UNIVERSIDADE ESTADUAL PAULISTA "JÚLIO DE MESQUITA FILHO"
FACULDADE DE ENGENHARIA
CÂMPUS DE ILHA SOLTEIRA

STHEFANY FERNANDES DE SOUZA

VIABILIDADE DA IMPLEMENTAÇÃO DO PROTOCOLO IPMI EM UM *SYSTEM-ON-CHIP*

Ilha Solteira
2019

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

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

STHEFANY FERNANDES DE SOUZA

VIABILIDADE DA IMPLEMENTAÇÃO DO PROTOCOLO IPMI EM UM *SYSTEM-ON-CHIP*

Dissertação apresentada à Faculdade de Engenharia de Ilha Solteira – UNESP como parte dos requisitos para obtenção do título de Mestre em Engenharia Elétrica. Especialidade: Automação

Prof. Dr. Aílton Akira Shinoda
Orientador

Ilha Solteira
2019

FICHA CATALOGRÁFICA
Desenvolvido pelo Serviço Técnico de Biblioteca e Documentação

S729v

Souza, Sthefany Fernandes de.
Viabilidade da implementação do protocolo IPMI em um SYSTEM-ON-CHIP / Sthefany Fernandes de Souza. -- Ilha Solteira: [s.n.], 2019
95 f. : il.

Dissertação (mestrado) - Universidade Estadual Paulista. Faculdade de Engenharia de Ilha Solteira. Área de conhecimento: Automação, 2019

Orientador: Aílton Akira Shinoda
Inclui bibliografia

1. Protocolo de comunicação IPMI. 2. Protocolo de barramento I²C. 3. Sistema-em-chip Zynq. 4. FreeRTOS.

Raiane da Silva Santos
Supervisora Técnica de Seção
Seção Técnica de Referência, Atendimento ao usuário e Documentação
Diretoria Técnica de Biblioteca e Documentação
CRB/8 - 9999

unesp

UNIVERSIDADE ESTADUAL PAULISTA

Câmpus de Ilha Solteira

CERTIFICADO DE APROVAÇÃO

TÍTULO DA DISSERTAÇÃO: Viabilidade da Implementação do Protocolo IPMI em um SYSTEM-ON-CHIP

AUTORA: STHEFANY FERNANDES DE SOUZA
ORIENTADOR: AILTON AKIRA SHINODA

Aprovada como parte das exigências para obtenção do Título de Mestra 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. CARLOS ANTONIO ALVES
Departamento de Engenharia Elétrica / Faculdade de Engenharia de Ilha Solteira

Prof. Dr. NELCILENO VIRGILIO DE SOUZA ARAÚJO
Instituto de Computação / Universidade Federal de Mato Grosso

Ilha Solteira, 05 de setembro de 2019

AGRADECIMENTOS

Gostaria de agradecer ao Prof. Dr. Aílton Akira Shinoda por todos os conselhos e encorajamento como orientador deste trabalho. Gostaria de agradecer também a comissão de tese como um todo, incluindo os Prof. Dr. Carlos Antônio Alves e o Prof. Dr. Nelcileno Virgílio de Souza Araújo, por todos os comentários. Gostaria de mencionar o Alison França Costa pelo *feedback* durante os resultados preliminares. Gostaria de agradecer ao Lucas Arruda Ramalho pela sua cooperação, dedicação e apoio durante o desenvolvimento deste projeto.

Sou grata aos grupos SPRACE/NCC e CMS/LHC/Cern, cuja colaboração forneceu o contexto e os materiais necessários neste projeto.

Agradeço aos professores e alunos que conheci durante as aulas na UNESP Ilha Solteira, por todo o conhecimento e amizade compartilhados. Gostaria de estender meu obrigado aos familiares e aos amigos(as) que me apoiaram durante este período. Agradeço ao Universo e ao meu ser pela coragem de vivenciar essa experiência nova, incrível e muito marcante na minha vida, espero que as etapas futuras sejam desafiadoras como esta.

“O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001”

“Esse tem sido um dos meus mantras – foco e simplicidade. Simples pode ser mais difícil de fazer do que complexo; você tem que trabalhar duro para clarear seu pensamento a fim de torná-lo simples.” Steve Jobs.

RESUMO

Bastidores eletrônicos de alta performance e disponibilidade utilizam o protocolo *Intelligent Platform Management Interface* (IPMI) para gerenciar seus dispositivos, controlando e monitorando os recursos disponíveis. Neste contexto para inserir dispositivos com tecnologia mais avançada, novos projetos foram elaborados para atualização dos sistemas de *hardware* e *software* baseados em *System-on-Chip* (SoC), principalmente na área de Física de Alta Energia. Uma aplicação existente, desenvolvida na parceira *São Paulo Research and Analysis Center – Fermi National Accelerator Laboratory* (SPRACE–FERMILAB) na colaboração internacional do *Compact Muon Solenoid detector/Large Hadron Collider/European Organization for Nuclear Research* (CMS/LHC/CERN), utiliza o protocolo IPMI implementado em um microcontrolador, contudo, para o processo de atualização vigente, há um interesse desta implementação em SoC. Assim, esta pesquisa foi desenvolvida como o estudo da viabilidade da implementação IPMI em um SoC. Para estabelecer e verificar o protocolo IPMI via barramento I²C, a plataforma Xilinx ZC702 *Evaluation Board* foi utilizada com os respectivos dispositivos SoC Zynq e *Erasable Programmable Memory* (EEPROM). Além disso foi desenvolvido uma estrutura simples do IPMI no sistema operacional em tempo real (*Free*RTOS) baseados em modelos de *hardware* e *software* criados na plataforma Xilinx IDE e SDK. Por meio dos resultados apresentados é possível constatar a viabilidade da implementação IPMI em sistemas como SoC Zynq como controlador de gerenciamento da plataforma, que possibilita a migração e o prosseguimento dos testes em plataformas *Advanced Telecom Computing Architecture* (ATCA) para uma comunicação real entre Zynq IPMC e o bastidor ATCA.

Palavras-chave: Protocolo de comunicação IPMI. Protocolo de barramento I²C. Sistema-em-chip Zynq. *Free*RTOS.

ABSTRACT

High performance and availability electronic racks use the Intelligent Platform Management Interface (IPMI) protocol to manage your devices by controlling and monitoring available resources. In this context to insert devices with more advanced technology, new projects were elaborated to update the System-on-Chip (SoC) based hardware and software systems, mainly in the area of High Energy Physics. An existing application developed at the São Paulo Research and Analysis Center partner - Fermi National Accelerator Laboratory (SPRACE – FERMILAB) in the international collaboration of the Compact Muon Solenoid detector/Large Hadron Collider/European Organization for Nuclear Research (CMS/LHC/CERN) uses The IPMI protocol implemented in a microcontroller, however, for the current update process, there is an interest of this implementation in SoC. Thus, this research was developed as the study of the viability of implementing IPMI in a SoC. To establish and verify the IPMI protocol via I²C bus, the Xilinx ZC702 Evaluation Board platform was used with the respective SoC Zynq and Erasable Programmable Memory (EEPROM) devices. In addition, a simple IPMI framework in the real time operating system (FreeRTOS) based on hardware and software models created on the Xilinx IDE and SDK platform was developed. From the results presented, it is possible to verify the viability of IPMI implementation in systems such as SoC Zynq as platform management controller, which allows migration and further testing on Advanced Telecom Computing Architecture (ATCA) platforms for real communication between Zynq IPMC and the shelf ATCA.

Keywords: IPMI communication protocol. I²C bus protocol. System-on-chip Zynq. FreeRTOS.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS E ABREVIATURAS

AMC	*Advanced Mezzanine Card*
A/D	*Analog-to-Digital*
APU	Application Processor Unit
ATCA	*Advanced Telecom Computing Architecture*
ATLAS	*A Toroidal LHC ApparatuS*
AXI	Advanced Extensible Interface
BMC	*Baseboard Management Controller*
BSP	*Board Support Package*
CERN	*European Organization for Nuclear Research*
CLB	*Configurable Logic Block*
CMOS	*Complementary Metal-Oxide Semiconductor*
CMS	*Compact Muon Solenoid*
cPCI	*compact Peripheral Component Interconnect*
DDR3L	*Double Date Rate Low-power*
D/A	*Digital-to-Analog*
DTC	*Detector, Trigger and Control*
EEPROM	*Electrically Erasable Programmable Read Only Memory*
FERMILAB	*Fermi National Accelerator Laboratory*
FF	*Flip-flop*
FPGA	*Field Programmable Gate Array*
FPU	*Floating Point Unit*
FreeRTOS	*Free Real-time Operation System*
FTDI	*Join Test Action Group*
GCC	*GNU Compiler Connection*
GPIO	*General Purpose Input/Output*
IDE	*Integrated Development Environment*
IOBs	Input/Output Blocks
IP	*Intellectual Property*
IPE	*Institute for Data Processing and Electronics*
I/O	*Input/Output*
I²C	*Inter-Integrated Circuit*

IC	*Integrated Circuit*
IPMC	*Intelligent Platform Management Controller*
IPMI	*Intelligent Platform Management Interface*
JTAG	*Join Test Action Group*
KIT	*Karlsruhe Institute of Technology*
LCD	*Liquid Crystal Display*
LED	*Light Emitting Diode*
LHC	*Large Hadron Collider*
LSB	*Least Significant Bit*
LUT	*Look-up Table*
MCU	*Microcontroller*
MMC	*Module Management Controller*
MSB	*Most Significant Bit*
MMU	*Memory Management Unit*
NCC	Núcleo de Computação Científica
NMOS	*Negative-channel Metal-Oxide Semiconductor*
OCM	*On Chip Memory*
OEMs	*Original Equipment Manufacturers*
PCB	*Printed-Circuit Board*
PCI	*Peripheral Component Interconnect*
PICMG	PCI *Industrial Computer Manufacturers Group*
PL	*Programmable Logic*
PMIC	*Power-management Integrated Circuit*
PS	*Processing System*
RAM	*Random Access Memory*
RISC	*Reduced Instruction Set*
ROM	*Read-Only Memory*
RTOS	*Real-time Operation System*
RTX	*Keil Real Time Operating System*
SCL	*Serial Clock*
SCU	*Snoop Control Unit*
SD	Secure Digital
SDA	*Serial Data*
SDK	*Software Development Kit*

ShMC	*Shelf Manager Controller*
SoC	*System on Chip*
SPI	*Serial Peripheral Interface*
SPRACE	São Paulo *Research and Analysis Center*
TFP	*Track Finding Processor*
UART	*Universal Asynchronous Receiver/Transmitter*
Unesp	Universidade Estadual Paulista
USB	*Universal Serial Bus*
VME	*Virtual Machine Environment*

SUMÁRIO

1 INTRODUÇÃO

Bastidores eletrônicos do tipo *Advanced Telecom Computing Architecture* (ATCA) utilizam o protocolo IPMI para realizar o gerenciamento entre os principais dispositivos que controlam o sistema. Este padrão de bastidor eletrônico é uma evolução dos bastidores do tipo *Virtual Machine Environment* (VME), utilizados em telecomunicações.

De acordo com (ROKOV, 2004), o IPMI é um protocolo de mensagens que define como monitorar o *hardware*, controlar dispositivos, recuperar *logs* de eventos e muito mais. Desta forma, o protocolo não é uma exclusividade dos bastidores ATCA, sendo encontrado em sistemas de *Data Centers* em geral que fazem o seu uso para monitoramento e controle também.

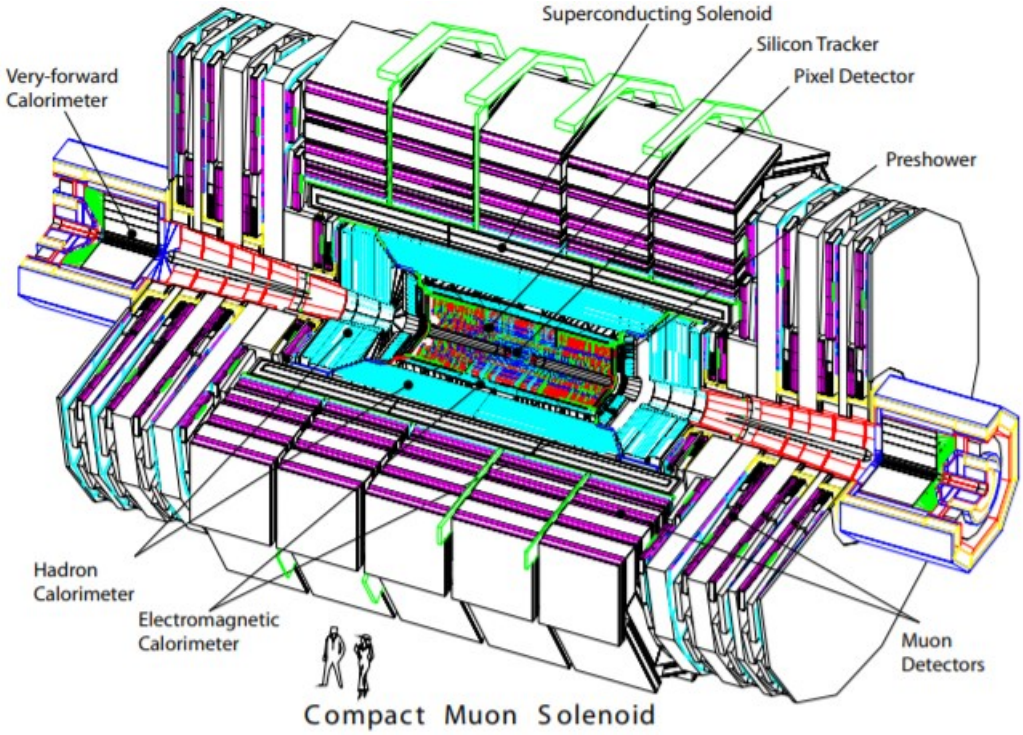
Como o padrão ATCA fornece uma infraestrutura eletrônica que proporciona alta disponibilidade e condições de alto desempenho, foi selecionado como plataforma em muitos projetos no *Large Hadron Collider* (LHC) em experimentos da *European Organization for Nuclear Research* (CERN) de acordo com (PEREK; MAKOWSK, 2011) (OKUMURA, 2013) (LARSEN, 2010) (2012).

Podemos destacar os detectores *A Toroidal LHC ApparatuS* (ATLAS) e *Compact Muon Solenoid* (CMS) que utilizam o padrão ATCA como plataformas de *hardware* para as atualizações dos sistemas eletrônicos do *back-end*, que possui cartões mezaninos com o *Intelligent Platform Management Controller* (IPMC) para a gestão das placas ATCA (CERN EP-ESE, 2017).

O detector CMS foi projetado para investigar uma ampla gama de fenômenos físicos, cuja característica do dispositivo é um solenoide supercondutor de 6 m de diâmetro interno, fornecendo um campo magnético de 3,8 T (CMS COLLABORATION, 2008), mostrado em perspectiva na Figura 1.

Com a atualização dos experimentos do LHC ocorridas para operação com alta luminosidade (ROSSI; BRÜNING, 2012), o detector CMS passou a fazer a aquisição dos dados baseado no rastreamento e reconstrução de nível 1 das partículas detectadas usando o *hardware* e *firmware Level 1 Tracking Trigger* (L1TT) que fornece uma filtragem dos dados altamente eficiente a altas taxas de disparo que detectam partículas menores nos limiares de momentum (CMS COLLABORATION *et al*., 2015).

Figura 1 – Vista em perspectiva do detector CMS



Fonte: CMS COLLABORATION (2008).

Especificamente, o sistema de processamento de dados de *back-end* do detector CMS utiliza as placas *Data, Trigger and Control* (DTC) e *Track Finding Processor* (TFP) que conecta o rastreador interno e externo e executa o primeiro nível da faixa de reconstrução (L1) a partir dos dados do rastreador externo respectivamente (AGGLETON *et al*., 2017).

Com o encerramento da segunda etapa de coleta de dados dos detectores do LHC, o acelerador de partículas foi desligado na manhã do dia 3 de dezembro de 2018 e a máquina permanecerá desativada por um período de dois anos, durante os quais grandes atualizações serão feitas para prepará-la para sua próxima fase de operações (SPRACE NEWS, 2018).

Neste intervalo de operação do LHC, grupos de tecnologia e desenvolvimento estão construindo *hardwares* para testar novas tecnologias para serem incorporadas ao sistema de processamento de *back-end* dos detectores em geral.

Uma das instituições que integram a colaboração internacional CMS/LHC/CERN, *Karlsruhe Institute of Technology* / *Institute for Data Processing and Electronics* (KIT/IPE), tem o interesse no desenvolvimento do protocolo IPMI em uma

solução integrada baseada em SoC como controlador de gerenciamento para uma nova plataforma ATCA em parceria com o SPRACE/NCC.

A tecnologia de um SoC é capaz de conter processador, memória e até placa de vídeo com uma arquitetura do tipo *Reduced Instruction Set* (RISC), proporcionando um baixíssimo consumo de energia, custo e boa performance, presentes em *smartphones* e *tablets* atualmente (CIPOLI, 2018).

Neste sentido, esta pesquisa foi desenvolvida para constatar a viabilidade da implementação do protocolo IPMI em um SoC, baseado em uma aplicação desenvolvida no trabalho de (PAIVA, 2016) que apresenta o protocolo implementado em um microcontrolador, chamado IPMC – FERMILAB, presente na placa ATCA Pulsar 2b.

Uma das principais funcionalidades IPMI inseridas no IPMC é o gerenciamento para o controle e monitoramento da inserção e retirada da placa ATCA seguindo o mecanismo *Hot-Swap* em um bastidor eletrônico, abordado no Capítulo 2.

Neste processo, a comunicação IPMI ocorre entre o controlador de gerenciamento da placa ATCA, o IPMC, com o controlador de gerenciamento do bastidor, *Shelf Manager Controller* (ShMC) para permitir ou não a ativação da placa no sistema.

Utilizando a programação do IPMC - FERMILAB com adaptações no código e sua inserção na plataforma ZC702 *Evaluation Board* (XILINX, 2018a) que possui o SoC Zynq, foi possível simular o primeiro estágio da comunicação entre os controladores IPMC e ShMC para verificação do padrão das mensagens IPMI de solicitação e reposta.

E, com o auxílio do dispositivo *Inter-Integrated Circuit* (I²C) EEPROM, também presente na plataforma ZC702, o cenário para a análise dos sinais de dados e *clock* com os pacotes IPMI enviados no barramento I²C foi criado, apresentado nos resultados. Além disso, a implementação IPMC foi adaptada para funcionar com o *Free Real Time Operating System* (*free*RTOS) que possui acesso livre, enquanto que na solução de (PAIVA, 2016) o desenvolvimento foi feito no *Keil Real Time Operating System* (RTX), uma versão do sistema em tempo real proprietária.

1.1 MOTIVAÇÃO

O estudo e aplicação de protocolos de comunicação de alto nível como IPMI, adaptável a sistemas com interface integrada como o SoC Zynq da Xilinx pode ser uma das novas tendências do mercado dos sistemas de processamentos de alto desempenho (XILINX, 2018a).

No mercado encontra-se disponível a família de SoCs Intel® Xeon® D (SUPERMICROCOMPUTER, 2019), que aliam confiabilidade, disponibilidade e facilidade de manutenção para classe de servidores com recursos disponíveis em soluções ultradensas e de baixo consumo de energia. E uma das líderes no segmento como a ASPEED (KENNEDY, 2018) com os controladores da série AST2500, utilizados em servidores de grandes *Original Equipment Manufacturers* (OEMs) para comutadores de hiperescala que possui o *Baseboard Management Controller* (BMC) com SoC baseado em Arm, com gráficos e lógica de controle incorporados.

Aliada a perspectiva de que muitos projetos na área de Física de Alta Energia estão sofrendo atualizações para um cenário com solução integrada como SoC, os resultados desta pesquisa podem contribuir para um panorama inicial do protocolo IPMI em um SoC como o Zynq da Xilinx e auxiliar nas etapas futuras da implementação Zynq IPMC em plataformas ATCA.

Outro aspecto, é diminuir as etapas no desenvolvimento e execução da implementação principal IPMC em um SO em tempo real livre para uma nova plataforma de *hardware* ATCA. Assim, esta pesquisa foi realizada para verificar a viabilidade da implementação IPMC em um SoC Zynq da Xilinx.

1.2 OBJETIVOS

A seguir são descritos os objetivos gerais e específicos da pesquisa assim como as etapas a serem seguidas.

1.2.1 Objetivo Geral

Implementar a programação do IPMC no dispositivo SoC Zynq da placa ZC702 e verificar a dinâmica da comunicação IPMI entre os controladores IPMC e ShMC via barramento I²C, com o auxílio de um dispositivo também I²C, presente na placa.

1.2.2 Objetivos Específicos

- Verificação do padrão de comunicação do barramento I²C no SoC Zynq;
- Especificação do dispositivo I²C EEPROM utilizado, e
- A viabilidade do desenvolvimento implementação do código IPMC em FreeRTOS na comunicação do dispositivo I²C escolhido com o SoC Zynq na placa ZC702.

1.3 ORGANIZAÇÃO DO TRABALHO

Este capítulo apresentou os aspectos gerais para o desenvolvimento desta pesquisa. No Capítulo 2 é feita uma breve revisão sobre o protocolo IPMI, o protocolo I²C, funcionamento do SoC Zynq, topologia do barramento I²C para comunicação entre dois dispositivos na placa ZC702, dispositivo I²C EEPROM e o *Free*RTOS. No Capítulo 3 é descrito a metodologia utilizada para o experimento. No Capítulo 4 é apresentado os resultados da implementação IPMI no SoC Zynq via terminal e osciloscópio. No Capítulo 5 é feita a discussão dos resultados obtidos desta implementação. No Capítulo 6 são apresentadas as conclusões sobre a pesquisa e a proposta para trabalhos futuros.

2 FUNDAMENTOS

Nesse capítulo são descritos os conceitos básicos sobre os protocolos IPMI e I²C, elementos básicos de um bastidor ATCA, estrutura básica do SoC Zynq, dispositivo I²C EEPROM e o sistema operacional FreeRTOS necessários para o desenvolvimento da pesquisa.

2.1 PROTOCOLO IPMI

Essencialmente, o protocolo IPMI é um padrão de instrumentação de *hardware* auto descritivo e baseado em mensagens. À medida que se executa o protocolo IPMI dentro de um sistema, é possível adicionar memórias externas (RAM/ROM), conectar outros barramentos I²C, usar pinos adicionais de entrada e saída (GPIO - *General Purpose Input Output*), alterar sensores, redimensionar *log* de eventos do sistema, etc. (ROKOV, 2004).

Desenvolvido pela Intel como um conjunto de interfaces comuns para um sistema de computador, ele pode ser usado por administradores de sistema para monitorar o estado do sistema e gerenciá-lo (INTEL *et al*., 2013). Também, independe de um sistema operacional e pode ser implementado sem um (KOZAK, 2011). A implementação de *hardware* é isolada da implementação de *software* desta forma novos sensores e eventos podem ser adicionados sem qualquer alteração do *software*.

O protocolo IPMI também define registros padronizados para descrever dispositivos de gerenciamento de plataformas e suas características (INTEL, 2015). Ter um sistema baseado na interface IPMI aumenta a confiabilidade de modo geral, com o monitoramento de parâmetros como temperaturas, tensões, ventiladores e intrusão do bastidor (opção que alerta se houver algum intruso dentro do bastidor ou se alguém abrir a porta sem autorização).

Outro aspecto interessante segundo (ROKOV, 2004), é que o protocolo também permite que o *software* de gerenciamento de vários fornecedores opere em diferentes plataformas de *firmware* e *hardware*.

Além disso, com a interface padronizada para *hardware* se tem um auxílio na previsão e monitoramento antecipado de falhas no diagnóstico de possíveis

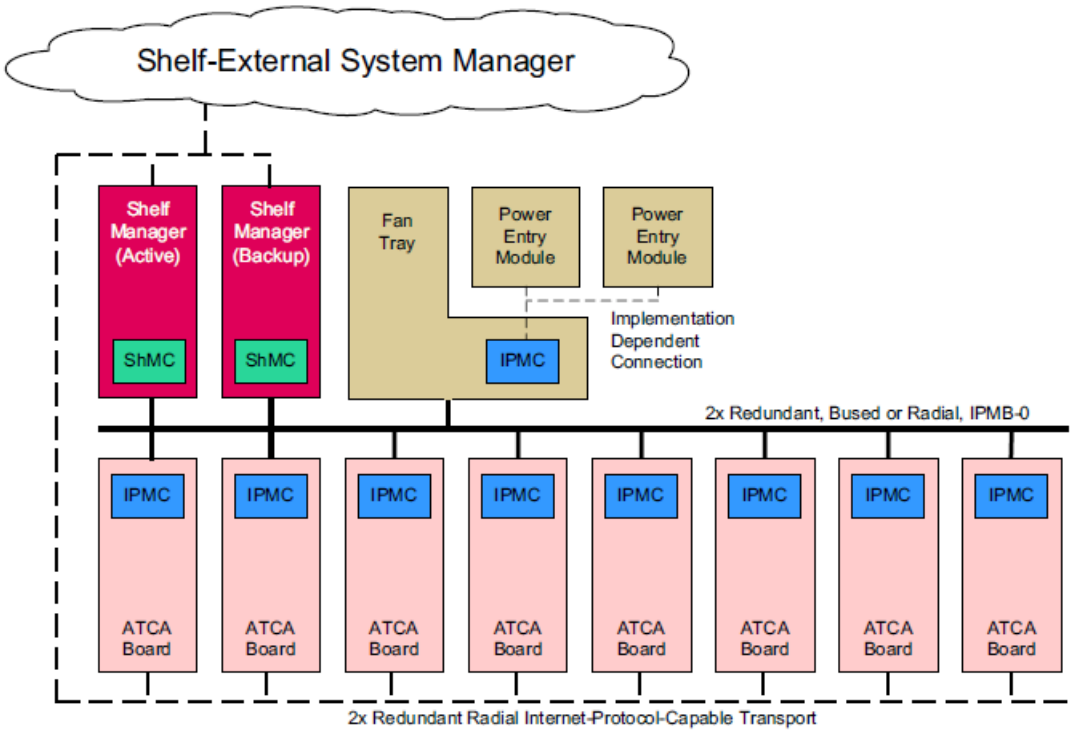
problemas com a mesma. Estendendo o controle de gerenciamento, monitoramento e entrega dos eventos dentro do bastidor.

O IPMI descreve como vários controladores de gerenciamento embarcados se colaboram (ORACLE, 2011). No topo da hierarquia IPMI, de modo geral, tem-se o *Baseboard Management Controller* (BMC) ao qual todos os outros controladores estão conectados. Estes por sua vez, são chamados de controladores satélite, que são componentes obrigatórios dos vários módulos do sistema que usam o *Intelligent Platform Management Bus* (IPMB) para a troca de mensagens com o BMC localizado no mesmo bastidor (KOZAK, 2011).

Uma aplicação muito difundida do protocolo IPMI são em bastidores com padrão ATCA. Neste padrão, o ShMC tem o papel de BMC e os IPMCs são os controladores satélites na arquitetura de gerenciamento de plataforma de *hardware*.

Na Figura 2, tem-se um exemplo de configuração de um bastidor ATCA, mostrando os elementos lógicos de gerenciamento do sistema, controladores ShMCs e IPMCs.

