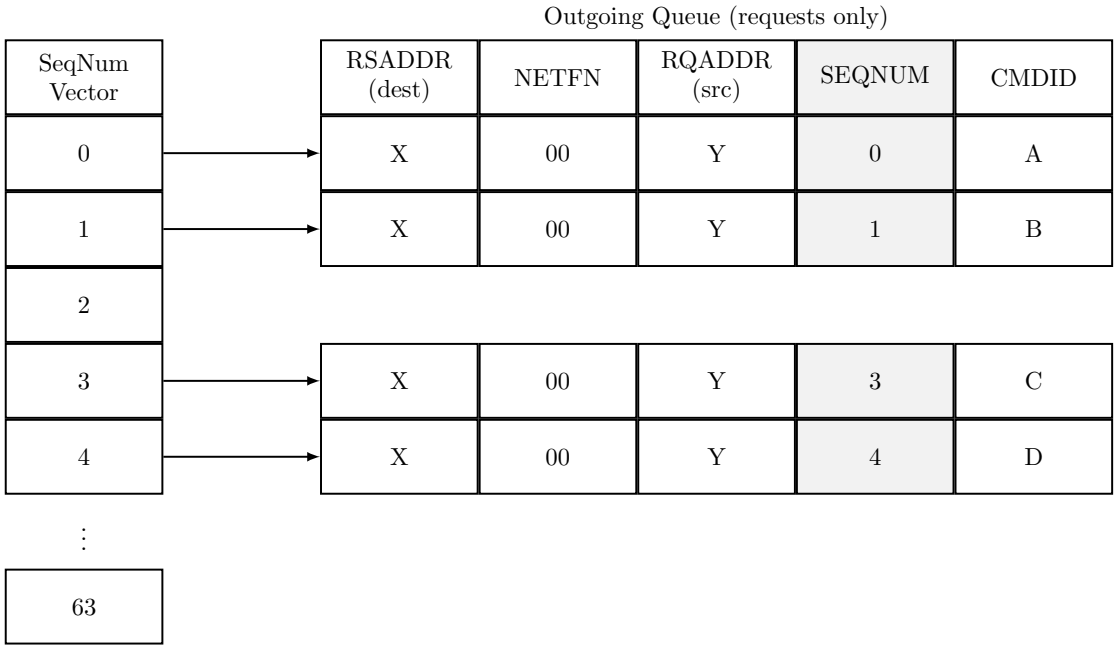
It is worth to emphasize that the communication of those I²C channels is made by peripherals, reducing the load of the microcontroller. In reception, the processor is only activated after the packet is fully assembled. For transmission, the processor is in charge of assembling the packets but does not need to take care of the transfer itself.

4.1.3 Send event processing

A send event is raised for any channel when a packet is assembled and ready in the related outgoing buffer. In the simplest case, it is a response and it is sent using the same channel that the request arrived. The response packet is discarded just after that.

For requests, however, the packet is held after sent for eventual retransmission. This process is controlled by time ticks and number of tries. There is a sequence number array tied to IPMB-0 and other to IPMB-L, that keep a pointer to each corresponding outgoing packet, which the sequence number is the same of the index where the pointer is stored, as shown in Figure 32. It must be clarified that there is only one sequence vector for channel, which means that there is no conflict between different buses of the same channel.

Figure 32: Each message in the outgoing queue is associated to a position in the vector of pointers, given by the SEQNUM value.



Source: Elaborated by the author.

It is worth pointing out that in the current implementation only one packet is covered for a given send event. If there are remaining time outed messages to process a new send event must be raised.

In addition, there is an exception for the expiration of the packets in the outgoing buffer during the initialization stage. It is assumed that the whole system was just powered on and that ShM is also in same process, resulting in its incapacity to answer. So, the Platform Event from `M0` to `M1` will be retransmitted until any response from ShM arrives.

4.1.4 Soft event processing

Soft events results from external stimuli such as hardware or state changes. Each one has its specific procedure to deal with the requisition bringing flexibility to the implementation.

There is another queue for the soft events which holds the event numbers or identifiers. The read and write indexes for this buffer are independent and work in a circular approach. New events when the buffer is full overwrite the oldest ones. Again, only on event is processed by turn. A new event is raised when the queue is not empty.

Soft events may result in a message to other devices. In this case, the packet is assembled with a sequence number allocated according to a free position in the related buffer. It is stored in the outgoing queue with flag as ready to send and the transmission event is raised.

4.1.5 Receive event processing

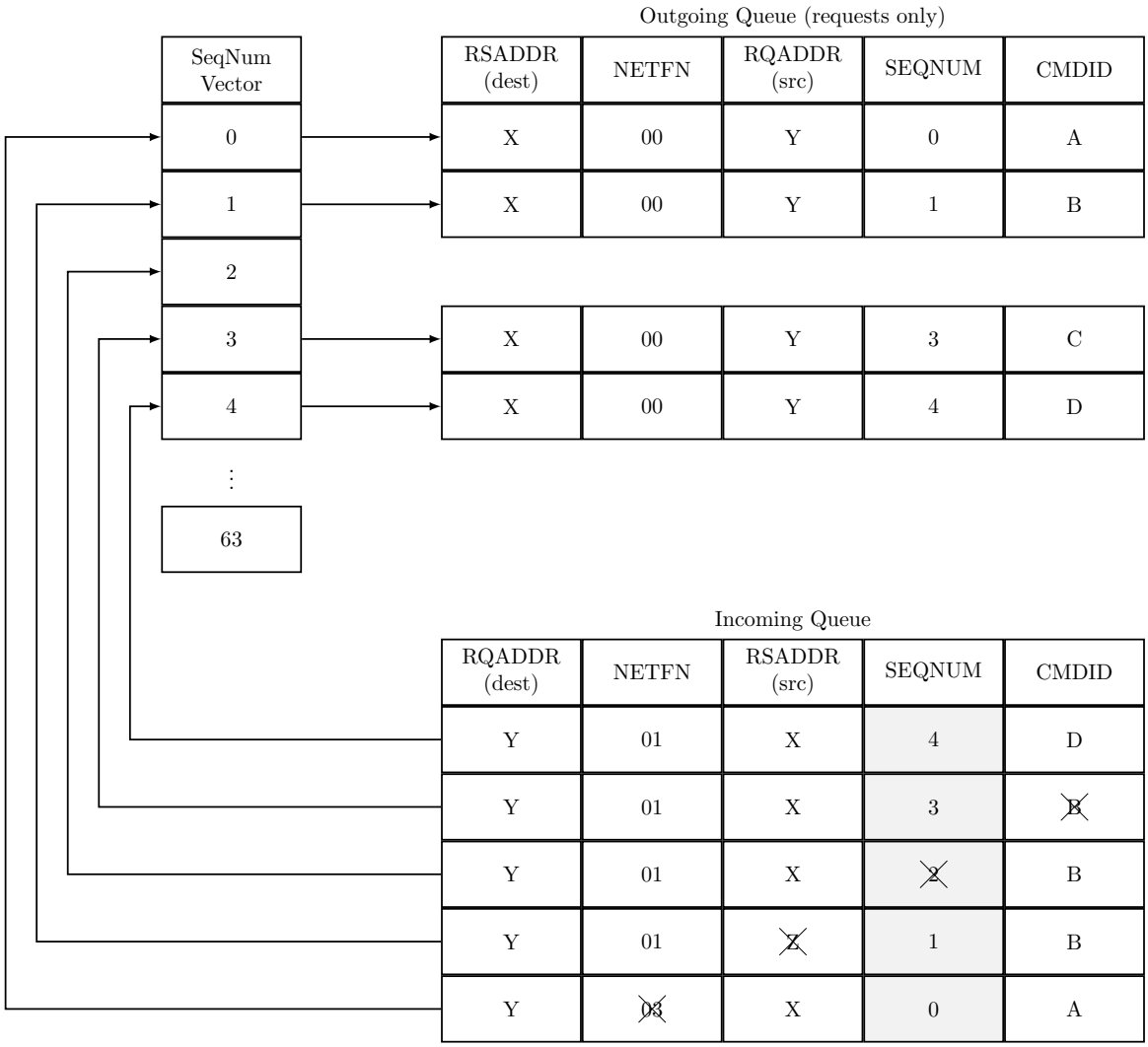
Each reception channel has its incoming queue to avoid conflicts. The queue is implemented in a circular approach, with independent read and write indexes. The processing sequence is kept according to the age of a packet given by ticks, with older transactions being processed before.

In a overall view, when a message is received and validated, the next stage is to perform the related actions. In the first moment, the process path is redirected according to the NETFN to inner functions of the identified group, gaining access to the internal commands. Then, the CMDID is retrieved and the corresponding function is called.

Even NETFN means a incoming request and a response must be sent. On the other hand, if NETFN is odd, the incoming packet is a response and the related request must be eliminated from the outgoing queue because retransmission is not needed anymore, as shown in Figure 33. Thus, as soon a response arrives and the fields are parsed, the value of the sequence number is used as index to retrieve the packet and check if it is the related request.

At this point, it must be highlighted that the sequence pointer vector provides a very efficient retrieving process. Instead of going thru each packet of each outgoing queue of the channel, the system is taken directly to the unique packet with the same sequence number.

Figure 33: Matching between request and response is oriented by the SEQNUM and depend on (physical and logical) addresses, NETFN, and CMDID fields.



Source: Elaborated by the author.

4.1.6 Periodic events processing

The periodic events check for the system status, for example handle switch, LED status, count down lockers. It will update also the ticks registers for each resource that depends on that, for instance, transmission packets waiting to be sent and queues. Last, it will act as a garbage collector checking if there are any untracked events.

4.1.7 FRU State Machine

Each edge of the state machine have its trigger by soft event, section 4.1.4, and a related process function, where the context verification is checked and the requests are assembled to the ShM. There is a function for all state changes in the IPMC FRU state machine, where LED behaviors and power management are configured according to each specific situation.

Important to add that there are two kinds of lock in this structure. First happens when a state change is dependent on further conditions, such as an answer from ShM. Second is related to time, mainly implemented to accomplish with the shelf operator physical interface.

4.1.8 FRU Data information

The Appendix A brings an example of how FRU information was implemented for the Pulsar IIb project. It is divided in three different pieces:

Common Header: All the info area are absent and the related offsets were set to zero but the Board Area Info set as `01h` and the MultiRecord Info Area set to `07h`, what indicates each start in the 8^{th} and 56^{th} bytes respectively.

Board Info Area: it was filled with general information about the Pulsar IIb project and Fermilab.

MultiRecord Info Area: The MultiRecord Area Record Type Identifier (ID) `C0h` indicates a format defined by the AdvancedTCA specification (PICMG, 2008). The Pulsar IIb IPMC code implements just the board point to point connectivity information with PICMG record ID of `14h`. This record informs which connections the AdvancedTCA carrier board has available.

Therefore, relating this process to Pulsar IIb IPMC, the FRU information about the E-Keying has to show the supported channels:

Base Interface 10/100/1000 Mbps: Pulsar IIb supports the Base Interface Channel 1 connected to the PICMG 3.0 10/100/1000 BASE-T Interface, Port 1. This Ethernet interface supports link speeds of 10, 100, and 1000 Mbps.

Fabric Interface 40 Gbps: Pulsar IIb supports the Fabric Interface Channel 1 connected to the PICMG 3.1 Ethernet Interface in ports 0/1/2/3 of 10 Gbps each.

Fabric Interface 20 Gbps: Pulsar IIb supports the Fabric Interface Channels 2 up to 13 connected to non-Ethernet PICMG 3.0 traffic in ports 0/1 of 10 Gbps each.

Other possible channels and ports are not available in Pulsar IIb, such the Update Channel Interface as example.

