

Zynq UltraScale+ MPSoC Software Developer Guide

UG1137 (2020.1) September 4, 2020



Revision History

The following table shows the revision history for this document.

Section	Revision Summary
09/04/2020 Version 2020.1	
Chapter 10: Platform Management Unit Firmware	Updated PMU Firmware Build Flags to add a new flag.
Chapter 12: Reset	Updated RPU Subsystem Restart for RPU only restart support details.
Appendix E: XilSecure Library v4.2	Added Additional References .
Appendix H: XilFPGA Library v5.2	Added Additional References .
12/05/2019 Version 2019.2	
Vitis Embedded Flow	Updated SDK flows to Vitis Embedded Flow throughout the document.
06/26/2019 Version 2019.1	
Chapter 4: Software Stack	Updated Multimedia Stack Overview .
Chapter 7: System Boot and Configuration	Updated Miscellaneous Functions
Chapter 10: Platform Management Unit Firmware	Added CSU/PMU Register Access and updated PMU Firmware Build Flags
Chapter 11: Power Management Framework	Updated Sub-system Power Management
	Added appendix
01/18/2019 Version 9.0	
Chapter 2: Programming View of Zynq UltraScale+ MPSoC Devices	Updated Boot Modes and System-Level Protections sections
Chapter 3: Development Tools	Added Device Tree Generator
Chapter 4: Software Stack	Removed XilRSA references
Chapter 8: Security Features	Updated Configuring XMPU Registers
Chapter 9: Platform Management	Updated Power Management Framework
Chapter 10: Platform Management Unit Firmware	Updated PMU Firmware Build Flags , FPD WDT , and PMU Firmware Memory Layout and Footprint
Chapter 12: Reset	Updated Warm Restart with a note about on-chip memory (OCM)
Chapter 16: Boot Image Creation	Removed content and updated the chapter with a short description and added a reference to the Bootgen user guide.
06/22/2018 Version 8.0	
Chapter 7: System Boot and Configuration	Added a note that SHA-2 will be deprecated from 2019.1 release with a recommendation to use SHA-3
Chapter 8: Security Features	Added Enhanced RSA Key Revocation Support
Chapter 10: Platform Management Unit Firmware	Updated PMU firmware Signals PLL Lock Errors on PS_ERROR_OUT section and PMU firmware Loading Options
05/04/2018 Version 7.0	
Chapter 8: Security Features	Added BIF File for Obfuscated Form (Gray) Key Stored in eFUSE and updated deprecation of SHA-2 authentication

Section	Revision Summary
Chapter 12: Reset	Added Warm Restart
Chapter 16: Boot Image Creation	Updated Boot Image format documentation
01/19/2018 Version 6.0	
Chapter 5: Software Development Flow	Updated Bare Metal Application Development
Chapter 7: System Boot and Configuration	Updated Boot Flow and Boot Modes sections
Chapter 8: Security Features	Updated BIF File with Multiple AESKEY Files
Chapter 13: High-Speed Bus Interfaces	Updated Ethernet flow figures
Chapter 16: Boot Image Creation	Updated example for [fsbl_config] parameter
11/15/2017 Version 5.0	
Chapter 1: About This Guide	Updated Prerequisites
Chapter 2: Programming View of Zynq UltraScale+ MPSoC Devices	Updated Boot Process and Security sections
Chapter 4: Software Stack	Updated FreeRTOS Software Stack
Chapter 7: System Boot and Configuration	Added FSBL Build Process and Setting FSBL Compilation Flags sections. Updated Boot Modes
Chapter 8: Security Features	Updated Boot Time Security
Chapter 9: Platform Management	Platform Management in PS and PMU Firmware sections
Chapter 10: Platform Management Unit Firmware	Added new chapter
Chapter 11: Power Management Framework	Updated Zynq UltraScale+ MPSoC Power Management Software Architecture, Using the API for Power Management, Sub-system Power Management, and XilPM Implementation Details sections
Chapter 16: Boot Image Creation	Updated BIF File Parameters, Boot Image Format and Boot Header Table.
05/03/2017 Version 4.0	
Chapter 2: Programming View of Zynq UltraScale+ MPSoC Devices	Added Boot Process
Chapter 4: Software Stack	Added information about Linux software stack exception levels ELO-EL3.
Chapter 7: System Boot and Configuration	Added QSPI24 and QSPI32 Boot Modes , eMMC18 Boot Mode , JTAG Boot Mode , USB Boot Mode . Updated Setting FSBL Compilation Flags to include FSBL_USB_EXCLUDE .
Chapter 8: Security Features	Added Bitstream Authentication Using External Memory , System Memory Management Unit , A53 Memory Management Unit , and R5 Memory Protection Unit . Updated Encryption and Authentication sections.
Chapter 16: Boot Image Creation	Added parameters and descriptions in Table 16-1. Added Boot Image Format. Added additional bit descriptions in Table 16-9.
	Added Appendixes for OS & Libraries content (Appendixes A-K).
12/15/2016 Version 3.0	
Chapter 1: About This Guide	Updated Introduction
Chapter 7: System Boot and Configuration	Updated Boot Modes
10/05/2016 Version 2.0	
Chapter 2: Programming View of Zynq UltraScale+ MPSoC Devices	Updated Boot Modes and removed Interrupt Features.

Section	Revision Summary
Chapter 3: Development Tools	Added Vivado Design Suite . Modified Supported features in Xilinx Software Development Kit. Added a link to the SDK_Download. Replaced PetaLinux figure with table in Arm GNU Tools section.
Chapter 4: Software Stack	Added FreeRTOS Software Stack
Chapter 5: Software Development Flow	Removed Developing Open Source Software.
Chapter 6: Software Design Paradigms	Added Frameworks for Multiprocessor Development
Chapter 7: System Boot and Configuration	Modified SD Mode diagram, Figure 7-2. Modified NAND Mode diagram Figure 7-4. Removed Keys organization in the CSU and Wake UP Mechanisms. Added Pre-Boot Sequence .
Chapter 8: Security Features	Updated chapter and removed Encryption Key Types and Key Registers table.
Chapter 9: Platform Management	Added Power Management Framework and updated PMU Firmware
DMA	Removed chapter
System Coherency	Removed chapter
Chapter 16: Boot Image Creation	Added new chapter
11/18/2015 Version 1.0	
Initial release.	N/A

Table of Contents

Revision History	2
Chapter 1: About This Guide	10
Introduction.....	10
Intended Audience and Scope of this Document.....	11
Prerequisites.....	11
Chapter 2: Programming View of Zynq UltraScale+ MPSoC	
Devices	13
Hardware Architecture Overview.....	13
Boot Process.....	16
Virtualization.....	19
System Level Reset Requirements.....	19
Security.....	20
Safety and Reliability.....	23
Memory Overview for APU and RPU Executables.....	26
Chapter 3: Development Tools	28
Vivado Design Suite.....	28
Vitis Unified Software Platform.....	30
Arm GNU Tools.....	32
Device Tree Generator.....	33
PetaLinux Tools.....	33
Linux Software Development using Yocto.....	34
Chapter 4: Software Stack	37
Bare Metal Software Stack.....	37
Linux Software Stack.....	40
Third-Party Software Stack.....	44
Chapter 5: Software Development Flow	45
Bare Metal Application Development.....	46
Application Development Using PetaLinux Tools.....	48

Linux Application Development Using Vitis.....	48
Chapter 6: Software Design Paradigms.....	53
Frameworks for Multiprocessor Development.....	53
Symmetric Multiprocessing (SMP).....	54
Asymmetric Multiprocessing (AMP).....	55
Chapter 7: System Boot and Configuration.....	59
Boot Process Overview.....	59
Boot Flow.....	59
Boot Image Creation.....	61
Boot Modes.....	64
Detailed Boot Flow.....	69
Disabling FPD in Boot Sequence.....	72
Setting FSBL Compilation Flags.....	72
FSBL Build Process.....	75
Chapter 8: Security Features.....	100
Boot Time Security.....	100
Bitstream Authentication Using External Memory.....	112
Run-Time Security.....	114
Arm Trusted Firmware.....	115
FPGA Manager Solution.....	118
Xilinx Memory Protection Unit.....	120
Xilinx Peripheral Protection Unit.....	121
System Memory Management Unit.....	121
A53 Memory Management Unit.....	122
R5 Memory Protection Unit.....	122
Chapter 9: Platform Management.....	123
Platform Management in PS.....	123
Wake Up Mechanisms.....	126
Platform Management for Memory.....	127
DDR Controller.....	127
Platform Management for Interconnects.....	127
PMU Firmware.....	128
Chapter 10: Platform Management Unit Firmware.....	129
Features.....	129

PMU Firmware Architecture.....	130
Execution Flow.....	131
Handling Inter-Process Interrupts in PMU firmware.....	133
PMU Firmware Modules.....	137
Error Management (EM) Module.....	140
Power Management (PM) Module.....	146
Scheduler.....	147
Safety Test Library.....	147
CSU/PMU Register Access.....	148
Timers.....	149
Configuration Object.....	152
PMU Firmware Loading Options.....	154
PMU Firmware Usage.....	160
PMU Firmware Memory Layout and Footprint.....	166
Dependencies.....	168
Chapter 11: Power Management Framework.....	169
Introduction.....	169
Zynq UltraScale+ MPSoC Power Management Overview.....	171
Power Management Framework Overview.....	175
Using the API for Power Management.....	188
XilIPM Implementation Details.....	194
Linux.....	197
Arm Trusted Firmware (ATF).....	214
PMU Firmware.....	217
Chapter 12: Reset.....	220
System-Level Reset.....	220
Block-Level Resets.....	220
Application Processing Unit Reset.....	221
Real Time Processing Unit Reset.....	222
Full Power Domain Reset.....	222
Warm Restart.....	222
Supported Use Cases.....	226
Chapter 13: High-Speed Bus Interfaces.....	248
USB 3.0.....	248
Gigabit Ethernet Controller.....	251
PCI Express.....	254

Chapter 14: Clock and Frequency Management	260
Changing the Peripheral Frequency.....	260
Chapter 15: Target Development Platforms	262
QEMU.....	262
Boards and Kits.....	262
Chapter 16: Boot Image Creation	263
Appendix A: Standalone Library v7.2	264
Xilinx Hardware Abstraction Layer API.....	264
MicroBlaze Processor API.....	278
Cortex R5 Processor API.....	287
ARM Processor Common API.....	310
Cortex A9 Processor API.....	313
Cortex A53 32-bit Processor API.....	345
Cortex A53 64-bit Processor Boot Code.....	358
Appendix B: LwIP 2.1.1 Library	370
Introduction.....	370
Using lwIP.....	371
LwIP Library APIs.....	381
Appendix C: XilIsf Library v5.15	387
Overview.....	387
XilIsf Library API.....	388
Library Parameters in MSS File.....	404
Appendix D: XilFFS Library v4.3	406
XilFFS Library API Reference.....	406
Library Parameters in MSS File.....	408
Appendix E: XilSecure Library v4.2	411
Overview.....	411
AES-GCM.....	412
RSA.....	423
SHA-3.....	429
XilSecure Utilities.....	436
Additional References.....	438

Appendix F: XilSkey Library v4.9	440
Overview	440
BBRAM PL API	445
Zynq UltraScale+ MPSoC BBRAM PS API	447
Zynq eFUSE PS API	449
Zynq UltraScale+ MPSoC eFUSE PS API	451
eFUSE PL API	464
CRC Calculation API	467
User-Configurable Parameters	469
Error Codes	493
Status Codes	503
Procedures	503
Data Structure Index	505
Appendix G: XilPM Library v3.1	509
XilPM Zynq UltraScale+ MPSoC APIs	509
Error Status	544
Data Structure Index	546
Appendix H: XilFPGA Library v5.2	549
Overview	549
XilFPGA APIs	558
Appendix I: XilMailbox v1.2	565
Overview	565
Data Structure Index	574
Appendix J: Additional Resources and Legal Notices	575
Xilinx Resources	575
Documentation Navigator and Design Hubs	575
References	575
Please Read: Important Legal Notices	578

About This Guide

Introduction

This document provides the software-centric information required for designing and developing system software and applications for the Xilinx® Zynq® UltraScale+™ MPSoCs. The Zynq UltraScale+ MPSoC family has different products, based upon the following system features:

- Application processing unit (APU):
 - Dual or Quad-core Arm® Cortex™-A53 MPCore
 - CPU frequency up to 1.5 GHz
- Real-time processing unit (RPU):
 - Dual-core Arm Cortex™-R5F MPCore
 - CPU frequency up to 600 MHz
- Graphics processing unit (GPU):
 - Arm Mali-400 MP2
 - GPU frequency up to 667 MHz
- Video codec unit (VCU):
 - Simultaneous Encode and Decode through separate cores
 - H.264 high profile level 5.2 (4Kx2K-60)
 - H.265 (HEVC) main, main10 profile, level 5.1, high Tier, up to 4Kx2K-60 rate
 - 8 and 10-bit encoding
 - 4:2:0 and 4:2:2 chroma sampling

For more details, see the [Zynq UltraScale+ MPSoC Product Table](#) and the [Product Advantages](#).