Figura 2 – Aspectos de gerenciamento de um exemplo de bastidor ATCA

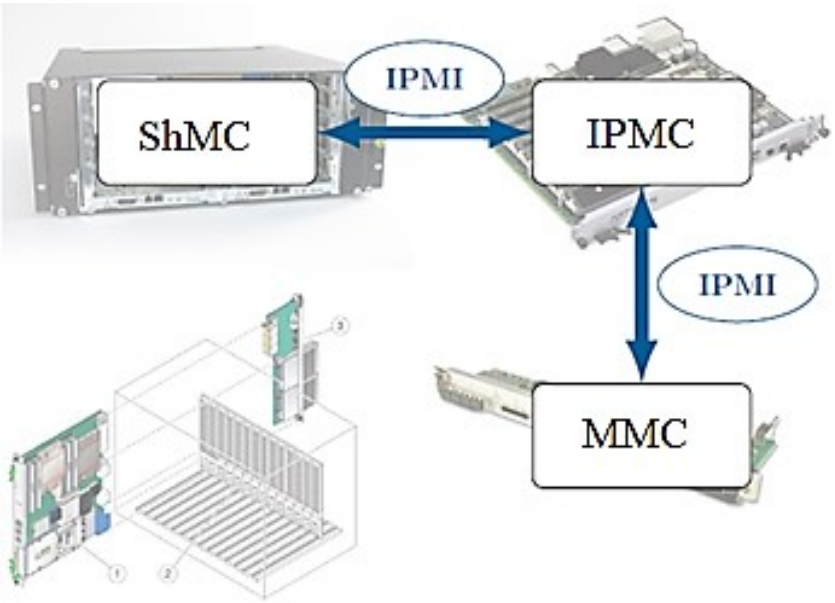


Fonte: PICMG 3.0 (2008).

O *Advanced Mezzanine Card* (AMC) é instalado em placas ATCA do tipo *carrier board* ou diretamente do tipo *Shelf* Manager ATCA. Esses cartões também utilizam o protocolo IPMI por meio do *Module Management Controller* (MMC) (KOZAK, 2011).

A Figura 3 ilustra uma aplicação simplificada da comunicação IPMI entre os principais elementos controladores em um bastidor ATCA.

Figura 3 – Exemplo de comunicação IPMI em um bastidor ATCA



Fonte: Paiva (2016).

Na Tabela 1 é mostrada a estrutura básica das mensagens IPMI para solicitação e resposta, utilizada pelos controladores de gerenciamento em geral. Em um bastidor ATCA, por exemplo, podem ser as mensagens transferidas do IPMC para o *Sh*MC e vice-versa.

As mensagens IPMI são divididas em categorias chamadas de *Network Functions* e cada uma delas consiste de um conjunto de comandos. A maioria dessas funções vem da especificação base IPMI. Cada solicitação IPMI deve ser seguida por uma resposta e o lado que inicia a troca de mensagens deve enviar novamente a mensagem original caso nenhuma resposta seja recebida (KOZAK, 2011).

Tabela 1 – Estrutura das mensagens IPMI

N° Byte	Solicitação	Resposta	
1	Endereço I²C do dispositivo que irá responder	1	Endereço I²C do dispositivo que fez a solicitação
2	*Network Function*	2	*Network Function*
3	*Header Checksum 1*	3	*Header Checksum 1*
4	Endereço I²C do dispositivo solicitante	4	Endereço I²C do dispositivo que responde
5	Comando com a sequência numérica	5	Comando com sequência numérica
6	Identificador do comando	6	Identificador do comando
		7	*Completion code* (complementar da solicitação)
7:N	Pacote de dados	8:N	Pacote de dados
N+1	Verificação da mensagem com um *Checksum 2*	N+1	Verificação da mensagem com um *Checksum 2*

Fonte: ADAPTADA DE KOZAK (2011).

A placa ZC702 da Xilinx possui uma arquitetura diferente da placa Pulsar 2b com IPMC programado, portanto comandos adicionais referentes a especificação e os dispositivos da plataforma escolhida, serão modificados. Contudo, mantendo a estrutura principal com os comandos IPMI de acordo com o protocolo.

Como a especificação IPMB define um transporte a nível de byte de transferências de mensagens IPMI sobre o I²C (INTEL et al., 1999), a aplicabilidade utilizando o barramento I²C presente na placa ZC702 pode ser constatada, com SoC Zynq operando ora como IPMC ora como ShMC (emulado), já que nesse cenário há a ausência de um bastidor ATCA, e a verificação é realizada através da comunicação com um dispositivo I²C para analisar as mensagens trocadas entre os controladores.

2.2 ELEMENTOS BÁSICOS DE UM BASTIDOR ATCA

O bastidor ATCA, é a estrutura mecânica que abriga as várias placas do tipo ATCA, possui recursos como fonte de alimentação, sistema refrigerado e uma interconexão *backplane*, de acordo com a Figura 4.

Uma camada de gerenciamento encontra-se na estrutura do bastidor ATCA, chamada *Shelf Manager* que serve para ligar e desligar os componentes, negociar energia, ler os sensores, executar decisões e disparar alarmes e atuadores (PICMG 3.0, 2008).

Figura 4 – Sistema Modular *Advanced*TCA ECO, com 14 slots, DC

Fonte: NVENT (2018).

Todos os elementos do bastidor ATCA praticamente são *Field Replaceable Unit* (FRU). Quando uma unidade FRU é inserida no bastidor, apenas parte da energia é fornecida a ela, permanecendo com as suas funções temporariamente incompletas.

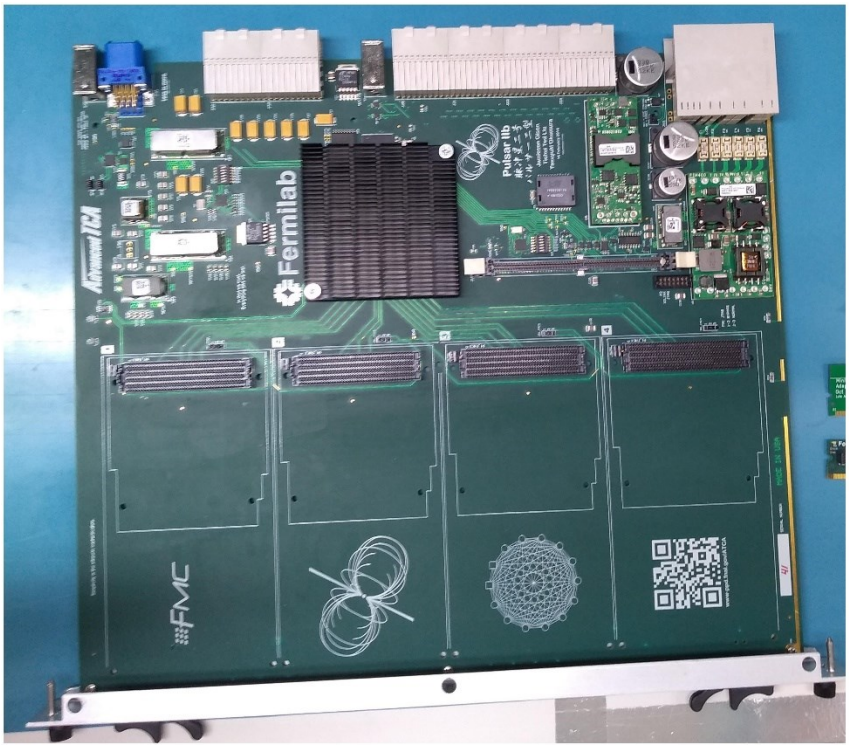
Uma das vantagens do padrão ATCA está na modularização do sistema, pois possui vários *slots* no *backplane* para interconexão das placas. Seguindo o padrão do *backplane* você pode plugar placas de propósitos diferentes em cada *slot*.

O ShMC inicia o processo de ativação da FRU, envolvendo a negociação de energia e transferência de informações. Estas etapas tornam o dispositivo operacional tanto para o procedimento de inserção como para extração desta unidade. Todas essas trocas de dados dependem do protocolo IPMI (PAIVA, 2016).

A maior de todas as placas, é conhecida como placa carregadora ou portadora, do inglês, *carrier board*. É uma placa-mãe que interconecta mezaninos AMC e o *backplane*, além de gerenciá-los.

Um elemento principal onde todas as outras placas se conectarão dentro do bastidor é mostrado na Figura 5.

Figura 5 – Placa ATCA Pulsar 2b do FERMILAB



Fonte: IMAGEM TIRADA EM 13/12/2018, SPRACE/NCC/SP.

Essas placas são de interconexão e roteamento, mas além disso podem conter FRUs, como os mezaninos. E assim com dois níveis de modularização. No entanto,

especificamente no caso dos mezaninos da Pulsar 2b, não são do tipo FRU, a placa carregadora sim.

Durante a ativação, essas placas devem reportar ao ShMC suas interconexões de *backplane* antes de configurá-las, permitindo que o ShMC detecte incoerência e informe a placa para não ativar esses *links* que sejam incompatíveis, impedindo-os de danificar (PAIVA, 2016).

A camada de gerenciamento que pode ser inserida na placa carregadora é o IPMC. Na Figura 6 é mostrado o módulo controlador de gerenciamento da placa Pulsar 2b, composto por microcontrolador ARM Cortex-M3.

Figura 6 – Cartão mezanino IPMC – FERMILAB da placa pulsar 2b

Fonte: Paiva (2016).

O canal de gerenciamento é um *link* do IPMB baseado no protocolo I²C, que interconecta o ShM ao IPMC dentro do bastidor ATCA, também chamado de canal principal, IPMB-0. Para este canal, o padrão ATCA garante a confiabilidade do barramento IPMB com a adição de um segundo canal para formar o IPMB-0, chamados de IPMB-A e IPMB-B. São canais redundantes com a mesma estrutura padronizada para o canal IPMB-0, para garantir a transmissão IPMI caso haja falha de um dos canais IPMB.

A especificação IPMI define os *links* e o controle de mensagens aos subsistemas de gerenciamento da plataforma dentro da placa carregadora, utilizando um canal de gerenciamento secundário, chamado IPMB-L, conectando ao IPMC os módulos de controladores de gerenciamento MMCs.

Outro tipo de placa que pode ser inserido no *slot*, conectado a placa carregadora, usando a parte traseira do bastidor é chamado de módulo de transição traseira, *Rear Transition Module* (RTM), mostrado na Figura 7.

Figura 7 – RTM436 da SANBlaze

Fonte: Sanblaze (2018).

Cada elemento é montado de acordo com o padrão ATCA, maiores detalhes na Figura 3 .

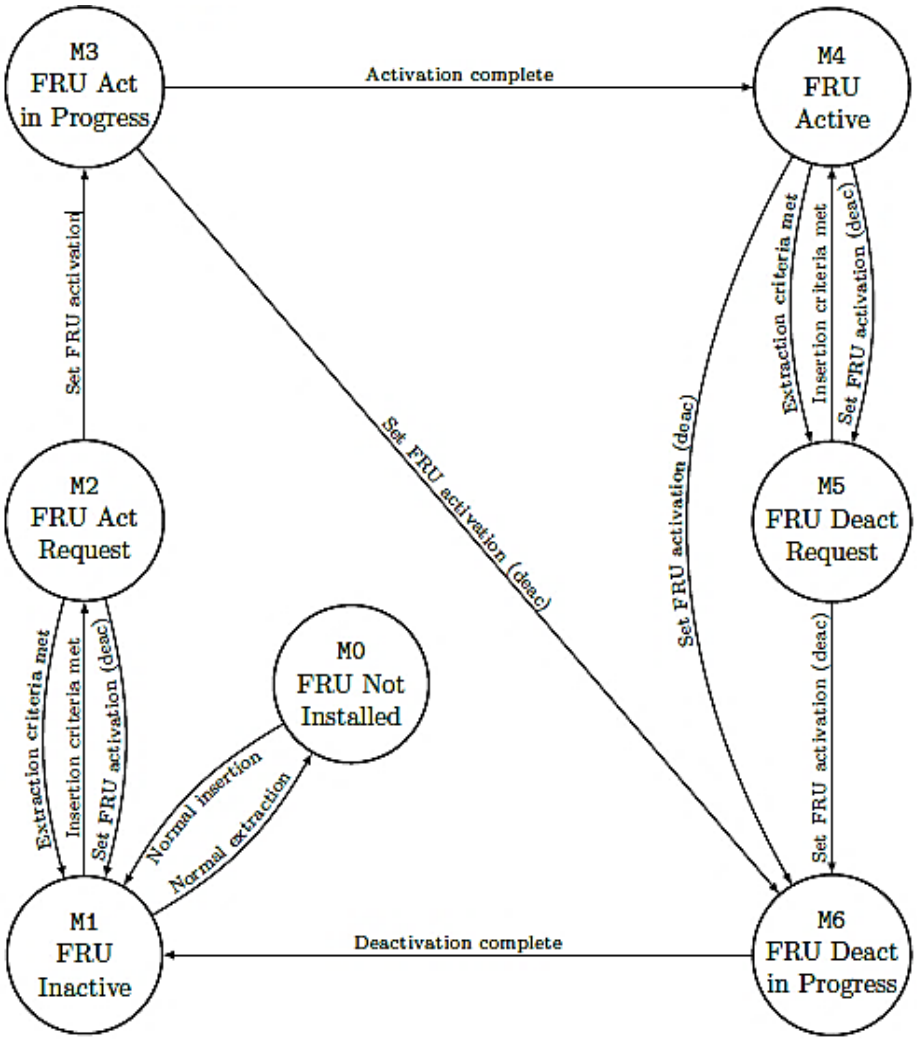
2.2.1 Comunicação IPMI do bastidor ATCA com IPMC da Pulsar 2b

A comunicação IPMI do bastidor ATCA com o IPMC da placa Pulsar 2b foi implementada seguindo as etapas do diagrama da máquina de estados do mecanismo *hot-swap* para inserção e retirada da placa do bastidor, mostrado na Figura 8.

No estado M0, a placa Pulsar 2b encontra-se desconectada. Quando inserida no bastidor, assume o estado inativo, M1. Seguindo a situação de uma transição normal da placa, o processo de ativação é acionado de M1 para M2, ocorrendo a transição para M2 e depois para M3, onde ela aguardará as instruções do ShMC.

Se a ativação da placa não for permitida, a transição da placa é direta do estado M3 para M6. Caso contrário, a ativação é concluída de M3 para M4. A placa permanece no estado M4 enquanto um processo de desativação não for acionado.

Figura 8 – Diagrama da máquina de estados para FRU definida pelo protocolo IPMI, utilizada no mecanismo *hot-swap*



Fonte: Adaptado de Perek (2010) E PICMG 3.0 (2008).

Ocorrendo uma solicitação de desativação, a placa é direcionada ao estado M5. Na situação do processo de desativação ser cancelado, a placa retorna ao estado M4. Do estado M6, em que a placa está em processo de desativação, só é possível ir para o estado M1.

Uma forma de verificar a operação da máquina de estados, é pelo comportamento do *Blue* LED (*Light Emitting Diode*), de acordo com a Tabela 2.

Tabela 2 – Comportamento do *Blue* LED em cada situação da máquina de estado do mecanismo *Hot-Swap*

Estado	Nome do Estado	Informações adicionais	Comportamento do Blue LED
M0	FRU não instalada	Placa em mãos	Desligado
M1	FRU inativa	*Handle switch aberta*	*Ligado*
M2	Solicitação de ativação da FRU	*Handle switch* fechada	*Piscada longa*
M3	Ativação da FRU em processo	ShMC valida a ativação	Desligado
M4	FRU ativada	Negociação de energia concedida	Desligado
M5	Solicitação de desativação da FRU	*Handle switch* aberta	Piscada curta
M6	Desativação da FRU em processo	ShMC valida a desativação	Piscada curta

Fonte: Adaptada de Paiva (2016).

Se o sistema está estável, a placa é ligada ou fica pronta para ser extraída. O *Blue* LED piscando significa que o procedimento *hot-swap* está em andamento, se estiver apagado, a placa está ativa.

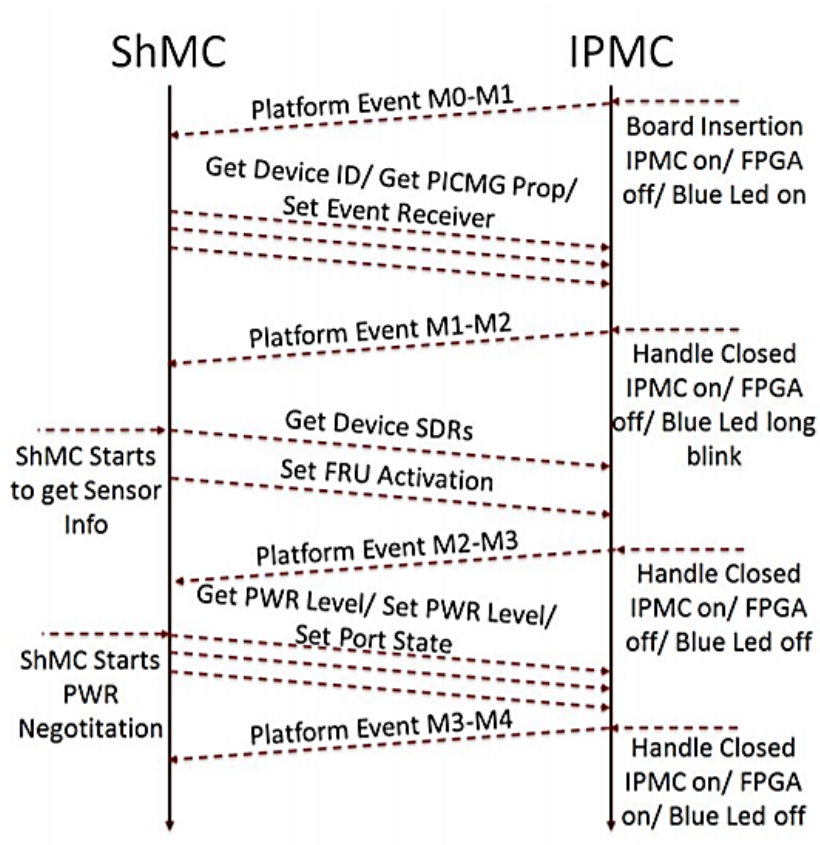
Todo estado de transição da Pulsar 2b no IPMC deve ser informado ao ShMC usando um comando IPMI específico sobre o evento que ocorre na plataforma "*Platform Event*". Este comando informa ao ShMC se a chave (*Handle Switch*) do mecanismo *hot-swap* está aberta ou fechada, os estados atual e anterior, e também a causa da transição.

Existem ainda, diferentes conjuntos de informações que o ShMC deve trocar com o IPMC, através de transações IPMI, dependendo do estado que IPMC se encontra em um instante específico.

2.2.1.1 Processo de ativação

No processo de ativação, a comunicação IPMI inicia com o estado de transição M0-M1, mostrada na Figura 9. Quando este evento de plataforma é relatado, o ShMC sabe qual endereço I²C do *hardware* que o IPMC controla, está disponível no bastidor e começa a obter informações de identificação dessa placa como ID do dispositivo e as propriedades do *Peripheral Component Interconnect Industrial Computer Manufacturers Group* (PICMG).

Figura 9 – Sistema de troca de mensagens IPMI no tempo do processo normal de ativação da placa



Fonte: Ramalho e Paiva (2018).

As respostas do comando mostrarão a identificação, versão de *firmware*, capacidades e suporte desse dispositivo. Somente quando o operador fecha a alça da chave *hot-swap*, a ativação da placa de fato é solicitada.

Nesse caso, o IPMC relata a transição do estado M1-M2 para o ShMC usando um comando de evento de plataforma. Depois disso, o ShMC começa a obter as informações do sensor da placa.

As informações do sensor, dados dos registradores do sensor (SDR), são gravadas em uma memória EEPROM acessível pelo IPMC e solicitada pelo ShMC através do comando IPMI para obter esses dados do sensor.

No entanto, a transição M2-M3 acontece apenas quando o ShMC valida o processo de ativação pelo comando IPMI com a definição de ativação da placa.

No estado M3, a placa está em processamento de ativação, e onde o ShMC e o IPMC fazem uma negociação de energia. O ShMC possui o conhecimento da potência disponível para cada *Slot* físico e o número de placas FRU ativas e inativas dentro do bastidor. Então ele analisa se o bastidor suporta ou não a ativação de mais uma placa FRU no sistema.

Neste estágio, o ShMC faz a solicitação com um comando IPMI ao IPMC para obter o nível de potência que a placa necessita. O IPMC responde, informando o nível constante de consumo de energia desejado.

Então, o ShMC pode validar qual o nível de consumo de energia a placa deve usar naquele momento, com o comando IPMI "*Set Power Level*". O estado M3 é utilizado pelo ShMC para coletar as informações da placa.

O IPMC deve responder à solicitação do ShMC para ler os dados da placa. Essas informações formam um repositório, onde encontram-se quais conexões ponto-a-ponto do bastidor, da interface base e da camada ATCA.

A segurança é obtida desabilitando o uso de alguns canais por meio do comando IPMI "*Set Port State*". Neste ponto, o IPMC pode executar a transição M3-M4, ativar a placa com capacidade total, informando ao ShMC outro *Platform Event*.

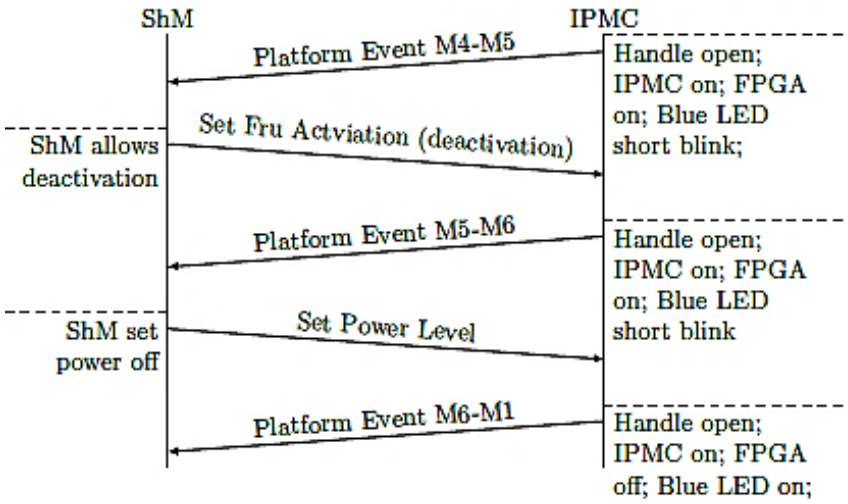
2.2.1.2 Processo de desativação

O processo de desativação da placa, inicia apenas quando o operador abre a alça da chave, *Handle switch*, ou quando o ShMC envia um comando IPMI “*Set FRU Activation*” com a desativação configurada (PICMG 3.0, 2008).

Desta forma, quando as transições dos eventos da plataforma acontecem em uma situação normal de operação, o relatório mostra M4-M5, M5-M6 e M6-M1, até o estado inativo da placa.

O processo de desativação em condições de transições normais da plataforma é mostrado no diagrama de tempo da comunicação IPMI na Figura 10.

Figura 10 – Sistema de troca de mensagens do ShMC com o IPMC no tempo no processo de desativação da placa



Fonte: Paiva (2016).

2.3 PROTOCOLO I²C

O protocolo I²C é um conjunto de regras específicas para a transmissão de sinais de *clock* e de dados seriais no barramento físico I²C. É o barramento físico onde ocorre a transmissão das mensagens IPMI dentro do bastidor ATCA.

A especificação IPMB o define como uma camada física para o controle de acesso de mídia simples, fazendo a sincronização de transmissores e receptores, e controlando a taxa de transferência em *links* multi mestre.

O barramento I²C é um padrão consolidado, implementado por várias tecnologias de circuitos integrados. Por ser um barramento versátil, o I²C é usado em várias arquiteturas de controle (NXP SEMICONDUCTORS, 2014).

A NXP Semicondutores o desenvolveu como um barramento simples e bidirecional com duas conexões para controlar de forma eficiente o circuito interconectado.

Todo barramento I²C é compatível com dispositivos incorporados em uma interface *on-chip* que lhes permite comunicar diretamente uns com os outros por meio do barramento I²C (NXP SEMICONDUCTORS, 2014).

Aspectos relevantes que se pode destacar sobre o barramento I²C são:

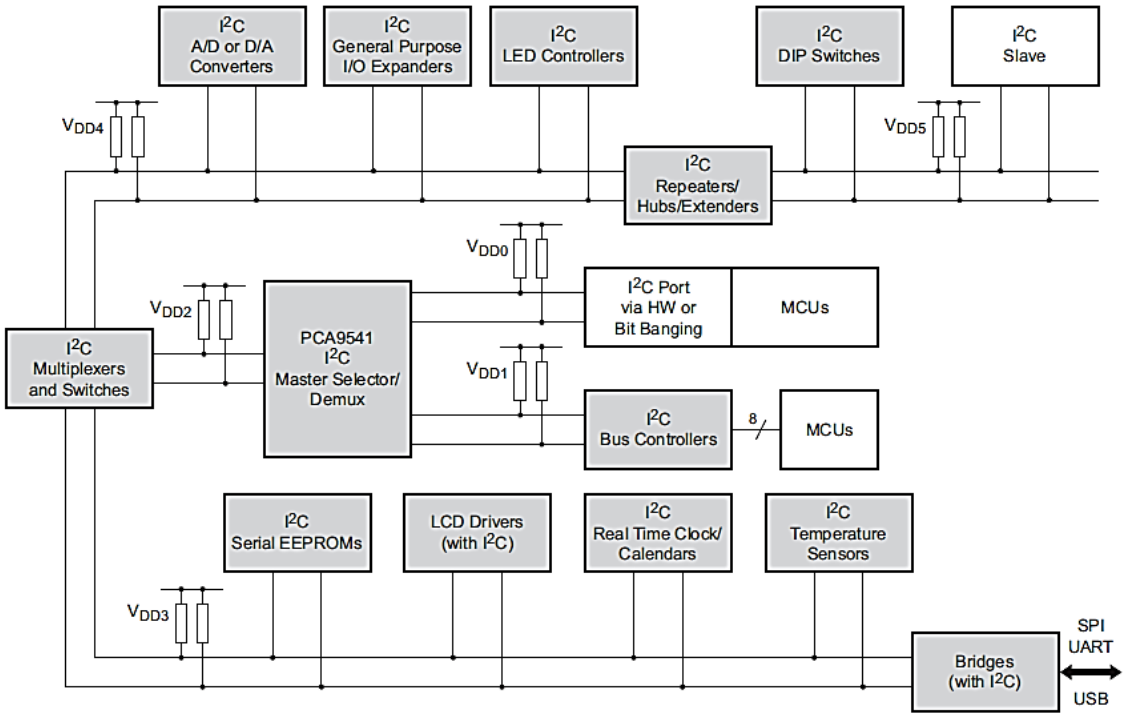
- Necessita apenas de duas linhas físicas para formar barramento, uma para os dados, *Serial Data* (SDA) e uma para o sinal de *clock*, *Serial Clock* (SCL);
- Cada dispositivo conectado ao barramento possui um único endereço para se comunicar;
- As relações que se estabelecem entre os dispositivos conectados ao barramento são simples e do tipo mestre/escravo em todos os momentos. Os dispositivos mestre podem operar como transmissores-mestre ou como receptores-mestre;
- O barramento é multi mestre, mais de um mestre pode controlar o barramento, incluindo a detecção de colisão e arbitração dos sinais enviados para evitar a corrupção dos dados, em casos de dois ou mais mestres solicitarem simultaneamente a transferência de dados;
- É do tipo serial, com 8 bits orientados e taxa de transferência de dados bidirecionais feitas com velocidade de transferências de até 100 Kbps para o modo padrão, até 400 Kbps no modo rápido, até 1 Mbps no modo rápido *Plus* e até 3,4 Mbps no modo de alta velocidade;
- A taxa de transferência de dados de 3,4 Mbps até 5 Mbps, configura o modo Ultrarrápido, e são excepcionalmente unidirecionais, serial e de 8 bits;
- O estágio de filtragem no chip rejeita picos de energia na linha de dados do barramento para preservar a integridade dos dados;
- O número de *Integrated Circuits* (CIs) conectados ao mesmo barramento é limitado pela capacidade máxima do mesmo.