4.1.9 Sensor Data Record

Each sensor available in this version is described as Full Sensor Record by an SDR Record ID and sensor number, as the example in the Appendix B. All SDR fields are detailed at Intel et al. (2013a) and Perek and Makowski (2011). Between those SDR fields, variables help ShM to convert the raw data of sensor reading in information compatible and with reasonable resolution (INTEL et al., 2013a; PEREK; MAKOWSKI, 2011). These variables are presented in the equation:

$$y = f(x)$$

where

$\boldsymbol{y}$ is the translated value of the sensor reading;

$\boldsymbol{f(x)}$ is the linearization function that can be chosen from linear, ln, log10, log2, e, exp10, exp2, 1/x, sqr(x), cube(x), sqrt(x), and $cube^{-1}$(x).

Once the linear function was chosen, the previous equation becomes:

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

where

$\boldsymbol{y}$ is the translated value of the sensor reading;

$\boldsymbol{x}$ is an 8-bits variable for the raw data from "Get Sensor Reading";

$\boldsymbol{M}$ is a 10-bits variable for the angular coefficient;

$\boldsymbol{B}$ is a 10-bits variable for the linear coefficient, or offset;

$\boldsymbol{Bexp}$ is a 4-bits variable for the order of magnitude of B;

$\boldsymbol{Rexp}$ is a 4-bits variable for the order of magnitude of the result.

The exchange messages related to SDR begins at `M2` FRU state. However, as the process can involve more than 100 requests and responses between ShM and IPMC, the SDR exchange messages continues regardless of the FRU state changes. The sensor readings are only performed when ShM has all SDR downloaded to it.

The Pulsar IIb IPMC implements the SDR information for all 27 sensors described at Table 5 in the section 3.1, including thresholds.

4.1.10 IPMI commands

In addition to the IPMI requests previously mentioned in section 2.2, other ancillary requests, listed below, were implemented or because they were useful during development process or because ShM might send them during moments that are not relevant for this project.

NETFN `2Ch/2Dh` CMDID `0Dh` (Get Device Locator Record ID) accelerates the access to the individual FRU records. If the ShM imports all SDR, the use of this command is not necessary.

NETFN `2Ch/2Dh` CMDID `10h` (Compute Power Properties) informs to the IPMC that it should lock in its desired power and cooling levels. Response includes information such as the number of Slots that the FRU spans, if the FRU is a Board and where the IPM Controller is located.

NETFN `04h/05h` CMDID `20h` (Get Device SDR Info) returns general information about the collection of sensors in a Dynamic Sensor Device, such as the IPMC.

NETFN `04h/05h` CMDID `22h` (Reserve Device SDR Repository) is used to obtain a Reservation ID, which is part a mechanism that is used to notify the Requester that a record may have changed during the process of a multi-part read.

NETFN `04h/05h` CMDID `27h` (Get Sensor Threshold) retrieves the threshold for the given sensor.

NETFN `0Ah/0Bh` CMDID `10h` (Get FRU Inventory Area Info) returns overall the size of the FRU Inventory Area in the IPMC device.

Essentially, a list with all the covered requests by this implementation is found below, separated by Network Function.

- Application
  - Get Device ID
- Storage
  - Get FRU Inventory Area Info
  - Read FRU Data
- Sensors and Events
  - Set Event Receiver
  - Get Device SDR Info
  - Get Device SDR
  - Reserve Device SDR Repository
  - Get Sensor Threshold
  - Get Sensor Reading
- PICMG
  - Get PICMG Properties
  - Set FRU Activation
  - Get Device Locator Record ID
  - Set Port State
  - Compute Power Properties
  - Set Power Level
  - Get Power Level

If any a Pulsar IIb IPMC receive any request that is not supported by this implementation, there is a default answer with completion code `C1h` indicating that.

The Pulsar IIb IPMC is able to create only one kind of request:

- Platform Event

which is created based on soft events, as explained in section 4.1.4, and is useful to inform the ShM about state changes in the FRU state machine. As a consequence, this implementation is capable of interpreting its response, which is needed for continuation by the hot-swap process.

4.2 XVC SERVICE IMPLEMENTATION

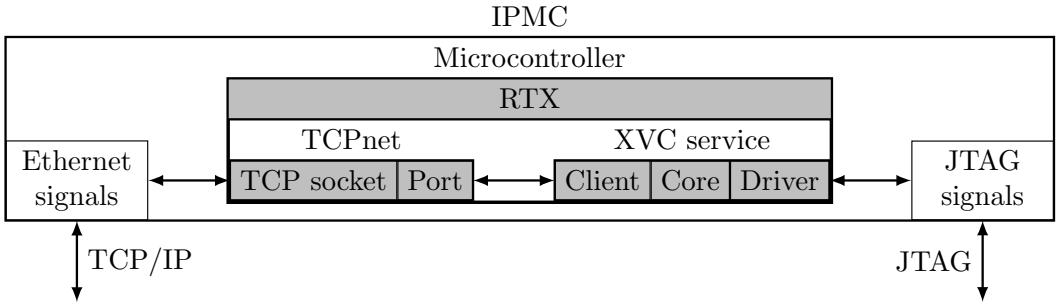
The XVC server is implemented as a TCP service and it is built based in three concepts, according to Figure 34:

XVC Client Interface: is the link between XVC and TCP/IP stack.

XVC Core: is the actual implementation of the XVC protocol. It processes the received commands and assemble the result to the XVC client.

JTAG Driver: deal with the input and output signals of the JTAG capable device side.

Figure 34: XVC service is designed in three layers: the client part establish the TCP/IP connection using resources provided by the TCPnet library, the core part is in charge of the XVC protocol and the driver part deal with the JTAG signals.



Source: Elaborated by the author.

4.2.1 XVC Client interface

The XVC Client Interface is in charge to deal with creation, binding, listening and closing of the TCP socket, making it possible transfer data using the TCPnet module to

the XVC client. To create the socket, it is needed to use the function

```
U8 tcp_get_socket (
    U8   type,          /* Type of TCP socket. */
    U8   tos,           /* Type Of Service. */
    U16  tout,          /* Idle timeout period before disconnecting. */
    U16 (*listener)(    /* Function to call when a TCP event occurs. */
        U8  socket,     /* Socket handle of the local machine. */
        U8  event,      /* TCP event such as connect, or close. */
        U8* ptr,        /* Pointer to IP address of remote machine, */
                        /*   or to buffer containing received data. */
        U16 par         /* Port number of remote machine, or length */
    )                   /* of received data. */
);
```

that will return a `handler id`, which is a integer from 0 to 255. This function is been called in the XVC initialization (`xvc_tcp_init` function). Its first parameter is the required type of the socket and for Vivado the modifiers are

```
TCP_TYPE_SERVER | TCP_TYPE_DELAY_ACK | TCP_TYPE_FLOW_CTRL
```

where:

`TCP_TYPE_SERVER`: means that the socket is open in passive mode and waits for incoming connections;

`TCP_TYPE_DELAY_ACK`: allows greater acknowledgment timeout on sequential segments;

`TCP_TYPE_FLOW_CTRL`: allows the TCP socket to control TCP Data Flow.

The other relevant parameter is the last one, which give to socket a callback function where to process the incoming packets. In the XVC implementation case it is the `xvc_tcp_callback` function. This function is then tied to the `socket_id` handler and it is event driven, so only the events defined by the TCPnet module are allowed. Notice that the socket need to be started after its creation, what is made by the XVC code using the

```
tcp_listen (socket_id, XVC_TCP_PORT_NUM);
```

function, where `socket_id` was retrieved from `tcp_get_socket` function and `XVC_TCP_PORT_NUM` is a constant defined earlier as 2542 (default port for XVC, defined by Xilinx).

Also, the TCP connection is built on a sequence of events and a minimum set of them should be met. For example:

`TCP_EVT_CONREQ`: is a connection request and this case tells to XVC server the IP of the client;

`TCP_EVT_ACK`: that process the acknowledgment received (do nothing in XVC case); and

TCP_EVT_DATA: is a event for the data itself.

The last event above is the core of the communication and it is tied to the function

```
xvc_tcp_process_received_data(*buf)
```

to process the data, meaning the actual implementation for the XVC protocol, what will be described in the next section.

Worth pointing that each TCP service has its own data buffer and also that the TCPnet module does not touch the content (payload) of the TCP frame and deliver it as is.

The process of sending data back to client must stay out of the TCP task because, according to the user guide, TCPnet is not a re-entrant module. Therefore, to overcome this situation, the `xvc_tcp_send_data` function is called just after the main TCPnet function (`main_TcpNet`) in the TCP task. This function deals with the `xvc_output` structure

```
typedef struct xvc_output_n {
    U8 data[XVC_OUTPUT_BUFFER_LEN];
    U32 nbytes;
} xvc_output_t;
```

where the data to be sent and its size relies.

The number of bytes to be sent is checked in the output structure. If this value is greater then zero, the function

```
BOOL tcp_send (
    U8  socket,     /* TCP socket to send the data packet from. */
    U8* buf,        /* Pointer to buffer containing the data to send. */
    U16 dlen );     /* Number of bytes of data to send. */
```

of the TCPnet module is called. It just needs the socket handler, a pointer for the data and the number of bytes to be sent.

Worth to say that the special TCPnet module function

```
tcp_reset_window()
```

must be executed at the end of the transaction to reset the available TCP window length, related to the `TCP_TYPE_FLOW_CTRL` socket modifier. One last point is that the connection remains open until the client closes it, as the socket is in passive mode.

4.2.2 XVC Core

When data arrives, there is a `TCP_EVT_DATA` event on the socket, and

```
xvc_tcp_process_received_data(*buf)
```

is called. There, the function

```
xcv_fill_cmd_structure(*buf)
```

is acted on and a basic command structure

```
typedef struct xvc_cmd_n {
    char * name;
    U8 * body;
} xvc_cmd_t;
```

is filled.

The first vector pointed by `* buf` contains a entire XVC command, because the first command in a XVC communication is a request about the capabilities (`getinfo`) of the service and the answer tell the client the amount of data that is allowed to send by frame, same as buffer size. Notice that one `shift` command is not directly related to one JTAG transaction, that is, the end of the `shift` command does not mean that the client process is over, it just got sliced in smaller pieces. The configuration of a device with a several megabytes firmware to the device is an obvious example of this procedure, where repeated `shift` commands are used to transfer the complete data.

According to the identified command, different routines are executed. For instance, the `getinfo` and the `settck` commands have very limited answers and there is not much information beyond what has been written. However, the `shift` command is a little bit more complex and deserves more clarification. Once this command is recognized, the `xvc_fill_shift_structure()` function is called and the structure

```
typedef struct xvc_shift_n {
    U32 nbits;
    U32 nbytes;
    U8 * tms;
    U8 * tdi;
} xvc_shift_t;
```

is filled. After that, each field become indexed, easily accessible by the JTAG driver, what is explained in the next section, 4.2.3.

However, it is adequate to point out that the obtained data from the JTAG capable device is returned by the JTAG driver in an already organized way within the XVC output structure, which will be used by the `xvc_tcp_send_data` function.