Intended Audience and Scope of this Document

The purpose of this guide is to enable software developers and system architects to become familiar with:

- Xilinx software development tools.
- Available programming options.
- Xilinx software components that include device drivers, middleware stacks, frameworks, and example applications.
- Platform management unit firmware (PMU firmware), Arm Trusted Firmware (ATF), OpenAMP, PetaLinux tools, Xen Hypervisor, and other tools developed for the Zynq UltraScale+ MPSoC device.

Prerequisites

This document assumes that you are:

- Experienced with embedded software development
- Familiar with Armv7 and Armv8 architecture
- Familiar with Xilinx development tools such as the Vivado® Integrated Design Environment (IDE), the Vitis™ unified software platform, compilers, debuggers, and operating systems.

This document includes the following chapters:

- [Chapter 2: Programming View of Zynq UltraScale+ MPSoC Devices](#): Briefly explains the architecture of the Zynq UltraScale+ MPSoC hardware. Xilinx recommends you to go through and understand each feature of this chapter.
- [Chapter 3: Development Tools](#): Provides a brief description about the Xilinx software development tools. This chapter helps you to understand all the available features in the software development tools. It is recommended for software developers to go through this chapter and understand the procedure involved in building and debugging software applications.
- [Chapter 4: Software Stack](#): Provides a description of various software stacks such as bare metal software, RTOS-based software and the full-fledged Linux stack provided by Xilinx for developing systems with the Zynq UltraScale+ MPSoC device.
- [Chapter 5: Software Development Flow](#): Walks you through the software development process. It also provides a brief description of the APIs and drivers supported in the Linux OS and bare metal.

- [Chapter 6: Software Design Paradigms](#): Helps you understand different approaches to develop software on the heterogeneous processing systems. After reading this chapter, you will have a better understanding of programming in different processor modes like symmetric multi-processing (SMP), asymmetric multi-processing (AMP), virtualization, and a hybrid mode that combines SMP and AMP.
- [Chapter 7: System Boot and Configuration](#): Describes the booting process using different booting devices in both secure and non-secure modes.
- [Chapter 8: Security Features](#): Describes the Zynq UltraScale+ MPSoC devices features you can leverage to enhance security during application boot- and run-time.
- [Chapter 9: Platform Management](#): Describes the features available to manage power consumption, and how to control the various power modes using software.
- [Chapter 10: Platform Management Unit Firmware](#): Describes the features and functionality of PMU firmware developed for Zynq UltraScale+ MPSoC device.
- [Chapter 11: Power Management Framework](#): Describes the functionality of the Xilinx Power Management Framework (PMF) that supports a flexible power management control through the platform management unit (PMU).
- [Chapter 12: Reset](#): Explains the system and module-level resets.
- [Chapter 13: High-Speed Bus Interfaces](#): Explains the configuration flow of the high-speed interface protocols.
- [Chapter 14: Clock and Frequency Management](#): Briefly explains the clock and frequency management of peripherals in Zynq UltraScale+ MPSoC devices.
- [Chapter 15: Target Development Platforms](#): Explains about the different development platforms available for the Zynq UltraScale+ MPSoC device, such as quick emulators (QEMU), and the Zynq UltraScale+ MPSoC boards and kits.
- [Chapter 16: Boot Image Creation](#): Describes Bootgen, a standalone tool for creating a bootable image for Zynq UltraScale+ MPSoC devices. Bootgen is included in the Vitis software platform.
- [Appendix A - Appendix K](#): Describe the available libraries and board support packages to help you develop a software platform.
- [Appendix J: Additional Resources and Legal Notices](#): Provides links to additional information that is cited throughout the document.

Programming View of Zynq UltraScale+ MPSoC Devices

The Zynq[®] UltraScale+[™] MPSoC supports a wide range of applications that require heterogeneous multiprocessing. Heterogeneous multiprocessing system consists of multiple single and multi-core processors of differing types. It supports the following features:

- Multiple levels of security
- Increased safety
- Advanced power management
- Superior processing, I/O, and memory bandwidth
- A design approach, based on heterogeneous multiprocessing presents design challenges, which includes:
 - Meeting application performance requirements within a specified power envelope
 - Optimizing memory access within heterogeneous multiprocessing system
 - Providing low-latency, coherent communications between various processing engines
 - Managing and optimizing system power consumption in all operational modes

Xilinx[®] provides comprehensive tools for hardware and software development on the Zynq UltraScale+ MPSoC, and various software modules such as operating systems, heterogeneous system software, and security management modules.

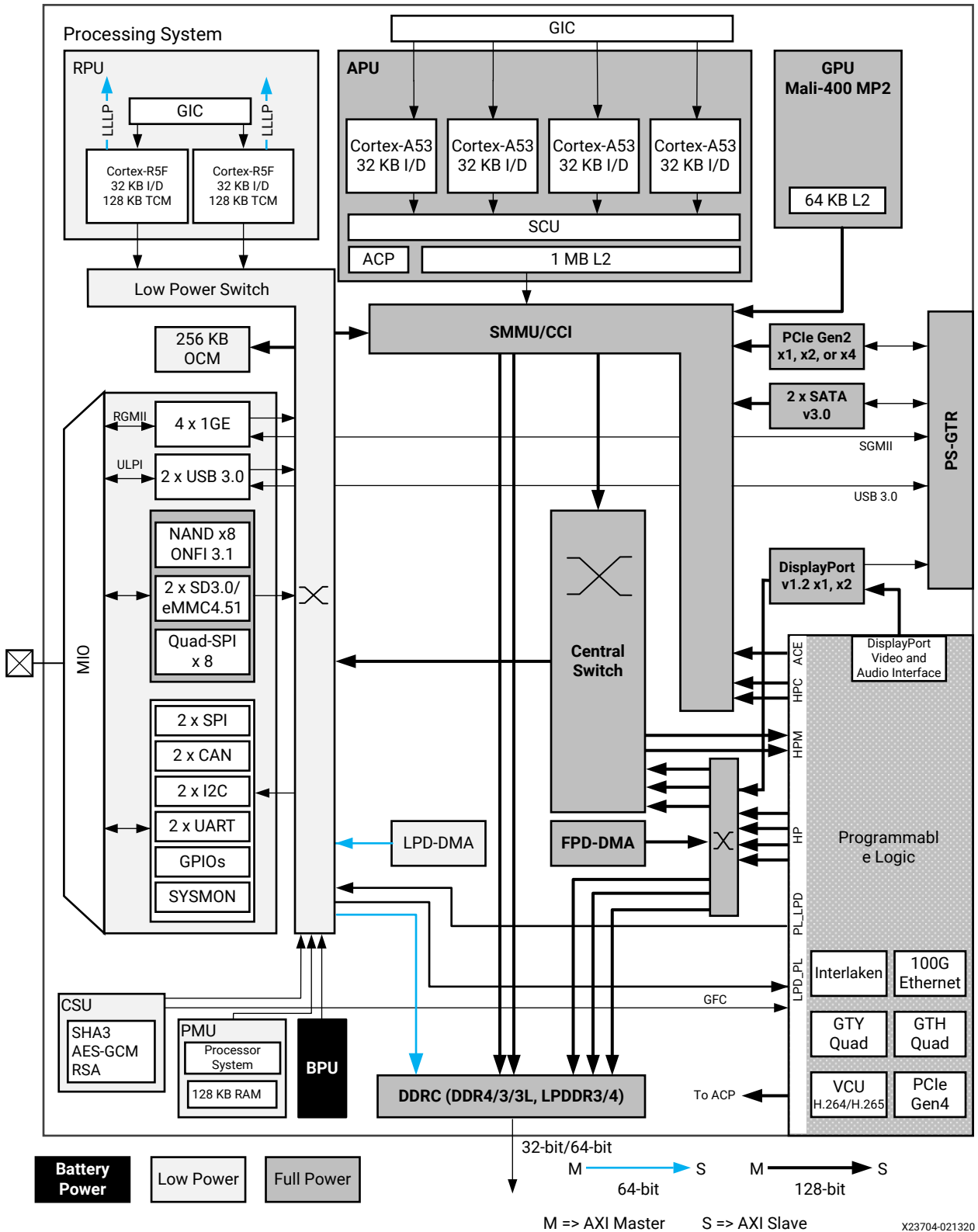
The Zynq UltraScale+ MPSoC is a heterogeneous device that includes the Arm[®] Cortex[™]-A53, high-performance, energy-efficient, 64-bit application processor, and also the 32-bit Arm Cortex[™]-R5F dual-core real-time processor.

Hardware Architecture Overview

The Zynq UltraScale+ MPSoCs provide power savings, programmable acceleration, I/O, and memory bandwidth. These features are ideal for applications that require heterogeneous multiprocessing.

The following figure shows the Zynq UltraScale+ MPSoC architecture with next-generation programmable engines for security, safety, reliability, and scalability.

Figure 1: Zynq UltraScale+ MPSoC Device Hardware Architecture



The Zynq UltraScale+ MPSoC features are as follows:

- Cortex-R5F dual-core real-time processor unit (RPU)
- Arm Cortex-A53 64-bit quad/dual-core processor unit (APU)
- Mali-400 MP2 graphic processing unit (GPU)
- External memory interfaces: DDR4, LPDDR4, DDR3, DDR3L, LPDDR3, 2x Quad-SPI, and NAND
- General connectivity: 2x USB 3.0, 2x SD/SDIO, 2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 1GE, and GPIO
- Security: Advanced Encryption Standard (AES), RSA public key encryption algorithm, and Secure Hash Algorithm-3 (SHA-3)
- AMS system monitor: 10-bit, 1 MSPS ADC, temperature, voltage, and current monitor
- The processor subsystem (PS) has five high-speed serial I/O (HSSIO) interfaces supporting the protocols:
 - PCIe®: base specification, version 2.1 compliant, and Gen2x4
 - SATA 3.0
 - DisplayPort: Implements a DisplayPort source-only interface with video resolution up to 4k x 2k
 - USB 3.0: Compliant to USB 3.0 specification implementing a 5 Gb/s line rate
 - Serial GMII: Supports a 1 Gb/s SGMII interface
- Platform Management Unit (PMU) for functions that include power sequencing, safety, security, and debug.

For more details, see the following sections of the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)): APU, RPU, PMU, GPU, and inter-processor interrupt (IPI).

Boot Process

The platform management unit (PMU) and configuration security unit (CSU) manage and perform the multi-staged booting process. You can boot the device in either secure or non-secure mode. See [Boot Process Overview](#) or, see the Boot and Configuration chapter of the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)).

Boot Modes

You can use any of the following as the boot mode for booting from external devices:

- Quad SPI flash memory (QSPI24, QSPI32)
- eMMC18
- NAND
- Secure Digital Interface Memory (SD0, SD1)
- JTAG
- USB

The bootROM does not directly support booting from SATA, Ethernet, or PCI Express (PCIe). The boot security does not rely on, and is largely orthogonal to TrustZone (TZ). The bootROM (running on the Platform Management Unit) performs the security resources management (for example, key management) and establishes root-of-trust. It authenticates FSBL, locks boot security resources, and transfers chain-of-trust control to FSBL (either on APU or RPU).

To understand more about the boot process in the different boot modes, see the 'Boot and Configuration' chapter of the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)).

QSPI24 and QSPI32

The QSPI boot mode supports the following:

- x1, x2, and x4 read modes for single Quad SPI flash memory (QSPI24) and x8 for dual QSPI
- Image search for MultiBoot
- I/O mode is not supported in FSBL

Note: Single Quad-SPI memory (x1, x2 and x4) is the only boot mode that supports execute-in-place (XIP).

For additional information, see [QSPI24 and QSPI32 Boot Modes](#).

eMMC18

The eMMC18 boot mode supports:

- FAT 16 and FAT 32 file systems for reading the boot images.
- Image search for MultiBoot. The maximum number of searchable files as part of an image search for MultiBoot is 8,191.

For additional information, see [eMMC18 Boot Mode](#).

NAND

The NAND boot supports the following:

- 8-bit widths for reading the boot images
- Image search for MultiBoot

For additional information, see [NAND Boot Mode](#).

SD

The SD boot supported version is 3.0. This version supports:

- FAT 16/32 file systems for reading the boot images.
- Image search for MultiBoot. The maximum number of searchable files as part of an image search for MultiBoot is 8,191.

For additional information, see [SD Boot Mode](#).

JTAG

You can download any software images needed for the PS and hardware images needed for the PL using JTAG.