Na Figura 11 é mostrado um tipo de configuração com vários dispositivos conectados ao barramento I²C.

Geralmente inclui algum tipo de controle inteligente, um microcontrolador, circuitos de uso geral como *drivers* de LCD (*Liquid Crystal Display*) e LED (*Light*

Emitting Diode), portas I/O (*Input/Output*) remotas, RAM, EEPROM, *clocks* em tempo real ou conversores A/D e D/A, circuitos orientados à aplicação como circuitos digitais de sintonização e processamento de sinais para sistemas de rádio e vídeo, sensores de temperatura e *smart cards*.

Figura 11 – Configuração de dispositivos conectados ao barramento I²C



Fonte: NXP SEMICONDUCTORS (2014).

2.3.1 Operação do protocolo I²C

Como cada dispositivo I²C é reconhecido por um endereço único (seja um microcontrolador, *driver* LCD, memória ou interface de teclado), ele pode operar como transmissor ou receptor, dependendo de sua funcionalidade. Por exemplo, um *driver* de LCD pode ser apenas um receptor, enquanto uma memória pode receber e transmitir dados.

Além dessa característica, os dispositivos podem ser considerados mestres ou escravos no barramento quando é realizado a transferências de dados. Um dispositivo considerado mestre inicia a transferência de dados e gera os sinais de *clock* para

permitir a transferência, controlando o barramento I²C. Nesse instante, qualquer dispositivo endereçado pelo mestre é considerado escravo na comunicação I²C.

Na Tabela 3 é mostrado as designações dos dispositivos quando utilizam o barramento I²C e os critérios estabelecidos quando o barramento está ocupado.

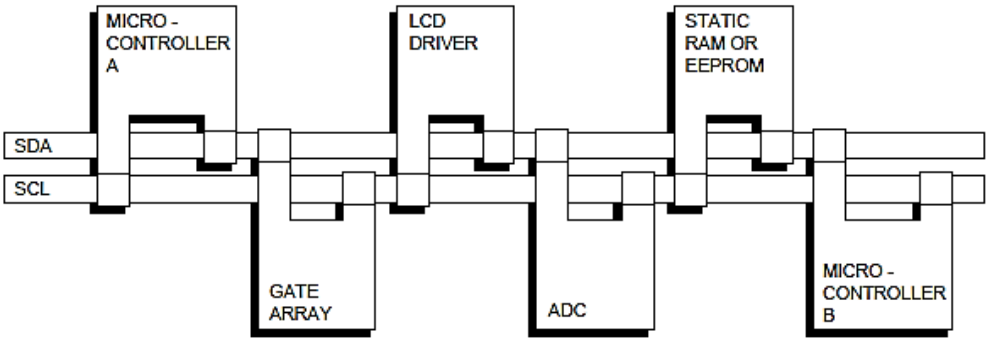
Tabela 3 – Definição da Terminologia do Barramento I²C

Terminologia	Descrição
Transmissor	Dispositivo que envia os dados no barramento
Receptor	Dispositivo que recebe os dados do barramento
Mestre	Dispositivo que inicia a transferência de dados, gera os sinais de *clock* e finaliza a transferência
Escravo	Dispositivo endereçado pelo Mestre
Multi mestre	Mais de um mestre pode tentar controlar o barramento ao mesmo tempo sem corromper a mensagem
Arbitração	Produzido para garantir, caso mais de um mestre tente controlar o barramento simultaneamente, somente um deles poderá fazê-lo e a mensagem vencedora não é corrompida
Sincronização	Produzido para sincronizar os sinais de *clock* de dois ou mais dispositivos

Fonte: Adaptada de NXP SEMICONDUCTORS (2014).

Pela característica multi mestre do barramento I²C, mais dispositivos mestres conectados, são capazes de controlá-lo, porém não simultaneamente. Geralmente são os microcontroladores. A seguir, na Figura 12, tem-se uma configuração com dois microcontroladores e outros dispositivos compartilhando o barramento I²C.

Figura 12 – Configuração de um barramento I²C utilizando dois microcontroladores



Fonte: NXP SEMICONDUCTORS (2014).

Considerando o cenário da Figura 12, suponhamos que se deseja estabelecer uma transferência de dados entre os microcontroladores A e B (µC A e µC B). As relações mestre-escravo e receptor-transmissor são estabelecidas no barramento I²C. Elas não são permanentes e dependem apenas da direção da transferência de dados num dado instante.

Caso o µC A queira enviar dados para o µC B, então a direção da transferência seguirá as seguintes etapas:

1°- O µC A é o mestre da comunicação e endereça o µC B, escravo. Como o µC A envia os dados para o µC B, ele é mestre-transmissor e o µC B é o escravo-receptor. O µC A termina a transferência.

2°- Se o µC A desejar receber os dados do µC B, o µC A que é o mestre, endereça o µC B, escravo, e o µC A torna-se mestre-receptor e recebe dados do µC B que agora é o escravo-transmissor. Novamente o µC A finaliza a transferência.

Em todo o processo o dispositivo mestre, µC A, controla o barramento gerando o sinal de *clock* e finalizando a transferência.

Com a possibilidade de conexão de mais dispositivos controladores ao barramento I²C, pode ocorrer que esses dispositivos tentem iniciar uma transferência de dados ao mesmo tempo, podendo ocasionar a colisão de dados. Neste sentido, para evitar tal evento, existe um critério de arbitragem.

Basicamente é um procedimento que utiliza uma conexão com uma porta lógica *AND* em todas as interfaces I²C dentro do barramento. Quando dois ou mais mestres tentam inserir informações no barramento, o primeiro a produzir o nível lógico '1' vence

e adquire o controle sobre o mesmo, o outro que produziu o nível lógico ‘0’ perde e deve aguardar.

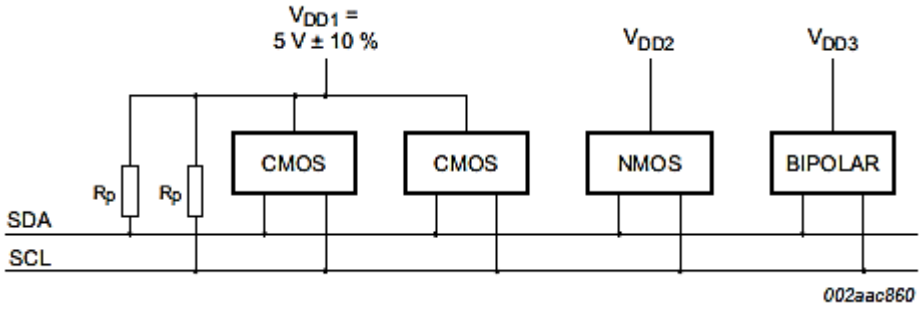
Durante esse critério, os sinais de *clock* são combinados e sincronizados com o sinal de *clock* do dispositivo mestre.

A geração de sinais de *clock* no barramento I²C sempre é responsabilidade dos dispositivos mestre. Cada mestre gera seus próprios sinais de *clock* ao transferir os dados para o barramento.

Os sinais de *clock* do dispositivo mestre podem ser alterados em situações quando são alongados por um dispositivo escravo lento que segura a linha do *clock* ou por outro mestre quando ocorre o critério de arbitragem.

Dispositivos com diferentes tensões também podem compartilhar o mesmo barramento I²C. Na Figura 13 é mostrado a conexão das linhas SDA e SCL do barramento conectadas a uma tensão de alimentação positiva via fonte de corrente ou por um resistor *pull-up*. As tensões V_{DD2} e V_{DD3} são fontes de dispositivos dependentes, NMOS e BIPOLAR.

Figura 13 – Dispositivos com diferentes tensões compartilhando o mesmo barramento



Fonte: NXP SEMICONDUCTORS (2014).

2.3.1.1 Estrutura do byte no barramento I²C

Para a validação dos dados no barramento, a transferência de bits deve seguir certas condições como os dados na linha SDA devem ser estáveis durante o nível lógico alto do *clock* e o estado alto ou baixo da linha SDA só pode mudar quando o

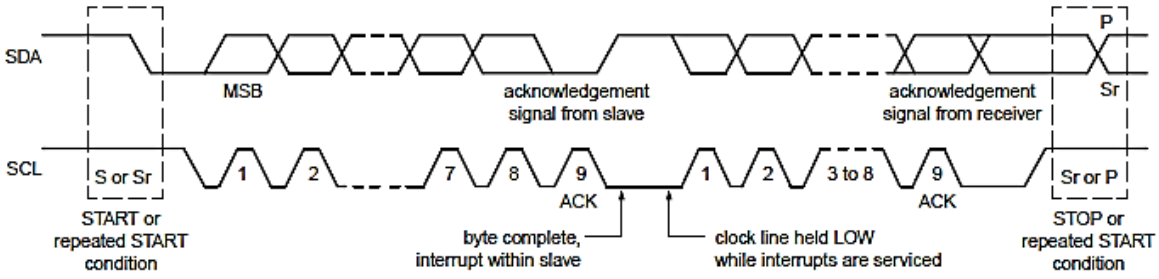
sinal o SCL estiver em nível lógico baixo. Um pulso de *clock* é gerado para cada bit de dado transferido.

Em todas as transferências de dados existe a condição inicial e uma condição de parada. A condição inicial (S) é configurada pela transição do nível lógico Alto para o nível Baixo na linha SDA enquanto a linha SCL está em nível Alto, e a condição de parada (P) pela transição do nível lógico Baixo para Alto na linha SDA enquanto SCL está no nível Alto.

Tais condições são sempre geradas pelo dispositivo mestre e o barramento se encontra ocupado após a condição inicial. Depois de concluída a transferência de dados após a condição de parada, o barramento estará livre.

Todas essas condições podem ser observadas na Figura 14 com o formato do byte na transferência de dados no barramento.

Figura 14 – Formato do byte na transferência de dados no barramento I²C



Fonte: NXP SEMICONDUCTORS (2014).

Cada byte inserido na linha SDA possui oito bits e a quantidade de bytes que pode ser transmitida é irrestrito. Cada byte enviado deve ser acompanhado por um bit de reconhecimento, ACK, e os dados são transferidos pelo primeiro bit mais significativo, MSB.

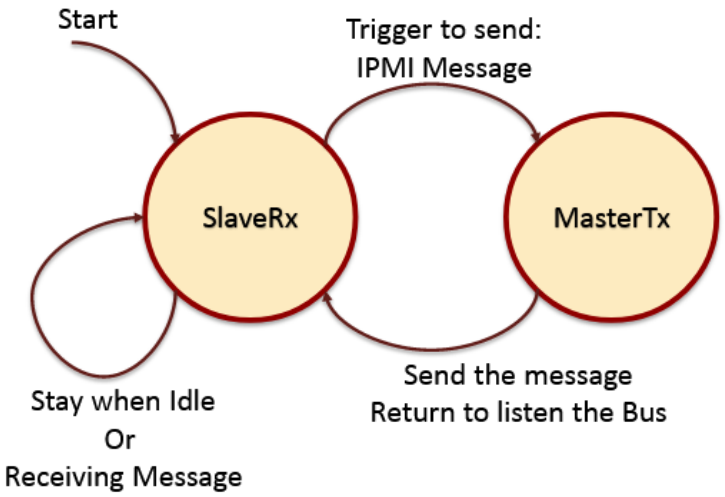
Se um dispositivo escravo não puder receber ou transmitir outro byte de dados completo, ele pode manter a linha SCL em nível Baixo e forçar o mestre a um estado de espera, por exemplo quando o dispositivo escravo necessita terminar a execução de alguma outra função como processar uma interrupção interna.

A transferência continua quando o dispositivo escravo está pronto para receber ou transmitir outro byte de dados, liberando assim a linha SCL.

Para estabelecer uma comunicação IPMI entre o IPMC de uma placa ATCA como a Pulsar 2b e o ShMC do bastidor, isso requer que esses dispositivos suportem uma máquina de estado I²C multi mestre.

Este modo de operação é responsável por mudar o comportamento de cada dispositivo individualmente entre mestre e escravo, quando conectados ao barramento I²C, seguindo a direção das mensagens, conforme ilustrado na Figura 15.

Figura 15 – Máquina de estados para o modo I²C multi mestre implementado no IPMC da Pulsar 2b



Fonte: Paiva e Ramalho (2015).

Quando o IPMC precisa transmitir mensagens IPMI para o ShMC, ele se comunica via barramento I²C no modo *Master* TX. Neste cenário o ShMC atua no modo *Slave* Rx para receber o quadro de dados IPMI.

Quando o ShMC precisa enviar uma mensagem IPMI, o comportamento é oposto, isto é, ele atua como *Master* TX na transmissão enquanto o IPMC atua como *Slave* RX.

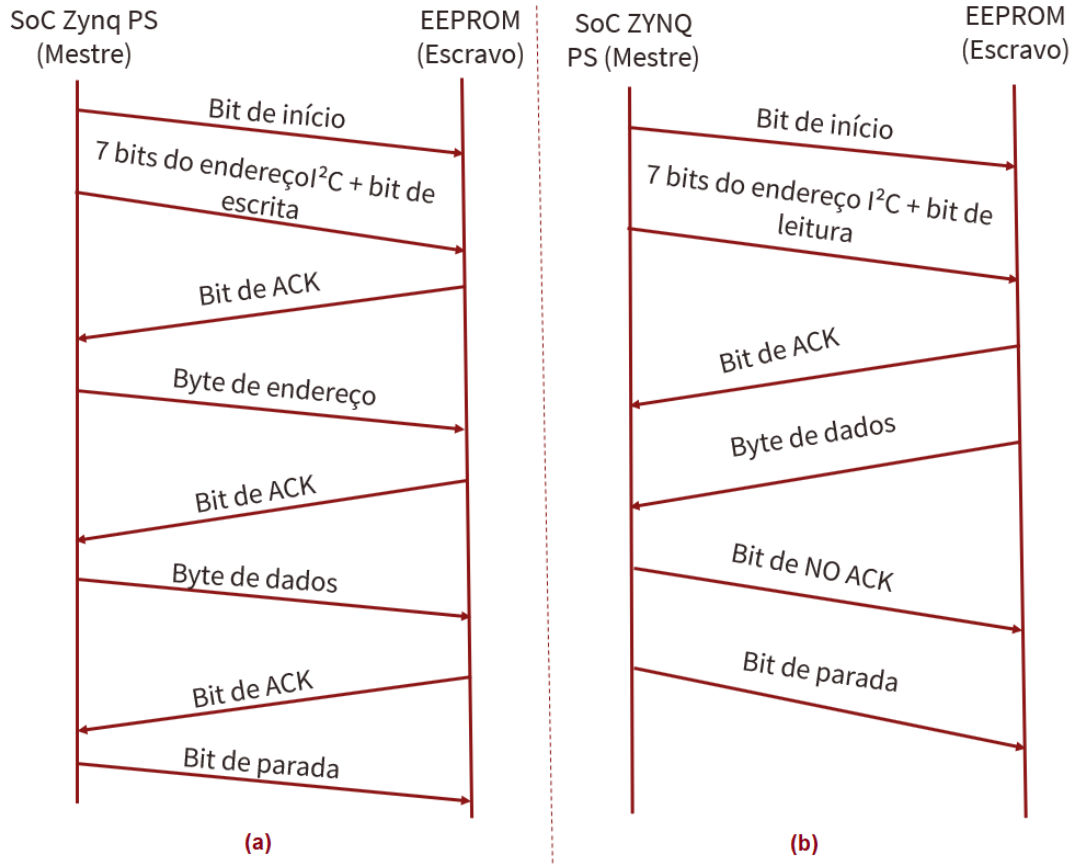
Entretanto no cenário de comunicação proposto para emular essa comunicação IPMI, com a EEPROM, esta funcionalidade não é usada diretamente, pela característica do próprio dispositivo ser passivo no sistema, atuando apenas como escravo na comunicação.

Desta forma apenas o SoC Zynq com o IPMC pode ter essa característica dual e escuta o canal I²C, esperando que a comunicação seja iniciada por outro dispositivo mestre conectado ao barramento.

A sequência da comunicação I²C no barramento segue então o padrão Mestre/Escravo para escrever e ler um conjunto de bytes de dados gravados em posições na memória EEPROM.

Assim, a sequência das mensagens trocadas no barramento no tempo durante um processo de escrita (a) ou de leitura (b) é de acordo com a Figura 16.

Figura 16 – Sequência da comunicação I²C no tempo da escrita (a) e da leitura (b) dos bytes de dados



Fonte: Próprio autor.

Outro aspecto a considerar é com um cenário onde a configuração do *hardware* apresentando possua o canal I²C direto entre o Zynq IPMC e o ShMC, sem passar por um dispositivo I²C *Bus Switch* como ocorre na ZC702, para reduzir um estágio de configuração para apenas acessar o dispositivo I²C desejado.

2.4 SoC Zynq

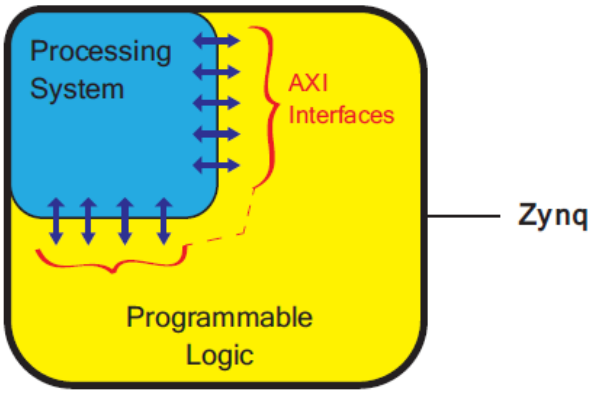
O Zynq é um sistema em *chip*, um circuito integrado capaz de implementar e executar sistemas operacionais inteiros, como o Linux. O SoC Zynq-7000 possui a característica da combinação de um processador ARM Cortex-A9 (CROCKETT, 2014) com dois núcleos e uma *Field-Programmable Gate Array* (FPGA), encontrado na placa da ZC702 da Xilinx.

De forma simplificada dentro de um sistema Zynq encontram-se elementos de memória, interfaces, *clocks,* lógica, aritmética, controle e funções básicas.

Possui a vantagem de ter um baixo custo além do tamanho reduzido, e com sua interface integrada em um mesmo *chip*, possíveis problemas de compatibilidade de interface das partes do sistema de processamento com a lógica programável (FPGA) são evitados.

O Zynq é composto por duas partes principais, o *Processing System* (PS) e a *Programmable Logic* (PL), mostrado na Figura 17. Partes fisicamente separadas dentro do *chip*, porém interconectadas pela interface *Advanced Extensible Interface* (AXI).

Figura 17 – Composição do Zynq de forma simplificada

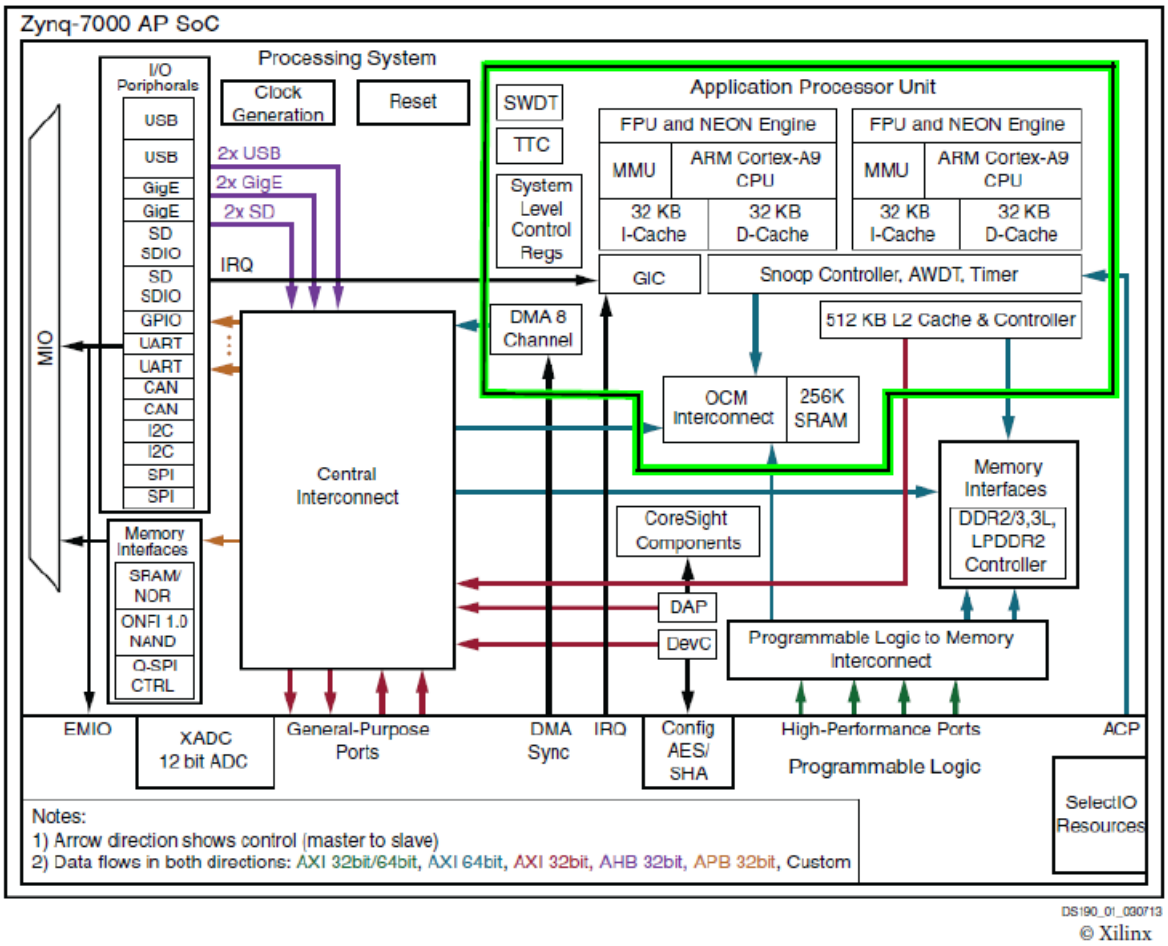


Fonte: Crockett (2014).

Dentro do Zynq PS, além do processador ARM, há um conjunto de recursos de processamento formados pela APU (*Aplication Processing Unit*), memória *cache*, interfaces de memória, interconexões e circuitos para geração do *clock*.

A APU geralmente possui dois núcleos de processamento ARM, e cada um deles possui suas próprias unidades computacionais. Ela possui uma memória *cache* de nível 2, uma *On Chip Memory* (OCM) e interligando essas memórias aos núcleos dos ARMs tem-se a *Snoop Control Unit (S*CU), mostrado na Figura 18.

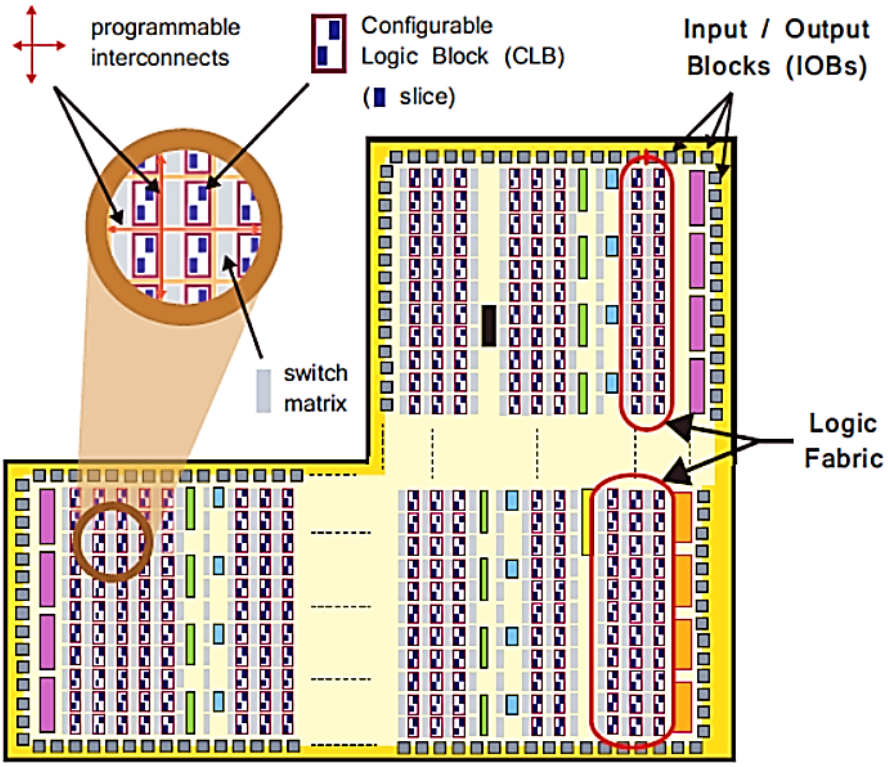
Figura 18 – Diagrama de blocos em destaque a APU em verde do SoC Zynq-7000



Fonte: Crockett (2014).

A parte Zynq PL é baseada nas FPGAS Artix-7 e a Kintex-7 (CROCKETT, 2014). Os principais componentes de uma estrutura lógica da parte PL são mostrados na Figura 19, com o *Configurable Logic Block* (CLB), a fatia (*slices*), o *Look-up table* (LUT), o *flip-flop* (FF), a matriz de comutação e os blocos de entrada e saída (IOBs).

Figura 19 – Tecido lógico do PL do Zynq com os seus componentes



Fonte: Crockett (2014).

O PS é executado independentemente do PL e pode ser feito na inicialização ou na redefinição do sistema.

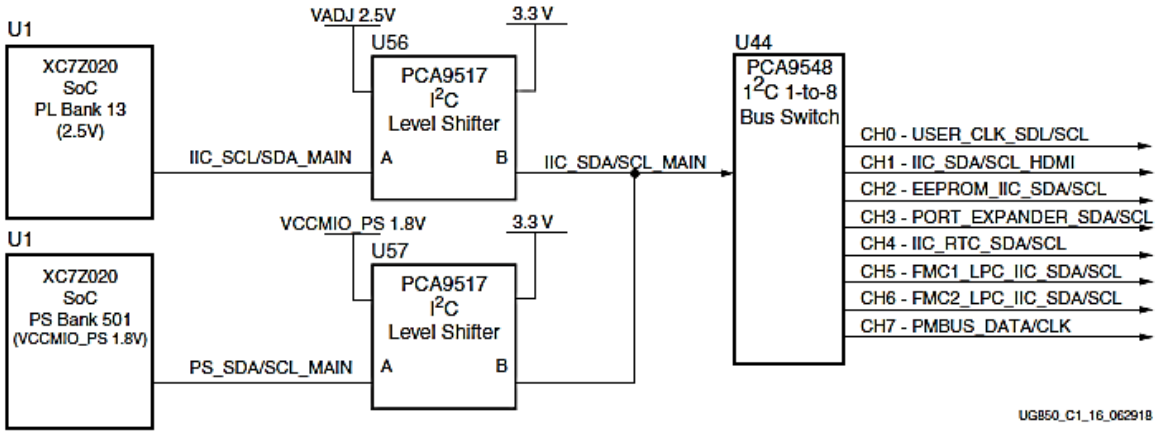
2.5 TOPOLOGIA DO BARRAMENTO I²C DO SoC ZYNQ DA ZC702

A placa de desenvolvimento ZC702 possui um dispositivo Zynq-7000 instalado e diversos recursos como circuitos de alimentação, memória externa, interfaces para programação e comunicação, entradas e saída simples para usuário, botões, LEDs, *switches* e várias outras interfaces e conectores periféricos.