4.2.3 XVC JTAG Driver

The JTAG driver is called when all the data from the client is placed in the `shift` structure and the XVC output structure is prepared to receive TDO directly. For each

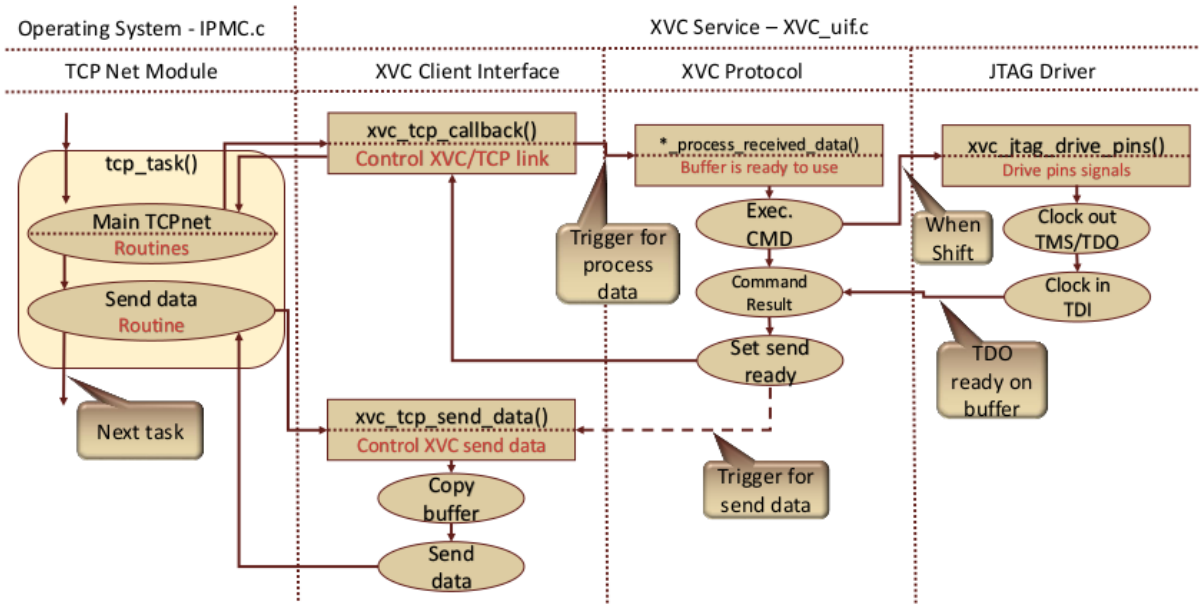
pair of TDI and TMS bits, one bit of input data is made available to the related pin. When TCK is made high level, TDO signal is sampled and after that TCK is lowered again. Thus, in the same clock cycle, TDI and TMS are sent and TDO is received. With the end of the vectors, the driver returns the number of bytes received and that need to be sent back to the client.

One important point is that the plain bit-by-bit loop approach is avoided for performance matters. So instead, the loop happens through bytes, which ensure eight times less tests. Also, each bit is processed without loops, using plain masks. If the last byte is not fully used, than only that is processed in a bit-by-bit loop fashion. All of this was reached after some optimization rounds.

4.2.4 XVC service architecture overview

The designed architecture for the XVC service is illustrated in the Figure 35, which make obvious the arrangement of the concept areas in layers: client interface, core and driver. It shows also how the functions of different layers are interconnected. In the top level, there is the TCP task provided by the TCPnet library which waits for the TCP packages. When a package is addressed to the XVC port, the payload of the frame is passed to the XVC core where the command is identified. If the command involves communication with the FPGA, the JTAG driver is called. After the execution of the command, the response is stored in a buffer and the TCP task is notified. The outgoing packet then is assembled and sent back.

Figure 35: Overview of the XVC service implementation architecture, showing the functions of each layer, the relationship between them, and how they are integrated to the system.



Source: Elaborated by the author.

5 RESULTS AND DISCUSSION

The Fermilab IPMC has implemented in it the IPMI protocol to monitor, control and ensure the appropriate operation of the AdvancedTCA carrier boards and other crate components. The platform system takes care of the hardware health, and takes corrective actions when needed and it is able to perform the hot-swap procedure.

Using this resources, ShM is able to perform AdvancedTCA cooling control. Asking for the Pulsar IIb sensor readings, it can set the fan levels, according to temperature of the components. Moreover, if sensor reading surpass a non-recoverable thresholds, predefined by SDR info, the ShM deactivates the board automatically in order to ensure the hardware appropriate operation.

Once the health of the environment is assured, the XVC implementation allows the programing and debug procedures over TCP/IP links, connecting Vivado IDE in a local computer to a Hardware Server in a remote gateway computer which, in turn, will provide the access to the reconfigurable components inside the AdvancedTCA shelf.

5.1 IPMI RESULTS

In this section, the results obtained with the IPMI implementation will be described. For that, information collected by using `clia` commands and also the `ipmb_traced` logs are shown.

At this point is interesting to keep in mind for the next plots that the `ipmb_traced` tool was identified as not able to register all the messages exchanged in the IPMB-0 channel. A simple procedure was repeated to prove that, involving:

1. insertion of any blade in the crate;
2. triggering the activation and waiting it to complete;
3. triggering the deactivation and waiting it to complete;
4. removing the board from the crate.

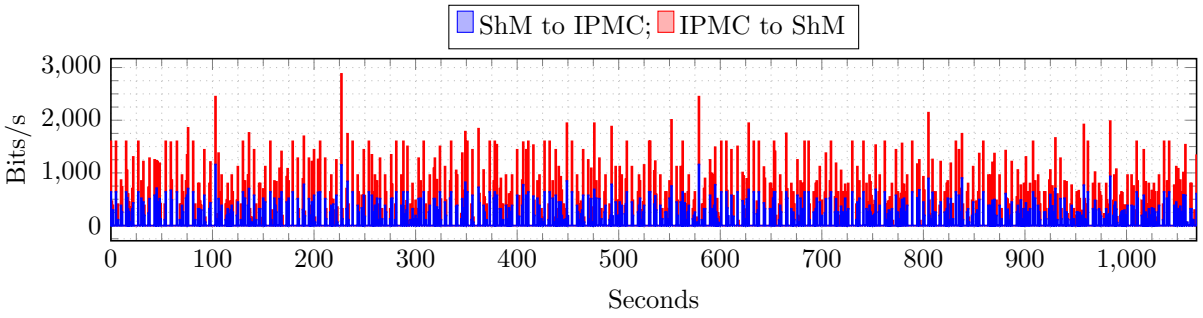
The processes finished without any error for all the tries, however not all the commands

needed for that were recorded and the missing messages were different for each trial. Thus, the appropriate way to look to the information taken with support of the `ipmb_traced` software is to become aware of what really is there, instead of looking for lacking transactions.

5.1.1 IPMI throughput

Despite the fact that the `ipmb_traced` software is not able to register all the exchanged packets, it is possible to plot the throughput os the IPMB channels to have a general idea about it, as shown in Figure 36. Even when compared to the nominal rate of approximately 88 kbps, that is, one byte for each 9 bits in a communication happening in 100 kHz, the value of 3 kbps is very distant from the limit. The direction of the communication is exposed in a stacked layout.

Figure 36: I^2C destination throughput of a communication between ShM and IPMC in both directions, presented in a stacked layout, using less than 5% of the nominal capacity of the channel.



Source: Author's measurement results.

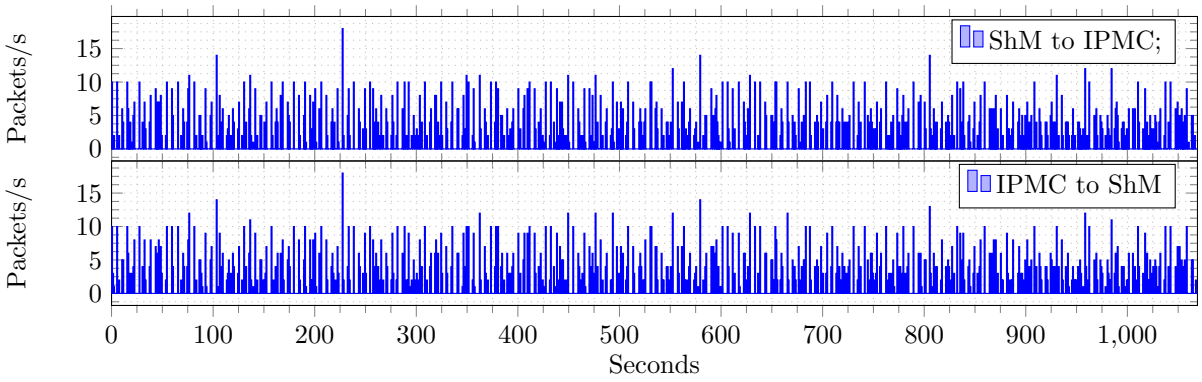
Another way of looking at the comunication is through the number of packets per second, as in Figure 37. It is easily seen that there is a well balanced distribution between packets from the ShM to the IPMC and in the reverse direction, as expected because there must be one answer for each request.

5.1.2 IPMI packet sending

The Figure 38 shows some requests generated by the soft events and tries to highlight also the round robin approach applied in the message dispatch, that is, they are sent using both available channels IPMB-A and IPMB-B, as instructed by the AdvancedTCA specification (PICMG, 2008).

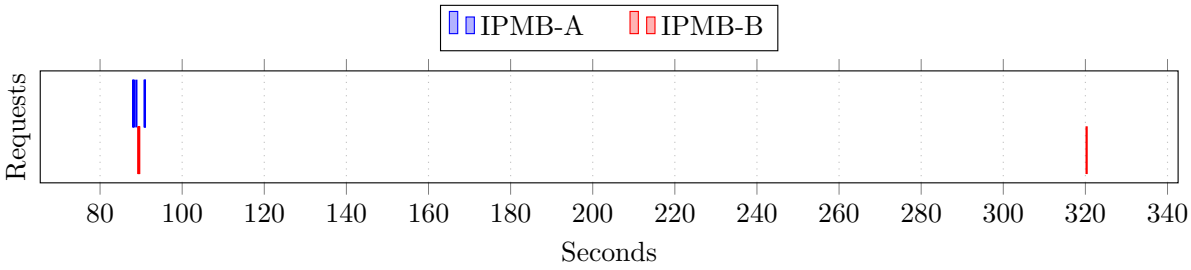
Other interesting feature to point out is the retransmission of a request in the case that

Figure 37: I²C destination throughput based on packet exchanged in both directions. Since a response is expected for each request, the balance between the plots indicate good behavior for the communication.



Source: Author's measurement results.

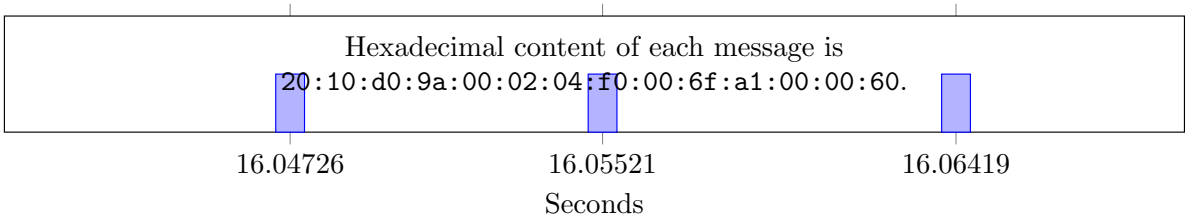
Figure 38: Requests generated by IPMC during a complete hot-swap cycle shows also the round robin approach in the use of the buses of the IPMB-0 channel.



Source: Author's measurement results.

the related response is missing (PICMG, 2008), as shown in Figure 39, where a message is preserved while a response is not received and it is re-sent after a while, in this case dependent of the task switching mechanism of the OS due to the soft events approach.

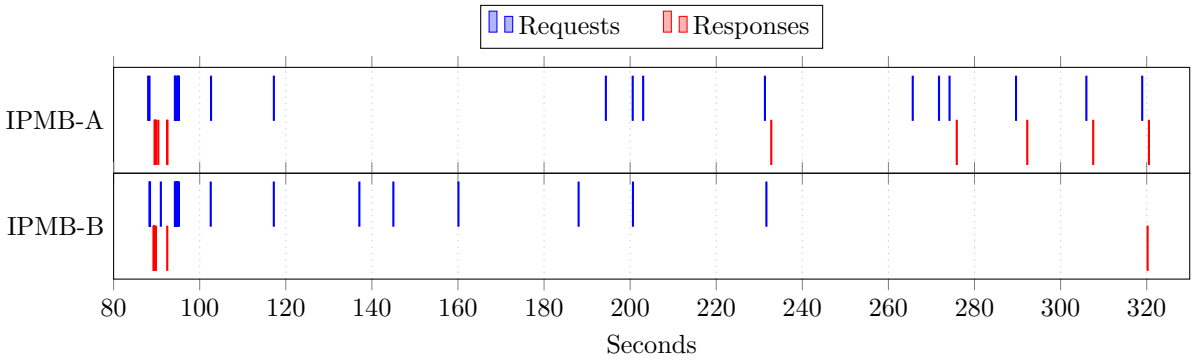
Figure 39: Retransmission mechanism working for time outed requests. Content of the messages are exactly the same.



Source: Author's measurement results.