IMPORTANT! *In JTAG mode, you can boot the Zynq UltraScale+ MPSoC in non-secure mode only.*

For additional information, see [JTAG Boot Mode](#).

Zynq UltraScale+ devices do not support JTAG accesses while the CPU cores are powered down randomly by the software running on the device.

In case of PetaLinux, these kernel configuration options are known to be incompatible with the JTAG debugger:

- CONFIG_PERF_EVENTS
- CONFIG_FREEZER
- CONFIG_SUSPEND
- CONFIG_PM
- CONFIG_CPU_IDLE

USB

USB boot mode supports USB 3.0. It does not support MultiBoot, image fallback, or XIP. It supports both secure and non-secure boot mode. It is not supported for systems without DDR. USB boot mode is disabled by default. For additional information, see [USB Boot Mode](#).

Virtualization

Virtualization allows multiple software stacks to run simultaneously on the same processor, which enhances the productivity of the Zynq UltraScale+ MPSoC. The role of virtualization varies from system to system. For some designers, virtualization allows the processor to be kept fully loaded at all times, saving power and maximizing performance. For others systems, virtualization provides the means to partition the various software stacks for isolation or redundancy.

For more information, see [System Virtualization](#) in the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*.

The support for virtualization applies only to an implementation that includes Arm exception level-2 (EL2). Armv8 supports virtualization extension to achieve full virtualization with near native guest operating systems performance. There are three key hardware components for virtualization:

- CPU virtualization
- Interrupt virtualization
- System MMU for I/O virtualization

System Level Reset Requirements

The system-level reset term is used to describe the system or subsystem level resets. ‘System’ reset (different from system-level resets) is a specific type of system-level reset. The following table provides summary of system-level resets, which are described in details in subsequent sections.

Table 1: System-Level Resets

Reset Type	Description
External POR	The external POR reset is triggered by external pin assertion. There are a number of software only registers which are not reset by the POR resets. At first POR boot, a safety system (requiring HFT1 by PS & PL) can be configured such that a subsequent POR only resets PS (and not PL).
Internal POR	Internal POR reset can be triggered by software register write, or by safety errors. With the exception of error status register (which are reset by external POR, but not by internal POR), internal POR resets the same thing as external reset does. Internal-POR cannot be guaranteed without silicon validation (due to in-rush power concern), so internal-POR is for internal purpose unless validated.
System Reset	System reset is to be able to reset system excluding debug logic. To simplify system reset, there are few other things (xBIST, scan clear, power gating) which are not reset by this reset. Also, boot mode information is not reset by system reset. The system reset can be triggered by external pin (SRST), or software register write, or by safety errors.

Table 1: System-Level Resets (cont'd)

Reset Type	Description
PS Only Reset	The PS only reset is to reset the PS while the PL remains active. This reset can be triggered by hardware error signals or by software register write. This reset is a subset of system reset (excluding the PL reset). If the PS reset is triggered by an error signal, then the error is also transmitted to the PL.
FPD Reset	The FPD reset resets all of the FPD power domain. It can be triggered by errors or software register write. If the FPD reset is triggered by an error signal, then the error is also transmitted to LPD & PL.
RPU Reset	The RPU Reset is to reset the RPU in case of errors. While each of the R5 core can be independently reset, but in lockstep, only R5_0 needs to be reset to reset both the R5 cores. This reset can be triggered by errors or software register write.

Security

The increasing ubiquity of Xilinx devices makes protecting the intellectual property (IP) within them as important as protecting the data processed by the device. As security threats have increased, the range of security threats or potential weaknesses that must be considered to deploy secure products has grown as well.

The Zynq UltraScale+ MPSoC provides the following features to help secure applications running on the SoC:

- Encryption and authentication of configuration files.
- Hardened crypto accelerators for use by the user application.
- Secure methods of storing cryptographic keys.

Methods for detecting and responding to tamper events. See the [Security](#) chapter of the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)* for more information.

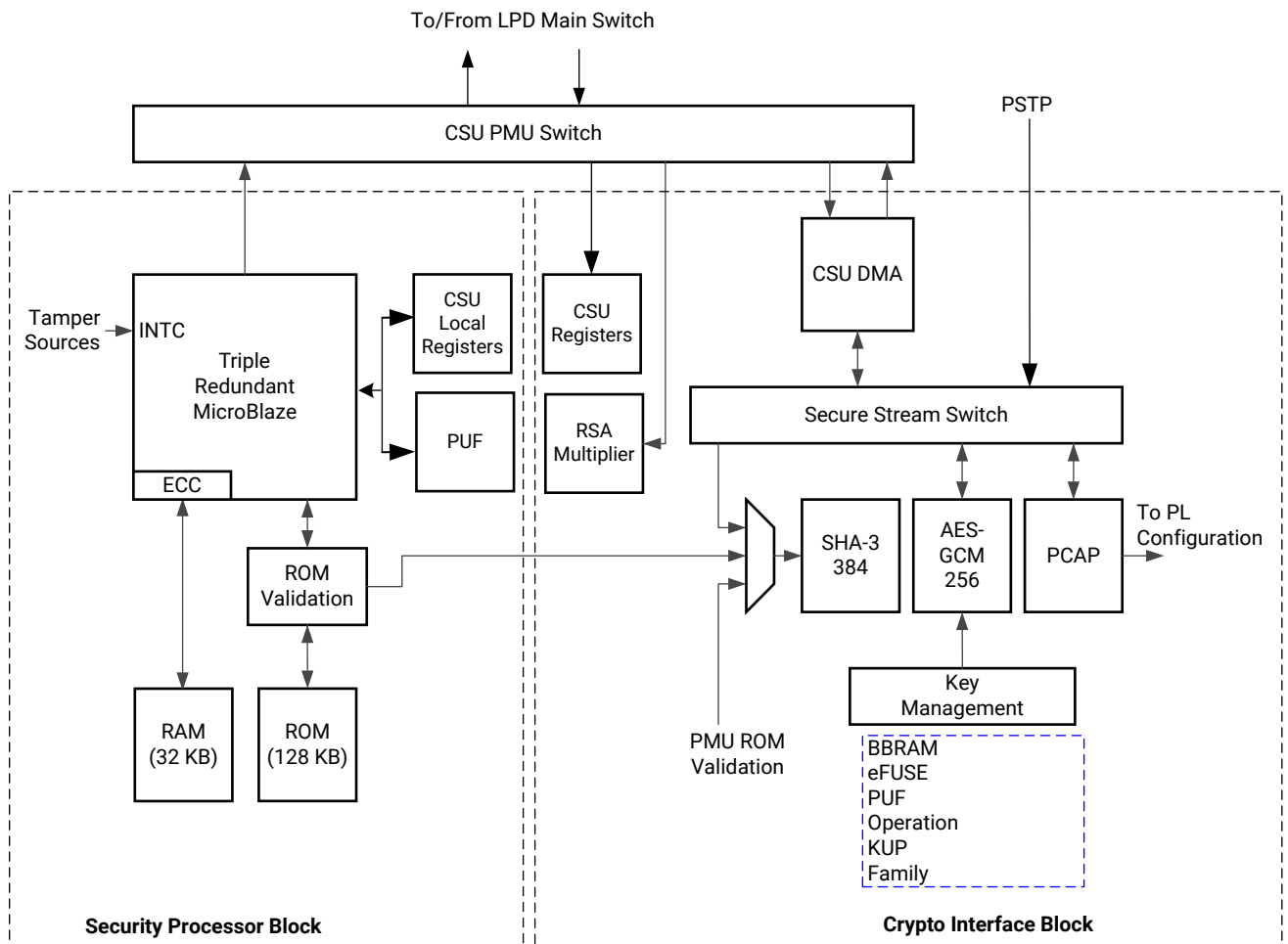
Configuration Security Unit

The following are some of the important responsibilities of the configuration security unit (CSU):

- Secure boot.
- Tamper monitoring and response.
- Secure key storage and management.
- Cryptographic hardware acceleration.

The CSU comprises two main blocks as shown in the following figure. On the left is the secure processor block that contains a triple redundant processor for controlling boot operation. It also contains an associated ROM, a small private RAM, and the necessary control/status registers required to support all secure operations. The block on the right is the crypto interface block (CIB) and contains the AES-GCM, DMA, SHA, RSA, and PCAP interfaces.

Figure 2: Configuration and Security Unit Architecture



After boot, the CSU provides tamper response monitoring. These crypto interfaces are available during runtime. To understand how to use these features, see Appendix K, XilFPGA Library v5.0. See the [Security](#) chapter of the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)* for more information.

- **Secure Processor Block:** The triple-redundant processor architecture enhances the CSU operations during single event upset (SEU) conditions.
- **Crypto Interface Block (CIB):** Consists of AES-GCM, DMA, SHA-3/384, RSA, and PCAP interfaces.

- **AES-GCM:** The AES-GCM core has a 32-bit word-based data interface, with 256-bits of key support.
- **Key Management:** To use the AES, a key must be loaded into the AES block. The key is selected by CSU bootROM.
- **SHA-3/384:** The SHA-3/384 engine is used to calculate a hash value of the input image for authentication.
- **RSA-4096 Accelerator:** Facilitates RSA authentication.

To understand boot image encryption or authentication, refer to the following:

- [Chapter 7: System Boot and Configuration](#)
- [Chapter 16: Boot Image Creation](#)
- The [Security](#) chapter of the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*.
- [Boot and Configuration](#) information in the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*.

System-Level Protections

The system-level protection mechanism involves the following areas:

- Zynq UltraScale+ MPSoC system software stack relies on the Arm Trusted Firmware (ATF). Protection can be enhanced even further by configuring the XMPU and XPPU to provide the system-level run-time security.
 - Protection against buggy or malicious software (erroneous software) from corrupting system memory or causing a system failure.
 - Protection against incorrect programming, or malicious devices (erroneous hardware) from corrupting system memory or causing a system failure.
 - Memory (DDR, OCM) and peripherals (peripheral control, SLCRs) are protected from illegal accesses by erroneous software or hardware to protect the system.
- The Xilinx memory protection unit (XMPU) enforces memory partitioning and TrustZone (TZ) protection for memory and FPD slaves. The XMPU can be configured to isolate a master or a given set of masters to a developer-defined set of address ranges.
- The Xilinx peripheral protection unit (XPPU) provides LPD peripheral isolation and inter-processor interrupt (IPI) protection. The XPPU can be configured to permit one or more masters to access an LPD peripheral. For more information, see the [XPPU Protection of Slaves](#) section of the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*.

Safety and Reliability

The Zynq UltraScale+ MPSoC architecture includes features that enhance the reliability of safety critical applications to give users and designers increased confidence in their systems. The key features are as follows:

- Memory and cache error detection and correction
- RPU safety features
- System-wide safety features

To understand how to use these features, see [Chapter 8: Security Features](#).

Safety Features

The Cortex-A53 MPCore processor supports cache protection in the form of ECC on all RAM instances in the processor using the following separate protection elements:

- SCU-L2 cache protection
- CPU cache protection

These elements enable the Cortex-A53 MPCore processor to detect and correct a 1-bit error in any RAM, and to detect 2-bit errors.

Cortex-A53 MPCore RAMs are protected against single-event-upset (SEU) such that the processor system can detect and then, take specific action to continue making progress without data corruption. Some RAMs have parity single-error detect (SED) capability, while others have ECC single-error correct, double-error detect (SECEDED) capability.

The RPU includes two major safety features:

- Lock-step operation, shown in the following figure.
- Error checking and correction, described further in [Error Checking and Correction](#).

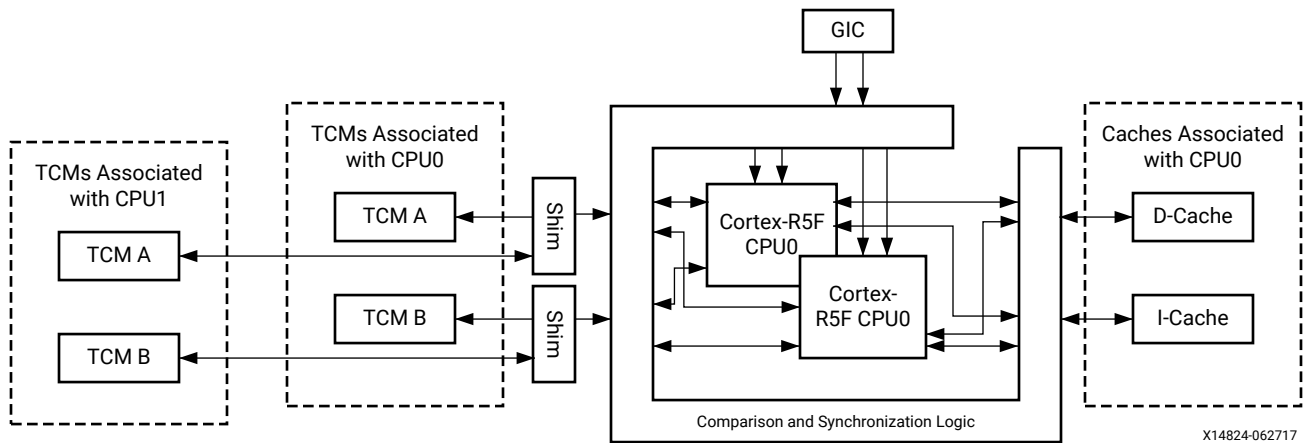
Lock-Step Operation

Cortex-R5F processors support lock-step operation mode, which operates both RPU CPU cores as a redundant CPU configuration called safety mode.