De acordo com a topologia do barramento I²C da placa ZC702 mostrada na Figura 20, a implementação é feita com uma única porta (IIC_SDA/SCL_MAIN) I²C no SoC XC7Z020 (dispositivo Zynq da placa). O canal I²C é instanciado no *Bus Switch* PCA9548 com saída para 8 canais específicos para os dispositivos I²C periféricos da placa. O comutador pode operar com frequência de até 400 kHz.

O endereço do comutador I²C é 0x74 (0b1110100), devendo ser endereçado e configurado na seleção do dispositivo I²C desejado.

Figura 20 – Topologia do barramento I²C da ZC702



Fonte: XILINX (2018a).

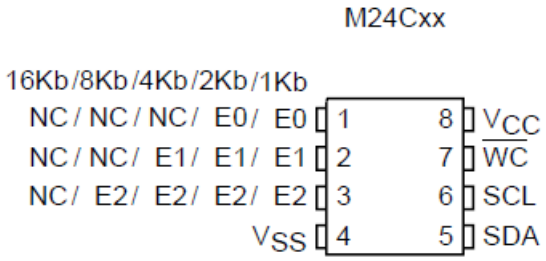
Para a aplicação feita com a placa ZC702, fez-se a configuração do caminho pelo barramento I²C, que pode ser pela parte PS ou PL até o comutador e o canal do endereço I²C do dispositivo desejado.

O dispositivo I²C EEPROM da placa ZC702 está configurado para o canal 2 ou posição I²C 2 no *Bus Switch* e com o endereço I²C 0b1010100.

2.6 DISPOSITIVO I²C EEPROM

Os dispositivos I²C EEPROM são organizados em categorias 2048/1024/512/256/128 x 8 bits (M24C16, M24C08, M24C04, M24C02, M24C01), mostrado na Figura 21.

Figura 21 – Dispositivo I²C EEPROM



Fonte: STMICROELECTRONICS (2004)

O I²C usa uma interface serial de dois fios, compreendendo linha de dados bidirecional e uma linha de relógio.

Os dispositivos portam um código de identificação do tipo de dispositivo de 4 bits (1010) de acordo com a definição de barramento I²C.

O dispositivo se comporta como um escravo no protocolo I²C, com todas as operações de memória sincronizadas pelo relógio serial. Operações de leitura e gravação são iniciadas por uma condição *Start*, gerada pelo mestre do barramento.

A condição inicial é seguida pelo código de seleção do dispositivo e o bit $R\overline{W}$ terminando por um bit de reconhecimento, conforme mostrado na Tabela 4.

Tabela 4 – Código de seleção do dispositivo

	Código de identificação do tipo de dispositivo[1]				Habilitação do *Chip*[2,3]			$R\overline{W}$
	b7	b6	b5	b4	b3	b2	b1	b0
Código de seleção M24C01	1	0	1	0	E2	E1	E0	$R\overline{W}$
Código de seleção M24C02	1	0	1	0	E2	E1	E0	$R\overline{W}$
Código de seleção M24C04	1	0	1	0	E2	E1	A8	$R\overline{W}$
Código de seleção M24C08	1	0	1	0	E2	A9	A8	$R\overline{W}$
Código de seleção M24C016	1	0	1	0	A10	A9	A8	$R\overline{W}$

Nota: 1. O bit mais significativo, b7, é enviado primeiro.

2. E0, E1 e E2 são comparados com os respectivos pinos externos no dispositivo de memória.

3. A10, A9 e A8 representam os bits mais significativos do endereço.

Fonte: Adaptada de STMICROELECTRONICS (2004)

Ao gravar dados na memória, o dispositivo insere um bit de reconhecimento, ACK, durante o tempo de 9 bits, seguindo a transmissão de 8 bits do mestre do barramento.

Quando os dados são lidos pelo mestre do barramento, o barramento mestre reconhece o recebimento do byte de dados do mesmo jeito. As transferências de dados são finalizadas por uma condição *Stop* após a confirmação da escrita ou após um bit de não reconhecimento, NO ACK, no processo de leitura.

2.7 FreeRTOS

O FreeRTOS é um SO em tempo real que permite multitarefas *preemptivas,* ou seja, um esquema de processamento computacional onde o kernel tem o controle do tempo que será usado por cada processo, e possui a capacidade de retomar esse tempo e disponibilizá-lo para outro processo segundo seu esquema de prioridades, ou não *preemptiva* com vários níveis de prioridade de tarefa. Os principais recursos são:

- Comunicação entre tarefas (*tasks*) através de filas (*queue*);
- Temporizadores de *software*;
- Semáforos e exclusões mútuas para sincronização de acesso a recursos compartilhados;
- Diversos algoritmos de gerenciamento de memória;
- Notificação e gerenciamento de eventos.

Ideal para aplicações com sistemas embarcados que usam microcontroladores ou microprocessadores, inclui uma combinação de requisitos *Hard* e *Soft Real Time* (BARRY, 2016).

Os requisitos *Soft Real Time* são os que estabelecem um prazo final, porém a violação deste prazo não tornaria o sistema inútil. Enquanto os requisitos *Hard Real Time* também estabelecem um prazo final, caso haja violação do prazo resultaria em falha absoluta do sistema.

Basicamente é um relógio em tempo real do Kernel sobre qual das aplicações inseridas podem ser construídas para atender os requisitos *Hard Real Time*.

Permitindo que as aplicações sejam organizadas como um conjunto de segmentos independentes (*threads)* para execução, cada *thread* de execução é chamado de *task (tarefa)*. Um exemplo é em um processador com apenas um núcleo, apenas uma *task* pode ser executada a qualquer momento e o Kernel decide qual *task* deve estar executando pelo nível da prioridade atribuída a cada *task* pelo usuário (BARRY, 2016).

Dentro do FreeRTOS é inserido o tempo de processamento para cada tarefa (*tasks*) de uma determinada aplicação. Para um caso simples, o usuário poderia atribuir prioridades mais altas as *tasks* que implementam os requisitos *Hard Real Time* e reduzir as prioridades das *tasks* de *Soft Real Time*, garantindo que as tarefas com aplicações para *hardware* sejam executadas primeiro que as de *software*.

As tarefas são implementadas como funções em linguagem C. Cada tarefa é um pequeno programa que possui uma entrada e normalmente será executado em um *loop* infinito, sem término.

Na Tabela 5, é mostrada a estrutura básica dos códigos de *hardware* dentro do FreeRTOS. Possui qualquer código que tenha qualquer interação direta com o *hardware* do processador e contido na camada dependente de *hardware*.

Tabela 5 – Estrutura básica dos códigos para *hardware* no FreeRTOS

Tarefas do usuário
Código independente de *hardware*
Código dependente do *hardware*

Fonte: Próprio autor.

Não é permitido que as tarefas do FreeRTOS retornem de suas funções de implementação de forma alguma, elas não devem conter uma instrução de 'retorno' e não devem poder executar após o final da função. Se uma tarefa não for mais necessária, ela deve ser explicitamente excluída.

O código-fonte principal do FreeRTOS está contido em apenas dois arquivos C que são comuns a todas as portas FreeRTOS, onde uma porta FreeRTOS é uma

combinação de compilador mais processador. Eles são chamados de tasks.c e list.c, e estão localizados diretamente no diretório FreeRTOS/*Source*, além de outros arquivos (BARRY, 2016).

Uma única definição de função de tarefa pode ser usada para criar qualquer número de tarefas - cada tarefa criada é uma instância de execução separada, com sua própria pilha e sua própria cópia de qualquer variável (pilha) automática definida dentro da própria tarefa (BARRY, 2016).

As tarefas criadas sempre têm um processamento a ser executado, sem a necessidade de esperar por algo, desta forma podem entrar em um estado *Running*. Esse tipo “processamento contínuo” da tarefa possui um uso limitado, porque só podem ser criadas com a prioridade mais baixa. Se forem executados em qualquer outra prioridade, impedirão que tarefas de menor prioridade sejam executadas.

Para tornar as tarefas úteis, elas devem ser reescritas para serem orientadas por eventos. Uma tarefa orientada a eventos torna-se ativa somente após a ocorrência do evento que a aciona e não pode inserir o estado em execução antes que o evento tenha ocorrido.

O gerenciador (*Scheduler)* sempre seleciona a tarefa de maior prioridade que pode ser executada. Tarefas de alta prioridade que não podem ser executadas significam que o *scheduler* não pode selecioná-las e deve, em vez disso, selecionar uma tarefa de prioridade mais baixa que possa ser executada. Portanto, o uso de tarefas orientadas por eventos, significa que as tarefas podem ser criadas com prioridades diferentes sem as tarefas de maior prioridade, privando as outras do tempo de processamento.

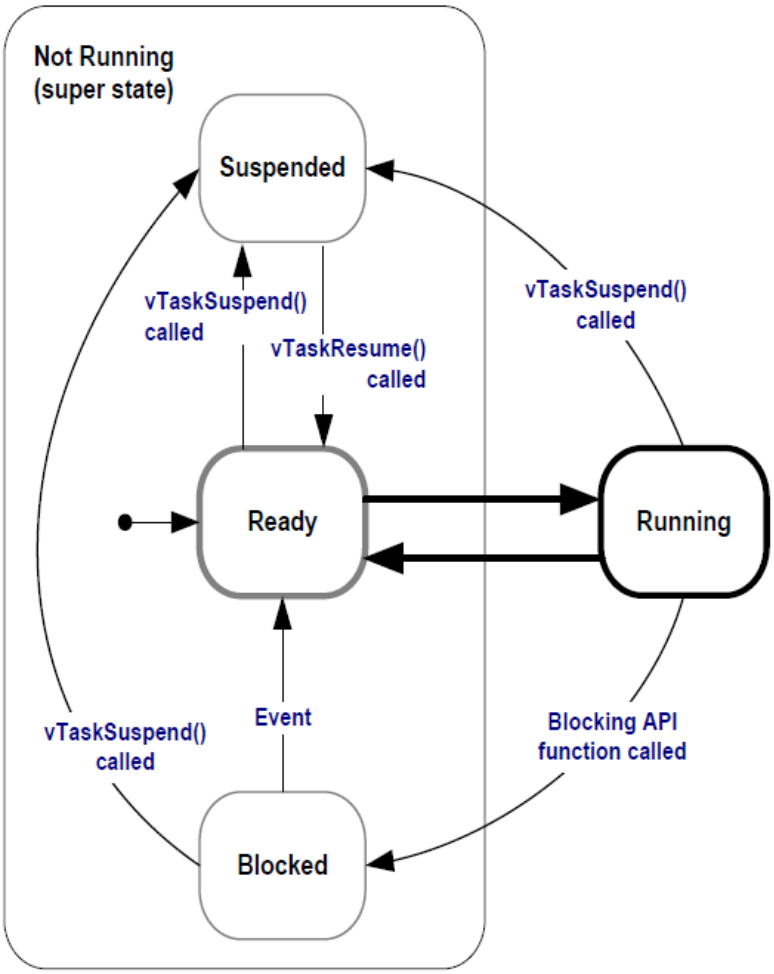
Os estados e sub estados de execução de uma tarefa dentro do FreeRTOS são mostrados na Figura 22. O estado bloqueado (*Blocked state*), significa que uma tarefa está aguardando um evento e permanece neste estado, que é um sub estado do estado “*Not Running”*.

Dois tipos diferentes de eventos que podem levar as tarefas para o estado *Blocked*:

- eventos temporais (relacionados ao tempo) - o evento sendo um período de atraso que expira ou um tempo absoluto sendo atingido, e

- eventos de sincronização - onde os eventos se originam de outra tarefa ou interrupção. Os eventos de sincronização cobrem uma ampla variedade de tipos de eventos.

Figura 22 – Máquina de estados e sub estados de uma tarefa no FreeRTOS



Fonte: Barry (2016).

Filas, semáforos binários, contagem de semáforos, exclusões mútuas, *mutexes* recursivas, grupos de eventos e notificações diretas para tarefas podem ser usados para criar eventos de sincronização. Exemplos de como construir esses recursos podem ser encontrados em (BARRY, 2016).

Ainda, é possível que uma tarefa bloqueie um evento de sincronização com um tempo limite, bloqueando efetivamente os dois tipos de evento simultaneamente.

O estado *Suspended* é um sub estado de "*Not Running*". Tarefas neste estado não estão disponíveis para o *scheduler*. O único caminho para o estado *Suspended* é através de uma chamada para a função da API vTaskSuspend() e a única saída é através de uma chamada para as funções da API vTaskResume() ou xTaskResumeFromISR() no código. A maioria das aplicações não usa esse estado.

No estado *Ready* as tarefas estão no estado *Not Running*, não estão bloqueadas nem suspensas, mas disponíveis para serem executadas, contudo não estão atualmente no estado *Running*.

O diagrama de estado "*Not Running*" e "*Running*" simplificado que inclui todos os sub estados (*Blocked, Suspended, Ready*) são apresentados na Figura 22. As tarefas são implementadas para a transição entre o sub estado *Ready* para o estado *Running*, destacados pelas linhas em negrito.

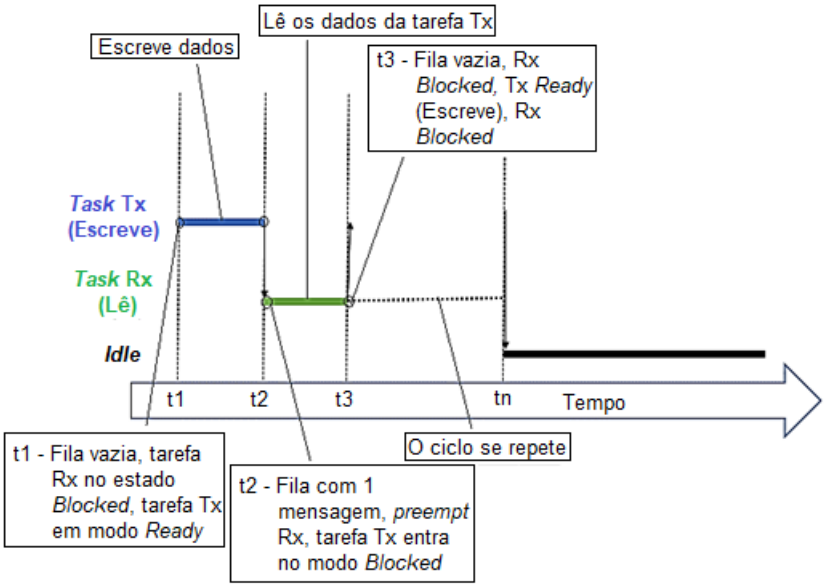
Algoritmos com *Scheduler*, descrevem um recurso opcional chamado *time slicing*. Para poder selecionar a próxima tarefa a ser executada, o *scheduler* deve executar no final de cada intervalo de tempo, uma interrupção periódica e rápida, chamada de "*Tick Interrupt*". O comprimento deste *time slicing*, é definido pela frequência de interrupção, configurada por uma constante de configuração de tempo de compilação (configTICK_RATE_HZ), definida dentro do FreeRTOSConfig.h (BARRY, 2016).

Por exemplo, se configTICK_RATE_HZ estiver definido como 100 Hz, o intervalo de tempo será 10 milissegundos. O tempo entre duas interrupções é chamado de "*Tick period*". Um *time slicing* é igual a um período de *Tick (interrupção)*.

Na Figura 23 é mostrado um exemplo com uma aplicação do processo de escrita e leitura na memória EEPROM inseridas como tarefas do FreeRTOS.

Com o auxílio do código xiicps_eeprom_polled_example.c para a comunicação I²C do Zynq com o dispositivo EEPROM e com a estrutura do código do helloworld.c do FreeRTOS, foi possível a criação da tarefa Rx responsável pela função de leitura dos dados do barramento I²C e a tarefa Tx responsável pela função de escrita de dados no barramento I²C.

Figura 23 – Sequência de execução no tempo expandida das tarefas no FreeRTOS



Fonte: Próprio autor.

A tarefa Tx grava um item específico, conjunto de bytes, na EEPROM memória e a tarefa Rx é responsável pela leitura desses bytes de dados escritos na EEPROM.

Para esta dinâmica, foi dada uma prioridade menor para tarefa Tx e maior prioridade para a tarefa Rx, mas a condição para a tarefa Rx ser executada, é tarefa Tx colocar um item na fila, portanto, a tarefa Rx estará em um estado bloqueado e antecipa a tarefa Tx quando houver um item na fila.

Na sequência de execução das tarefas, a tarefa Tx escreve uma mensagem em 1 segundo (t1-t2) e a tarefa Rx executa a leitura no segundo seguinte (t2-t3), quando o *timer* expira (tn), o processo de leitura/escrita na EEPROM termina, as duas tarefas são excluídas e o sistema FreeRTOS executa uma tarefa *Idle,* ociosa.

3 METODOLOGIA

Neste capítulo, descreve-se os materiais utilizados no teste, o *software*, diagramas para implementação do projeto de *hardware* e o cenário de teste implementado na placa ZC702.

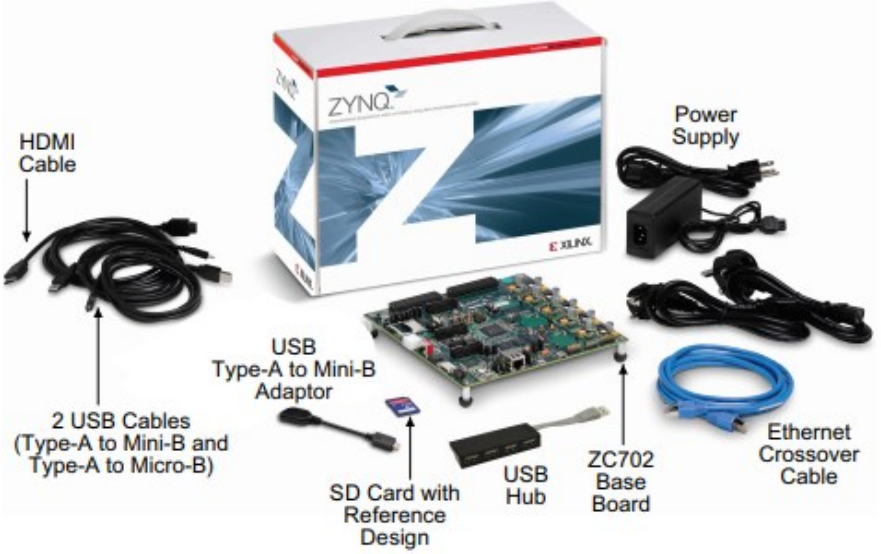
3.1 MATERIAIS

Os materiais utilizados no experimento para viabilizar a comunicação IPMI com o SoC Zynq da placa ZC702 foram:

- Vivado 2018.2 da Xilinx;
- Computador;
- ZC702 *Evaluation kit*;
- Fonte de alimentação;
- Osciloscópio digital.

Na Figura 24, é mostrado o kit de desenvolvimento da ZC702 da Xilinx, contendo a placa e seus dispositivos, cabo de alimentação, cabos USB para conexão da placa ao computador, cartão SD dentre outros itens.

Figura 24 – *kit* de desenvolvimento da ZC702



Fonte: XILINX (2017).

3.2 FERRAMENTAS

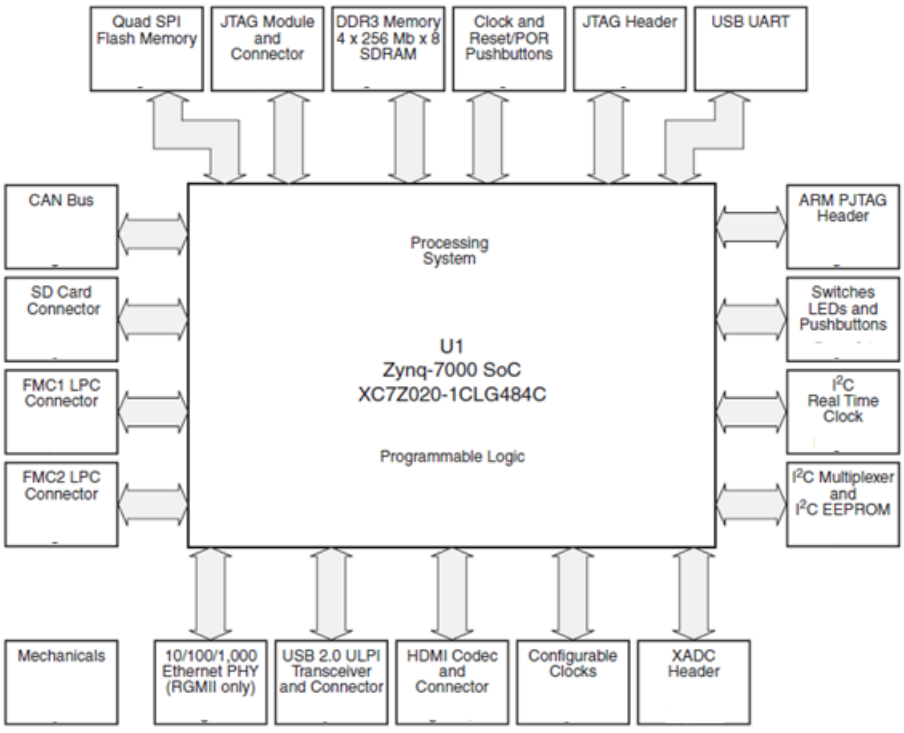
A ZC702 é uma placa desenvolvida pela Xilinx com um sistema de processamento SoC ZYNQ-7000 e com inúmeros periféricos para serem utilizados e configurados, além de fornecer um ambiente de *hardware* para desenvolvimento e avaliação de projetos.

Possui recursos comuns a muitos sistemas de processamento incorporados, incluindo memória de componentes DDR3 (*Double Data Rate 3*), uma Ethernet PHY de modo triplo, I/O de uso geral e duas interfaces UART.

Suporta outros recursos com a utilização de cartões mezanino FPGA VITA-57 (FMC) quando conectado a um dos dois conectores FMC (*FPGA Mezzanine Card*) do LPC (*Low Pin Count*).

Na Figura 25 é possível ver o diagrama de blocos contendo todos os recursos presentes na placa ZC702.

Figura 25 – Diagrama de blocos da placa ZC702



Fonte: XILINX (2018a).

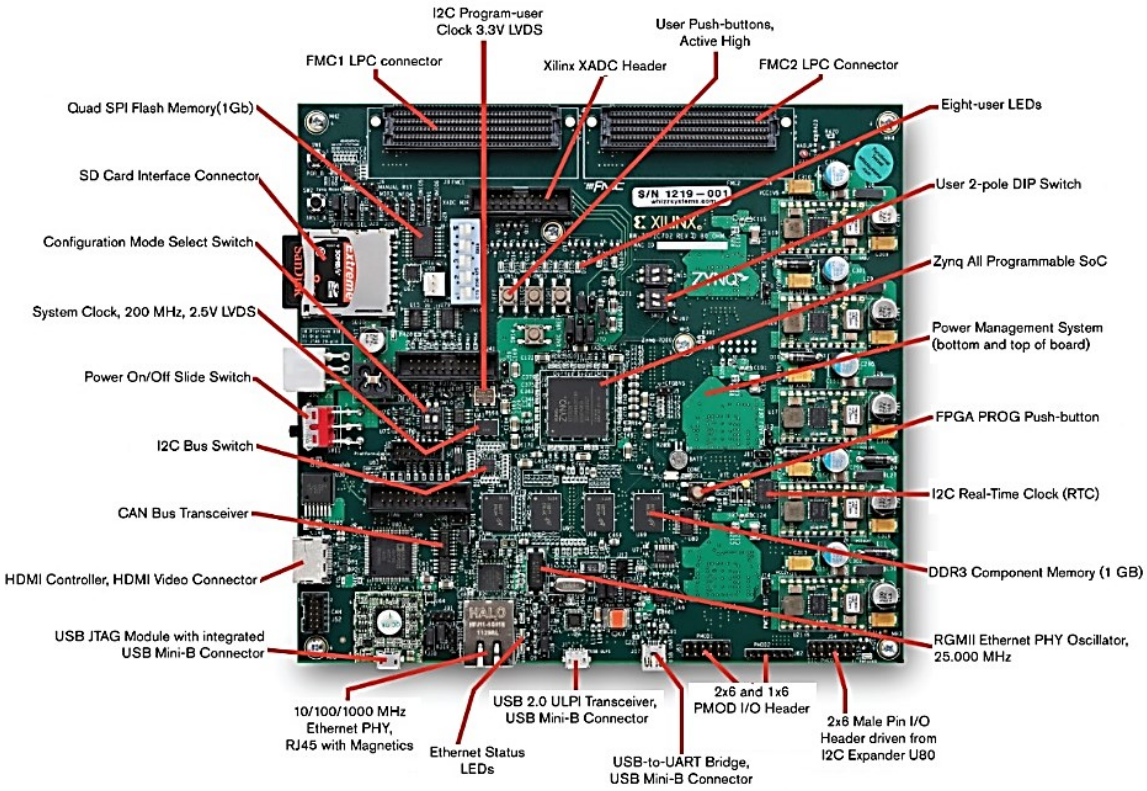
Para aplicação desenvolvida nesta pesquisa, os recursos utilizados da placa foram:

- SoC Zynq XC7Z020-1CLG484C;
- 128 MB de memória flash Quad SPI;
- Interface USB JTAG (*Joint Test Action Group*) usando um módulo Digilent;
- ponte USB-para-UART;
- Barramento I²C multiplexado para memória EEPROM M24C08 (1 kB)

No experimento que será apresentado, foi realizado o cenário da comunicação do SoC Zynq com a M24C08 EEPROM (1 kB) via barramento I²C, utilizando o FreeRTOS.

Na Figura 26 é mostrado a vista superior da placa ZC702 com a indicação de cada componente físico presente na placa e seu correspondente.

Figura 26 – Placa ZC702 com seus componentes

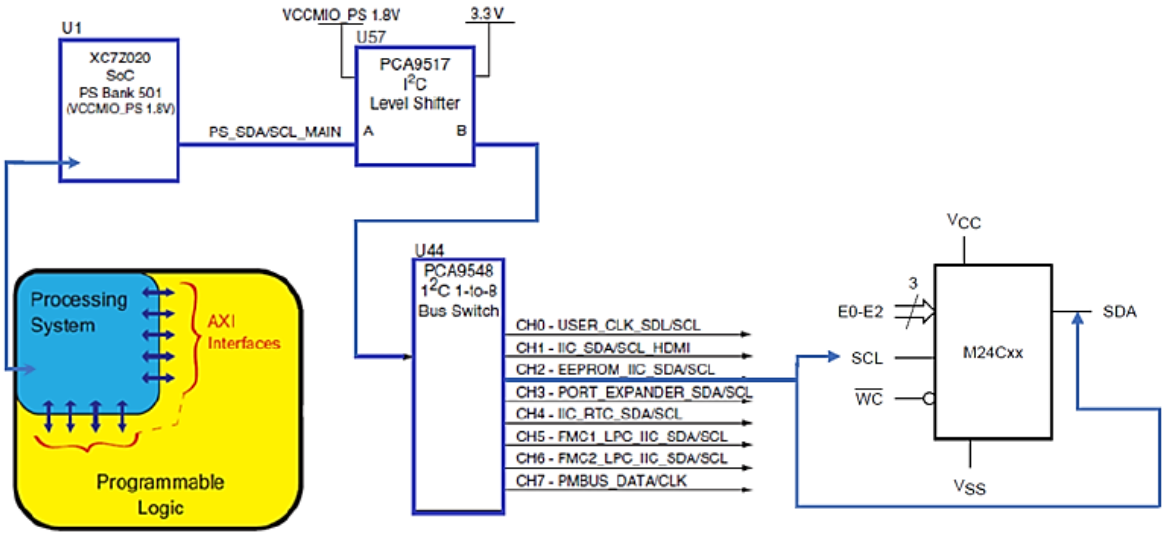


Fonte: XILINX (2018b).