Responses are sent using the same channel that the request arrived (PICMG, 2008), as shown in Figure 40. It is expected one response for each request unless of retransmissions, however the `ipmb_traced` tool loose many of them. The plot can still be used as basis for messages that were really exchanged and they shown that the responses alternate buses following the requests.

Figure 40: Responses being sent in the same channel they were received, as required by the IPMB specification.



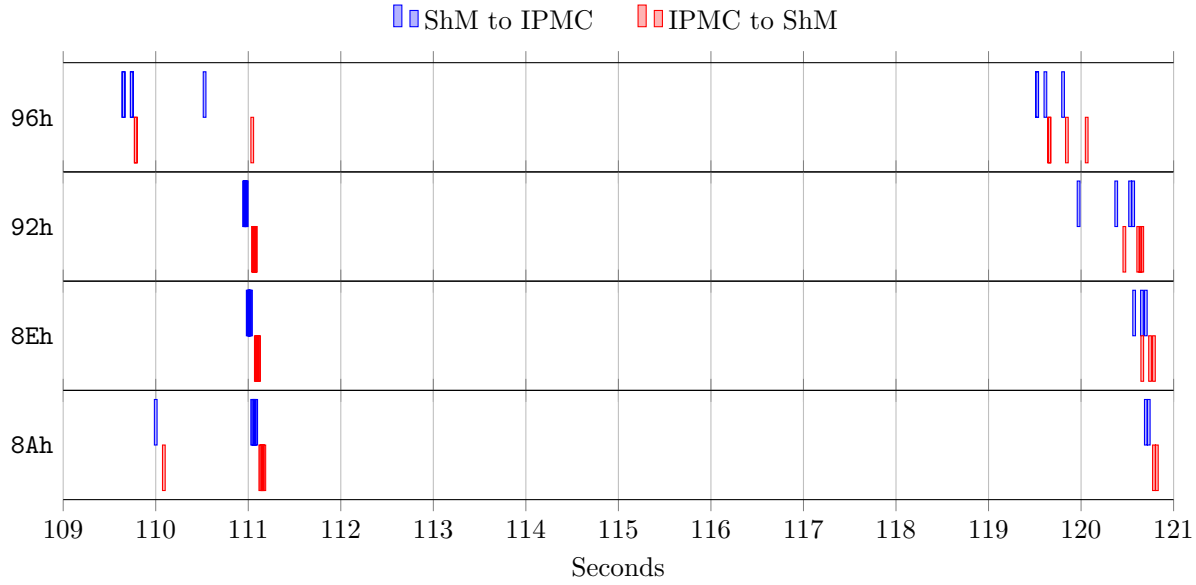
Source: Author's measurement results.

5.1.3 IPMI monitoring stage

Figure 41 shows messages exchanged between the ShM and different blades in an AdvancedTCA shelf during the monitoring stage, that is, after the board is activated and is fully operational. Multiple requests are needed to read the multiple sensors in the board and the procedure is repeated each few seconds.

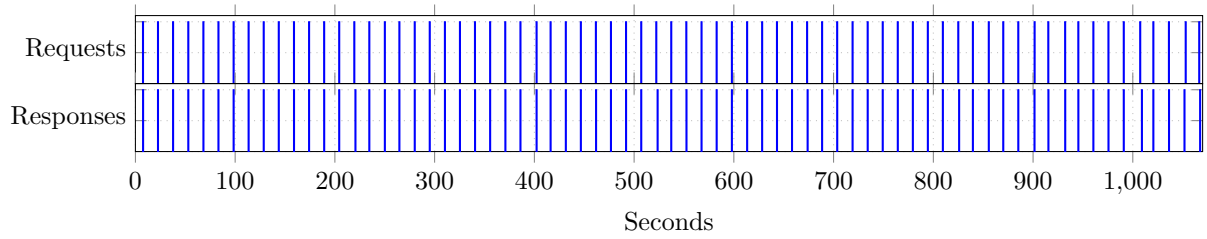
In the Figure 42, the Get Device ID commands used to ensure the presence of the IPMC are shown. Again, as a monitoring process, this procedure is periodically executed.

Figure 41: Periodic bursts of Get Sensor Reading messages exchanged during the monitoring stage of blade operation.



Source: Author's measurement results.

Figure 42: Periodic Get Device ID commands exchanged between ShM and IPMC during the monitoring stage of the blade operation, cheking the communication channel, the presence of carrier board and its identification.

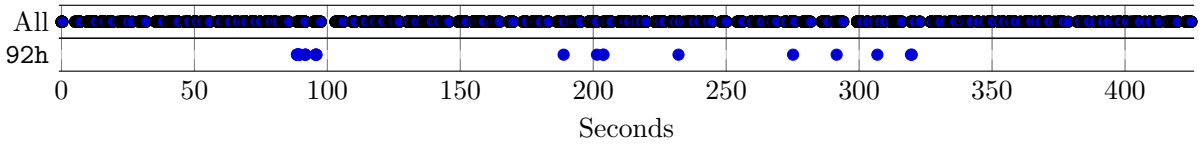


Source: Author's measurement results.

5.1.4 FRU state machine

The Figure 43 tries to make clear the ShM behavior change along with the activation and deactivation process. In the beginning the board is not in the crate so no messages are exchanged. Then there is a period with many communications due to the activation of an inserted board. Then the situation reach the monitoring stage where the ShM gets information from IPMC periodically. After that, the board is removed from the shelf and some messages are seen for the deactivation procedure followed with no more messages exchanged.

Figure 43: Messages exchanged during a complete hot-swap cycle of a Pulsar IIb carrier board. In the beginning there is no communication between ShM and IPMC, followed by the activation process. Then the monitoring stage is reached with periodic verification. And no messages are registered after deactivation of the blade.



Source: Author's measurement results.

Even if the `ipmb_traced` tool was not able to sniff all the communication, the process is registered in the ShM event log along with the related reason for the change:

```
# clia sel | tail
0x00F3: Event: at Aug 31 15:43:13 2016; from:(0x92,0,0); sensor:(0xf0,0);
event:0x6f(asserted): HotSwap: FRU 0 M0->M1, Cause=0x0
0x00F4: Event: at Aug 31 15:43:15 2016; from:(0x92,0,0); sensor:(0xf0,0);
event:0x6f(asserted): HotSwap: FRU 0 M1->M2, Cause=0x2
0x00F5: Event: at Aug 31 15:43:16 2016; from:(0x92,0,0); sensor:(0xf0,0);
event:0x6f(asserted): HotSwap: FRU 0 M2->M3, Cause=0x1
0x00F6: Event: at Aug 31 15:43:16 2016; from:(0x92,0,0); sensor:(0xf0,0);
event:0x6f(asserted): HotSwap: FRU 0 M3->M4, Cause=0x0
0x00F7: Event: at Aug 31 15:47:00 2016; from:(0x92,0,0); sensor:(0xf0,0);
event:0x6f(asserted): HotSwap: FRU 0 M4->M5, Cause=0x0
0x00F8: Event: at Aug 31 15:47:01 2016; from:(0x92,0,0); sensor:(0xf0,0);
event:0x6f(asserted): HotSwap: FRU 0 M5->M6, Cause=0x0
0x00F9: Event: at Aug 31 15:47:01 2016; from:(0x92,0,0); sensor:(0xf0,0);
event:0x6f(asserted): HotSwap: FRU 0 M6->M1, Cause=0x0
0x00FA: Event: at Aug 31 15:47:18 2016; from:(0x92,0,0); sensor:(0xf0,0);
event:0x6f(asserted): HotSwap: FRU 0 M1->M0, Cause=0x6
```

5.2 XVC RESULTS

This section describes the verification performed on the implemented XVC service and the obtained results relative to three different scenarios. First one provided the behavior

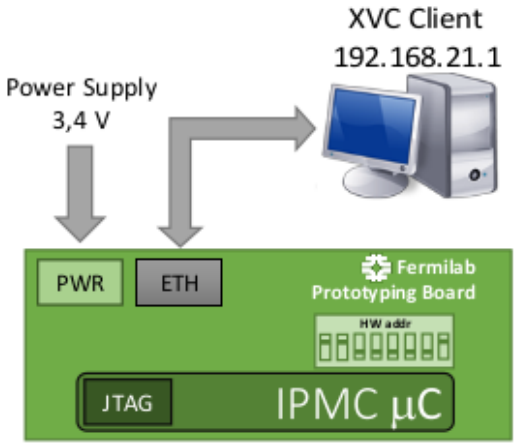
analysis of the behavior of the server out of the Xilinx resources context, making sure that it was acting as expected by the protocol description. Second, the implementation was subjected to work with the Xilinx tools and devices to ensure compatibility. And, finally, the system was plugged into an AdvancedTCA shelf and a remote setup were deployed.

5.2.1 Standalone Test

The primary scenario was assembled in the simplest way to ensure that the XVC server works as expected before trying it with vendor tools or devices. According to Figure 44, the assembling consisted of:

- a prototyping board designed by Fermilab, where the IPMC could be plugged in and the interfaces needed by this project, Ethernet and JTAG pins, were accessible. No JTAG capable devices were available.
- a computer to run an emulated XVC client to send and receive data through a TCP/IP link to the prototyping board.

Figure 44: Test scenario for the XVC implementation with no dependency of Xilinx tools or devices.



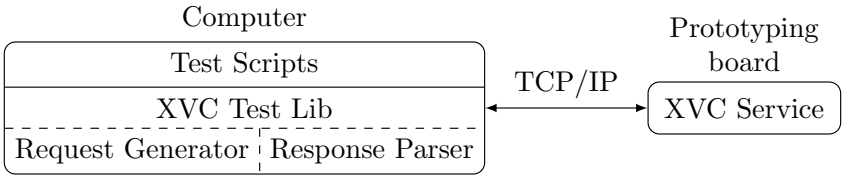
Source: Elaborated by the author.

First and foremost, the use of the Vivado software was not appropriate at this point because there is no way to establish an actual XVC connection without real Xilinx device. Also, even if it was possible, there was no easy way to control the data sent to the device, thus it would be hard to predict which answer should be received.

For that, a very simple XVC client emulator was developed using a Python framework. This client was built as a library with functions to stablish TCP/IP connections, assemble

the three XVC commands, parse XVC responses and print debug information. Then, the needed tests were implemented as scripts using that library, as illustrated in the Figure 45.

Figure 45: Diagram for the XVC client emulator framework built with Python scripts, capable to connect to the XVC service over a TCP/IP link.



Source: Elaborated by the author.

With the XVC client emulator, a series of verifications were allowed:

- Command behavior test: XVC commands (`getinfo`, `settck` and `shift`) were assembled and sent, and the response checked. In the case of the `shift` command, an array of bits was repeated for the TMS and the TDI vectors and expected as the answer, in a way that the data was copied directly from the input buffer.
- JTAG driver test: the emulator provided `shift` commands with random vectors for the TDI array, and again the answer should contain the same data. This test was made in two stages: first was used a internal loop to bit-by-bit copy the TDI array to the TDO array; after that, a electrical short-circuit was made between the TDI to the TDO pins and the full driver with real signals was used.
- Reliability test: shift commands with random content and length were assembled and sent, expecting to receive the same data back. The client emulator counted zero wrong responses after tests performing 2 hundred thousand transactions each. Once FPGA configuration is one of the longest operations involving around 50 thousand transactions for 28 MB firmware files, the XVC service code was considered reliable.

After that, the XVC service was considered ready to be tested with vendor tools and devices.