The Cortex-R5F processor set to operate in the lock-step configuration exposes only one CPU interface. Because Cortex-R5F processor only supports the static split and lock configuration, switching between these modes is permitted only while the processor group is held in power-onreset (POR). The input signals SLCLAMP and SLSPLIT control the mode of the processor group.

These signals control the multiplex and clamp logic in the lock-step configuration. When the Cortex-R5F processors are in the lock-step mode (shown in the following figure), there must be code in the reset handler to manage that the distributor within the GIC dispatches interrupts only to CPU0. The RPU includes a dedicated interrupt controller for Cortex-R5F MPCore processors. This Arm PL390 generic interrupt controller (GIC) is based on the GICv1 specification.

Figure 3: RPU Lock-Step Operation



Tightly coupled memories (TCMs) are mapped in the local address space of each Cortex-R5F processor; however, they are also mapped in the global address space where any master can access them provided that the XPPU is configured to allow such accesses.

The following table lists the address maps from the RPU point of view:

Table 2: RPU Address Maps

Operation Mode	Memory	R5_0 View (Start Address)	R5_1 View (Start Address)	Global Address View (Start Address)
Split Mode	R5_0 ATCM (64 KB)	0x0000_0000	N/A	0xFFE0_0000
	R5_0 BTCM (64 KB)	0x0002_0000	N/A	0xFFE2_0000
	R5_0 instruction cache	I-Cache	N/A	0xFFE4_0000
	R5_0 data cache	D-Cache	N/A	0xFFE5_0000
Split Mode	R5_1 ATCM (64 KB)	N/A	0x0000_0000	0xFFE9_0000
	R5_1 BTCM (64 KB)	N/A	0x0002_0000	0xFFEB_0000
	R5_1 instruction cache	I-Cache	N/A	0xFFEC_0000
	R5_1 data cache	D-Cache	N/A	0xFFED_0000
Lock-step Mode	R5_0 ATCM (128 KB)	0x0000_0000	N/A	0xFFE0_0000
	R5_0 BTCM (128 KB)	0x0002_0000	N/A	0xFFE2_0000
	R5_0 instruction cache	I-Cache	N/A	0xFFE4_0000
	R5_0 data cache	D-Cache	N/A	0xFFE5_0000

Error Checking and Correction

The Cortex-R5F processor supports error checking and correction (ECC) schemes of data. The data has similar properties although the size of the data chunk to which the ECC scheme applies is different.

For each aligned data chunk, the processor computes and stores a number of redundant code bits with the data. This enables the processor to detect up to two errors in the data chunk or its code bits, and correct any single error in the data chunk or its associated code bits. This is also referred to as a single-error correction, double-error detection (SEC-DED) ECC scheme.

System-Wide Safety Features

The system-wide safety features are designed to address error-free operation of the Zynq UltraScale+ MPSoC.

These features include the following:

Platform Management Unit

The platform management unit (PMU) in the Zynq UltraScale+ MPSoC executes the code loaded from ROM and RAM within a flat memory space, implements power safety routines to prevent tampering of PS voltage rails, performs logic built-in self-test (LBIST), and responds to a user-driven power management sequence.

The PMU also includes some registers to control the functions that are typically very critical to the operation and safety of the device. Some of the registers related to safety are as follows:

- GLOBAL_RESET: Contains reset for safety-related blocks.
- SAFETY_GATE: Gates hardware features from accidental enablement.
- SAFETY_CHK: Checks the integrity of the interconnect data lines by using target registers for safety applications by periodically writing to and reading from these registers.

PMU Triple-Redundancy

The power management unit (PMU) contains triple-redundant embedded processors for a high-level of system reliability and strong SEU resilience. PMU controls the power-up, reset, and monitoring of resources within the entire system. The PMU performs multiple tasks including the following tasks:

- Initializing the system during boot
- Managing power gating and retention states for different power domains and islands
- Communicating the supply voltage settings to the external power control devices
- Managing sleep states including the deep-sleep mode and processing of wake functions

More details about PMU are available in [Chapter 9: Platform Management](#).

Interrupts

The generic interrupt controller (GIC) handles interrupts. Both the APU and the RPU have a separate dedicated GIC for interrupt handling. The RPU includes an Arm PL390 GIC, which is based upon the GICv1 specification due to its flexibility and protection. The APU includes a GICv2 controller. The GICv2 is a centralized resource for supporting and managing interrupts in multi-processor systems. It aids the GIC virtualization extensions that support the implementation of the GIC in systems supporting processor virtualization.

The Zynq UltraScale+ MPSoC embeds an inter-processor interrupt (IPI) block that aids in communication between the heterogeneous processors. Because PMUs can communicate with different processors simultaneously, the PMU has four IPIs connected to the GIC of the PMU.

For more information on IPI routing to different processors, see the “Interrupts” chapter in the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)).

Memory Overview for APU and RPU Executables

The following tables give the configurable memory regions for APUs and RPUs.

Note:

- In RPU lock-step mode ([Lock-Step Operation](#)), R5_0_ATCM_MEM_0 and R5_0_BTCM_MEM_0 memory address are mapped to R5_0_ATCM_LSTEP and R5_0_BTCM_LSTEP memory ranges respectively in the system address map.
- In RPU split mode, R5_x_ATCM_MEM_0 and R5_x_BTCM_MEM_0 memory address are mapped to R5_x_ATCM_SPLIT and R5_x_BTCM_SPLIT memory ranges respectively in the system address map.
- QSPI memory is accessible when QSPI controller is in linear mode.

See the [System Addresses](#) chapter of the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)) for more information.

See Real-time Processing Unit (RPU) and On-Chip Memory (OCM) sections of the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)) for more information on RPU, R5 and OCM.

Table 3: Configurable Memory Regions for APUs

Memory Type	Start Address	Size
DDR Low	0x00000000	2 GB
DDR High	0x80000000	2 GB

Table 3: Configurable Memory Regions for APUs (cont'd)

Memory Type	Start Address	Size
OCM	0xFFFFC0000	256 KB
QSPI	0xC0000000	512 MB

Table 4: Configurable Memory Regions for RPU Lock-Step Mode

Memory Type	Start Address	Size
DDR Low	0x100000	2047 MB
OCM	0xFFFFC0000	256 KB
QSPI	0xC0000000	512 MB
R5_0_ATCM_MEM_0	0x000000	64 KB
R5_0_BTCM_MEM_0	0x200000	64 KB
R5_TCM_RAM_0_MEM	0x000000	256 KB

Table 5: Configurable Memory Regions for RPU Split Mode

Memory Type	Start Address	Size
R5_0		
DDR Low	0x100000	2047 MB
OCM	0xFFFFC0000	256 KB
QSPI	0xC0000000	512 MB
R5_0_ATCM_MEM_0	0x000000	64 KB
R5_0_BTCM_MEM_0	0x200000	64 KB
R5_1		
DDR Low	0x100000	2047 MB
OCM	0xFFFFC0000	256 KB
QSPI	0xC0000000	512 MB
R5_1_ATCM_MEM_0	0x000000	64 KB
R5_1_BTCM_MEM_0	0x200000	64 KB

Note: BootROM always copies First Stage Boot Loader (FSBL) from 0xFFFFC0000 and it is not configurable. If FSBL is compiled for a different load address, Bootgen may refuse it as CSU bootROM (CBR) does not parse partition headers in the boot image but merely copies the FSBL code at a fixed OCM memory location (0xffffc0000). See [Chapter 7: System Boot and Configuration](#) for more information on Bootgen.

Development Tools

This chapter focuses on Xilinx[®] tools and flows available for programming software for Zynq[®] UltraScale+™ MPSoCs. However, the concepts are generally applicable to third-party tools as the Xilinx tools incorporate familiar components such as an

Eclipse-based integrated development environment (IDE) and the GNU compiler tool chain.

This chapter also provides a brief description about the open source tools available that you can use for open source development on different processors of the Zynq UltraScale+ MPSoC.

A comprehensive set of tools for developing and debugging software applications on Zynq UltraScale+ MPSoC devices includes:

- Hardware IDE
- Software IDEs
- Compiler toolchain
- Debug and trace tools
- Embedded OS and software libraries
- Simulators (for example: QEMU)
- Models and virtual prototyping tools (for example: emulation board platforms)

Third-party tool solutions vary in the level of integration and direct support for Zynq UltraScale+ MPSoC devices.

The following sections provide a summary of the available Xilinx development tools.

Vivado Design Suite

The Xilinx Vivado[®] Design Suite contains tools that are encapsulated in the Vivado integrated design environment (IDE). The IDE provides an intuitive graphical user interface (GUI) with powerful features.

The Vivado Design Suite supersedes the Xilinx ISE software with additional features for system-on-a-chip development and high-level synthesis. It delivers a SoC-strength, IP- and system-centric, next generation development environment built exclusively by Xilinx to address the productivity bottlenecks in system-level integration and implementation.

All of the tools and tool options in Vivado Design Suite are written in native Tool Command Language (Tcl) format, which enables use both in the Vivado IDE or the Vivado Design Suite Tcl shell. Analysis and constraint assignment is enabled throughout the entire design process. For example, you can run timing or power estimations after synthesis, placement, or routing. Because the database is accessible through Tcl, changes to constraints, design configuration, or tool settings happen in real time, often without forcing re-implementation.

The Vivado IDE uses a concept of opening designs in memory. Opening a design loads the design netlist at that particular stage of the design flow, assigns the constraints to the design, and then applies the design to the target device. This provides the ability to visualize and interact with the design at each design stage.



IMPORTANT! *The Vivado IDE supports designs that target 7 series and newer devices only.*

You can improve design performance and ease of use through the features delivered by the Vivado Design Suite, including:

- The Processor Configuration Wizard (PCW) within the IP integrator with graphical user interfaces to let you create and modify the PS within the IP integrator block design.



VIDEO: For a better understanding of the PCW, see the Quick Take Video: [Vivado Processor Configuration Wizard Overview](#).

- Register transfer level (RTL) design in VHDL, Verilog, and SystemVerilog.
- Quick integration and configuration of IP cores from the Xilinx IP catalog to create block designs through the Vivado IP integrator.
- Vivado synthesis.
- C-based sources in C, C++, and SystemC.
- Vivado implementation for place and route.
- Vivado serial I/O and logic analyzer for debugging.
- Vivado power analysis.
- SDC-based Xilinx Design Constraints (XDC) for timing constraints entry.
- Static timing analysis.
- Flexible floorplanning.
- Detailed placement and routing modification.
- Bitstream generation.

- Vivado Tcl Store, which you can use to add to and modify the capabilities in Vivado.

You can download the Vivado Design Suite from the [Xilinx Vivado Design Suite – HLx Editions](#).

Vitis Unified Software Platform

The Vitis™ unified software platform is an integrated development environment (IDE) for the development of embedded software applications targeted towards Xilinx embedded processors. The Vitis software platform works with hardware designs created with Vivado Design Suite. The Vitis software platform is based on the Eclipse open source standard and the features for software developers include:

- Feature-rich C/C++ code editor and compilation environment
- Project management
- Application build configuration and automatic Makefile generation
- Error navigation
- Integrated environment for seamless debugging and profiling of embedded targets
- Source code version control
- System-level performance analysis
- Focused special tools to configure FPGA
- Bootable image creation
- Flash programming
- Script-based command-line tool

The Vitis IDE lets you create software applications using a unified set of Xilinx tools for the Arm® Cortex™-A53 and Cortex™-R5F processors as well as for Xilinx MicroBlaze™ processors. It provides various methods to create applications, as follows:

- Bare metal and FreeRTOS applications for MicroBlaze
- Bare metal, Linux, and FreeRTOS applications for APU
- Bare metal and FreeRTOS applications for RPU
- User customization of PMU firmware
- Library examples are provided with the Vitis tool (ready to load sources and build), as follows:
 - OpenCV
 - OpenAMP RPC
 - FreeRTOS “HelloWorld”

- lwIP
- Performance tests (Dhrystone, memory tests, peripheral tests)
- RSA authentication for preventing tampering or modification of images and bitstream
- First stage boot loader (FSBL) for APU or RPU.