O modo de acesso ao dispositivo I²C EEPROM presente na ZC702, via barramento I²C foi pela parte Zynq PS, em azul, mostrado na Figura 27.

A conexão passa pelo dispositivo I²C *Bus Switch*, então é necessário configurar *driver* I²C na implementação de *software* para selecionar o endereço do canal que dá acesso ao dispositivo I²C EEPROM.

Figura 27 – Arquitetura simplificada do SoC Zynq com a topologia do barramento I²C da ZC702 e acesso ao dispositivo I²C EEPROM



Fonte: Adaptado de Crockett (2014), XILINX (2018a) E STMICROELECTRONICS (2004).

3.2.1 Vivado design suíte

Para o desenvolvimento do projeto, utilizou-se o programa *Vivado design suite*, versão 2018.2, edição *Web*PACK, que é uma versão gratuita. Inclui todo o *software* principal para criação de projetos com o Zynq, contém um ambiente de Desenvolvimento Integrado (IDE) para projeto de *hardware*, o *Software Development Kit* (SDK), compilador *GNU Compiler Connection* (GCC) para desenvolvimento de *software* e Vivado *Simulator* para verificação.

As ferramentas mais importantes utilizadas deste pacote são Vivado IDE e o SDK, servindo como base para o *design* do sistema de *hardware* e *software*.

No Vivado IDE, cria-se toda a parte do sistema de *hardware* do projeto SoC, com o processador, as memórias, periféricos, interfaces externas e conexões do barramento, ao mesmo tempo que interage com outras ferramentas do Vivado *design*

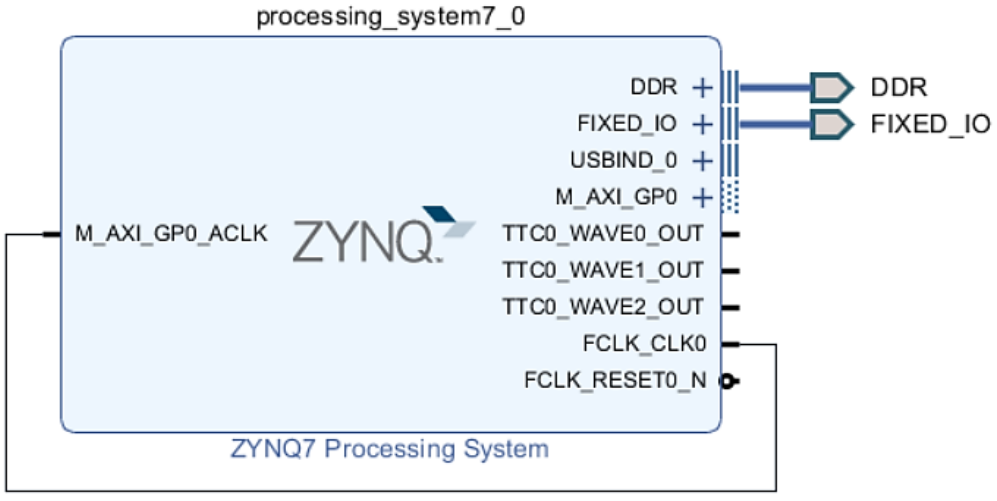
suíte, inclui também recursos para integração e armazenamento de IPs (*Intellectual Property*), possibilitando a reutilização do *design* em outras aplicações.

No SDK encontram-se todo o conjunto do *design* de *software* baseado na plataforma Eclipse (CROCKETT, 2014), com suporte a todos os IPs da Xilinx, suporte a bibliotecas GCC para extensões ARM e NEON (extensão avançada *Single instruction multiple data* (SIMD)), usado em processadores ARM, também chamado de *Media Processing Engine* (MPE) usando as linguagens C e C++ e ferramentas para depuração e criação de perfil.

Para o desenvolvimento da aplicação, o primeiro passo é criar o IP do Zynq PS no Vivado IDE, gerar um arquivo *bitstream* FPGA (arquivo que contém as informações de programação da FPGA, programada usando um fluxo de bits específico para que se comporte como uma plataforma de *hardware* incorporada), exportar o *hardware* e enviar para a ferramenta SDK.

O projeto desenvolvido para ZC702 utilizou o *design* do IP *Integrator* composto pelo bloco Zynq PS. O bloco e as portas externas são mostrados no diagrama da Figura 28.

Figura 28 – Diagrama de bloco do componente IP *Integrator* do Vivado IDE



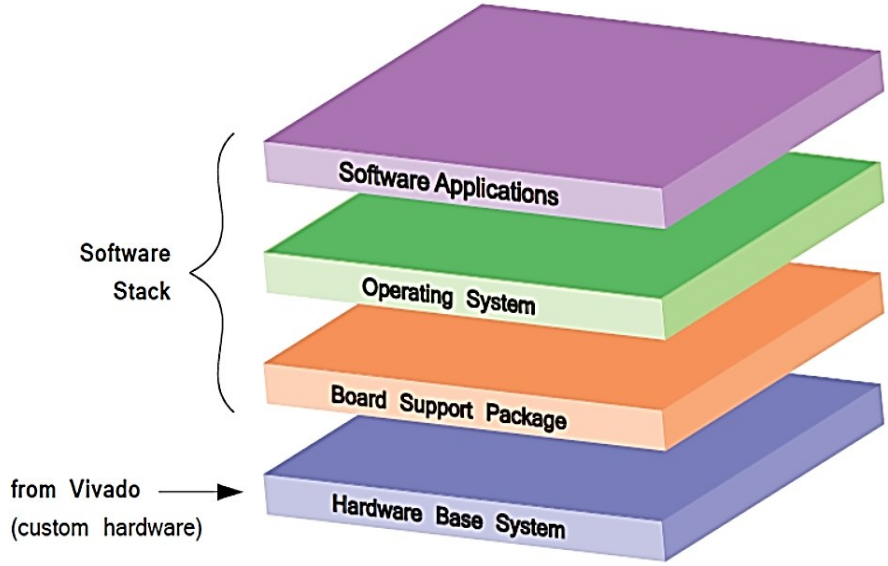
Fonte: Próprio autor.

Após a validação do diagrama de blocos IPs e a geração do arquivo *bitstream*, o projeto pode ser exportado para o SDK para o projeto de *software* ser desenvolvido.

E, seu desenvolvimento é realizado com base nos dados da aplicação de *hardware* feitos no Vivado IDE.

Basicamente, o sistema de *software* pode ser considerado como uma pilha ou um conjunto de camadas construído sobre a camada *Hardware Base System*. Toda a estrutura desenvolvida no Vivado IDE e no SDK é mostrada na Figura 29.

Figura 29 – Pilha de desenvolvimento do projeto no Vivado *design suite*

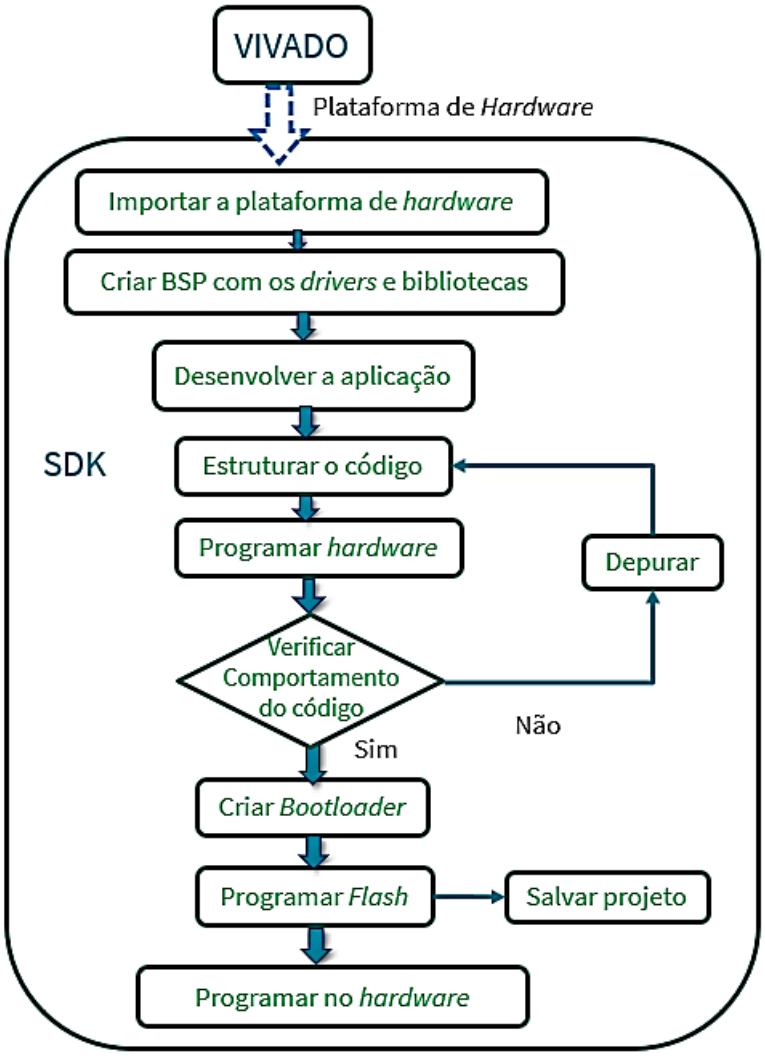


Fonte: Crockett (2014).

A camada *Board Support Package* (BSP) está diretamente acima da camada de *Hardware Base System* e onde encontram-se o conjunto de *drivers* e funções de baixo nível que serão usados na camada chamada *Operation System* (OS) para comunicar-se com o *hardware*. Na camada OS, as aplicações de *software* são executadas, essas aplicações estão na camada superior da pilha, no nível mais alto de abstração do *software*.

O diagrama de desenvolvimento no SDK é mostrado na Figura 30. O projeto exportado do Vivado IDE representa o *hardware* personalizado no qual o *software* é baseado, chamado de plataforma do *hardware*.

Figura 30 – Diagrama de blocos da implementação do projeto no SDK



Fonte: Próprio autor.

3.2.1.1 Requisitos mínimos para testes

Para execução do Vivado *design suite*, é importante que o sistema operacional do computador do usuário conceda permissões de gravação para todos os diretórios que contenham arquivos do *design*. A especificação de *hardware* do computador que será utilizado é significativa.

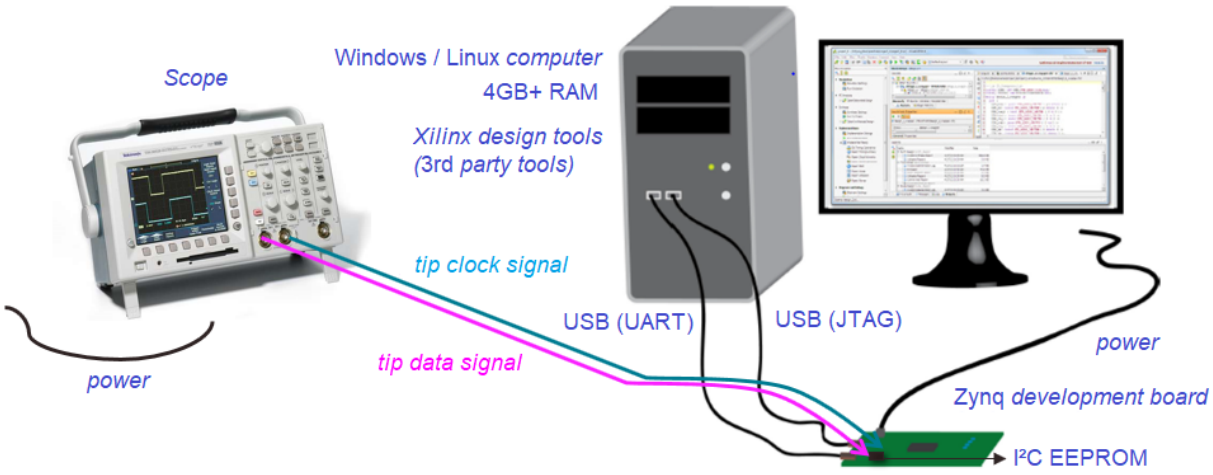
A Xilinx fornece algumas indicações dos requisitos de memória por dispositivo necessário, particularmente os sistemas operacionais de 32 bits não são muito adequados para aplicação. Basicamente os requisitos que o computador deve ter são

pelo menos 4 GB de RAM para os três dispositivos menores do ZYNQ e um processador *dual-core*.

Na Figura 31 é mostrado o esquemático básico da configuração de *hardware* para o teste. O computador deve ter uma porta USB para programar o Zynq por um JTAG e outra porta USB para a comunicação do PC-Zynq através do aplicativo UART para o Terminal, para a depuração dos projetos. O usuário deve ter a placa de desenvolvimento com o Zynq para testar os projetos.

Neste esquemático tem-se um osciloscópio digital com as pontas de prova acessando o barramento I²C conectado ao dispositivo I²C EEPROM da placa para captura dos sinais de dados e *clock*.

Figura 31 – Esquemático básico da bancada para o teste



Fonte: Próprio autor.

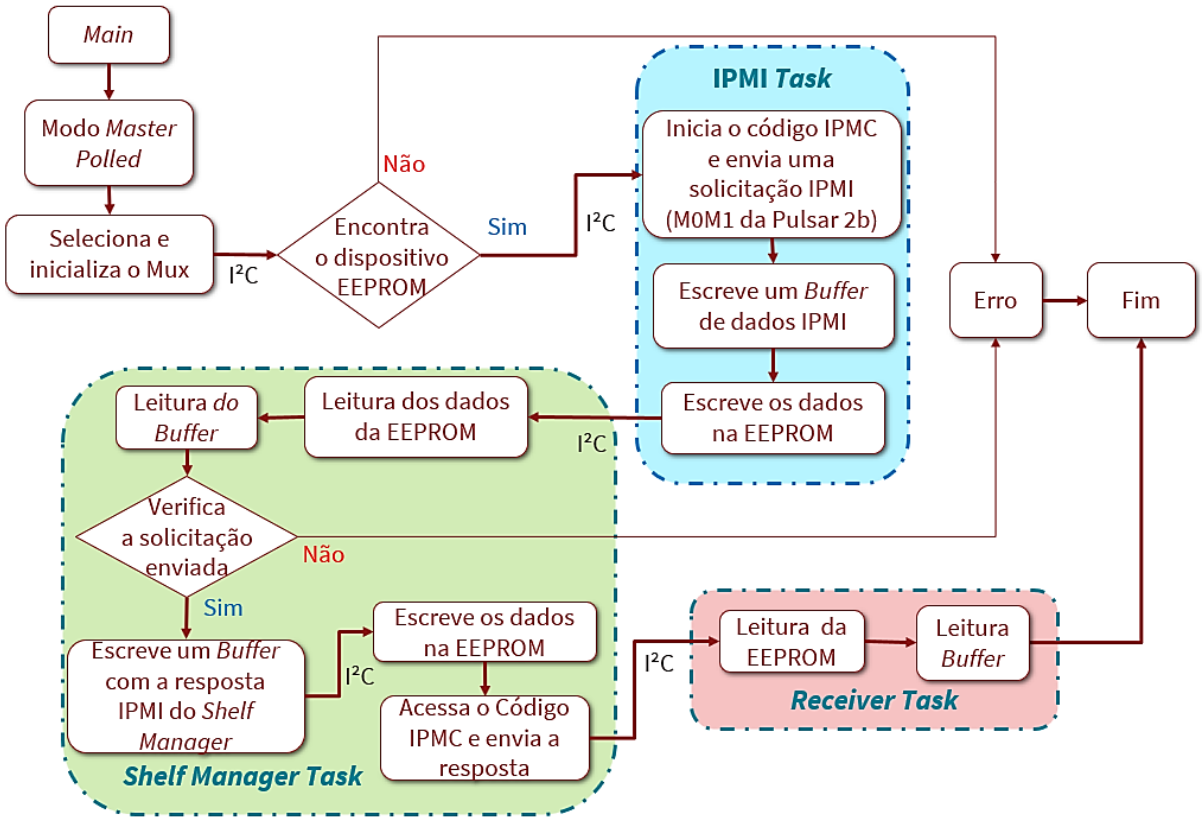
4 IMPLEMENTAÇÃO

Com os arquivos exportados do Vivado IDE para o SDK, o desenvolvimento do projeto foi construído no ambiente do FreeRTOS para possibilitar a criação de tarefas a serem gerenciadas.

O código IPMC da Pulsar 2b e a comunicação I²C entre o Zynq e a EEPROM foram inseridos dentro de tarefas no FreeRTOS. Como o sistema operacional é um Kernel em tempo real no qual as aplicações podem ser construídas para atender aos requisitos de tempo real, a aplicação foi organizada como um conjunto de tarefas para execução.

Os códigos fornecidos pela Xilinx em implementações FreeRTOS foram modificados para testar o barramento I²C e estabelecer uma comunicação entre o Zynq PS e a EEPROM. A implementação desta aplicação é mostrada na Figura 32.

Figura 32 – Fluxograma do teste da comunicação I²C implementado em um sistema operacional em tempo real



Fonte: Próprio autor.

Toda vez que o *driver* I²C do Mux precisa ser acessado, o modo *Master Polled* é utilizado. Verificações são feitas durante o processo para realização da escrita e leitura correta dos dados IPMI na EEPROM.

Durante a comunicação I²C entre o Zynq e a EEPROM, as tarefas IPMI, *Shelf Manager* e *Receiver* incluem as funções de escrita e leitura e são executadas pelo FreeRTOS na APU do Zynq. Elas possuem a mesma prioridade de execução, contudo comandadas por filas para que sejam executadas em sequência, criando uma dependência entre elas. Uma temporização de 5 s até finalizar a implementação.

Na Figura 33 é mostrado o *setup* experimental para depuração no SDK, utilizado para o teste da implementação na placa ZC702.

Figura 33 – *Setup* experimental com a ZC702

Fonte: Próprio autor.

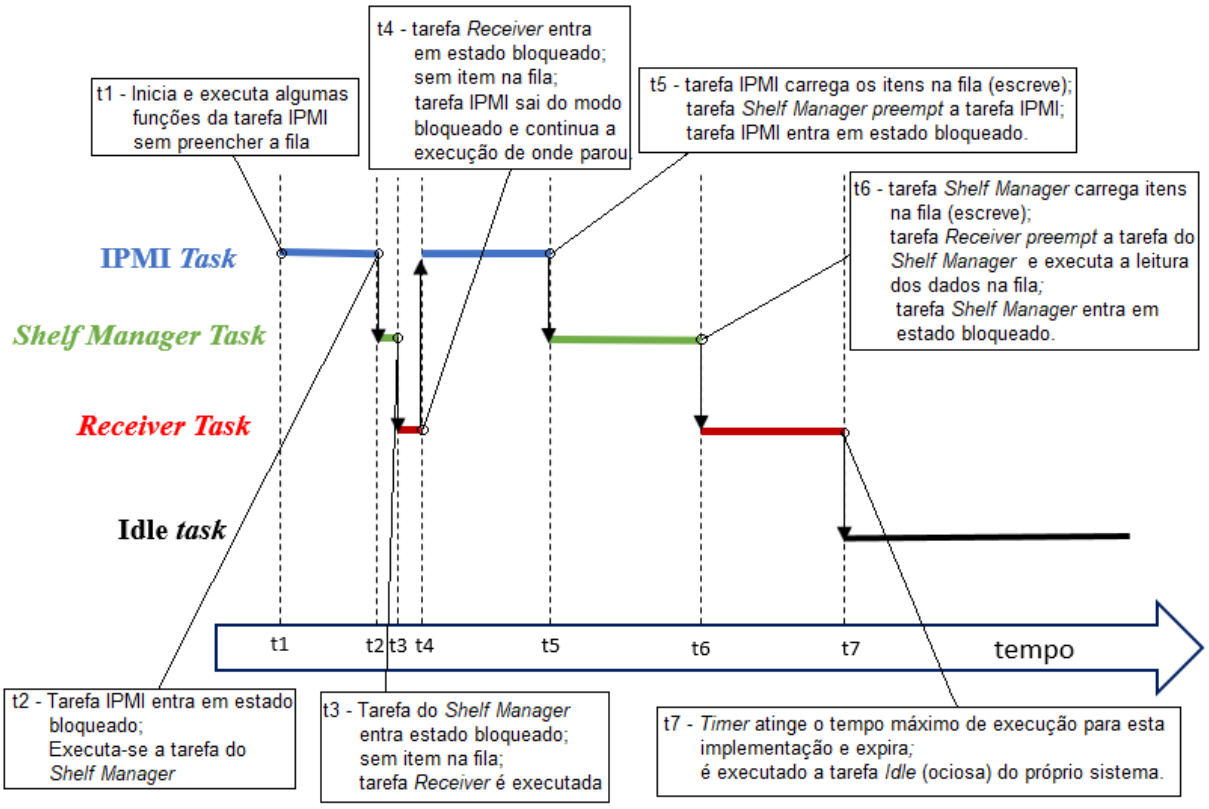
Utilizando o terminal do SDK, para visualização do resultado do teste, verificou-se o *status* da comunicação entre o SoC Zynq (mestre) com o dispositivo I²C EEPROM (escravo) e a dinâmica da troca de mensagens no barramento I²C emulando o primeiro estágio da comunicação IPMI entre o IPMC e o ShMC.

4.1 RESULTADOS

Na Figura 34 é mostrado o diagrama expandido no tempo da execução das tarefas implementadas no FreeRTOS. A solicitação IPMI é escrita na memória EEPROM na tarefa com o mesmo nome (t4-t5) e a tarefa do *Shelf Manager* (emulado) executa a leitura desta solicitação, faz a verificação e em seguida executa a escrita da resposta IPMI na memória para esta transição (t5-t6).

No instante (t6-t7) é executado a leitura dos dados de resposta IPMI pelo IPMC. Após 5 segundos o tempo expira para a execução desta implementação, as tarefas são finalizadas e descartadas, desta forma o sistema operacional passa executar uma tarefa ociosa.

Figura 34 – Diagrama expandido de tempo de execução das tarefas no FreeRTOS

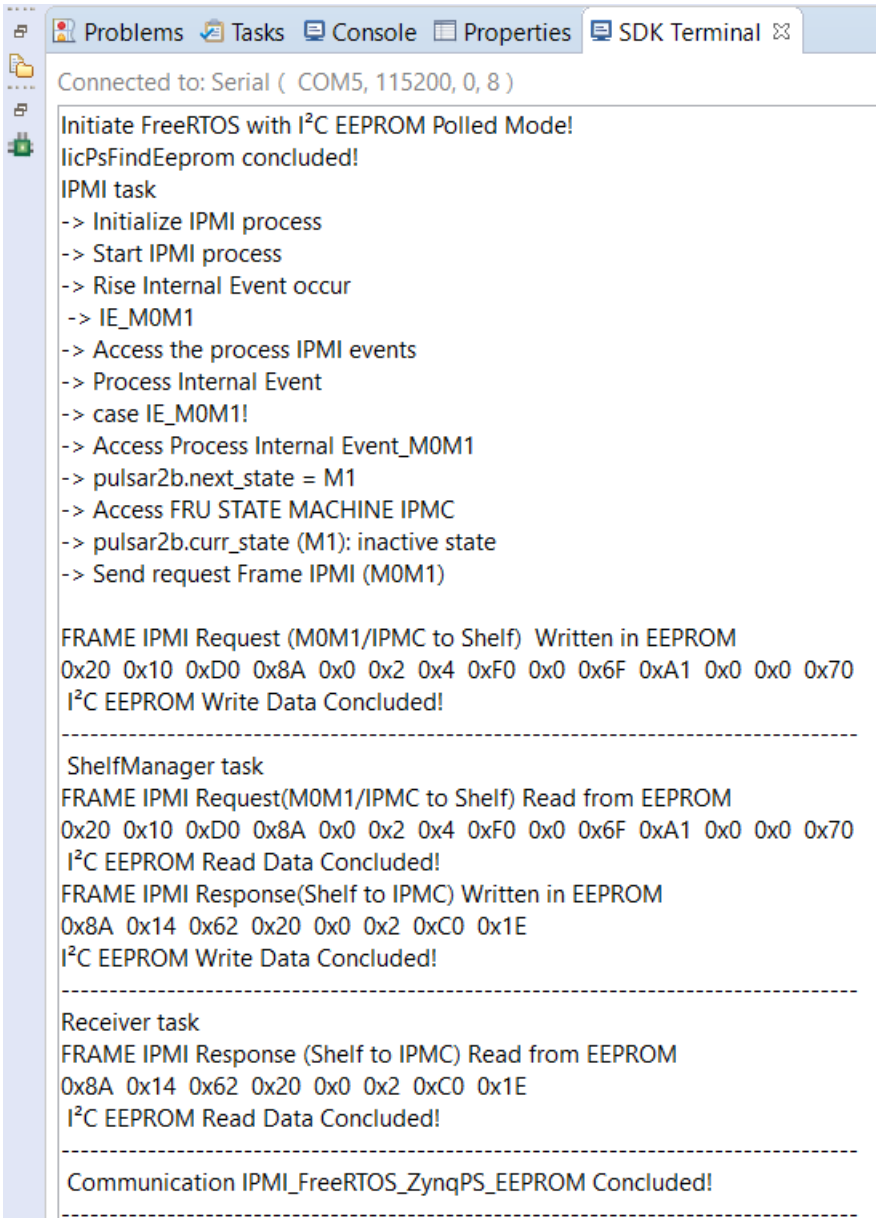


Fonte: Próprio autor.

Como o sistema de tarefas é gerenciado por filas e elas possuem a mesma prioridade de execução o próprio *scheduler* do sistema operacional faz com que cada tarefa entre em modo de operação, como pode ser visto nos instantes (t1-t2), (t2-t3) e

(t3-t4). Contudo pelo fato das tarefas *Shelf Manager* e *Receiver* dependerem de dados enviados na fila, entram em estado bloqueado até que seja satisfeita esta condição. A partir do instante t4, o controle pelas filas na execução das tarefas em sequência é atingido. O resultado da depuração da implementação no FreeRTOS utilizando a plataforma da ZC702 é mostrado na Figura 35 no terminal do SDK.

Figura 35 – Resultado no terminal do SDK



Fonte: Próprio autor.

Com a inicialização do barramento, a escrita e a leitura dos dados IPMI na memória, verificação da mensagem transmitida e da resposta da comunicação IPMI,

executada pelas tarefas. Na Tabela 6 é mostrado o padrão das mensagens de solicitação e de resposta IPMI e o significado dos valores hexadecimais dentro da comunicação IPMI entre os controladores de plataforma para uma transição de M0 para M1 da placa Pulsar 2b.

Tabela 6 – Estrutura da comunicação IPMI da transição M0M1 da placa Pulsar 2b sendo executada no Zynq da ZC702

Byte	Solicitação IPMI do IPMC		Resposta IPMI do ShMC do bastidor		
1	Endereço I²C do dispositivo que irá responder	0x20	1	Endereço I²C do dispositivo que fez a solicitação	0x8A
2	*Network Function_LUN*	0x10	2	*Network Function_LUN*	0x14
3	*Header Checksum 1*	0xD0	3	*Header Checksum 1*	0x62
4	Endereço I²C do dispositivo solicitante	0x8A	4	Endereço I²C do dispositivo que responde	0x20
5	Comando com a sequência numérica	0x00	5	Comando com sequência numérica	0x00
6	Identificador do comando	0x02	6	Identificador do comando	0x02
-	-	-	7	*Completion code* (complementar da solicitação)	0xC0
7:N	Pacote de dados	{0x04, 0xf0, 0x00, 0x6f,0xa1, 0x00, 0x00}	8:N	Pacote de dados	-
N+1	Verificação da mensagem com um *Checksum 2*	0x70	N+1	Verificação da mensagem com um *Checksum 2*	0x1E

Fonte: Próprio autor.