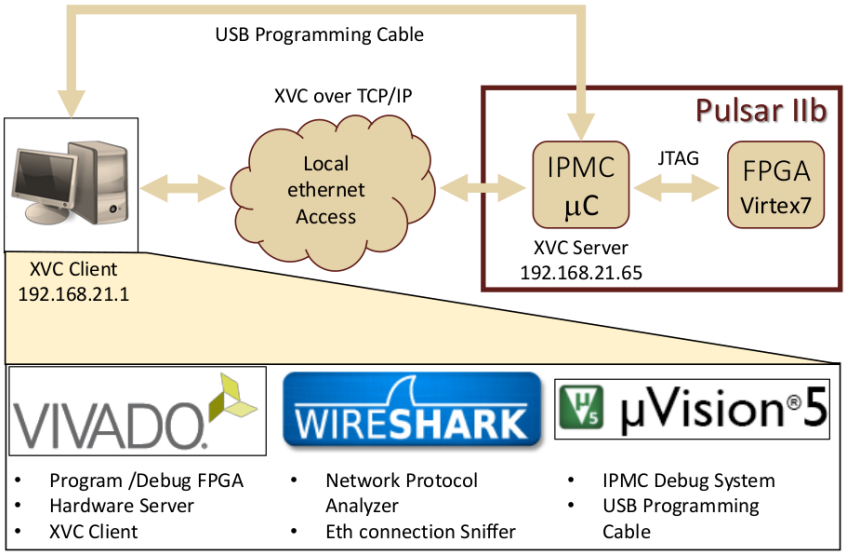
5.2.2 Xilinx Resources Verification

With the standalone structure verified, the implementation could move forward to the next level, which includes the use of vendor specific tools. This part of the tests aimed to ensure the compatibility with that and the Figure 46 describes the topology of this test. A workstation is mainly used to

1. run Xilinx tools and establish a internet link to the XVC server, allowing programming and debug actions;

2. run a sniffer in the connection between the Xilinx tools and the XVC server, showing what is inside the TCP/IP frames;

Figure 46: Test scenario for the XVC implementation with vendor tools and devices.



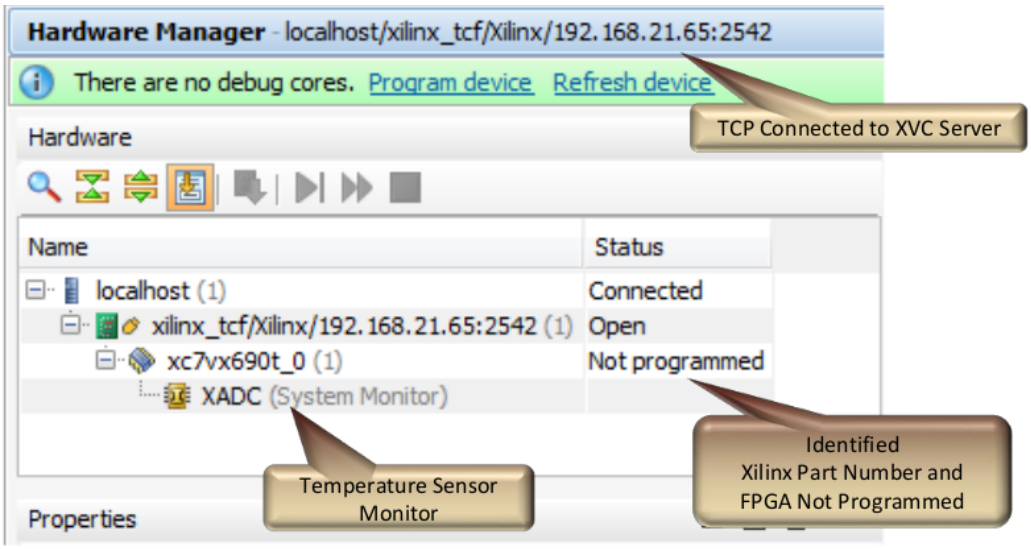
Source: Elaborated by the author.

In a communication between the Xilinx tools with their devices, some steps are performed to establish a communication over XVC protocol and Vivado sends a sequence of JTAG commands through Hardware Server. First, `getinfo` command is used to get the version of the protocol and the maximum buffer length supported by IPMC. After that, Vivado uses the `settck` to set the TCK clock frequency applied at the JTAG driver. Then, `shift` commands are used to identify the devices that are part of that JTAG chain. As illustrated at Figure 47, those steps were successful completed and information from a Virtex 7 FPGA is shown.

Important to mention that, once the device is identified, it is possible to get some data from FPGA. For example, that Virtex 7 exposed a Xilinx Analog to Digital Converter (XADC) module right out of the box, useful for temperature monitoring, shown in Figure 48. Other resources, like the Integrated Logic Analyzer (Integrated Logic Analyzer (ILA)) core, are available to directly inspect the internal signals inside the device, but are not part of the scope of this document.

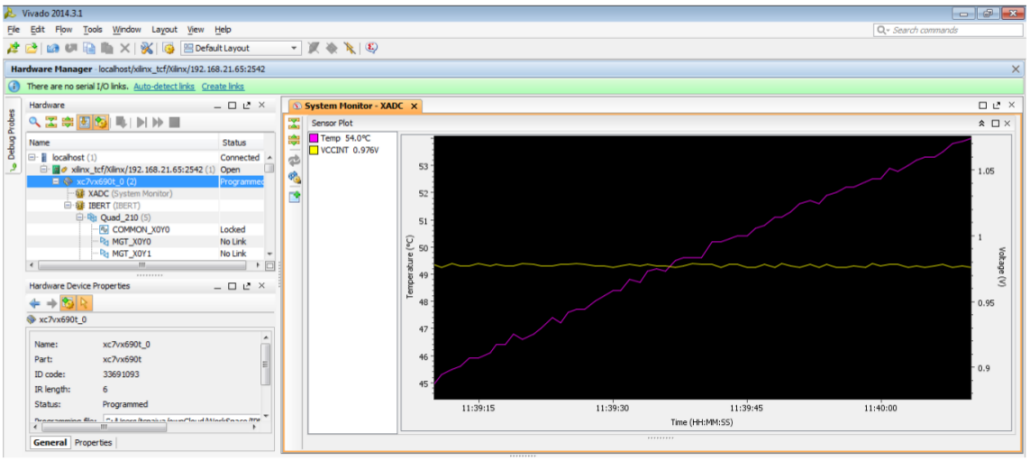
However, the messages exchanged throught XVC link in this situation were not enough to force the connection to its limits. Therefore, a remote 28 MB size firmware programming

Figure 47: Xilinx JTAG capable devices identified in a chain over a XVC link, corroborated by the IP address in the description of the link.



Source: Obtained as a result of the author's research.

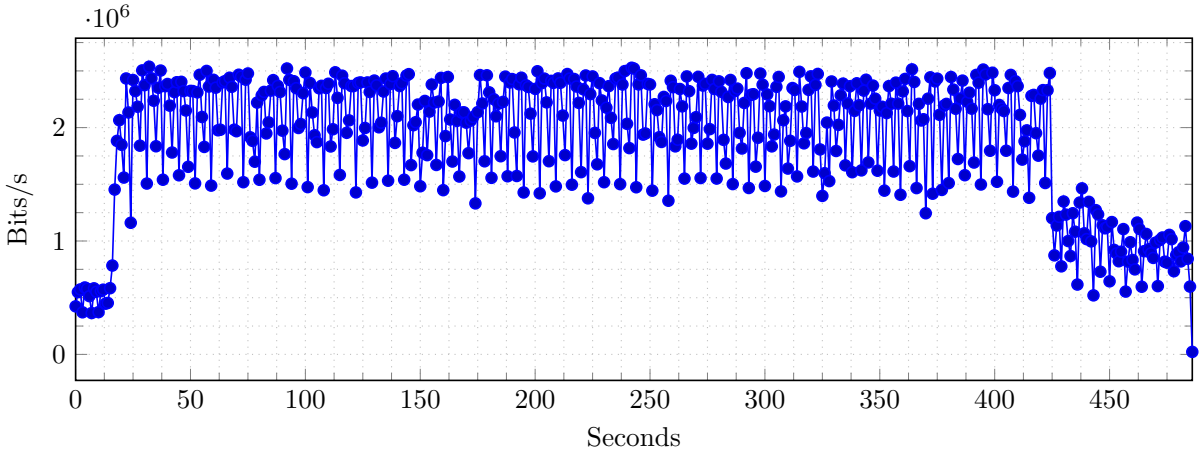
Figure 48: Plot obtained direct in the Vivado IDE of temperature and voltage sensors monitoring over an XVC link.



Source: Obtained as a result of the author's research.

was performed. Inline with the maximum buffer length informed in the `getinfo` response, the image file was divided in more then 56 thousand `shift` packets with length from 1020 to 1024 bytes. Also, the same number of `shift` responses where obtained and each packet with sizes between 505 and 507 bytes, what makes sense because it is approximately half of the size of the related requisition. The throughput measured with Wireshark during this test was around 2.4 Mbps and the complete operation took about 412 seconds, as illustrated in Figure 49.

Figure 49: Throughput of a firmware programming procedure measured with Wireshark, before optimization.



Source: Author's measurement results.

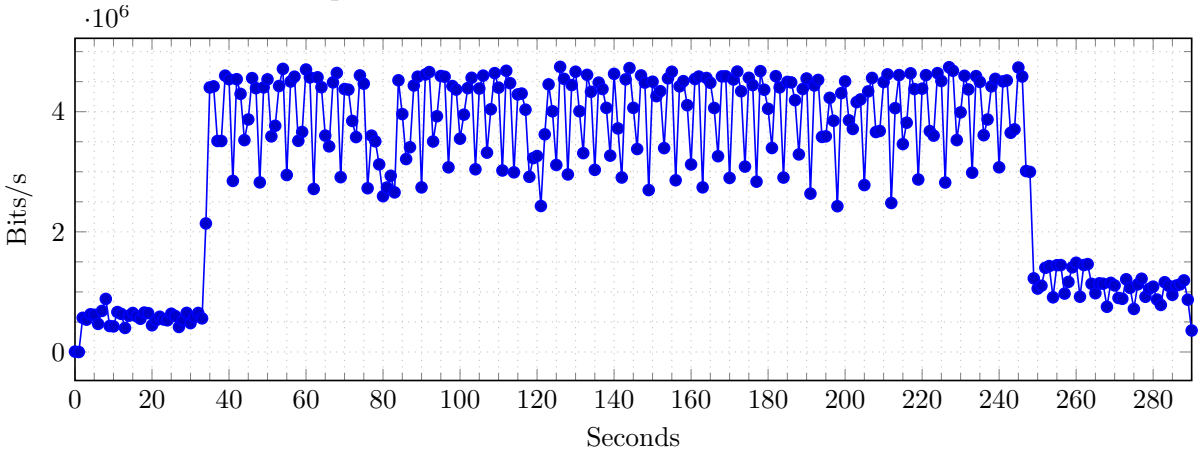
Some effort were dispended to optimize the code with focus on reducing the number of loops targetting bytes instead of bits, bit masking with plain approach instead of computated one and the use of variables rather than recalculations. With those simple principles, the throughput reached 4.7 Mbps, which shortened the configuration time to about 210 seconds, as seen in the Figure 50.

The performed tests certified that the XVC server implemented is ready to be used. This new IPMC function will bring speed and mobility to the firmware development in the Pulsar IIb project because all the debug and programming resources are available over traditional network links. Next section shows how to use the XVC service within the AdvancedTCA topology.

5.2.3 XVC service within AdvancedTCA shelf scope

This section describes the final sets made to the project to make the Xilinx JTAG capable devices inside of an AdvancedTCA shelf available over internet, as illustrated in the Figure 51. Worth pointing that the Hardware Server is a automatically installed

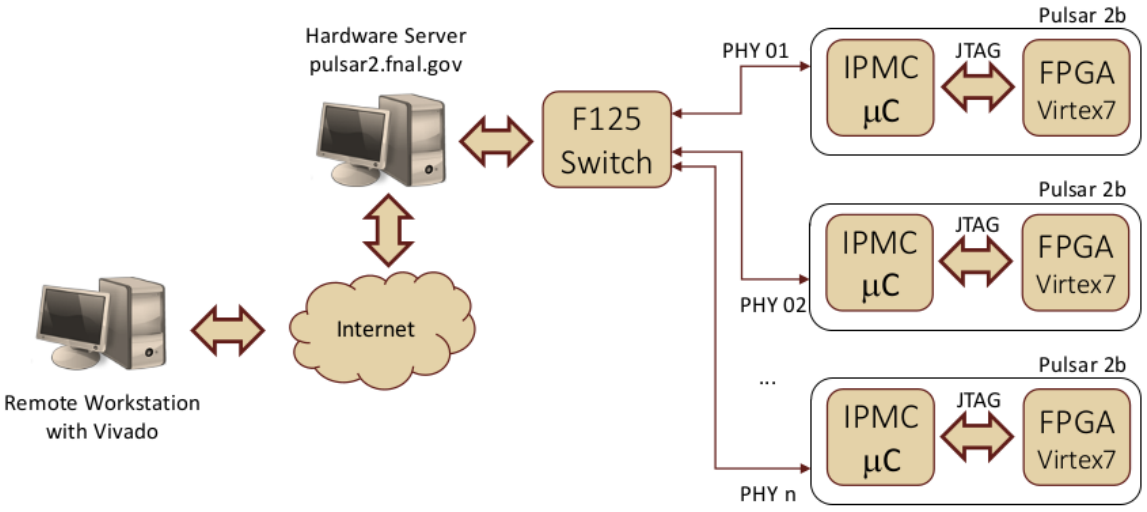
Figure 50: Throughput of a firmware programming procedure measured with Wireshark, after optimization.