You can export a block design, hardware design files, and bitstream files to the export directory directly from the Vivado Project Navigator. For more information regarding the Vivado Design Suite, see the [Vivado Design Suite Documentation](#).

All processes necessary to successfully complete this export process are run automatically. The Vitis IDE creates a new hardware platform project within the workspace containing the following files:

- `.project`: Project file
- `psu_init.tcl`: PS initialization script
- `psu_init.c`, `psu_init.h`: PS initialization code
- `psu_init.html`: Register summary viewer
- `system.hdf`: Hardware definition file

The compiler can be switched as follows:

- 32-bit or 64-bit (applications that are targeted to Cortex-A53)
- 32-bit only (applications targeted to Cortex-R5F, and Xilinx MicroBlaze devices)

For the list of build procedures, see the *Vitis Unified Software Platform Documentation: Embedded Software Development* ([UG1400](#)), where built-in help content lets you explore further after you launch the Vitis IDE.

The Vitis software platform has the following IDE extensions.

- **XSCT Console:** Xilinx Software Command-line Tool (XSCT) is an interactive and scriptable command-line interface to the Vitis software platform. As with other Xilinx tools, the scripting language for XSCT is based on Tools Command Language (Tcl). You can run XSCT commands interactively or script the commands for automation. XSCT supports the following actions.
 - Creating platform projects and application projects
 - Manage repositories
 - Manage domain settings and add libraries to domains
 - Set toolchain preferences
 - Configure and build applications
 - Download and run applications on hardware targets

- Create and flash boot images by running Bootgen and program_flash tools
- **Bootgen Utility:** Bootgen is a Xilinx tool that lets you stitch binary files together and generate device boot images. Bootgen defines multiple properties, attributes and parameters that are input while creating boot images for use in a Xilinx device. Bootgen comes with both a graphical user interface and a command line option. The tool is integrated into the Vitis software platform for generating basic boot images using a GUI, but the majority of Bootgen options are command line-driven. For more information on the Bootgen utility, see the *Bootgen User Guide* ([UG1283](#)).
- **Program Flash:** Program Flash is a tool used to program the flash memories in the design. Various types of flash types are supported by the Vitis software platform for programming.
- **Repositories:** A software repository is a directory where you can install third-party software components, as well as custom copies of drivers, libraries, and operating systems. When you add a software repository, the Vitis software platform automatically infers all the components contained with the repository and makes them available for use in its environment. Your workspace can point to multiple software repositories.
- **Program FPGA:** You can use the Program FPGA feature to program FPGA using bitstream.
- **Device Tree Generation:** Device tree (DT) is a data structure that describes hardware. This describes hardware that is readable by an operating system like Linux so that it does not need to hard code details of the machine. Linux uses the DT basically for platform identification, runtime configuration like bootargs, and device node population.

For a detailed explanation on the Vitis IDE features, and to understand the embedded software design flow, see the *Vitis Unified Software Platform Documentation: Embedded Software Development* ([UG1400](#)).

You can download the Vitis tool from the [Embedded Design Tools Download](#).

Arm GNU Tools

The Arm GNU open source toolchain is adopted for the Xilinx software development platform. The GNU tools for Linux hosts are available as part of Vitis software platform. This section details the open source GNU tools and Linux tools available for the processing clusters in the Zynq UltraScale+ MPSoC.

The following table lists some of the Xilinx Arm GNU tools available for programming the APU, RPU, and embedded MicroBlaze processors.

Table 6: Xilinx Arm GNU Tools

Tool	Description
aarch64-none-elf-gcc aarch64-none-elf-g++	GNU C/C++ compiler.

Table 6: Xilinx Arm GNU Tools (cont'd)

Tool	Description
aarch64-none-elf-as	GNU assembler.
aarch64-none-elf-ld	GNU linker.
aarch64-none-elf-ar	A utility for creating, modifying, and extracting from archives.
aarch64-none-elf-objcopy	Copies and translates object files.
aarch64-none-elf-objdump	Displays information from object files.
aarch64-none-elf-size	Lists the section sizes of an object or archive file.
aarch64-none-elf-gprof	Displays profiling information.
aarch64-none-elf-gdb	The GNU debugger.

Device Tree Generator

The device tree (DT) data structure consists of nodes with properties that describe a hardware. The Linux kernel uses the device tree to support a wide range of hardware configurations.

In FPGAs, it is possible to have different combinations of peripheral logics, each using a different configuration. For all the different combinations, the device tree generator (DTG) generates the `.dts/.dtsi` device tree files.

The following is a list of the `.dts/.dtsi` files generated by the device tree generator:

- `pl.dtsi`: Contains all the memory mapped peripheral logic (PL) IPs.
- `pcw.dtsi`: Contains the dynamic properties for the PS IPs.
- `system-top.dts`: Contains the memory, boot arguments, and command line parameters.
- `zynqmp.dtsi`: Contains all the PS specific and the CPU information.
- `zynqmp-clk-ccf.dtsi`: Contains all the clock information for the PS peripheral IPs.

For more information, see the [Build Device Tree Blob](#) page on the Xilinx Wiki.

PetaLinux Tools

The PetaLinux tools offer everything necessary to customize, build, and deploy open source Linux software to devices.

PetaLinux tools include the following:

- Build tools such as GNU, `petalinux-build`, and `make` to build the kernel images and the application software.
- Debug tools such as GDB, `petalinux-boot`, and `oprofile` for profiling.

The following table shows the supported PetaLinux tools.

Table 7: PetaLinux Supported Tools

Tools	Description
GNU	Arm GNU tools.
<code>petalinux-build</code>	Used to build software image files.
Make	Make build for compiling the applications.
GDB	GDB tools for debugging.
<code>petalinux-boot</code>	Used to boot Linux.
QEMU	Emulator platform for the Zynq UltraScale+ MPSoC device.
OProfile	Used for profiling.

See the following documentation for more details:

- [PetaLinux Tools documentation](#)
- *Zynq UltraScale+ MPSoC: Embedded Design Tutorial (UG1209)*
- *Libmetal and OpenAMP for Zynq Devices User Guide (UG1186)*

Linux Software Development using Yocto

Xilinx offers the `meta-xilinx` Yocto/OpenEmbedded recipes to enable those customers with in-house Yocto build systems to configure, build, and deploy Linux for Zynq® UltraScale+™ MPSoCs.

The `meta-xilinx` layer also provides a number of BSPs for common boards which use Xilinx devices.

The `meta-xilinx` layer provides additional support for Yocto/OE, adding recipes for various components. See [meta-xilinx](#) for more information.

You can develop Linux software on Cortex-A53 using open source Linux tools. This section explains the Linux Yocto tools and its project development environment.

The following table lists the Yocto tools.

Table 8: Yocto Tools

Tool Type	Name	Description
Yocto build tools	Bitbake	Generic task execution engine that allows shell and Python tasks to be run efficiently, and in parallel, while working within complex inter-task dependency constraints.
Yocto profile and trace tools	Perf	Profiling and tracing tool that comes bundled with the Linux Kernel.
	Ftrace	Refers to the ftrace function tracer but encompasses a number of related tracers along with the infrastructure used by all the related tracers.
	Oprofile	System-wide profiler that runs on the target system as a command-line application.
	Sysprof	System-wide profiler that consists of a single window with three panes, and buttons, which allow you to start, stop, and view the profile from one place.
	Blktrace	A tool for tracing and reporting low-level disk I/O.

Yocto Project Development Environment

Developers can configure the Yocto project development environment to support developing Linux software for Zynq UltraScale+ MPSoCs through Yocto recipes provided from the Xilinx GIT server. You can use components from the Yocto project to design, develop, and build a Linux-based software stack.

The following figure shows the complete Yocto project development environment. The Yocto project has wide range of tools which can be configured to download the latest Xilinx kernel and build with some enhancements made locally in the form of local projects.

You can also change the build and hardware configuration through BSP.

Yocto combines a compiler and other tools to build and test images. After the images pass the quality tests and package feeds required for SDK generation are received, the Yocto tool launches the Vitis IDE for application development.

The important features of the Yocto project are, as follows:

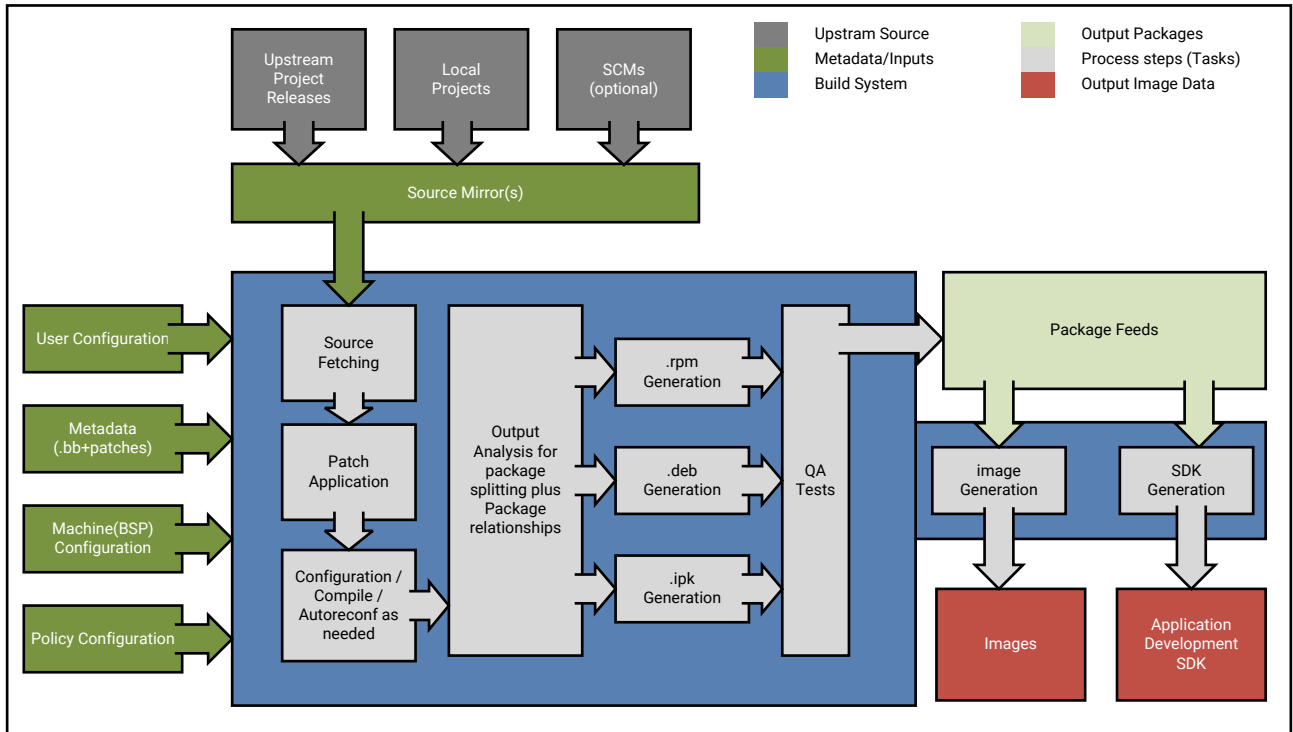
- Provides a recent Linux kernel along with a set of system commands and libraries suitable for the embedded environment.
- Makes available system components such as X11, GTK+, Qt, Clutter, and SDL (among others) so you can create a rich user experience on devices that have display hardware. For devices that do not have a display or where you wish to use alternative UI frameworks, these components need not be installed.

- Creates a focused and stable core compatible with the OpenEmbedded project with which you can easily and reliably build and develop Linux software.
- Supports a wide range of hardware and device emulation through the quick emulator (QEMU). See the *Xilinx Quick Emulator User Guide: QEMU (UG1169)* for more information.



IMPORTANT! Enabling full Yocto of Xilinx QEMU is not available.

Figure 4: Yocto Project Development Environment



X14841-021317

You can download the Yocto tools and the Yocto project development environment from the [Yocto Project Organization](#).

For more information about Xilinx-provided Yocto features, see Yocto Features in the *PetaLinux Tools Documentation: Reference Guide (UG1144)*.

Software Stack

This chapter provides an overview of the various software stacks available for the Zynq[®] UltraScale+™ MPSoC devices.

For more information about the various software development tools used with this device, see [Chapter 3: Development Tools](#). For more information about bare metal and Linux software application development, see [Chapter 5: Software Development Flow](#).