Os dados IPMI de solicitação advém da própria configuração e programação IPMC da Pulsar 2b, com mínimas alterações para funcionar no sistema do SoC Zynq e no SO FreeRTOS.

Para os dados IPMI de resposta do *Shelf Manager* para IPMC, uma tarefa implementada do sistema operacional executa a função do ShMC, pois não há a presença de um bastidor ATCA acessível neste cenário.

Para se conhecer os dados IPMI transmitidos no barramento I²C em um bastidor ATCA e para conferência, foram utilizados os dados medidos por um analisador, chamado *Wireshark*.

Esta ferramenta compreende os aspectos estruturais do protocolo, como o IPMB-0 composto pelos canais IPMB-A e IPMB-B. Capaz de classificar e filtrar os pacotes, é útil para criar pequenos conjuntos de dados e solucionar problemas com o desenvolvimento de avaliação da comunicação.

É utilizada para rastrear comunicações de rede, fornecendo as informações sobre o que está dentro de um pacote, mostrado na Figura 36.

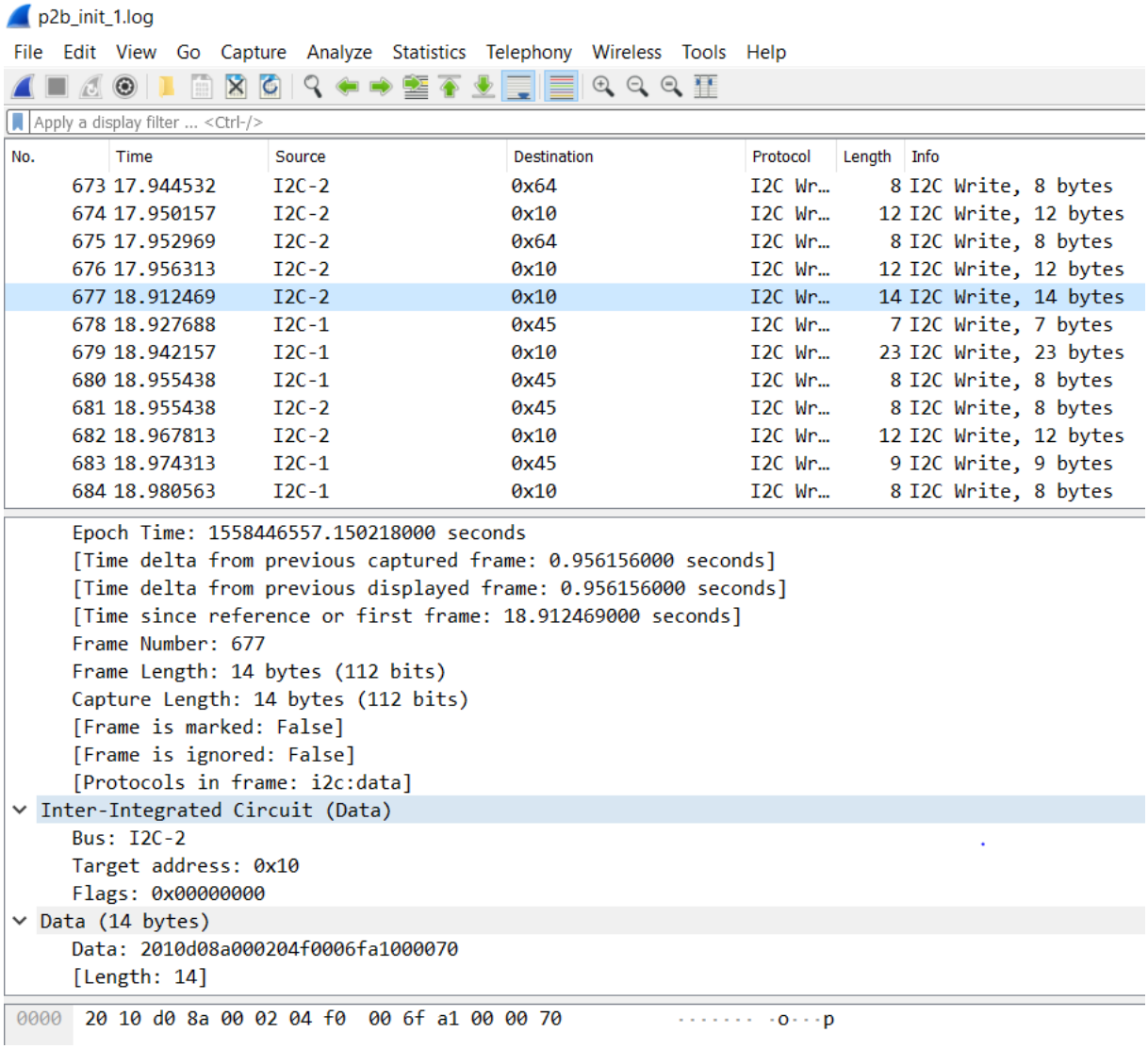
No teste da Figura 36, destaca-se o pacote IPMI 677, contendo a primeira solicitação que o IPMC (0x45) faz ao ShMC (0x10) para a primeira transição de estado (M0M1) da plataforma Pulsar 2b, valores que não consideram o bit de escrita.

Porém, dentro do pacote esses valores consideram o bit de escrita, sendo o IPMC (0x8A) e o ShMC (0x20).

Os valores para o comando da transição M0M1 da plataforma FRU Pulsar 2b possuem os valores de acordo com a especificação de (INTEL HEWLETT-PACKARD NEC DELL, 2013), em que:

- (0x04) é a versão do pacote configurado para o IPMI 2.0;
- (0xF0) é o tipo do sensor, FRU *hot swap*;
- (0x00) Número do Sensor, cuja escolha é arbitrária desde que tenha um valor único;
- (0x6F) Tipo de evento, dado pré-definido;
- (0xA1) Evento do dado 1, A (código OEM) 0x1 = estado atual M1 com a FRU inativa;
- (0x00) Evento do dado 2, transição normal de estado da FRU com estado anterior M0,
- (0x00) Evento do dado 3, identificação (ID) do dispositivo FRU.

Figura 36 – Ambiente do analisador Wireshark, descrição do pacote IPMI de solicitação



Fonte: Próprio autor.

Embora este analisador não seja capaz de descrever todos os campos dos vários comandos IPMI, ainda é capaz de fornecer informações interessantes como destino, duração, tempo e dados brutos em formato hexadecimal para cada pacote.

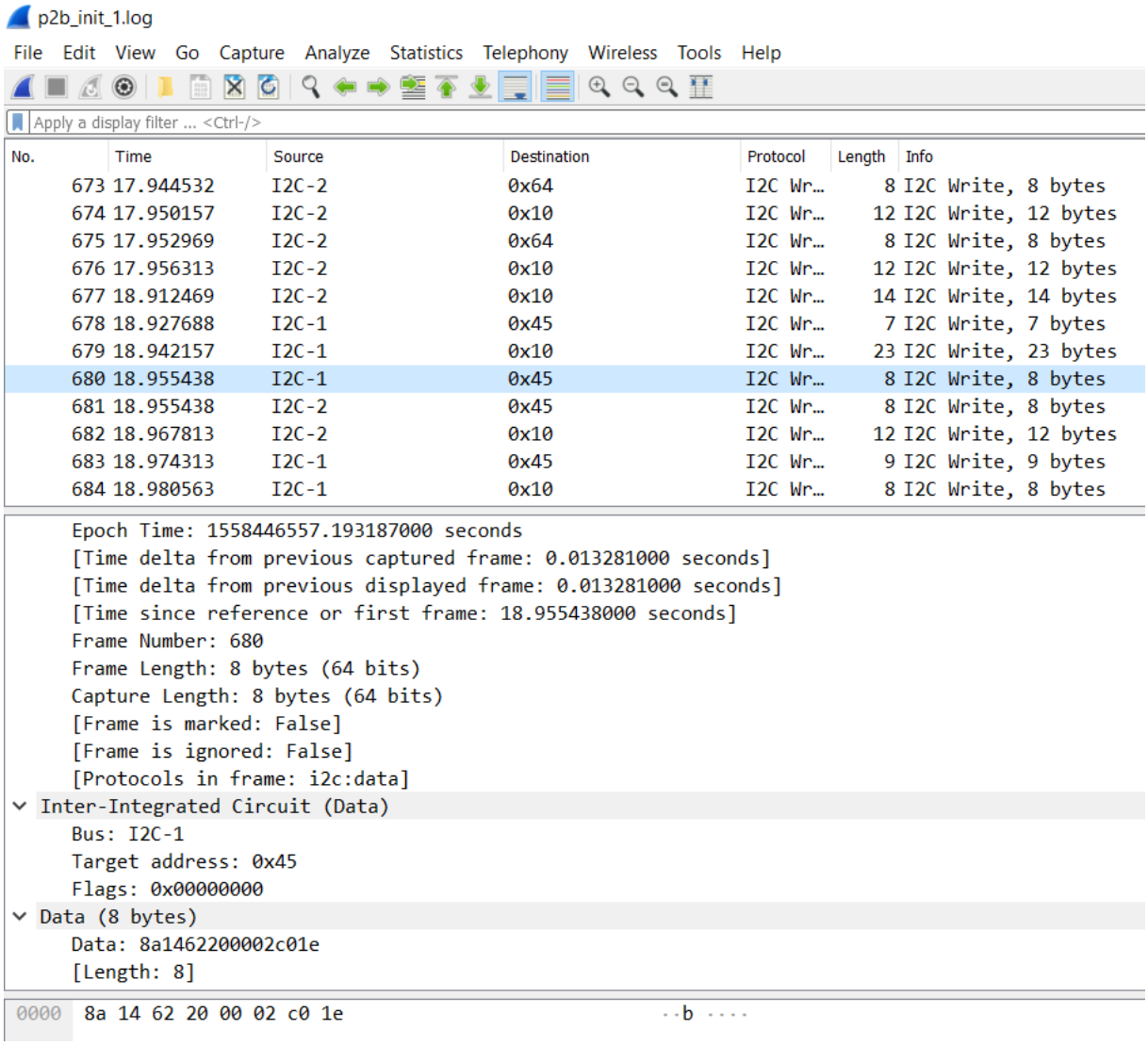
Na Figura 37 é mostrado, em destaque, o pacote IPMI 680, contendo a resposta do ShMC (0x10) a solicitação do IPMC (0x45) para a primeira transição de estado (M0M1) da plataforma Pulsar 2b.

Nesta situação, o *Shelf Manager* não autoriza a transição da plataforma Pulsar 2b no bastidor naquele instante, identificado pelo dado C0h dentro da mensagem IPMI

enviada ao IPMC, significando que o canal do ShMC está com o *link* ocupado (INTEL HEWLETT-PACKARD NEC DELL, 2013).

Uma nova tentativa será feita pelo IPMC do mesmo comando para esta transição, porém com uma outra sequência numérica, até que o ShMC autorize este evento dentro do bastidor ATCA.

Figura 37 – Ambiente do analisador Wireshark, descrição do pacote IPMI de resposta



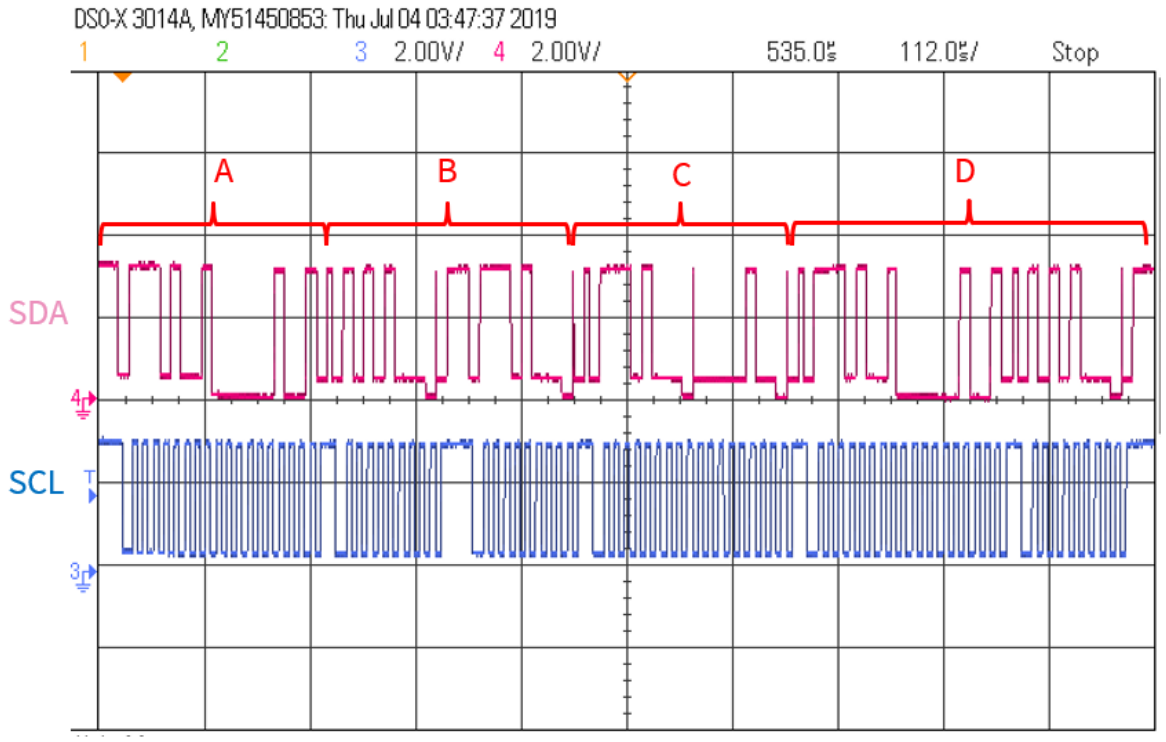
Fonte: Próprio autor.

A seguir, são apresentados os resultados de medição no osciloscópio dos dados sendo transmitidos no barramento I²C da ZC702.

O Zynq é sempre mestre na comunicação com a EEPROM (escravo), portanto comanda o barramento I²C com o sinal de *clock*, funcionando como IPMC ou como ShMC emulado.

Na Figura 38, encontra-se o primeiro estágio desta comunicação I²C entre o Zynq PS até dispositivo I²C EEPROM, mostrando a configuração e seleção dos dispositivos da placa ZC702 para ocorrência da comunicação no barramento I²C.

Figura 38 – Comunicação I²C com a verificação e seleção dos dispositivos Mux e I²C EEPROM da placa ZC702



Fonte: Próprio autor.

Pela topologia do barramento da plataforma tem-se o dispositivo I²C EEPROM instanciado no multiplexador PCA9548 (Mux) da placa, portanto este elemento também é selecionado e configurado.

Observa-se a sequência dos bytes enviados no barramento, com um conjunto de 9 bits em sequência, os primeiros 7 bits do endereço do dispositivo seguido do bit de escrita ou leitura, depende do processo, terminando com o bit de ACK ou NO ACK e o envio do pacote de dados também com a mesma estrutura. E de acordo com o protocolo I²C a quantidade de dados enviado no barramento é ilimitada.

Na Tabela 7 é mostrado a sequência dos bytes enviados no barramento I²C com a descrição detalhada dos campos da Figura 38.

Tabela 7 – Descrição do primeiro sinal enviado no barramento I²C para configuração dos dispositivos

Campos	Descrição dos sinais no barramento I²C da ZC702
(A)	Bit de início + Endereço do Mux (0x74) + Bit de leitura (1) + Bit de ACK (0) + Canal da EEPROM no Mux (0x02) + Bit de escrita (0) + Bit de NO ACK (1) + Bit de parada;
(B)	Bit de início + Endereço I²C EEPROM (0x54) + Bit de escrita (0) + Bit de ACK (0) + Bit de parada + Bit de início + Endereço do Mux (0x74) + bit de escrita (0) + Bit de ACK (0);
(C)	Bit de início + Endereço do Mux (0x74) + bit de escrita (0) + Bit de ACK (0) + Canal da EEPROM no Mux (0x02) + Bit de escrita (0) +Bit de ACK (0) + Bit de parada;
(D)	Bit de início + Endereço do Mux (0x74) + bit de leitura (1) + Bit de ACK (0) + Canal da EEPROM no Mux (0x02) + Bit de escrita (0) +Bit de NO ACK (1) + Bit de parada + Bit de início + Endereço da EEPROM (0x54) + Bit de escrita (0) + Bit de ACK (0) + Bit de parada.

Fonte: Próprio autor.

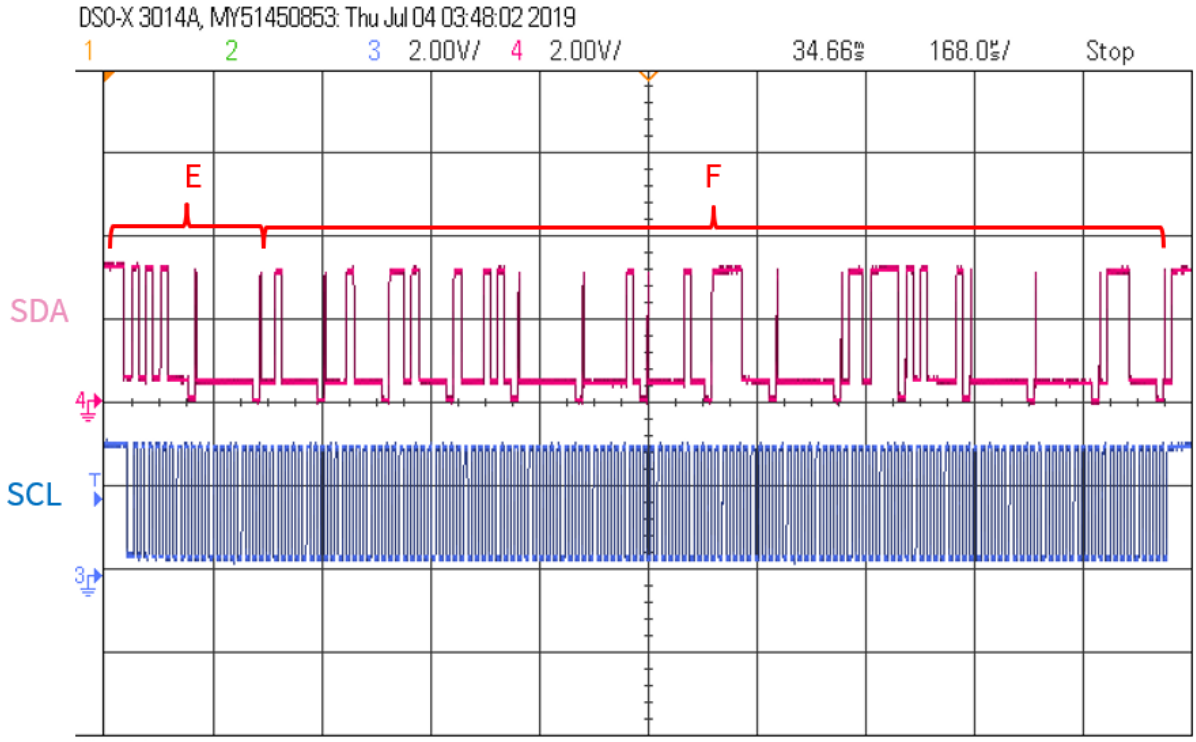
Os campos descritos na Tabela 7 referem-se a parte inicial da implementação no FreeRTOS que verifica a existência dos dispositivos, Mux e I²C EEPROM, na plataforma.

A configuração é feita na sequência e uma nova verificação é realizada para constatar que os dispositivos foram selecionados. E, assim iniciar o processo de escrita e leitura dos dados na memória.

Para etapas de verificação dos dispositivos o processo de leitura é realizado e para configuração tem-se o processo de escrita no barramento I²C.

Na Figura 39 é mostrado o processo de escrita dos dados na memória EEPROM (Escravo) com os dados de solicitação IPMI enviados pelo Zynq PS (Mestre) na função IPMC.

Figura 39 – Comunicação I²C com o processo de escrita da solicitação IPMI na memória



Fonte: Próprio autor.

Na Tabela 8 tem-se a descrição detalhada da sequência dos bytes da mensagem IPMI enviados no barramento I²C com o processo de escrita na memória da Figura 39.

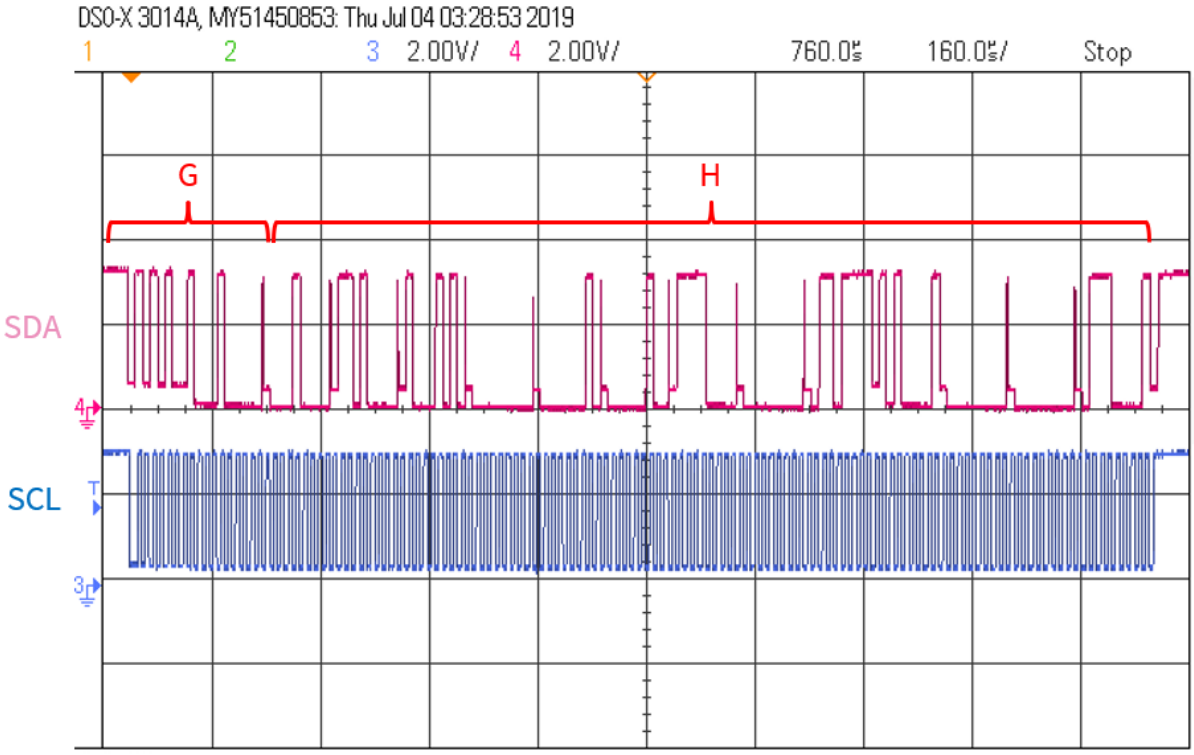
Tabela 8 – Descrição dos campos com o processo de escrita na memória

Campos	Descrição dos sinais no barramento I²C da ZC702
(E)	Bit de início + Endereço da EEPROM (0x54) + Bit de escrita (0) + Bit de ACK (0) + Endereço da 1ª posição de memória (0x00);
(F)	[Endereço do *Shelf Manager* (0x20) + bit ACK] + [*network function* com LUN (0x10) + bit ACK] + [CHK1 (0xD0) + bit ACK] + [Endereço do IPMC (0x8A)+ bit ACK] + [Sequência numérica (0x00) bit ACK] + [Identificador do comando (0x02) bit ACK] + [Comando para transição M0M1 (0x04 + 0xF0 + 0x00 + 0x6F + 0xA1 + 0x00 + 0x00) + bit ACK] + [CHK2 (0x70) + bit ACK] + Bit de parada.

Fonte: Próprio autor.

Na Figura 40 é mostrado o processo de leitura dos dados gravados na memória. O Zynq PS (Mestre) executa a função de ShMC emulado como uma tarefa do FreeRTOS, comunicando-se via barramento I²C com o dispositivo I²C EEPROM (Escravo).

Figura 40 – Comunicação I²C com o processo de leitura da solicitação IPMI da memória



Fonte: Próprio autor.

O processo de leitura dos dados da memória é semelhante ao do padrão de escrita. Contudo, sem a necessidade de especificação da posição de memória acompanhados sempre com o bit de NO ACK (1) no fim de cada dado lido, de acordo com o protocolo I²C.

Assim, para o sinal medido da Figura 40, tem-se a descrição dos campos na Tabela 9.

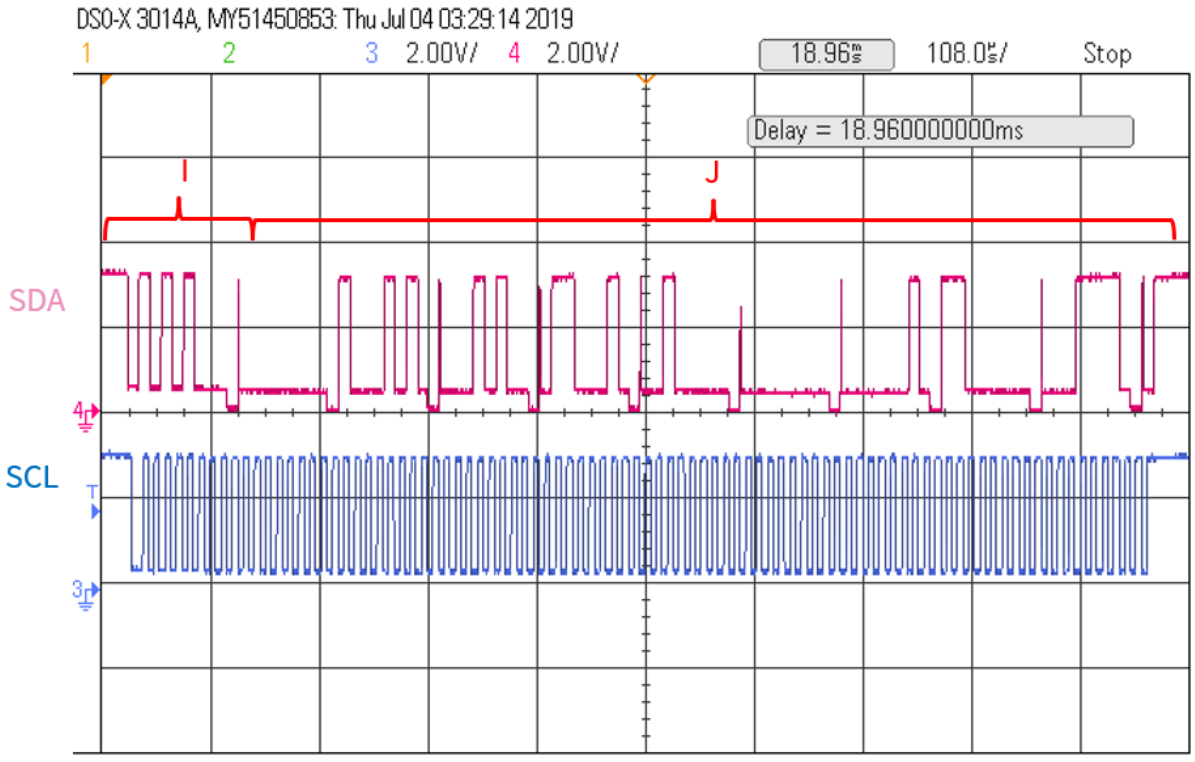
Tabela 9 – Descrição dos campos com o processo de escrita na memória

Campos	Descrição dos sinais no barramento I²C da ZC702
(G)	Bit de início + Endereço I²C EEPROM (0x54) + Bit de leitura (1) + Bit de ACK (0) + [Endereço do ShMC (0x20) + bit NO ACK]
(H)	[*network function*_LUN (0x10) + bit NO ACK] + [CHK1 (0xD0) + bit NO ACK] + [Endereço do IPMC (0x8A)+ bit NO ACK] + [Sequência numérica (0x00) + bit NO ACK] + [Identificador do comando (0x02) + bit NO ACK] + [Comando da transição M0M1 (0x04 + 0xF0 + 0x00 + 0x6F + 0xA1 + 0x00 + 0x00) + bit NO ACK] + [CHK2 (0x70) + bit NO ACK] + Bit de parada.