Source: Author's measurement results.

with Vivado but it can be used in a standalone way, and taking advantage of this fact, a gateway machine with a known IP was inserted in the structure to deal with the local hardware and to wait for external connections. Also, as the standard requires a backplane inside shelf interconnecting all blades, an AdvancedTCA switch was used to expose the network interface of each IPMC, making the XVC service available outside of the shelf.

Figure 51: Shelf topology used for tests of the XVC services in the AdvancedTCA context. An AdvancedTCA network switch exposes the XVC links to a hardware gateway with Xilinx Hardware Serverinstalled.

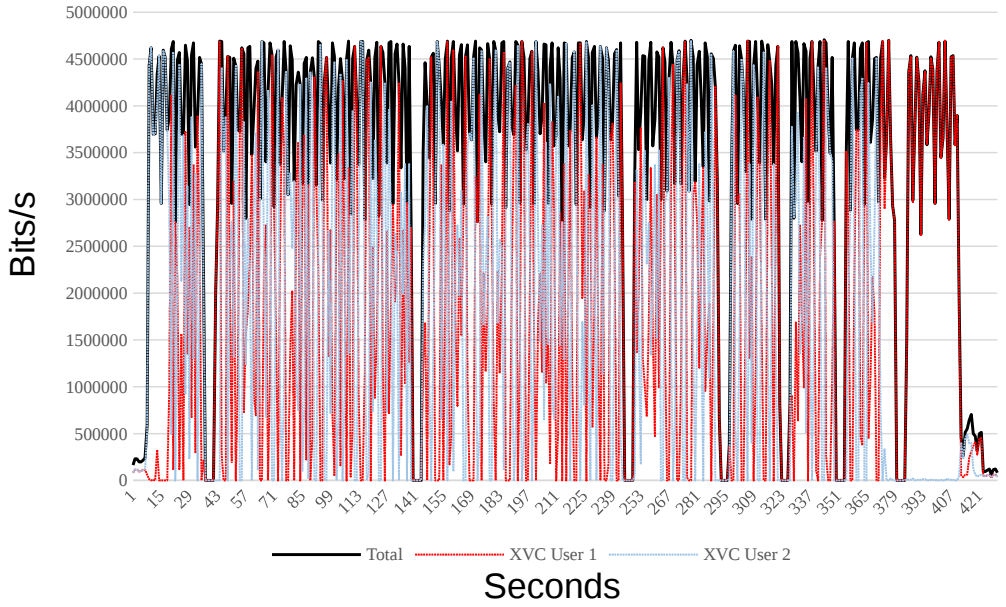


Source: Provided by SPRACE.

In the first attempt, the Hardware Server was installed and started with no custom settings at the gateway machine. The remote user was able to connect using the default port (3121) using Vivado and it was possible to list all the Pulsar IIb blades available in the shelf, nevertheless each XVC service was identified by its local IP address. Also, more than one remote user could be connected to different boards at same time, however if one

of then began a programming procedure, the service became very slow for the others. This behavior was confirmed using Wireshark to sniff simultaneous programming operations of two remote users and the graph in Figure 52 shows that the total throughput is saturated in no more than 5 Mbps, even though the link between gateway and each XVC server were distinct, and the duration was 402 seconds for one user and 377 for the other, which means almost twice the time needed for a independent operation.

Figure 52: FPGAs of two Pulsar IIb blades were simultaneously configured with one instance of the Hardware Server. Reduced throughput obtained due to switching between both configuration operations, which is reinforced by the multiple times the plots touch the horizontal axis.



Source: Author's measurement results

As the expected performance should be twice as faster as the obtained result and the Hardware Server instance was taken as a possible bottleneck due to the high variation in the thoughput plot, indicating that the software was switching between targets during the configuration. A second approach was proposed where an instance of the Hardware Server would be created for each XVC service available. This way, every set of Hardware Server and XVC server would be accessed by a different TCP port (XILINX, 2012):

```
hw_server -e "set auto-open -servers xilinx-xvc:<ipmc ip>:2542" -stcp::31<xx>
```

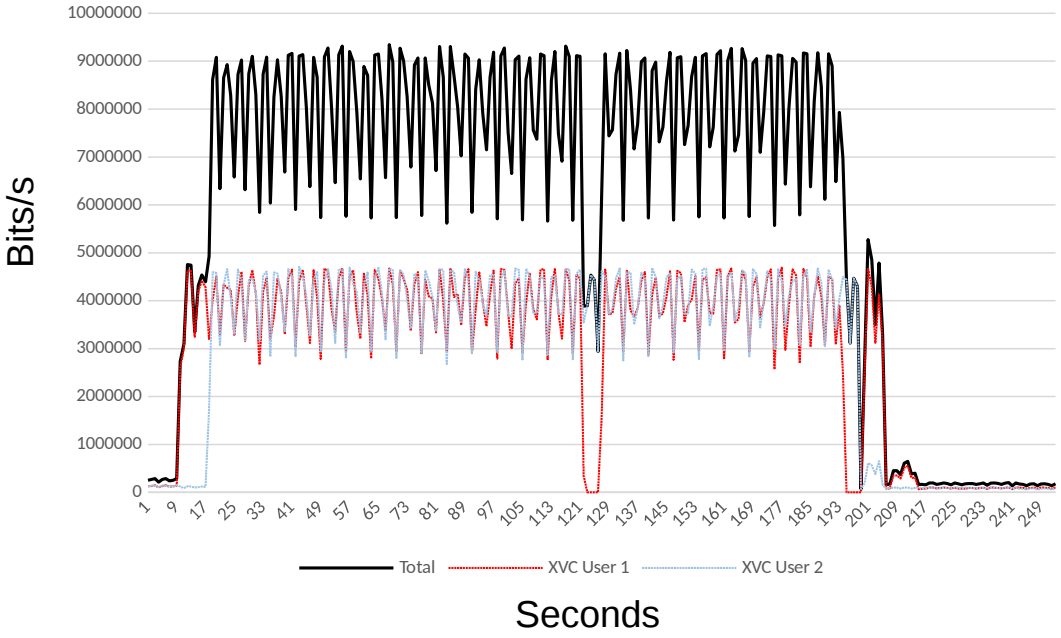
where `<ipmc ip>` was the IP address of each Pulsar IIb and `<xx>` was the number of the physical slot where it was plugged in. For example:

- XVC service of the physical slot 01 is connected to the Hardware Server with TCP port 3101

- XVC service of the physical slot 05 is connected to the Hardware Server with TCP port 3105
- XVC service of the physical slot 14 is connected to the Hardware Server with TCP port 3114

Therefore, now each remote user must only know which physical slot should be used. The total throughput graph in the Figure 53 of more than 9 Mbps is a sum of the throughput of both programming operation, and this was the previously expected result, with no influence between the remote user connections.

Figure 53: FPGAs of two Pulsar IIb blades were simultaneously configured with one instance of the Hardware Server for each blade. Obtained throughput of twice the rate for a single FPGA blade configuration was in accordance with the expected rate, since configuration channels should be independent.

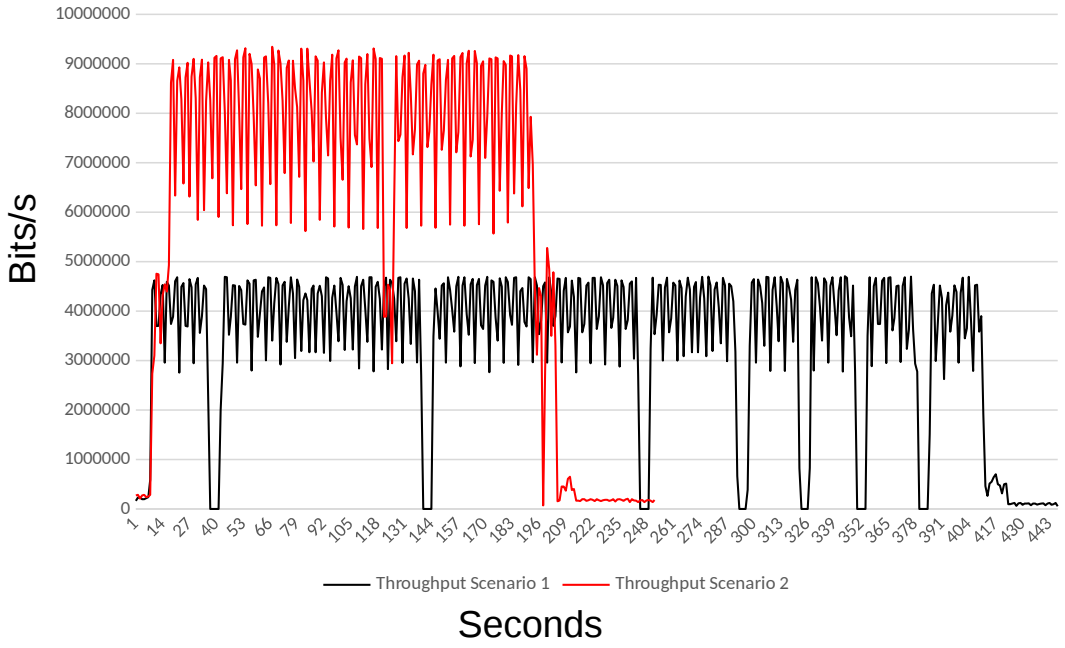


Source: Author's measurement results

Thus, a new plot was assembled with the throughput of both scenarios in Figure 54. One possible reason for the little throughput of the first case is that the Hardware Server instance is not a multi-threaded process, so the programming procedure made by one remote user make the system busy for the other connections.

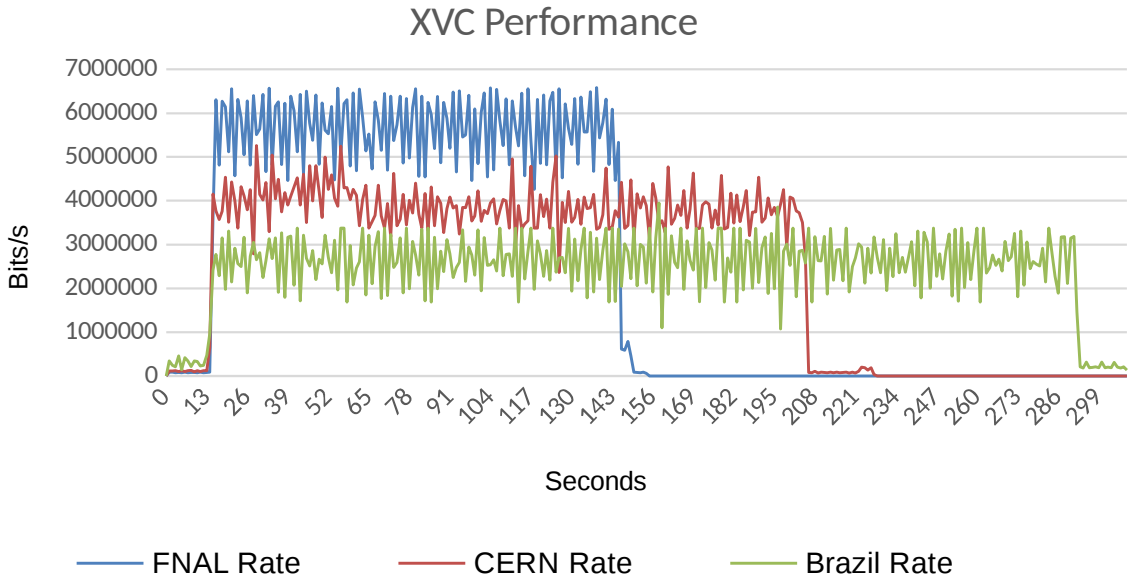
Finally, Figure 55 presents plots for Pulsar IIb FPGA being configured over XVC links from different places in the world. When compared to the local throughput at Fermilab, the external connections from Europe and South America showed a suitable performance for a remote development environment.