Bare Metal Software Stack

Xilinx[®] provides a bare metal software stack called the standalone board support package (BSP) as part of the Vitis™ software platform. The Standalone BSP gives you a simple, single-threaded environment that provides basic features such as standard input/output and access to processor hardware features. The BSP and included libraries are configurable to provide the necessary functionality with the least overhead. You can locate the standalone drivers at the following path:

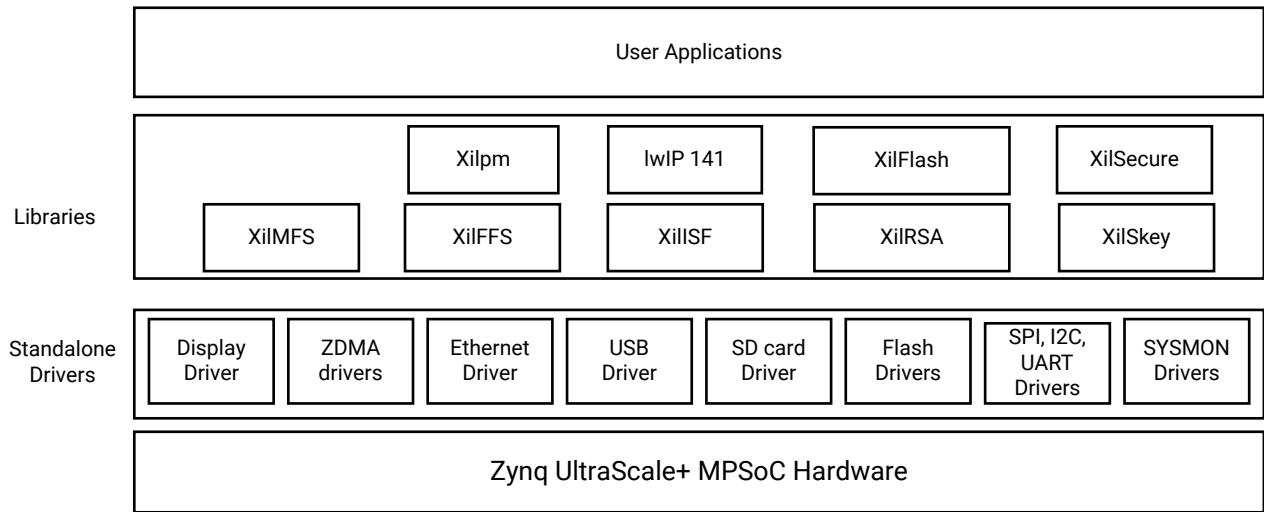
```
<Xilinx Installation Directory>\Vitis\<version>\data\embeddeds\nXilinxProcessorIPLib\drivers
```

You can locate libraries at the following path:

```
<Xilinx Installation Directory>\Vitis\<version>\data\embeddeds\nlib\nsw_services
```

The following figure illustrates the bare metal software stack in the APU.

Figure 5: Bare-Metal Software Development Stack



X17169-062717

Note: The software stack of libraries and drivers layer for bare metal in RPU is same as that of APU.

The key components of this bare metal stack are:

- Software drivers for peripherals including core routines needed for using the Arm® Cortex™-A53, Arm® Cortex™-R5F processors in the PS as well as the Xilinx® MicroBlaze™ processors in the PL.
- Bare metal drivers for PS peripherals and optional PL peripherals.
- Standard C libraries: libc and libm, based upon the open source Newlib library, ported to the Arm Cortex-A53, Arm Cortex-R5F, and the MicroBlaze processors.
- Additional middleware libraries that provide networking, file system, and encryption support.
- Application examples including the first stage boot loader (FSBL) and test applications.

The C Standard Library (libc)

libc library contains standard functions that all C programs can use. The following table lists the libc modules:

Table 9: Libc.a Functions and Descriptions

Header File	Description
alloca.h	Allocates space in the stack
assert.h	Diagnostics code
ctype.h	Character operations
errno.h	System errors

Table 9: Libc.a Functions and Descriptions (cont'd)

Header File	Description
inttypes.h	Integer type conversions
math.h	Mathematics
setjmp.h	Non-local goto code
stdint.h	Standard integer types
stdio.h	Standard I/O facilities
stdlib.h	General utilities functions
time.h	Time function

The C Standard Library Mathematical Functions (libm)

The following table lists the libm mathematical C modules:

Table 10: libm.a Function Types and Function Listing

Function Type	Supported Functions
Algebraic	cbirt, hypot, sqrt
Elementary transcendental	asin, acos, atan, atan2, asinh, acosh, atanh, exp, expm1, pow, log, log1p, log10, sin, cos, tan, sinh, cosh, tanh
Higher transcendentals	j0, j1, jn, y0, y1, yn, erf, erfc, gamma, lgamma, and gamma_ramma_r
Integral rounding	eil, floor, rint
IEEE standard recommended	copysign, fmod, ilogb, nextafter, remainder, scalbn, and fabs
IEEE classification	isnan
Floating point	logb, scalb, significand
User-defined error handling routine	matherr

Standalone BSP

The libraries available with the standalone BSP are as follows:

- XilFatFS: Is a LibXil FATFile system and provides read/write access to files stored on a Xilinx system ACE compact flash.
- XilFFS: Generic Fat File System Library.
- XilFlash: Xilinx flash library for Intel/AMD CFI compliant parallel flash.
- XilISF: In-System Flash library that supports the Xilinx in-system flash hardware.
- XilMFS: Memory file system.
- XilSecure: Xilinx Secure library provides support to access secure hardware (AES, RSA and SHA) engines.

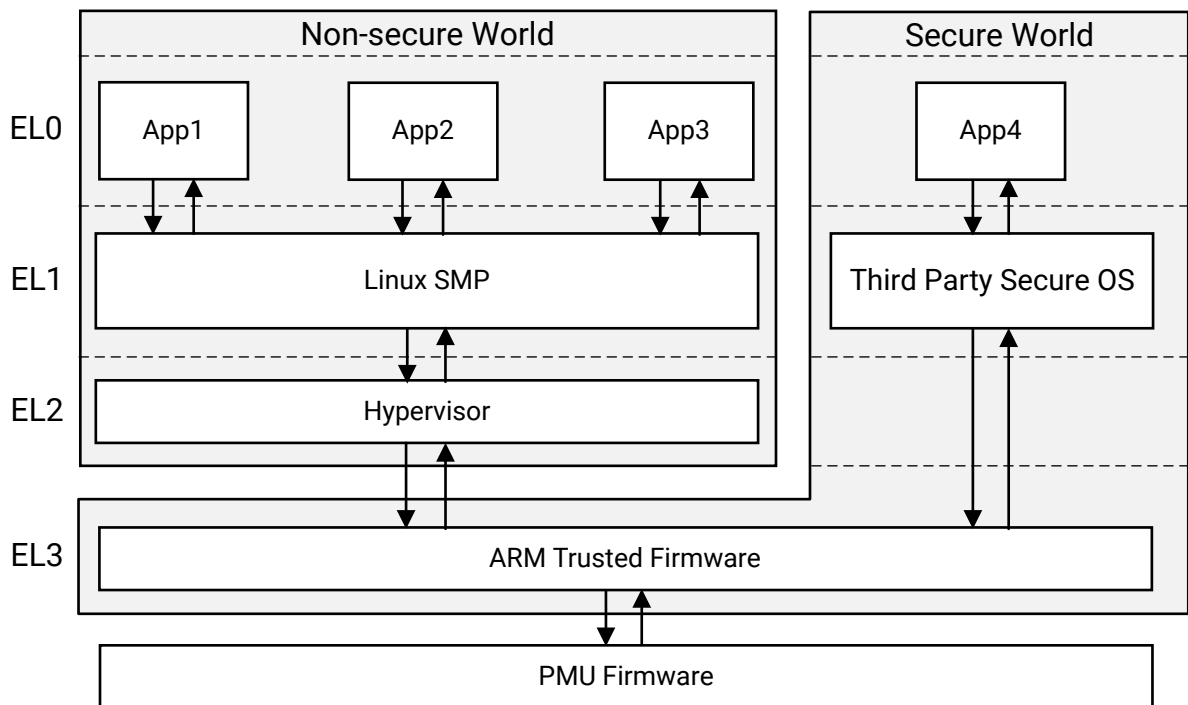
- XilSKey: Xilinx secure key library.
- lwIP Library: An open source TCP/IP protocol suite that provides access to the core lwIP stack and BSD (Berkeley Software Distribution) sockets style interface to the stack.

These libraries are documented in Appendix B, Xilinx Standard C Libraries.

Linux Software Stack

The Linux OS supports the Zynq UltraScale+ MPSoC. With the sole exception of the Arm GPU, Xilinx provides open source drivers for all peripherals in the PS as well as key peripherals in the PL. The following figure illustrates the full software stack in APU, including Linux and an optional hypervisor.

Figure 6: Linux Software Development Stack



X18968-071217

The Armv8 exception model defines exception levels EL0–EL3, where:

- EL0 has the lowest software execution privilege. Execution at EL0 is called unprivileged execution.
- Increased exception levels, from 1 to 3, indicate an increased software execution privilege.

- EL2 provides support for processor virtualization. You may optionally include an open source or commercial hypervisor in the software stack.
- EL3 provides support for a secure state. The Cortex-A53 MPCore processor implements all the exception levels (EL0-EL3) and supports both execution states (AArch64 and AArch32) at each exception level.

You can leverage the Linux software stack for the Zynq UltraScale+ MPSoC in multiple ways. The following are some of your options:

- **PetaLinux Tools:** The PetaLinux tools include a branch of the Linux source tree, U-Boot as well as Yocto-based tools to make it easy to build complete Linux images including the kernel, the root file system, device tree, and applications for Xilinx devices. See the [PetaLinux Product Page](#) for more information. The PetaLinux tools work with the same open source Linux components described immediately below.
- **Open Source Linux and U-Boot:** The Linux Kernel sources including drivers, board configurations, and U-Boot updates for the Zynq UltraScale+ MPSoC are available from the [Xilinx Github link](#), and on a continuing basis from the main Linux kernel and U-Boot trees as well. Yocto board support packages are also available from the main Yocto tree.
- **Commercial Linux Distributions:** Some commercial distributions also include support for Xilinx UltraScale+ MPSoC devices and they include advanced tools for Linux configuration, optimization, and debug. You can find more information about these from the [Xilinx Embedded Computing page](#).

Multimedia Stack Overview

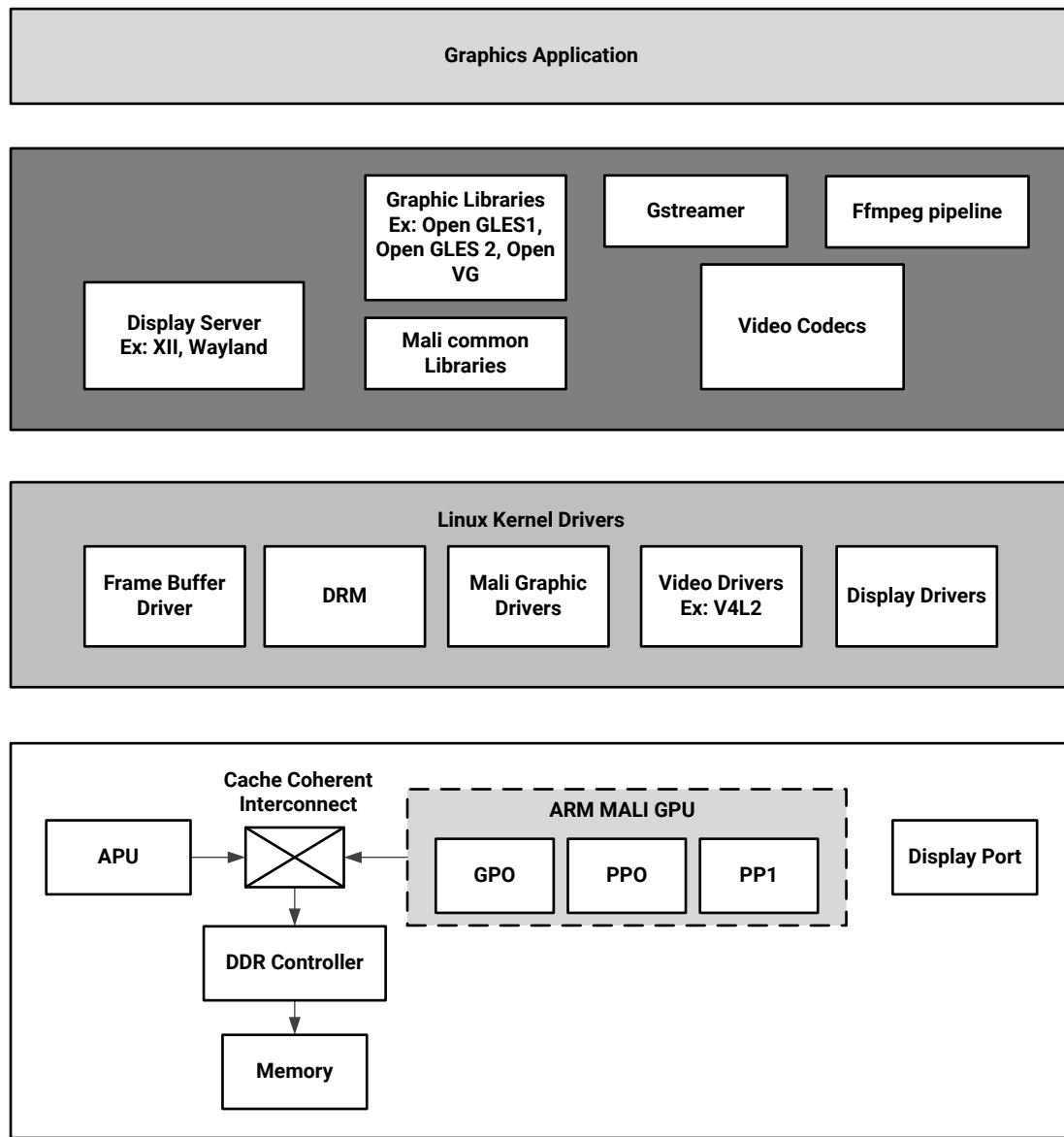
This section describes the multimedia software stack in the Zynq UltraScale+ MPSoC.