Fonte: Próprio autor.

Na Figura 41 é mostrado o processo de escrita da resposta IPMI do *Shelf Manager* emulado, executado pelo Zynq PS baseado na leitura da solicitação IPMI.

Figura 41 – Processo de escrita da resposta IPMI do ShMC para o IPMC



Fonte: Próprio autor.

E, na Tabela 10, tem-se a descrição detalhada dos campos da Figura 41.

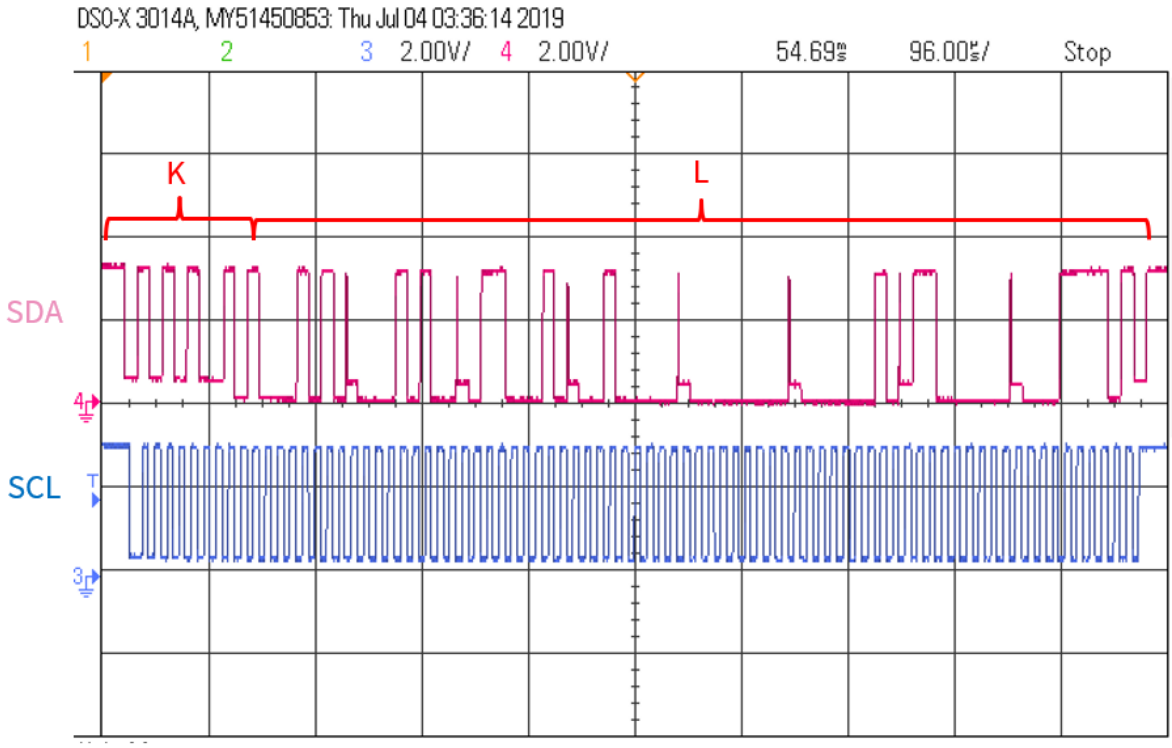
Tabela 10 – Descrição dos campos com o processo de escrita na memória

Campos	Descrição dos sinais no barramento I²C da ZC702
(I)	Bit de início + Endereço da EEPROM (0x54) + Bit de escrita (0) + Bit de ACK (0)
(J)	Endereço da posição de memória (0x00) + Dados de resposta IPMI com o bit de escrita: [Endereço do IPMC (0x8A) + bit ACK] + [*network function*_LUN (0x14) + bit ACK] + [CHK1 (0x62) + bit ACK] + [Endereço do ShMC (0x20) + bit ACK] + [Sequência numérica (0x00) bit ACK] + [Identificador do comando (0x02) + bit ACK] + [*Completion code* do ShMC (0xC0) + bit ACK] + [CHK2 (0x1E) + bit ACK] + Bit de parada.

Fonte: Próprio autor.

Na Figura 42 é mostrado o processo de leitura da memória com a resposta IPMI do ShMC para o IPMC.

Figura 42 – Comunicação I²C com o processo de leitura na EEPROM do quadro IPMI de resposta do ShMC para o IPMC



Fonte: Próprio autor.

Na Tabela 11, encontra-se a descrição detalhada dos campos referente ao sinal da Figura 42.

Tabela 11 – Descrição dos campos no processo de leitura dos dados da memória

Campos	Descrição dos sinais no barramento I²C da ZC702
(K)	Bit de início + Endereço I²C EEPROM (0x54) + Bit de leitura (1) + Bit de ACK (0)
(L)	Endereço do IPMC (0x8A) + bit NO ACK (1)] + [*network function*_LUN (0x14) + bit NO ACK] + [CHK1 (0x62) + bit NO ACK] + [Endereço do ShMC (0x20) + bit NO ACK] + [Sequência numérica (0x00) + bit NO ACK] + [Identificador do comando (0x02) + bit NO ACK] + [*Completion code* do ShMC (0xC0) + bit NO ACK] + [CHK2 (0x1E) + bit NO ACK] + Bit de parada;

Fonte: Próprio autor.

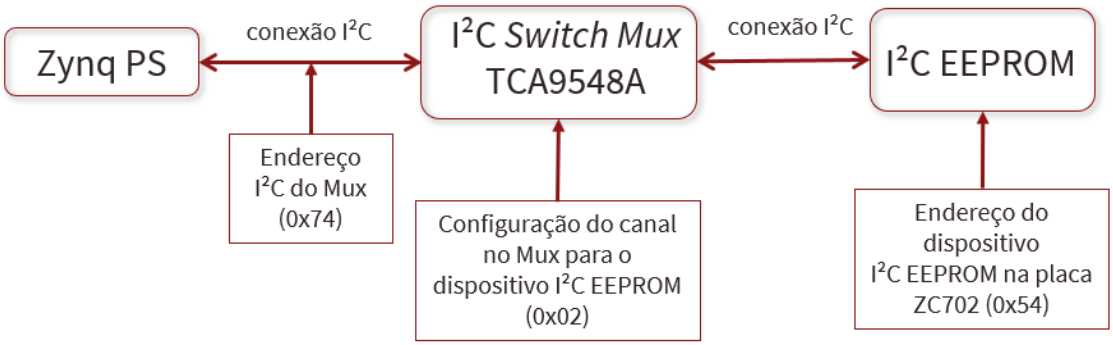
5 ANÁLISE E DISCUSSÃO DOS RESULTADOS

Com os resultados apresentados no Capítulo 4 para constatar a viabilidade do protocolo IPMI dentro do contexto do SoC Zynq, o cenário na plataforma ZC702 foi utilizado. Desta forma, a configuração e seleção dos endereços e registradores de cada dispositivo utilizado na placa deve ser especificado na implementação.

Dependendo da plataforma de *hardware* utilizada, os valores dos endereços e registradores podem ser alterados, além disso cada dispositivo I²C quando definido possui um endereço I²C único para ser acessado na plataforma geralmente especificado como é no caso da ZC702.

Na Figura 43 é mostrado o diagrama de blocos do cenário com os respectivos endereços I²C dos elementos utilizados na plataforma ZC702.

Figura 43 – Diagrama de blocos do cenário da comunicação I²C da ZC702 com os endereços dos dispositivos utilizados



Fonte: Próprio autor.

Essas configurações são inseridas na implementação no FreeRTOS e na análise do sinal da Figura 38, é possível ver a estrutura do protocolo I²C sendo executado várias vezes, quando os dados são transmitidos no barramento, seja no processo de leitura (verificar a existência do dispositivo na plataforma) ou escrita (selecionar os dispositivos para operação) dos dados. Isso se deve ao fato de as funções implementadas na rotina do *software* fazerem inúmeras verificações para selecionar os dispositivos corretos da plataforma.

A comunicação do Zynq PS com a memória segue os seguintes passos:

1°- O Zynq PS é o Mestre da comunicação I²C e endereça o dispositivo Escravo I²C EEPROM. Como passa por outro dispositivo (Mux) este também é endereçado pelo mestre. Como o Mestre envia os dados para o Escravo, ele é mestre-transmissor enquanto o outro é o escravo-receptor. O Mestre então termina a transferência.
2°- Se o Zynq PS deseja receber os dados da memória, ele o endereça novamente e nesta situação, torna-se mestre-receptor e recebe dados do Escravo que agora é o escravo-transmissor. Novamente o Mestre finaliza a transferência.

Neste cenário, o Zynq PS é o elemento transmissor (Tx) e a memória o receptor da comunicação I²C (Rx), como pode ser observado no sinal da Figura 39 (processo de escrita na memória) e Figura 40 (processo de escrita).

Em uma comunicação IPMI, o Zynq deve ter o papel tanto de transmissor (TX) como receptor (RX), para se ter uma condição multi-mestre exigida pelo protocolo IPMI nos controladores da plataforma.

Neste sentido, para satisfazer esta condição, houve a inserção da implementação IPMC como função de uma tarefa do FreeRTOS e outra tarefa pra emulação do ShMC, possibilitando um cenário com a dinâmica da comunicação IPMI neste contexto. Neste processo testou-se apenas uma transição da máquina de estado de uma FRU ATCA para transferência de dados reais de mensagens IPMI no barramento I²C desta plataforma.

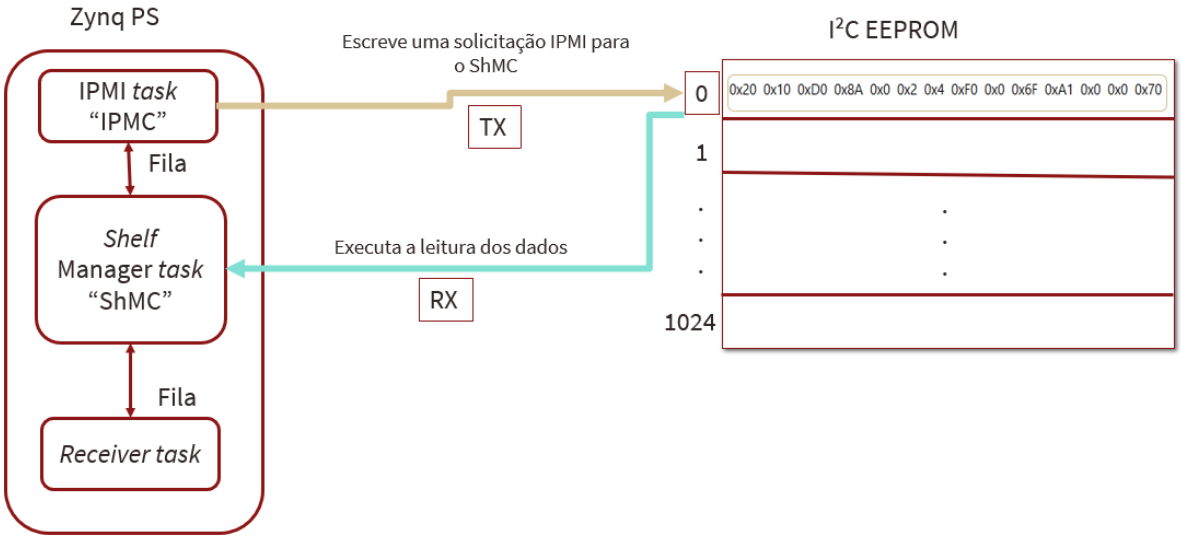
Baseado nos resultados das Figura 39-Figura *42*, os esquemáticos para o processo da comunicação IPMI verificado foram elaborados. Na Figura 44 com uma solicitação IPMI e na Figura 45 com a resposta IPMI.

A tarefa IPMI tem a principal função de escrever um conjunto de bytes específicos (solicitação IPMI) na EEPROM utilizando as funções e estruturas do código IPMC da Pulsar 2b adaptado ao contexto do FreeRTOS e do SoC Zynq.

A tarefa do *Shelf Manager* emulado executa a leitura dos dados da memória, compara os dados e escreve na mesma posição de memória a resposta IPMI com o envio para IPMC. A tarefa *Receiver* faz a leitura dos dados IPMI de resposta do *Shelf Manager* enviados ao IPMC, fechando assim o primeiro ciclo da comunicação.

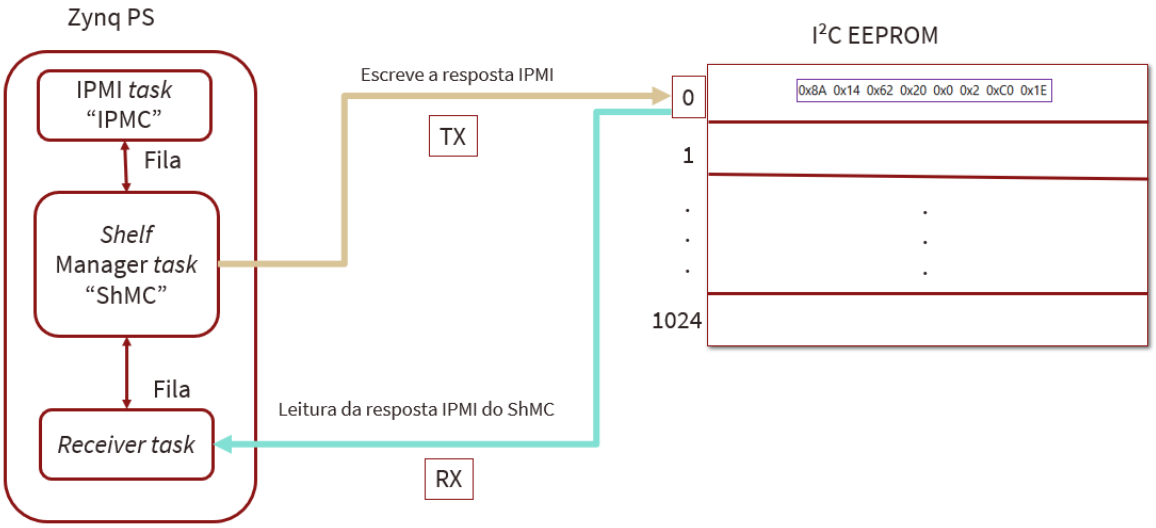
O dispositivo I²C EEPROM é utilizado para verificação e leitura das mensagens IPMI no barramento I²C executada pelo Zynq PS, na emulação desta comunicação.

Figura 44 – Esquemático da escrita e leitura dos dados de solicitação IPMI da memória executado pelas tarefas implementadas no FreeRTOS. Emulação da comunicação do IPMC com o ShMC



Fonte: Próprio autor.

Figura 45 – Esquemático da escrita e leitura dos dados de resposta IPMI da memória executado pelas tarefas implementadas no FreeRTOS. Emulação da comunicação do ShMC com o IPMC



Fonte: Próprio autor.

6 CONCLUSÕES E TRABALHOS FUTUROS

Na utilização de um sistema operacional de tempo real com um microcontrolador como o IPMC – FERMILAB, consegue-se o controle em um sistema que gerencia múltiplas tarefas simultaneamente sem a necessidade de um SoC.

Contudo, quando se pensa em expandir os tipos de aplicações, seria necessário a inserção de mais microcontroladores na mesma plataforma de *hardware*. Uma alternativa de custo e benefício é a utilização de um sistema multiplataforma no mesmo dispositivo como o SoC Zynq, pois há a compactação das várias funcionalidades com diferentes finalidades integradas em um único *chip*: FPGA, Linux, RTOS, controle, gerenciamento, acesso, e roteamento de informações entre outros.

Atualmente, cada plataforma de desenvolvimento que compõe o detector CMS, usa diferentes dispositivos e arquiteturas, mesmo que a funcionalidade das placas seja diferente, todas compartilham requisitos de gerenciamento semelhantes, como o protocolo IPMI, assegurando uma operação confiável do bastidor ATCA com controle e gerenciamento automático dos dispositivos com quase nenhuma perda de desempenho.

Além disso, a implementação IPMI em um contexto de *system-on-chip* como o Zynq utilizando um sistema em tempo real como o *Free*RTOS de acesso livre, é uma função conveniente para uma das finalidades de aplicação de um bastidor ATCA para o projeto do KIT/CERN.

No cenário apresentado neste trabalho, utilizando o Zynq da placa ZC702, tem-se o primeiro estágio de verificação da aplicabilidade em um SoC, sendo possível a análise do conjunto de funções necessárias para inserir o protocolo IPMI em um sistema operacional de tempo real como o *Free*RTOS.

A abordagem escolhida envolve verificações para a correta seleção dos dispositivos de *hardware* utilizado. Como o Zynq é capaz de operar como multi-mestre na comunicação I²C, ou seja, se comportar tanto como mestre ou escravo, ele pode ser utilizado na comunicação IPMI com a execução apenas da função de escrita no barramento IPMB (que é um canal I²C em essência).

Porém, nesta simulação ele foi utilizado apenas como mestre da comunicação I²C, pois fisicamente é o único controlador da plataforma ZC702. Para emulação do início da comunicação IPMI entre os controladores de gerenciamento IPMC e ShMC,

houve a necessidade de implementá-los como tarefas do *Free*RTOS para que o SoC Zynq operasse como os dois em instantes distintos.

Por meio da programação IPMC – FERMILAB adaptado ao SoC Zynq da plataforma ZC702, pode-se verificar o padrão de uma comunicação IPMI com o barramento I²C e testar a implementação com o dispositivo I²C EEPROM. Constatou-se a viabilidade do envio da estrutura dos quadros IPMI de solicitação e resposta no barramento com o SoC Zynq atuando como o controlador da plataforma.

Pelo fato da plataforma ZC702 ser diferente, a configuração dos canais IPMB que utiliza os *links* físicos I²C para transmissão de dados deve ser toda reconfigurada. Desta forma, a configuração dos canais I²C da ZC702 basearam-se na especificação da Xilinx, utilizando o modo *master polled* para enviar e receber dados no barramento.

O dispositivo I²C EEPROM serviu apenas para verificação das mensagens IPMI na dinâmica da comunicação entre as tarefas implementadas no *Free*RTOS.

Neste estágio de verificação de viabilidade da implementação IPMI no SoC Zynq, o uso da plataforma ZC702 foi satisfatória. Contudo, para implementação em um bastidor ATCA, como a configuração desta placa possui todos os canais I²C ocupados e ligados a outros dispositivos, além dela não ser uma placa do tipo ATCA, há a limitação nesta aplicação seja para expansão e configuração dos canais I²C para comunicação com outros controladores.

Uma alternativa seria utilizar outra plataforma de desenvolvimento que possua o SoC Zynq com os canais I²C livres que possam ser configurados para função multi-mestre do I²C diretamente, atuando somente como IPMC e acesso remoto ao *Shelf Manager* do SPRACE/NCC. Esta nova placa seria acoplada à placa Pulsar 2b com o SoC Zynq substituindo o IPMC – FERMILAB.

Neste sentido, a placa Ultra96 da Xilinx (XILINX, 2019) atende os requisitos, com um *Multiprocessor* (MP) SoC Zynq da família *Ulta Scale*+ (UE+) e com as devidas adaptações para esta topologia *hardware*, a implementação da programação IPMC utilizando o *Free*RTOS seria feita diretamente.

O IPMC implementado no Zynq UE+ (Zynq UE+ IPMC) será testado como controlador da plataforma ATCA Pulsar 2b e posteriormente poderá ser utilizado em outra etapa com uma outra plataforma ATCA, que está em fase de desenvolvimento pelo grupo do KIT/CERN.

REFERÊNCIAS

AGGETON, L. ARDILA-PEREZ, F. A. BALL, M. N. BALZER, G. BOUDOUL, J. BROOKE, M. CASELLE, L. CALLIGARIS, D. CIERI, E. J. CLEMENT, S. DUTTA, G. HALL, K. HARDER, P. R. HOBSON, G. M. ILES, T. JAMES, K. MANOLOPOULOS, T. MATSUSHITA, A. D. MORTON, D. NEWBOLD, S. PARAMESVARAN, M. PESARESI, N. POZZOBON, I. D. REID, A. W. ROSE, O. SANDER, C. SHEPHERD-THEMISTOCLEOUS, A. SHTIPLIYSKI, T. SCHUH, L. SKINNARI, S. P. SUMMERS, A. TAPPER, I. TOMALIN, A. THEA, K. UCHIDA, P. VICHOUDIS, S. VIRET, e M. WEBER. **An FPGA-Based Track Finder for the L1 Trigger of the CMS Experiment at the High Luminosity LHC**. CMS NOTE -2017/009. Genebra, Suíça. 2017.

BARRY, R. **Mastering the FreeRTOS real time Kernel a hands on tutorial guide**. [*S. l.: s. n*], 2016.

CERN EP-ESE. **CERN-IPMC: Technical overview**. v.1, 2017. Disponível em: <https://espace.cern.ch/ph-dep-ESE-BE ATCAEvaluationProject/PP_IPMC/Public%20documents/Forms/AllItems.aspx>. Acesso em: 07 jul. 2019.

CIPOLI, P. **O que é um SoC?** [*S. l.: s. n*], 2018. Disponível em: https://canaltech.com.br/hardware/O-que-e-um-SoC/. Acesso em: 02 jan. 2019.

CMS COLLABORATION. **“The CMS experiment at the CERN LHC”. JINST 3** S08004, 2008.

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

CROCKETT, L. H. **The Zynq book. Embedded Processing with the ARM® Cortex®-A9 on the Xilinx® Zynq®-7000 All Programmable SoC**. Glasgow, Scotland, United Kingdom: Strathclyde Academic Media, 2014. Disponível em: http://www.zynqbook.com/download-book.php.

INTEL. **IPMI -Intelligent platform management interface specification update.** rev. 7, 2015. Disponível em: https://www.intel.com/content/www/us/en/servers/ipmi/ipmi-home.html. Acesso em: 2018.

INTEL *et al*. **IPMI -Intelligent platform management bus communications protocol specification.** v 1, 1999. Disponível em: https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ipmp-spec-v1.0.pdf. Acesso em: 2018.

INTEL *et al*. **IPMI v2.0 rev. 1.1 specification**. I**ntelligent platform management interface specification**. 2013. Disponível em: https://www.intel.com/content/www/us/en/servers/ipmi/ipmi-intelligent-platform-mgt-interface-spec-2nd-gen-v2-0-spec-update.html. Acesso em: 03 ago. 2018.

INTEL HEWLETT-PACKARD NEC DELL. I**ntelligent Platform Management, Interface Specification, Second Generation v2.0**. rev 1.1. [*S.l*], 2013.

KENNEDY, P. **Explaining the Baseboard Management Controller or BMC in Servers.** ServeTheHome. Los Angeles, Chicago, United State of America. 2018. Disponível em: https://www.servethehome.com/explaining-the-baseboard-management-controller-or-bmc-in-servers/. Acesso em: 03 ago. 2018.

KOZAK, P. P. **Real-time IPMI protocol analyzer**. IEEE TRANSACTIONS ON NUCLEAR SCIENCE, Vol. 58, N° 4, pp. 1857-1863. 2011. Disponível em: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5766799. Acesso em: mar. 2018.

LARSEN, R. S. P**ICMG xTCA standards extensions for physics new devepolments and future plans**. pp. v.1, n.1, p. 1-7. Lisbon: 17th IEEE-NPSS Real Time Conference (RT), 2010.

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

NVENT. **NVENT SCHROFF.** Disponível em: https://schroff.nvent.com/en/schroff-na/advancedtca-eco-modular-system--14-slot--dc-advancedtca-eco-modular-system--14-slot--dc. Acesso em: 08 mar. 2018.

NXP SEMICONDUCTORS. **UM10204 I²C-bus specification and user manual**. rev.6. 2014.

OKUMURA, J. O. **Prototype performande studies of a full mesh ATCA-based general purpose data processing board**. IEEE Nuclear Science Synposium and Medical Imaging Conference (pp. 1-6). Seoul: v.1, n.1. 2013.

ORACLE, S. N. **Enabling end-to-end 10 Gigabit ethernet in Oracle's Sun Netra ATCA product family.** 2011. Disponível em: <http://www.oracle.com/technetwork/articles/systems-hardware-architecture/atcafamily-architecture-163505.pdf>. Acesso em: 03 mai. 2018.

PAIVA, T. C. **Remote development environment with reconfigurable components in the advanced telecom computing architecture context**. P149r, Dissertação de Mestrado, Unesp, Ilha Solteira, SP, Brasil. 2016.

PAIVA, T. C., RAMALHO, L. A. **ATCA/IPMI Service. A New Pulsar 2b IPMC Functionality**. SPRACE - INTERNAL NOTE SIN-R 02/2015, São Paulo, Brasil. 2015.

PEREK M. D. **ATCA carrier board with dedicated IPMI controller**. [*S.l.:s.n*], IEEE Explore Library, 2010. Disponível em: https://ieeexplore.ieee.org/document/5551663/?reload=true. Acesso em: 08 mar. 2018.

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

PICMG 3.0. **AdvancedTCA base specification**. version R3.0. [*S. l.*], 2008. Disponível em: https://www.picmg.org/wp-content/uploads/PICMG_ATCA_3_7R1_0.pdf.

RAMALHO, L. A., PAIVA, T. C. **FRU state machine is done**! [*S. l.: s. n.*], 2018.

ROKOV, O. T. **Making IPMI work in ATCA designs. Making IPMI Work in ATCA Designs**. Brighton, Reino Unido, Inglaterra. 2004. Disponível em https://www.embedded.com/design/connectivity/4009335/Making-IPMI-Work-in-ATCA-Designs. Acesso em: 01 nov. 2018.

ROSSI, L., BRÜNING, O. **High Luminosity Large Hadron Collider. A description for the European Strategy Preparatory Groups**. Genebra, Suíça, 2012. Disponível em: https://cds.cern.ch/record/1471000. Acesso em: 03 ago. 2018.

SANBLAZE. **SB-RTM436, RTM w/6Gb SAS Expander with 2 disks**. [*S.l.:s.n*], 2018. Disponível em: https://www.sanblaze.com/copy-of-sb-hda20. Acesso em: 08 mar. 2018.

SPRACE NEWS. **LHC se prepara para novas conquistas**. São Paulo, SP, Brasil, 2018. Disponível em: https://sprace.org.br/index.php/lhc-se-prepara-para-novas-conquistas/. Acesso em: 05 jul. 2019.

STMICROELECTRONICS. **M24C16, M24C08, M24C04, M24C02, M24C01. 16Kbit, 8Kbit, 4Kbit, 2Kbit and 1Kbit Serial I²C Bus EEPROM**. [*S.l.:s.n*], 2004. Disponível em: <www.st.com>.

SUPERMICROCOMPUTER. **Supermicro Solutions based on Intel® Xeon®-D processors**. [*S.l.:s.n*], 2019. Disponível em: https://www.supermicro.com/products/nfo/Xeon-D.cfm. Acesso em: 05 jul. 2019.