Figure 54: Comparizon between the throughput of the scenario with one shared Hardware Server and other with two dedicated Hardware Servers.



Source: Author's measurement results.

Figure 55: Thoughput of FPGA configuration operation over XVC links from different places in the world, demonstrating reasonable performance for a remote development environment.



Source: Author's measurement results

5.3 PUBLICATIONS

- PAIVA, T. C.; RAMALHO, L. A.; SHINODA, A. A.; VAZ, M.; CASCADAN, A.; FERREIRA, V. F.. A framework for development and test of xTCA modules with FPGA based systems for particle detectos. In: International Workshop on Personal Computers and Particle Accelerator Controls – PCAPAC, 2016, Campinas, Brazil.

- PAIVA, T. C.; RAMALHO, L. A.; SHINODA, A. A.; VAZ, M.; IOPE, R. L.; LEAL, B. C.. Implementation of the XVC protocol for the development of FPGA-based processor board in High Energy Physics instrumentation. In: Conferência Brasileira de Dinâmica, Controle e Aplicações – DINCON, 2015, Natal, Brazil. Anais do DINCON 2015, 2015. p. 1–7.

- RAMALHO, L. A.; PAIVA, T. C.; LIU, T. T.; SHINODA, A. A.; OLSEN, J.; LEAL, B. C.; SILVA FILHO, M. V.; IOPE, R. L.. Development of an Intelligent Platform Management Controller for the Pulsar IIb. In: 2015 IEEE Nuclear Science Symposium and Medical Imaging Conference, 2015, San Diego – California – USA. Proceedings of 2015 NSS, 2015. p. 1–4.

6 CONCLUSIONS AND FUTURE WORK

The IPMI implementation for the Pulsar IIb IPMC is a convenient function in the AdvancedTCA context proposed by Fermilab for the L1TT project. Services like XVC, that allows developers to perform remote FPGA programming and debugging operations, offer benefits of availability provided by AdvancedTCA standard. The remote programming is performed while the IPMI task ensures a healthy operation, with auto-cooling control and almost no performance loss. The XVC and the IPMI services had been working simultaneously and shown that there is almost no interference of each other when the OS processing is taking into account.

Availability test of the IPMI system were performed during 5 days with in an AdvancedTCA shelf with Pulsar IIb active and its operation were stable. Moreover, the AdvancedTCA stand assembled at Fermilab has been used for several months receiving XVC connections from different parts of the world without glitches.

The implementation had been debugged in two ways. The first one was the Keil μVision Debug System with ULINK programming cable connected directly to the IPMC microcontroller. This tool was the most reliable but the number of cables involved is a limiting factor when an AdvancedTCA shelf is been debugged. The second tool used, `ipmb_traced` sniffer, brought some fast evolve in the early steps of the implementation. Communication between AdvancedTCA carrier boards such as fan set, switch, and Pulsar IIb were followed using this sniffer tool. Although being very useful, some packets were lost and important requests and responses in the communication were not visible. Considering this disadvantage, metrics like throughput are not reliable.

The primary point is that the implemented XVC service works reliably: the client interface succeed to open and keep a TCP connection, the core performed the expected behavior for the `getinfo`, `settck` and `shift` commands, and the JTAG driver was able to communicate directly to the device. Besides that, the reliability test also showed that no errors were found during several sequential transactions.

The Wireshark Protocol Analyser was a strong ally to investigate and to provide simple statistics about the links. With that, an unexpected behavior of the Hardware Server was found in the AdvancedTCA shelf context: it was restricting the throughput of the link with

the XVC server when there were more than one remote user connected. Multiple Hardware Servers improved the situation, however at this point is worth emphasize that the use of proprietary solutions limits the analysis and the ability to act in the enhancement of that and related resources.

While the presented solution demonstrated that a remote development environment with reconfigurable components is suitable in the context of the AdvancedTCA standard, it is really a Xilinx centered solution. For the future work, an interesting approach is the implementation of the current setup in a more agnostic fashion, to include tools of other vendors. It would be needed a driver in the host machine used by the developer, reconized by the FPGA vendor software as a common JTAG cable. However, JTAG transactions would be sent by internet network links instead, and in the other end, the packets should be received by a service similar to XVC, translating them to JTAG signals. Furthermore, in-depth studies would not be limited by closed source tools anymore, and a very appealing direction would be the formation of virtual JTAG rings assembled with devices from independent JTAG chains.

In the IPMI implementation scenario only, the next steps lies in the extension of the current code to support all the features defined by AdvancedTCA standard, converting the current restrict project in a real option for the AdvancedTCA context.

REFERENCES

ARTESYN. *ATCA-8330 – AdvancedTCA DSP Blade*. Ed. by Artesyn. Available at: <https://www.artesyn.com/computing/products/product/atca-8330-advancedtca-dsp-blade>. Access: Oct. 6, 2016.

ATLAS LAPP. *LAPP IPMC mezzanine*. Ed. by LAPP. [S.l.], Jan. 11, 2016.

BHATTI, A; CANEPA, A; CASARSA, M; CONVERY, M; CORTIANA, G; DONATI, S; FLANAGAN, G; GRECO, V; FRISCH, H; FUKUN, T et al. Level-2 calorimeter trigger upgrade at CDF. *IEEE Transactions on Nuclear Science*, IEEE, New York, v. 56, n. 3, p. 1685–1689, June 16, 2009.

CMS COLLABORATION et al. Technical proposal for the Phase-II upgrade of the Compact Muon Solenoid. *CERN-LHCC-2015-010, LHCC-P-008*, June 1, 2015.

GILBERT, Jay; RASOVSKY, Karel. *Choosing the right ATCA fabric*. Ed. by Embedded. Intel Corporation. Aug. 3, 2004. Available at: <http://www.embedded.com/design/connectivity/4009302/Choosing-the-Right-ATCA-Fabric>. Access: Sept. 15, 2016.

INTEL; HEWLETT-PACKARD; NEC; DELL. *Intelligent Platform Management Bus communications protocol specification*. Ed. by Intel. [S.l.], Nov. 15, 1999.

_____. *IPMI specification*. Ed. by Intel. Version 2.0. [S.l.], Oct. 1, 2013.

_____. *Platform management FRU information storage definition*. Ed. by Intel. Version 1.0. [S.l.], Feb. 28, 2013.

KEIL. *RL-RTX user guide*. [S.l.: s.n.]. Available at: <http://www.keil.com/support/man/docs/rlarm/rlarm_ar_artxarm.htm>. Access: Apr. 10, 2016.

_____. *TCPnet user guide*. [S.l.: s.n.]. Available at: <http://www.keil.com/support/man/docs/rlarm/rlarm_tn_tcpip_prot.htm>. Access: Apr. 10, 2016.

LARSEN, R. S. PICMG xTCA standards extensions for physics new developments and future plans. *17th IEEE-NPSS Real Time Conference (RT)*, Lisbon, v. 1, n. 1, p. 1–7, May 2010.

_____. Recent progress in next generation platform standards for physics instrumentation and controls. *18th IEEE-NPSS Real Time Conference (RT)*, Berkeley, CA, v. 1, n. 1, p. 1–5, June 2012.

MAUNDER, Colin M; TULLOSS, Rodham E. *The test access port and boundary scan architecture*. Washington, DC: IEEE Computer Society Press Los Alamitos, 1990.

MENDEZ, J; BOBILLIER, V; HAAS, S; JOOS, M; VASEY, F. Evaluation of a commercial AdvancedTCA board management controller solution (IPMC). *Journal of Instrumentation*, Bristol, v. 11, n. 02, p. c02044, 2016.

NXP SEMICONDUCTORS. *I^2C-bus specification and user manual*. Version Rev. 6. [S.l.], Apr. 2014.

OKUMURA, Y.; OLSEN, J.; LIU, T.T.; Y., HANG. Prototype performance studies of a full mesh ATCA-based general purpose data processing board. *IEEE Nuclear Science Symposium and Medical Imaging Conference*, Seoul, v. 1, n. 1, p. 1–6, Oct. 2013.

OLSEN, J.; LIU, T.T.; OKUMURA, Y. A full mesh ATCA-based general purpose data processing board. *Topical Workshop on Electronics for Particle Physics*, Perugia, v. 9, n. 1, p. 1–7, Sept. 2013.

_____. Data Formatter design specification. *FERMILAB-TM-2553-E-PP*, Perugia, v. 1, n. 1, p. 1–88, Mar. 2013.

ORACLE. *Sun Netra CP3270 ATCA blade server users guide*. Ed. by Oracle. [S.l.], June 2010.

PENTAIR. *Schroff AdvancedTCA Systems*. Ed. by Pentair. Available at: <http://schroff.pentair.com/en/schroff-na/ATCA>. Access: Oct. 6, 2016.

PEREK, P.; MAKOWSKI, D. *Intelligent Platform Management Controller for ATCA carrier boards*. Saarbrucken: Lambert Academic Publishing, 2011.

PICMG. *PICMG 3.0 – AdvancedTCA base specification*. Version R3.0. [S.l.], Mar. 2008.

_____. *PICMG AMC.0 – AdvancedMC base specification*. Version R2.0. [S.l.], Nov. 2006.

PIGEON POINT. *External interface reference*. [S.l.], 2015.

ROSSI, L; BRÜNING, O. *High Luminosity Large Hadron Collider A description for the European Strategy Preparatory Group*. Geneva, Aug. 2012. Available at: <https://cds.cern.ch/record/1471000>. Access: Oct. 6, 2016.

SANBLAZE. *RTM with 6Gb SAS Expander with 2 Disks*. Ed. by SANBlaze. Available at: <http://www.sanblaze.com/products/sb-rtm436/>. Access: Oct. 6, 2016.

SPRACE. *SPRACE research group*. Available at: <http://sprace.org.br/>. Access: Jan. 15, 2015.

VADATECH. *ART132 – ATCA Rear Transition Module*. Ed. by Vadatech. Available at: <http://www.vadatech.com/product.php?product=244>. Access: Oct. 6, 2016.

_____. *ATCA Carrier for VMEbus Board*. Ed. by Vadatech. Available at: <http://www.vadatech.com/product.php?product=270&catid_prev=0&catid_now=207&parentcat=35&parentarc=>. Access: Oct. 6, 2016.

XILINX. *Vivado Design Suite User guide – programming and debugging*. Ed. by Xilinx. [S.l.], Dec. 18, 2012.

_____. *Xilinx Virtual Cable protocol*. [S.l.: s.n.]. Available at: <https://raw.githubusercontent.com/Xilinx/XilinxVirtualCable/master/README_XVC_v1_0.txt>. Access: Apr. 10, 2016.

_____. *Xilinx Virtual Cable repository*. [S.l.: s.n.]. Available at: <https://github.com/Xilinx/XilinxVirtualCable>. Access: Apr. 10, 2016.