The GPU and a high performance DisplayPort accelerate the graphics application. The GPU provides hardware acceleration for 2D and 3D graphics by including one geometry processor (GP) and two pixel processors (PP0 and PP1), each having a dedicated memory management unit (MMU). The cache coherency between the APU and the GPU is achieved by cache-coherent interconnect (CCI), which supports the AXI coherency extension (ACE) only.

CCI in-turn connects the APU and the GPU to the DDR controller, which arbitrates the DDR access.

The following figure shows the multimedia stack.

Figure 7: Multimedia Stack



X14795-071317

The Linux kernel drivers for multimedia enables the hardware access by the applications running on the processors.

The following table lists the multimedia drivers through the middleware stack that consists of the libraries and framework components the applications use.

Table 11: Libraries and Framework Components

Component	Description
Display server	Coordinates the input and output from the applications to the operating system.
Graphics library	The Zynq UltraScale+ MPSoC architecture supports OpenGL ES 1.1 and 2.2, and Open VG 1.1.
Mali™-400 MP2 common libraries	Mali-400 MP2 graphic libraries. For more details on how to switch between different EGL backends, refer to Xilinx MALI Driver .
Gstreamer	A freeware multimedia framework that allows a programmer to create a variety of media handling components.
Video codecs	Video encoders and decoders.

The following table lists the Linux kernel graphics drivers.

Table 12: Linux Kernel Drivers

Drivers	Description
Frame buffer driver	Kernel graphics driver exposing its interface through /dev/fb*. This interface implements limited functionality (allowing you to set a video mode and drawing to a linear frame buffer).
Direct rendering manager (DRM)	Serves in rendering the hardware between multiple user space components.
Mali-400 MP2 graphics drivers	Provides the hardware access to the GPU hardware.
Video drivers	Video capture and output device pipeline drivers based on the V4L2 framework. The Xilinx Linux V4L2 pipeline driver represents the whole pipeline with multiple sub-devices. You can configure the pipeline through the media node, and you can perform control operations, such as stream on/off, through the video node. Device nodes are created by the pipeline driver. The pipeline driver also includes the wrapper layer of the DMA engine API, and this enables it to read/write frames from RAM.
Display port drivers	Enables the hardware access to the display port, based on DRM framework.

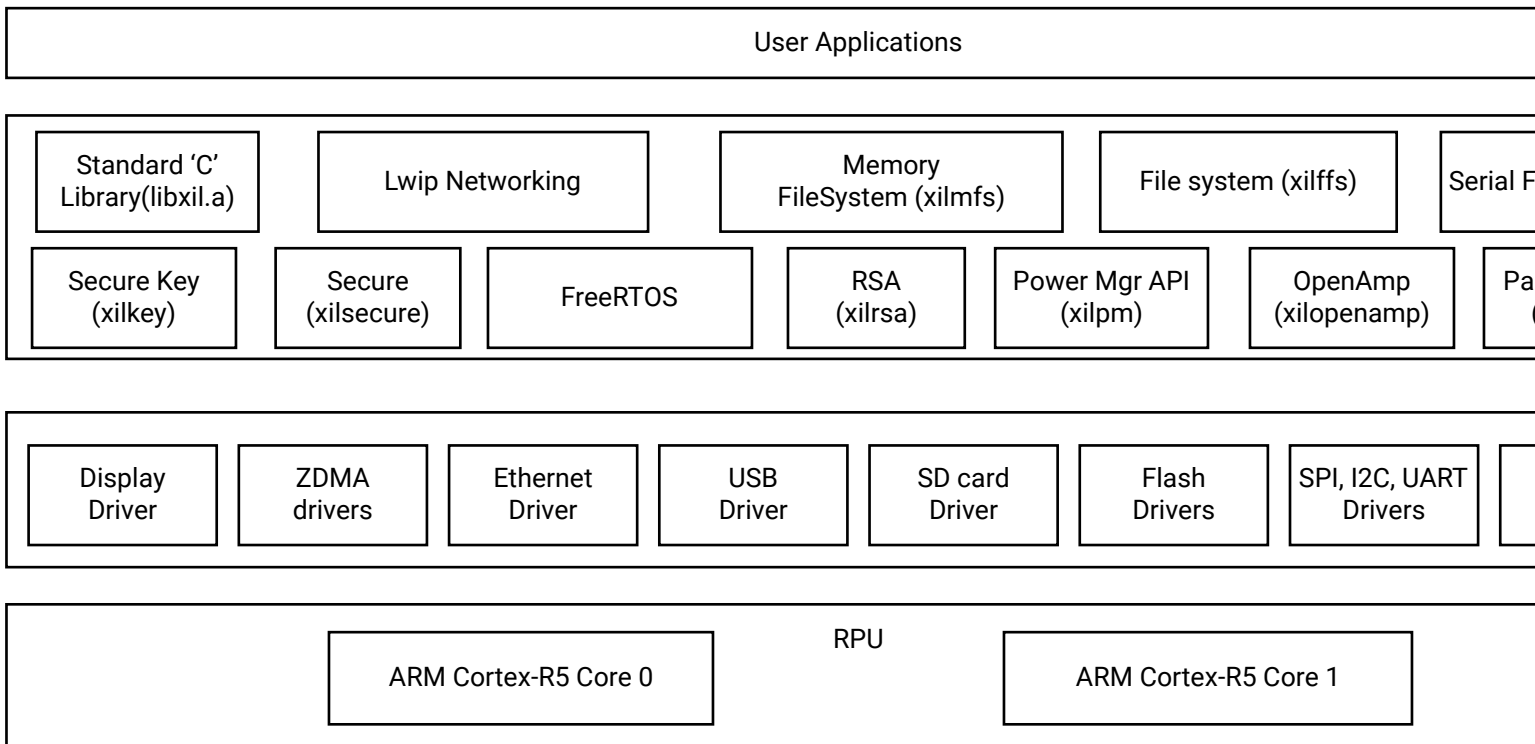
FreeRTOS Software Stack

Xilinx provides a FreeRTOS board support package (BSP) as a part of the Vitis™ software platform. The FreeRTOS BSP provides you a simple, multi-threading environment with basic features such as, standard input/output and access to processor hardware features. The BSP and the included libraries are highly configurable to provide you the necessary functionality with the least overhead. The FreeRTOS software stack is similar to the bare metal software stack, except that it contains the FreeRTOS library. Xilinx device drivers included with the standalone libraries

can typically be used within FreeRTOS provided that only a single thread requires access to the device. Xilinx bare metal drivers are not aware of Operating Systems. They do not provide any support for mutexes to protect critical sections, nor do they provide any mechanism for semaphores to be used for synchronization. While using the driver API with FreeRTOS kernel, you must take care of this aspect.

The following figure illustrates the FreeRTOS software stack for RPU.

Figure 8: FreeRTOS Software Stack



Note: The FreeRTOS software stack for APU is same as that for RPU except that the libraries support both 32-bit and 64-bit for APU.

Third-Party Software Stack

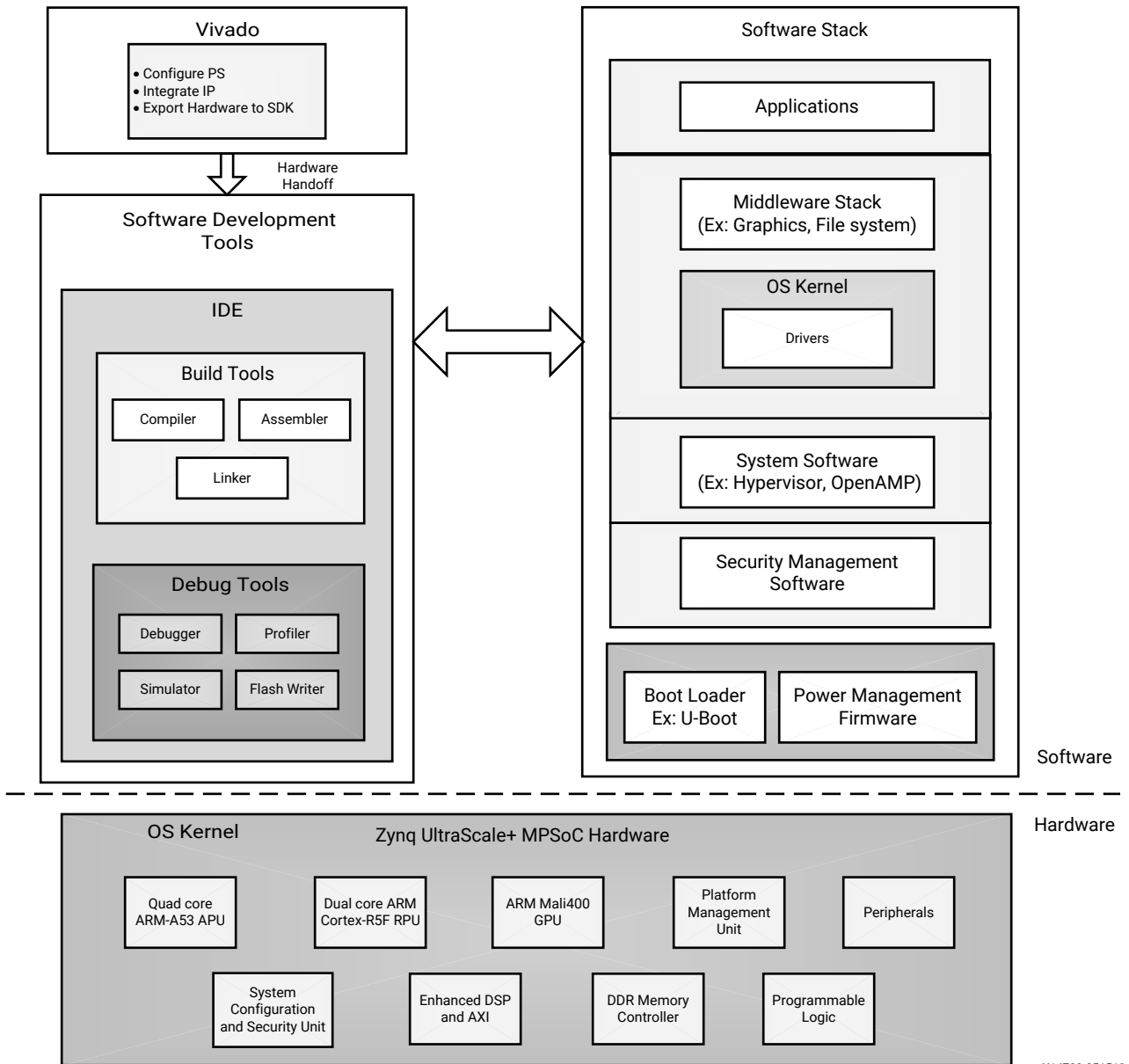
Many other embedded software solutions are also available from the Xilinx partner ecosystem. More information is available from the Xilinx website, [Embedded Computing](#) and the website, [Xilinx Third Party Tools](#).

Software Development Flow

This chapter explains the bare metal software development for RPU and APU using the Vitis™ unified software platform as well as Linux software development for APU using PetaLinux tools and the Vitis software platform.

The following figure depicts the top-level software architecture of the Zynq® UltraScale+™ MPSoC.

Figure 9: Software Development Architecture



Bare Metal Application Development

This section assists you in understanding the design flow of bare metal application development for APU and RPU using the Vitis software platform. The following figure shows the top-level design flow in the Vitis software platform. You can create a C or C++ standalone application project by using the New Application Project wizard.