XILINX.. **Zynq-7000 All programmable SoC ZC702 evaluation kit. Quick Start Guide XTP310***,* v1.4, [*S.l.:s.n*], 2017. Disponível em: https://www.xilinx.com/support/documentation/boards_and_kits/zc702_zvik/xtp310-zc702-quickstart.pdf. Acesso em 2018.

XILINX. **ZC702 evaluation board for the Zynq-7000 XC7Z020 SoC user guide**. UG850 (v1.6.1), [*S.l.:s.n*], 2018. Disponível em: https://www.xilinx.com/support/documentation/boards_and_kits/zc702_zvik/ug850-zc702-eval-bd.pdf.

XILINX. **Board features**. **Featuring the ZC702 base board,** [*S.l.:s.n*], 2018. Disponível em: https://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html#hardware. Acesso em: 03 ago. 2018.

XILINX. **Zynq UltraScale+ MPSoC product advantages**. [*S.l.:s.n*]**,** 2019. Disponível em: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html. Acesso em: 11 fev. 2019.

ZHAO, H. X. **Analysis of Out-of-Band Management Security Based on IPMI Protocol**. [*S. l.: s. n.*], 2017.

APÊNDICE – Código principal da implementação IPMI inserida no FreeRTOS

```
/*
 * Author: Sthefany Fernandes de Souza
 * 28 June 2019
 * Adaptation IPMC code to FreeRTOS with queue system to by-pass a frame IPMI
 * Request of IPMC (M0M1 of Pulsar2b) to Shelf Manager (Emulated) using ZynqPS
 * of ZC702 and send this information to I²C EEPROM device via I²C bus
 * emulated(responser)ZynqPS one hour is IPMC (requester) another time is shelf
 * manager ZynqPS is Master and EEPROM is Slave of all communication by I²C bus
*/

/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "timers.h"
#include "semphr.h"
/* Xilinx includes. */
#include "xil_printf.h"
#include "xparameters.h"

/*IIC_ZC702 includes*/
#include "sleep.h"
#include "xiicps.h"
#include "xplatform_info.h"

/* IPMI */
#include "ipmi.h"
#include "ipmi_internal_events.h"
#include "ipmi_ipmb.h"
#include "ipmi_requests.h"
#include "ipmi_requests_application.h"

#define TIMER_ID	1
#define TIMER_CHECK_THRESHOLD	1

#define IIC_SCLK_RATE		100000
#define SLV_MON_LOOP_COUNT 0x00FFFFFF  /**< Slave Monitor Loop Count*/
#define MUX_ADDR 0x74
#define MAX_CHANNELS 0x04
#define MAX_SIZE	32
#define PAGE_SIZE_16	16
#define PAGE_SIZE_32	32
#define EEPROM_START_ADDRESS	0

typedef u16 AddressType;
/*-----------------------------------------------------------*/

/* The IPMI and Rx tasks as described at the top of this file. */
void prvIPMITask( void *pvParameters );
void prvReceiverTask( void *pvParameters );
void prvShelfManagerTask(void *pvParameters);
void vTimerCallback( TimerHandle_t pxTimer );
/*-----------------------------------------------------------*/

/* Functions to initiate the IIC Polled mode by ZynqPS, to select
 * and set the MUX and EEPROM devices of board, to write and read
```

```
 * data bytes (page data) on EEPROM. */
s32 IicPsEepromPolledInit(void);
s32 EepromWriteData(XIicPs *IicInstance, u16 ByteCount);
s32 EepromReadData(XIicPs *IicInstance, u8 *BufferPtr, u16 ByteCount);
s32 IicPsSlaveMonitor(u16 Address, u16 DeviceId);
s32 MuxInitChannel(u16 MuxIicAddr, u8 WriteBuffer);
s32 FindEepromDevice(u16 Address);
s32 IicPsFindEeprom(u16 *Eeprom_Addr, u32 *PageSize);
s32 IicPsConfig(u16 DeviceId);
s32 IicPsFindDevice(u16 addr);
/*-----------------------------------------------------------*/
/* The queue used by the IPMI and Receiver tasks, as described at the top of this
file. */
TaskHandle_t xIPMITask, xShelfManagerTask, xReceiverTask;
QueueHandle_t xQueue,xQueue_shelf,xQueue_shelf_response;
TimerHandle_t xTimer = NULL;
long RxtaskCntr = 0;
/*-----------------------------------------------------------*/
/* Variables used in this file. */
u8 WriteBuffer[sizeof(AddressType) + MAX_SIZE];
u8 ReadBuffer[MAX_SIZE];
u16 EepromAddr[] = {0x54,0x55,0};
u16 MuxAddr[] = {0x74,0};
u16 EepromSlvAddr;
u32 PageSize;
XIicPs IicInstance;
u32 Platform;
u32 WrBfrOffset;
ipmb_tx_cel_t out;
ipmb_tx_queue_t flag;

int main( void )
{
const TickType_t x5seconds = pdMS_TO_TICKS(5000);
s32 Status;
AddressType Address = EEPROM_START_ADDRESS;
Status = IicPsFindEeprom(&EepromSlvAddr,&PageSize); /*Find EEPROM device:
                                      Select the address and page size. */

  if (PageSize == PAGE_SIZE_16) {   //PAGE SIZE TO EEPROM (1kB) ZC702---(M24C08)
      WriteBuffer[0] = (u8) (Address); //1° position EEPROM address, 1 byte
      WrBfrOffset = 1;
      }
  else {
      WriteBuffer[0] = (u8) (Address >> 8);//1° position EEPROM address, 2 bytes
      WriteBuffer[1] = (u8) (Address);
      WrBfrOffset = 2;
  }
  xil_printf( "Initiate FreeRTOS with I²C EEPROM Polled Mode!\r\n" );

  /*Function writes, reads, and verifies the data to the IIC EEPROM.
    It does the write as a single page write, performs a buffered read. */
   Status = IicPsEepromPolledInit();

   if (Status != XST_SUCCESS) {
       xil_printf("IicPsFindEeprom Test Failed \r\n");
   }
   else {
          xil_printf("IicPsFindEeprom conclude! \r\n");
```

```
    }

    /* Create the queues used by the tasks and functions along the code.
     * The tasks have the same priority and are controlled by queues. Example:
     * the "IPMI" task, so will preempt the "IPMI" task and remove values from
     * the queue as soon as the "IPMI" task writes to the queue - therefore
     * the xQueue can never have more than one item in it. */
     xQueue = xQueueCreate( 1, sizeof( WriteBuffer ) );
     /* Check the queue was created. */
     configASSERT( xQueue );

     /* Queue_flag used by functions inside "IPMI" task to signal a internal
      * event flag or event flags to executed the (procedures-retirar) of
      * (t_IPMI file IPMI.c). */
      xQueue_flag = xQueueCreate( 1, sizeof( u16 ) );
      configASSERT( xQueue_flag );

     /* Queue_M0M1 used by function 'SendRequest(ipmb_tx_cel_t out)' and return
      * the data of frame IPMB request(M0M1). OBS: In this case
      * the size of queue is a structure  'ipmb_tx_cel_t'. */
      xQueue_M0M1 = xQueueCreate( 1, sizeof( ipmb_tx_cel_t) );
      configASSERT( xQueue_M0M1 );
     /* Queue between Shelf Manager task and Receiver task */
       xQueue_shelf = xQueueCreate( 1, sizeof(WriteBuffer) );
       configASSERT( xQueue_shelf );

       xQueue_shelf_response = xQueueCreate( 1, sizeof(ipmb_rx_cel_t) );
       configASSERT( xQueue_shelf_response);

/* Create the two tasks.  The "IPMI" task is given a lower priority than the
"Receiver" task, so the "Receiver" task will leave the Blocked state and preempt
"IMPI" task as soon as the "IPMI" task places an item in the queue.*/

xTaskCreate( prvIPMITask, /* The function that implements the task. */
          (const char *) "IPMI", /* Text name for the task, provided to assist
debugging only.*/
       configMINIMAL_STACK_SIZE, /* The stack allocated to the task. */
       NULL,                  /* The task parameter is not used, so set to NULL. */
       tskIDLE_PRIORITY,         /* The task runs at the idle priority. */
       &xIPMITask );             /* Task handle. */

       xTaskCreate( prvShelfManagerTask,
                    ( const char * ) "ShelfManager",
                    configMINIMAL_STACK_SIZE,
                    NULL,
                    tskIDLE_PRIORITY,
                    &xShelfManagerTask );

       xTaskCreate( prvReceiverTask,
                    ( const char * ) "Receiver",
                    configMINIMAL_STACK_SIZE,
                    NULL,
                    tskIDLE_PRIORITY,
                    &xReceiverTask);

/* Create a timer with a timer EXPIRY of 2 seconds. The timer would expire
after 2 seconds and the TimerCallback would get called. In the timer call back
checks are done to ensure that the tasks have been running properly till then.
```

```
The tasks are deleted in the timer call back and a message is printed to convey
that the example has run successfully. The timer EXPRIRY is set to 2 seconds and
the timer set to not auto reload. */
xTimer = xTimerCreate( (const char *) "Timer",
                           x5seconds,
                           pdFALSE,
                           (void *) TIMER_ID,
                           vTimerCallback);
/* Check the timer was created. */
configASSERT( xTimer );

/* Start the timer with a block time of 0 ticks. This means as soon as the
schedule starts the timer will start running and will expire after 2 seconds */
xTimerStart( xTimer, 0 );

/* Start the tasks and timer running. */
vTaskStartScheduler();

/* If all is well, the scheduler will now be running, and the following line
will never be reached.  If the following line does execute, then there was
insufficient FreeRTOS heap memory available for the idle and/or timer tasks
to be created.  See the memory management section on the FreeRTOS web site
for more details. */
for( ;; );
}

/*-----------------IPMITask-----------------------------------*/
void prvIPMITask( void *pvParameters )
{
/*Initialization and start IPMI functions. */
IPMI_init();
IPMI_start(); // First transition internal of Pulsar2b (IE_M0M1)
t_IPMI();     //task IPMI of IPMC code

u32 Index;
int Status;
ipmb_tx_cel_t * rec_out;

for( ;; )
{
       xQueueReceive(xQueue_M0M1,&rec_out,portMAX_DELAY);

       WriteBuffer[1] = rec_out->pkt.head.dst;
       WriteBuffer[2] = rec_out->pkt.head.netfn_lun;
       WriteBuffer[3] = rec_out->pkt.head.chk1;
       WriteBuffer[4] = rec_out->pkt.head.src;
       WriteBuffer[5]  = rec_out->pkt.head.seq_lun;
       WriteBuffer[6]  = rec_out->pkt.head.cmd;
       WriteBuffer[7]  = rec_out->pkt.body[0];
       WriteBuffer[8]  = rec_out->pkt.body[1];
       WriteBuffer[9]  = rec_out->pkt.body[2];
       WriteBuffer[10]  = rec_out->pkt.body[3];
       WriteBuffer[11]  = rec_out->pkt.body[4];
       WriteBuffer[12]  = rec_out->pkt.body[5];
       WriteBuffer[13]  = rec_out->pkt.body[6];
       WriteBuffer[14]  = rec_out->pkt.body[7];

Status = EepromWriteData(&IicInstance, WrBfrOffset + PageSize-2);/*This function
                           writes a buffer of data to the IIC serial EEPROM. */
```

```
xil_printf("\r\n------------------------------------------------------------\r\n ");
xil_printf("FRAME IPMI Request (M0M1/IPMC to Shelf)  Written in EEPROM\r\n");
      for(Index = 1; Index < 15; Index++){
          xil_printf("0x%x  ", WriteBuffer[Index]);
      }
      xil_printf("\r\n ");
      if (Status != XST_SUCCESS) {
      xil_printf("IIC EEPROM Write Data Failed\r\n");
      }
      else {
      xil_printf("I²C EEPROM Write Data Conclude! \r\n");
      xil_printf("\r\n---------------------------------------\r\n ");
      }
      /* Block to send data on the queue. */
      xQueueSend( xQueue, WriteBuffer, 0UL );//Send WriteBuffer to ReceiverTask
      }
}

/*-------------------ReceiverTask--------------------------*/
void prvShelfManagerTask( void *pvParameters )
{
ipmb_channel_t * ch;
ipmb_rx_cel_t * in;
int Status, i;
u32 Index;
i=0;
u8 Data[] = {0x20, 0x10,0xd0, 0x8a, 0x00, 0x02, 0x04, 0xf0, 0x00, 0x6f,0xa1, 0x00,
0x00, 0x70};

    for( ;; )
    {
     /* Block to wait for data arriving on the queue. */
     xQueueReceive( xQueue, ReadBuffer, portMAX_DELAY );
     Status = EepromReadData(&IicInstance, ReadBuffer, 14); /*This function reads
                                                  data from the IIC serial EEPROM
                                                  into a specified buffer. */
     xil_printf("FRAME IPMI Request(M0M1/IPMC to Shelf) Read from EEPROM\r\n");
     for(Index = 0; Index < 14; Index++){
        xil_printf("0x%x  ",ReadBuffer[Index]);
     }
     xil_printf("\r\n ");
     if (Status != XST_SUCCESS) {
         xil_printf("IIC EEPROM Read Data Failed\r\n");
     }
     else {
         xil_printf("I²C EEPROM Read Data Conclude! \r\n");
         xil_printf("\r\n ");
     }

     u8 frame_IPMI_shelf[] = {0x8a, 0x14, 0x62, 0x20, 0x00, 0x02, 0xc0, 0x1e};
     //Answer Shelf to IPMC

     WriteBuffer[1] = frame_IPMI_shelf[0];
     WriteBuffer[2] = frame_IPMI_shelf[1];
     WriteBuffer[3] = frame_IPMI_shelf[2];
     WriteBuffer[4] = frame_IPMI_shelf[3];
     WriteBuffer[5]  = frame_IPMI_shelf[4];
     WriteBuffer[6]  = frame_IPMI_shelf[5];
     WriteBuffer[7]  = frame_IPMI_shelf[6];
```

```
      WriteBuffer[8]  = frame_IPMI_shelf[7];

      for(Index = 0;Index < 14; Index++){ //verify data read from EEPROM == data
                                          //expect in the shelf
          if(Data[Index] == ReadBuffer[Index]){
              i++;
          }
      }
      if(i==14){
      Status = EepromWriteData(&IicInstance, 9);/*This function writes a buffer of
                                                  data to the IIC serial EEPROM. */
      }
      xil_printf("FRAME IPMI Response(Shelf to IPMC) Written in EEPROM\r\n");
      for(Index = 1; Index < 9; Index++){
          xil_printf("0x%x  ", WriteBuffer[Index]);
      }
      xil_printf("\r\n ");
      if (Status != XST_SUCCESS) {
          xil_printf("I²C EEPROM Write Data Failed\r\n");
      }
      else {
      xil_printf("I²C EEPROM Write Data Conclude!\r\n");
      xil_printf("\r\n----------------------------------------------------\r\n ");

      xQueueSend( xQueue_shelf_response,WriteBuffer, 0UL);
      xil_printf("Send queue response\r\n");
      ProcessResponse(ch,in);
      /* Block to send data on the queue. */
        xQueueSend ( xQueue_shelf,WriteBuffer, 0UL);//Send WriteBuffer to
                                                    ReceiverTask
      }
     }
}
/*-------------------ShelfManagerTask--------------------------*/
void prvReceiverTask( void *pvParameters )
{
int Status;
u32 Index;

 for( ;; )
    {
    /* Block to wait for data arriving on the queue. */
       xQueueReceive( xQueue_shelf,ReadBuffer,portMAX_DELAY );

       Status = EepromReadData(&IicInstance, ReadBuffer, 8); /*This function
                                              reads data from the IIC serial
                                             EEPROM into a specified buffer. */
       xil_printf("FRAME IPMI Response (Shelf to IPMC) Read from EEPROM\r\n");

       for(Index = 0; Index < 8; Index++){
           xil_printf("0x%x  ",ReadBuffer[Index]);
        }
        xil_printf("\r\n ");
        if (Status != XST_SUCCESS) {
             xil_printf("IIC EEPROM Read Data Failed\r\n");
        }
        else {
            xil_printf("I²C EEPROM Read Data Conclude! \r\n");
            xil_printf("\r\n--------------------------------------------\r\n ");
```

```
        }
    RxtaskCntr++;
   }
}

/*----------------TimerCallback-----------------------------*/
void vTimerCallback( TimerHandle_t pxTimer )
{
 long lTimerId;
 configASSERT( pxTimer );
 lTimerId = ( long ) pvTimerGetTimerID( pxTimer );
 if (lTimerId != TIMER_ID) {
     xil_printf("FreeRTOS frame IPMI from ZynqPS to EEPROM FAILED");
 }

 /* If the RxtaskCntr is updated every time the "Receiver" task is called.
  * The "Receiver" task is called every time the "IPMI" task sends a message.
  * The IPMI task sends a message every 1 second. The timer expires after 2
  * seconds. We expect the RxtaskCntr to at least have a value of
  * (TIMER_CHECK_THRESHOLD) when the timer expires. */

  if (RxtaskCntr >= TIMER_CHECK_THRESHOLD) {
      xil_printf("Communication IPMI_FreeRTOS_ZynqPS_EEPROM Conclude!");
  }
  else {
     xil_printf("Communication IPMI_FreeRTOS_ZynqPS_EEPROM FAILED!");
  }
}
/*****************************************************************************/
/*This function writes, reads, and verifies the data to the IIC EEPROM. It
* does the write as a single page write, performs a buffered read.
* @param     None.
* @return    XST_SUCCESS if successful else XST_FAILURE.
* @note            None. */
/*****************************************************************************/
s32 IicPsEepromPolledInit()
{
 u32 Index;
 s32 Status;

 Status = IicPsFindEeprom(&EepromSlvAddr,&PageSize);
 if (Status != XST_SUCCESS) {
     return XST_FAILURE;
 }
 /*Initialize the data to write (FFh) and the read buffer (0h).*/
 for (Index = 0; Index < PageSize; Index++) {
      WriteBuffer[WrBfrOffset + Index] = 0xFF;
      ReadBuffer[Index] = 0;
    }
    return XST_SUCCESS;
}

/*****************************************************************************/
/*This function writes a buffer of data to the IIC serial EEPROM.
* @param     ByteCount contains the number of bytes in the buffer to be written.
* @return    XST_SUCCESS if successful else XST_FAILURE.
* @note      The Byte count should not exceed the page size of the EEPROM as
            noted by the constant PAGE_SIZE.*/
/*****************************************************************************/
```

```
s32 EepromWriteData(XIicPs *IicInstance, u16 ByteCount)
{
 s32 Status;

 /* Send the Data for IIC bus of ZC702. */
 Status = XIicPs_MasterSendPolled(IicInstance, WriteBuffer, ByteCount,
EepromSlvAddr);

 if (Status != XST_SUCCESS) {
     return XST_FAILURE;
 }

 /* Wait until bus is idle to start another transfer. */
    while (XIicPs_BusIsBusy(IicInstance));

 /* Wait for a bit of time to allow the programming to complete */
    usleep(250000);
    return XST_SUCCESS;
}
/*****************************************************************************/
/*This function reads data from the IIC serial EEPROM into a specified buffer.
* @param     BufferPtr contains the address of the data buffer to be filled.
* @param     ByteCount contains the number of bytes in the buffer to be read.
* @return    XST_SUCCESS if successful else XST_FAILURE.
* @note            None.
*****************************************************************************/
s32 EepromReadData(XIicPs *IicInstance, u8 *BufferPtr, u16 ByteCount)
{
s32 Status;
AddressType Address = EEPROM_START_ADDRESS;
u32 WrBfrOffset;

/* Position the Pointer in EEPROM. */
if (PageSize == PAGE_SIZE_16) {
    WriteBuffer[0] = (u8) (Address);
    WrBfrOffset = 1;
}
else {
     WriteBuffer[0] = (u8) (Address >> 8);
     WriteBuffer[1] = (u8) (Address);
     WrBfrOffset = 2;
    }

Status = EepromWriteData(IicInstance, WrBfrOffset);
if (Status != XST_SUCCESS) {
    return XST_FAILURE;
}

/* Receive the Data.*/
Status = XIicPs_MasterRecvPolled(IicInstance, BufferPtr, ByteCount,
EepromSlvAddr);
if (Status != XST_SUCCESS) {
    return XST_FAILURE;
 }

/* Wait until bus is idle to start another transfer.*/
  while (XIicPs_BusIsBusy(IicInstance));
  return XST_SUCCESS;
}
```

```
/*****************************************************************************/
/* This function initializes the IIC MUX to select the required channel.
* @param     MuxAddress and Channel select value.
* @return    XST_SUCCESS if pass, otherwise XST_FAILURE.
* @note      None.
******************************************************************************/
s32 MuxInitChannel(u16 MuxIicAddr, u8 WriteBuffer)
{
 u8 Buffer = 0;
 s32 Status = 0;
 /* Wait until bus is idle to start another transfer.*/
  while (XIicPs_BusIsBusy(&IicInstance));
  /* Send the Data.*/
  Status = XIicPs_MasterSendPolled(&IicInstance, &WriteBuffer,1, MuxIicAddr);
  if (Status != XST_SUCCESS) {
      return XST_FAILURE;
  }
  /* Wait until bus is idle to start another transfer. */
  while (XIicPs_BusIsBusy(&IicInstance));
  /* Receive the Data.*/
  Status = XIicPs_MasterRecvPolled(&IicInstance, &Buffer,1, MuxIicAddr);

  if (Status != XST_SUCCESS) {
     return XST_FAILURE;
  }
  /* Wait until bus is idle to start another transfer.*/
  while (XIicPs_BusIsBusy(&IicInstance));
  return XST_SUCCESS;
}

/*****************************************************************************/
/*This function perform the initial configuration for the IICPS Device.
* @param     DeviceId instance.
* @return    XST_SUCCESS if pass, otherwise XST_FAILURE.
* @note      None.
******************************************************************************/
s32 IicPsConfig(u16 DeviceId)
{
 s32 Status;
 XIicPs_Config *ConfigPtr;      /* Pointer to configuration data */

 /* Initialize the IIC driver so that it is ready to use.*/
 ConfigPtr = XIicPs_LookupConfig(DeviceId);
 if (ConfigPtr == NULL) {
     return XST_FAILURE;
 }
 Status = XIicPs_CfgInitialize(&IicInstance, ConfigPtr, ConfigPtr->BaseAddress);
 if (Status != XST_SUCCESS) {
     return XST_FAILURE;
 }
 /* Set the IIC serial clock rate. */
 XIicPs_SetSClk(&IicInstance, IIC_SCLK_RATE);
 return XST_SUCCESS;
}
/*****************************************************************************/
/*This function is use to figure out the slave device is alive or not.
* @param        slave address and Device ID .
* @return       XST_SUCCESS if successful, otherwise XST_FAILURE.
```

```
* @note         None.
*****************************************************************************/
s32 IicPsFindDevice(u16 addr)
{
 s32 Status;
 Status = IicPsSlaveMonitor(addr,0);
 if (Status == XST_SUCCESS) {
     return XST_SUCCESS;
 }

 Status = IicPsSlaveMonitor(addr,1);
 if (Status == XST_SUCCESS) {
     return XST_SUCCESS;
 }
 return XST_FAILURE;
}

/*****************************************************************************/
/*This function is use to figure out the Eeprom slave device
* @param     addr: u16 variable
* @return    XST_SUCCESS if successful and also update the epprom slave
* device address in addr variable else XST_FAILURE.
* @note            None.
*****************************************************************************/
s32 IicPsFindEeprom(u16 *Eeprom_Addr,u32 *PageSize)
{
 s32 Status;
 u32 MuxIndex,Index;
 u8 MuxChannel;

 for(MuxIndex=0;MuxAddr[MuxIndex] != 0;MuxIndex++){
 Status = IicPsFindDevice(MuxAddr[MuxIndex]);
 if (Status == XST_SUCCESS) {
 for(Index=0;EepromAddr[Index] != 0;Index++) {
 for(MuxChannel = MAX_CHANNELS; MuxChannel > 0x0; MuxChannel = MuxChannel >> 1) {
        Status = MuxInitChannel(MuxAddr[MuxIndex], MuxChannel);
        if (Status != XST_SUCCESS) {
        xil_printf("Failed to enable the MUX channel\r\n");
        return XST_FAILURE;
        }
        Status = FindEepromDevice(EepromAddr[Index]);
        if (Status == XST_SUCCESS) {
        *Eeprom_Addr = EepromAddr[Index];
        *PageSize = PAGE_SIZE_16;
         return XST_SUCCESS;
         }
        }
    }
    }
    }
    for(Index=0;EepromAddr[Index] != 0;Index++) {
    Status = IicPsFindDevice(EepromAddr[Index]);
    if (Status == XST_SUCCESS) {
        *Eeprom_Addr = EepromAddr[Index];
        *PageSize = PAGE_SIZE_32;
        return XST_SUCCESS;
      }
 }
 return XST_FAILURE;
```

```
}
/*****************************************************************************/
/*This function checks the availability of EEPROM using slave monitor mode.
* @param     EEPROM address.
* @return    XST_SUCCESS if successful, otherwise XST_FAILURE.
* @note      None.
******************************************************************************/
s32 FindEepromDevice(u16 Address)
{
 u32 Index,IntrStatusReg;
 XIicPs *IicPtr = &IicInstance;
 XIicPs_EnableSlaveMonitor(&IicInstance, Address);
 Index = 0;

 /* Wait for the Slave Monitor status */
 while (Index < SLV_MON_LOOP_COUNT) {
        Index++;

/* Read the Interrupt status register.*/
IntrStatusReg = XIicPs_ReadReg(IicPtr->Config.BaseAddress,(u32)XIICPS_ISR_OFFSET);

if (0U != (IntrStatusReg & XIICPS_IXR_SLV_RDY_MASK)) {
    XIicPs_DisableSlaveMonitor(&IicInstance);
    XIicPs_WriteReg(IicPtr->Config.BaseAddress,(u32)XIICPS_ISR_OFFSET,
IntrStatusReg);
     return XST_SUCCESS;
     }
 }
 XIicPs_DisableSlaveMonitor(&IicInstance);
 return XST_FAILURE;
}

/*****************************************************************************/
/*This function checks the availability of a slave using slave monitor mode.
* @param     DeviceId is the Device ID of the IicPs Device and is the
*            XPAR_<IICPS_instance>_DEVICE_ID value from xparameters.h
* @return    XST_SUCCESS if successful, otherwise XST_FAILURE.
* @note      None.
******************************************************************************/
s32 IicPsSlaveMonitor(u16 Address, u16 DeviceId)
{
 u32 Index,IntrStatusReg;
 s32 Status;
 XIicPs *IicPtr;

 /* Initialize the IIC driver so that it is ready to use. */
 Status = IicPsConfig(DeviceId);
 if (Status != XST_SUCCESS) {
     return XST_FAILURE;
 }

 IicPtr = &IicInstance;
 XIicPs_EnableSlaveMonitor(&IicInstance, Address);
  Index = 0;

  /* Wait for the Slave Monitor Status*/
  while (Index < SLV_MON_LOOP_COUNT) {
        Index++;
```

```
  /* Read the Interrupt status register.*/
 IntrStatusReg = XIicPs_ReadReg(IicPtr>Config.BaseAddress,(u32)XIICPS_ISR_OFFSET);

 if (0U != (IntrStatusReg & XIICPS_IXR_SLV_RDY_MASK)) {
         XIicPs_DisableSlaveMonitor(&IicInstance);
         XIicPs_WriteReg(IicPtr->Config.BaseAddress,(u32)XIICPS_ISR_OFFSET,
IntrStatusReg);
        return XST_SUCCESS;
 }
 }
 XIicPs_DisableSlaveMonitor(&IicInstance);
 return XST_FAILURE;
}
```