APPENDIX A – FRU DATA INFORMATION EXAMPLE

```
U8 FRU_Pulsar2b[] = {
        // [08] <- common area checksum (to calculate),
        // [56] <- board info area checksum (to calculate)
        // [60] <- record checksum (to calculate)
        // [61] <- multi record header checksum (to calculate)

        // [01-08] -- common area header 1-8
        0x01, 0x00, 0x00, 0x01, 0x00, 0x07, 0x00, 0xFF,

        // board area info 9-56
        0x01, 0x06, 0x19, 0xa0, 0x72, 0x92, 0xc8, 0x46, // [09-16] --
        0x65, 0x72, 0x6d, 0x69, 0x6c, 0x61, 0x62, 0xca, // [17-24] --
        0x50, 0x75, 0x6c, 0x73, 0x61, 0x72, 0x20, 0x49, // [25-32] --
        0x49, 0x62, 0x04, 0x00, 0x00, 0x00, 0x00, 0xca, // [33-40] --
        0x50, 0x75, 0x6c, 0x73, 0x61, 0x72, 0x20, 0x49, // [41-48] --
        0x49, 0x62, 0xc1, 0x31, 0x00, 0x00, 0x00, 0xFF, // [49-56] --
        // Description of Board info of Pulsar 2b
        // 9 - 0x01 Format Version 01
        // 10 - 0x06 Board Area Length 6 * 8 = 48
        // 11 - 0x19 Language Code 25
        // 12, 13, 14 - 0xa0, 0x72, 0x92 in LS 0x92072a0 = 1 Apr 2014
        //         (9597600 minutes since 1996)
        // 15 - 0xc8 = Board Manuf. Type Ascii 8bits Length 12 bytes
        // 16:23 - 0x46, 0x65, 0x72, 0x6d, 0x69, 0x6c, 0x61, 0x62 = "Fermilab"
        // 24 - 0xca = Product Name Type Ascii 8bits Length 10 bytes
        // 25:34 - 0x50, 0x75, 0x6c, 0x73, 0x61, 0x72, 0x20, 0x49, 0x49, 0x62
        //         = "Pulsar IIb"
        // 35 - 0x04 = Serial Number Binary
        // 36:39 - 0x00, 0x00, 0x00, 0x00 = Will change from IPMC to IPMC
        // 40 - 0xca = Board Part Number Type Ascii 8bits Length 10 bytes
        // 41:50 = 0x50, 0x75, 0x6c, 0x73, 0x61, 0x72, 0x20, 0x49, 0x49, 0x62
        // 51 - 0xc1 = FRU File ID Type Ascii 8bits Length 1 bytes
        // 52 - 0x31 = "1"
        // 53 - 0xc1 = no more fields
        // 54:55 - 0x00, 0x00 = complete
        // 56 - 0x00 = checksum board area info

        // multi record area info 57-END
        0xc0, 0x82, 0x4e, 0xDC, 0xFF, 0x5a, 0x31, 0x00, // [57-64]
        0x14, 0x00, 0x01, 0xd7, 0x7c, 0x77, 0xd7, 0xab, // [65-72]
        0x11, 0xad, 0xb9, 0x22, 0x4d, 0xfe, 0x7b, 0x9c, // [73-80]
        0x2d, 0x37, 0x9b,                               // [81-83]
        // E-Keying info Pulsar 2b
```

```
        //          = Available Point-to-Point Connections base/fabric/update
        //          interface channels and ports
        //  Base Interface Channel 1 Port 1 Ethernet 10/100/1000
        //  Link Descriptor 01
        //  0x00, 0x00, 0x12, 0x01 (LS Byte) => 0x01, 0x12, 0x00, 0x00
        //  (31:24) grouping ID (8) = 0000 0000
        //  (23:20) link type extension (4) = 0000 10/100/1000BASE-T Link
        //          (four-pair)
        //  (19:12) link type (8) = 0000 0001 - PICMG 3.0 Base Interface
        //          10/100/1000 BASE-T
        //  (11:00) link designator (12) = 0010 0000 0001 - Port 1 Included -
        //          Base Interface - Channel 1
        0x01, 0x12, 0x00, 0x00, // BI channel 1 // [84-87]

        //  Fabric Interface Channel 1 Port 0/1/2/3
        //  Link Descriptor 02
        //  0x00 0x00 0x2f 0x41(LS Byte) => 0x41, 0x2f, 0x00, 0x00
        //  (31:24) grouping ID (8) = 0000 0000
        //  (23:20) link type extension (4) = 0000 10/100/1000BASE-T Link
        //          (four-pair)
        //  (19:12) link type (8) = 0000 0010 - PICMG 3.1 Ethernet Fabric
        //           Interface
        //  (11:00) link designator (12) = 1111 0100 0001 - Port 0/1/2/3
        //           Included - Fabric Interface - Channel 1
        0x41, 0x2f, 0x00, 0x00, //FI channel 1 // [88-91]

        //  Fabric Interface Channel 2-13 Port 0/1
        //  Link Descriptor 03
        //  0x00 0x00 0x23 0x42(LS Byte) => 0x42, 0x23, 0x00, 0x00
        //  (31:24) grouping ID (8) = 0000 0000
        //  (23:20) link type extension (4) = 0000 10/100/1000BASE-T Link
        //           (four-pair)
        //  (19:12) link type (8) = 0000 0010 - PICMG 3.1 Ethernet Fabric
        //           Interface
        //  (11:00) link designator (12) = 0011 0100 0010 - Port 0/1 Included -
        //           Fabric Interface - Channel 2
        0x42, 0x23, 0x00, 0x00, //FI channel 2  //  [92-95]
        0x43, 0x23, 0x00, 0x00, //FI channel 3  //  [96-99]
        0x44, 0x23, 0x00, 0x00, //FI channel 4  // [100-103]
        0x45, 0x23, 0x00, 0x00, //FI channel 5  // [104-107]
        0x46, 0x23, 0x00, 0x00, //FI channel 6  // [108-111]
        0x47, 0x23, 0x00, 0x00, //FI channel 7  // [112-115]
        0x48, 0x23, 0x00, 0x00, //FI channel 8  // [116-119]
        0x49, 0x23, 0x00, 0x00, //FI channel 9  // [120-123]
        0x4a, 0x23, 0x00, 0x00, //FI channel 10 // [124-127]
        0x4b, 0x23, 0x00, 0x00, //FI channel 11 // [128-131]
        0x4c, 0x23, 0x00, 0x00, //FI channel 12 // [132-135]
        0x4d, 0x23, 0x00, 0x00  //FI channel 13 // [136-139]
};
```

APPENDIX B – SDR FULL SENSOR RECORD EXAMPLE

```
U8 SDR_U2_FPGATemperature[] = { // LTC2990
        0x03, 0x00, 0x51, 0x01, 0x34, 0x00, 0x00, 0x03, // 01-08
        0xa0, 0x60, 0x77, 0x66, 0x01, 0x01, 0x00, 0x00, // 09-16
        0x7c, 0x00, 0x38, 0x38, 0x00, 0x01, 0x00, 0x00, // 17-24
        0x01, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x22, // 25-32
        0x56, 0x14, 0xff, 0x00, 0x55, 0x4b, 0x32, 0x00, // 33-40
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xc9, // 41-48
         'F',  'P',  'G',  'A',  ' ',  'T',  'E',  'M', // 49-56
         'P'                                            // 57-64
};

// SDR comments - Example Sensor Threshold
// More Details IPMI Spec 2.0 SDR Type 01h, Full Sensor Record
// 1        - 0x02  =>  record ID LS
// 2        - 0x00  =>  record ID MS
// 3        - 0x51  =>  SDR version 1.5
// 4        - 0x01  =>  Record Type Full Sensor Record
// 5        - 0x34  =>  Length from byte 6 (Sensor Owner ID) up to end
// 6        - 0x00  =>  Hw_Addr of IPMC (Sensor Owner ID)
// 7        - 0x00  =>  Sensor Owner LUN
// 8        - 0x02  =>  Sensor Number # 1
// 9        - 0xa0  =>  Entity ID Chassis-Specific
// 10       - 0x60  =>  '1... ....'  =>  treat entity as physical
//                      '.110 0000'  =>  device-relative Entity Instance
// 11       - 0x77  =>  '0... ....'  =>  Not Settable
//                      '.1.. ....'  =>  Scanning Enable
//                      '..1. ....'  =>  Events Enable
//                      '...1 ....'  =>  Threshold
//                      '.... 0...'  =>  No hysteresis
//                      '.... .1..'  =>  Sensor Type
//                      '.... ..11'  =>  Event Gen/Scanning Enable
// 12       - 0x66  =>  '0... ....'  =>  not ignore this sensor
//                      '.1.. ....'  =>  auto rearm
//                      '..00 ....'  =>  no hysteresis
//                      '.... 01..'  =>  threshold is readable but not settable
//                      '.... ..00'  =>  event generation just when threshold
// 13       - 0x01  =>  Sensor type Temperature
// 14       - 0x01  =>  Event/Reading Type Code Threshold
// 15,16    - 0x0000  =>  '0000 .... .... ....'  =>  Not Return Lower threshold
//                        on Get Sensor reading
//                        '.... 0000 0000 0000'  =>  No Threshold can cause
//                        Assertions
// 17,18    - 0x7000  =>  '0111 .... .... ....'  =>  Return Upper threshold on
```

```
//                          Get Sensor reading
//                          '.... 1100 0000 0000' => Non-Recoverable Threshold
//                          can cause Deassertion
// 19,20    - 0x3838  =>  '0011 1000 .... ....' => Init upper thresholds mask
//                          '.... .... 0011 1000' => Upper critical thresholds
//                          are readable
// 21       - 0x00  =>  '00.. ....' => unsigned
//                         '..00 0...' => no rate unit
//                         '.... .00.' => no modifier
//                         '.... ...0' => no percentage
// 22       - 0x01  =>  Sensor Base Unit Type degree C
// 23       - 0x00  =>  Sensor Modifier Unit Type not used
// 24       - 0x00  =>  Conversion Function 'f' is linear
// 25       - 0x01  =>  8 LS bits 'M' = 1
// 26       - 0x00  =>  '00.. ....' 2 MS bits 'M'
//                         '..00 0000' No Tolerance
// 27       - 0x00  =>  8 LS bits 'B' = 0
// 28       - 0x01  =>  '00.. ....' 2 MS bits 'B'
//                         '..00 0000' 6 LS bits Accuracy
// 29       - 0x00  =>  '0000 ....' => 4 MS bits Accuracy
//                         '.... 00..' => Accuracy exp
//                         '.... ..00' => Sensor Direction Unspecified
// 30       - 0x00  =>  '0000 ....' => 'Rexp' = 0
//                         '.... 0000' => 'Bexp' = 0
// 31       - 0x00  =>  '0000 0...' => reserved
//                         '.... .000' => normal min/ normal max/ nominal
//                        readings unspecified
// 32       - 0x22  =>  Ignore Nominal Reading
// 33       - 0x56  =>  Ignore Normal Max
// 34       - 0x14  =>  Ignore Normal Mini
// 35       - 0xff  =>  Sensor Max  => 255
// 36       - 0x00  =>  Sensor Min  => 0
// 37       - 0x55  =>  Init Upper Non-Recoverable Threshold = 85 C
// 38       - 0x4b  =>  Init Upper Critical Threshold = 75 C
// 39       - 0x32  =>  Init Upper Non-Critical Threshold = 50 C
// 40       - 0x00  =>  Not has Init Lower Non-Recoverable Threshold
// 41       - 0x00  =>  Not has Init Lower Critical Threshold
// 42       - 0x00  =>  Not has Init Lower Non-Critical Threshold
// 43       - 0x00  =>  No Positive Hysteresis
// 44       - 0x00  =>  No Negative Hysteresis
// 45,46  - 0x0000  =>  2 bytes reserved
// 47       - 0x00  =>  reserved for OEM use
// 48       - 0xc9  =>  '11.. ....' => ID String Type 8-bit ASCII
//                         '..0. ....' => reserved
//                         '...0 1001' => Length Code 9 bytes
// 49       - 0x46  =>  'F'
// 50       - 0x50  =>  'P'
// 51       - 0x47  =>  'G'
// 52       - 0x41  =>  'A'
// 53       - 0x20  =>  ' '
// 54       - 0x54  =>  'T'
// 55       - 0x45  =>  'E'
// 56       - 0x4D  =>  'M'
// 57       - 0x50  =>  'P'
```