To create a project, follow these steps:

1. Click **File** → **New** → **Application Project**. The New Application Project dialog box appears.
Note: This is equivalent to clicking on **File** → **New** → **Project** to open the New Project wizard, selecting **Xilinx** → **Application Project**, and clicking **Next**.
2. Type a project name into the Project Name field.
3. Select the location for the project. You can use the default location as displayed in the Location field by leaving the Use default location check box selected. Otherwise, click the check box and type or browse to the directory location.
4. Select **Create a new platform from hardware (XSA)**. The Vitis IDE lists the all the available pre-defined hardware designs.
5. Select any one hardware design from the list and click **Next**.
6. From the CPU drop-down list, select the processor for which you want to build the application. This is an important step when there are multiple processors in your design. In this case you can either select **psu_cortexa53_0** or **psu_cortexr5_0**.
7. Select your preferred language: C or C++.
8. Select an OS for the targeted application.
9. Click **Next** to advance to the Templates screen.

The Vitis software platform provides useful sample applications listed in Templates dialog box that you can use to create your project. The Description box displays a brief description of the selected sample application. When you use a sample application for your project, the Vitis software platform creates the required source and header files and linker script.

10. Select the desired template. If you want to create a blank project, select **Empty Application**. You can then add C files to the project, after the project is created.
11. Click **Finish** to create your application project and board support package (if it does not exist).

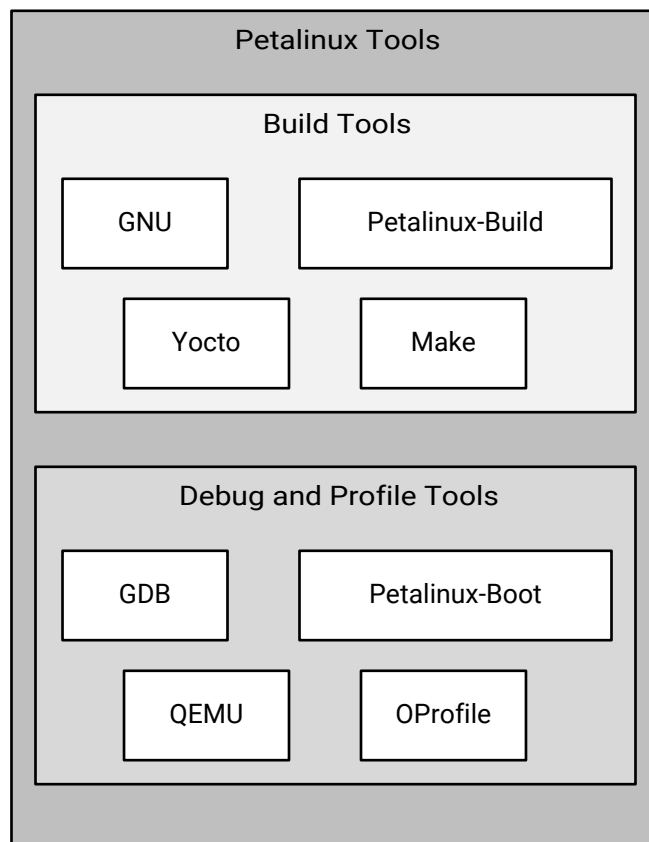
Note:

1. Xilinx recommends that you use the Managed Make flow rather than Standard Make C/C++ unless you are comfortable working with make files. For more details on QEMU, see the *Xilinx Quick Emulator User Guide: QEMU* ([UG1169](#)).
2. Cortex™-R5F and Cortex™-A53 32-bit bare metal software do not support 64-bit addressed data transfer using device DMA.
3. By default, all standalone applications will run only on APU0. The other APU cores will be off.

Application Development Using PetaLinux Tools

Software development flow in the PetaLinux tools environment involves many stages. To simplify understanding, the following figure shows a chart with all the stages in the PetaLinux tools application development.

Figure 10: PetaLinux Tool-Based Software Development Flow



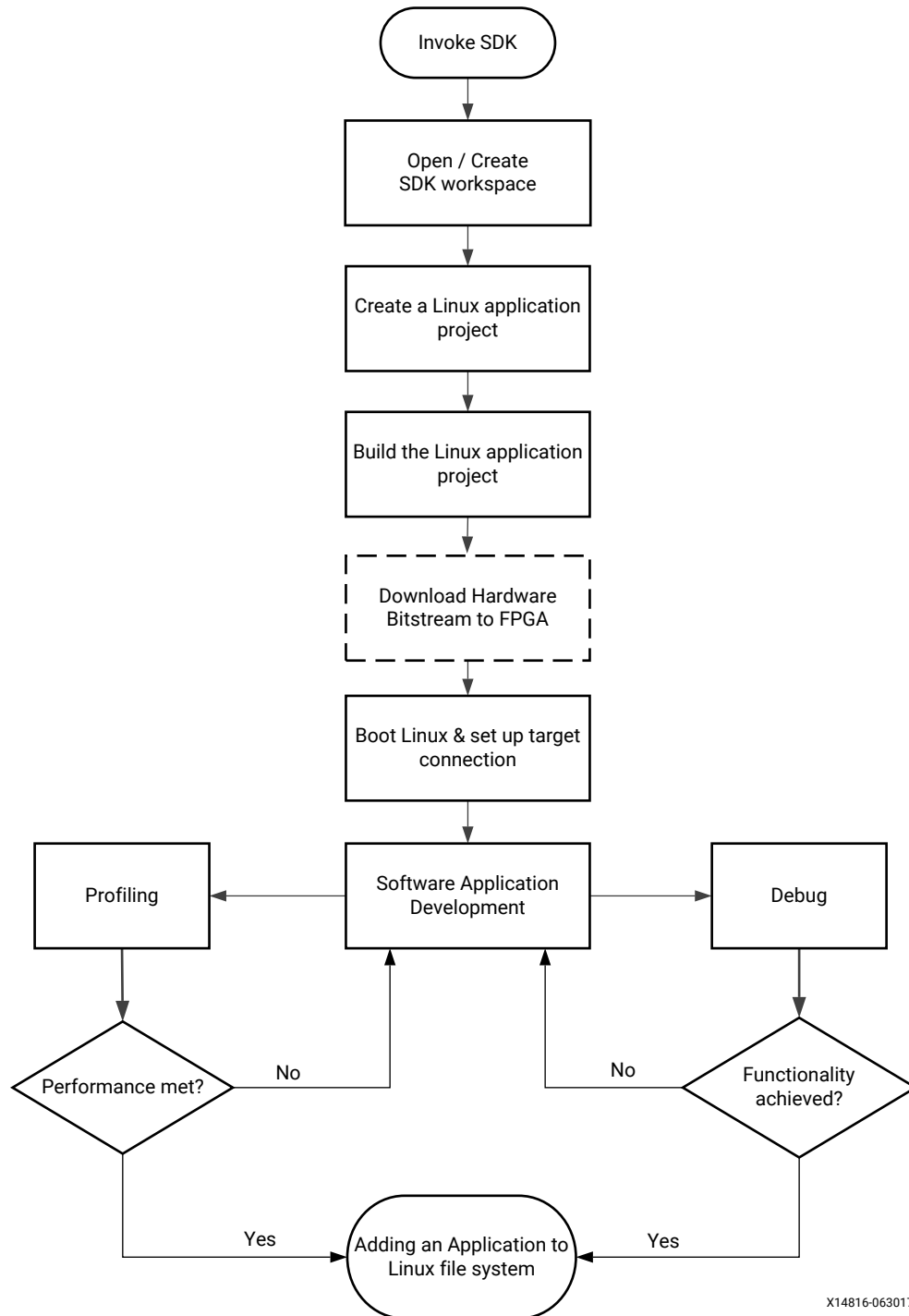
X14815-063017

Linux Application Development Using Vitis

Xilinx software design tools facilitate the development of Linux user applications. This section provides an overview of the development flow for Linux application development.

The following figure illustrates the typical steps involved to develop Linux user applications using the Vitis software platform.

Figure 11: Linux Application Development Flow




X14816-063017

Creating a Linux Application Project

You can create a C or C++ Linux application project by using the New Application Project wizard.

To create a project:

1. Click **File** → **New** → **Application Project**. The New Application Project dialog box appears.
2. Type a project name into the **Project Name** field.
3. Select the location for the project. You can use the default location as displayed in the Location field by leaving the **Use default location** check box selected. Otherwise, click the check box and type or browse to the directory location.
4. Select **Next**.
5. On the Select platform tab, select the Platform that has a Linux domain and click **Next**.
Note: If your platform does not yet have a Linux domain, refer to .
6. On the Domain window, select the domain from the Domain drop-down.
7. Select your preferred language: **C** or **C++**.
8. Optionally, select **Linux System Root** to specify the Linux sysroot path and select **Linux Toolchain** to specify the Linux toolchain path.
9. Click **Next** to move to the **Templates** screen.
10. The Vitis software platform provides useful sample applications listed in the Templates dialog box that you can use to create your project. The Description box displays a brief description of the selected sample application. When you use a sample application for your project, the Vitis software platform creates the required source and header files and linker script.
11. Select the desired template. If you want to create a blank project, select **Empty Application**. You can then add C files to the project, after the project is created.
12. Click **Finish** to create your Linux application project.
13. Click the  icon to generate or build the application project.

Create a Hello World Application


After installing the Vitis™ software platform, the next step is to create a software application project. Software application projects are the final application containers. The project directory that is created contains (or links to) your C/C++ source files, executable output file, and associated utility files, such as the Makefiles used to build the project.

Note: The Vitis software platform automatically creates a system project for you. A system project is a top-level container project that holds all of the applications that can run in a system at the same time. This is useful if you have many processors in your system, especially if they communicate with one another, because you can debug, launch, and profile applications as a set instead of as individual items.

Build a Sample Application

This section describes how to create a sample Hello World application using an existing template.

1. Launch the Vitis software platform.

2. Select a workspace directory for your first project.
3. Click **Launch**. The welcome page appears.
4. Close the welcome page. The development perspective opens.
5. Select **File → New → Application Project**.
6. Enter a name in the Project name field and click **Next**. The Select platform tab opens. You should choose a platform for your project. You can either use a pre-supplied platform (from Xilinx or another vendor), a previously created custom platform, or you can create one automatically from an exported Vivado® hardware project.
7. On the **Select platform** tab, click the platform you just created and click **Next**. To use your own hardware platform, click the  icon and add your platform to the list.
8. Select the system configuration for your project and click **Next**. The Templates window opens.
9. Select **Hello World** and click **Next**. Your workspace opens with the Explorer pane showing the `hello_world_system` system project and the `zcu102` platform project.
10. Right-click the system project and select **Build Project**. You have now built your application and the Console tab shows the details of the file and application size.

Debug and Run the Application

Now that you have generated the executable binary, you can test it on a board. To run the application on the board, perform the following preliminary steps:

- Connect a JTAG cable to the computer.
 - Set the Boot Mode switch of the board to JTAG mode.
 - Connect a USB UART cable and setup your UART console.
 - Power up the board.
1. Expand the system project and choose the application project you want to debug. Right-click the application and select **Debug As → Launch on Hardware (Single Application Debug)**.
 2. On the Confirm Perspective Switch dialog, click **Yes**. The Vitis IDE switches to the Debug perspective and the debugger stops at the entry to your `main()` function.
 3. Using the commands in the toolbar, step through the application. After you step through the `print()` function, `Hello World` appears in the UART console.

Adding Driver Support for Custom IP in the PL

The Vitis software platform supports Linux BSP generation for peripherals in the PS as well as custom IP in the PL. When generating a Linux BSP, the Vitis software platform produces a device tree, which is a data structure describing the hardware system that passes to the kernel when you boot.

Device drivers are available as part of the kernel or as separate modules, and the device tree defines the set of hardware functions available and features enabled.

Additionally, you can add dynamic, loadable drivers. The Linux kernel supports these drivers. Custom IP in the PL are highly configurable, and the device tree parameters define both the set of IP available in the system and the hardware features enabled in each IP.

See [Chapter 3: Development Tools](#) for additional overview information on the Linux Kernel and boot sequence.

Software Design Paradigms

The Xilinx[®] Zynq[®] UltraScale+[™] MPSoC architecture supports heterogeneous multiprocessor engines targeted at different tasks. The main approaches for developing software to target these processors are by using the following:

- **Frameworks for Multiprocessor Development:** Describes the frameworks available for development on the Zynq UltraScale+ MPSoC.
- **Symmetric Multiprocessing (SMP):** Using SMP with PetaLinux is the most simple flow for developing an SMP with a Linux platform for the Zynq UltraScale+ MPSoC.
- **Asymmetric Multiprocessing (AMP):** AMP is a powerful mode to use multiple processor engines with precise control over what runs on each processor. Unlike SMP, there are many different ways to use AMP. This section describes two ways of using AMP with varying levels of complexity.

The following sections describe these development methods in more detail.

Frameworks for Multiprocessor Development

Xilinx provides multiple frameworks for Zynq UltraScale+ MPSoCs to facilitate the application development on the heterogeneous processors and Xilinx 7 series FPGAs. The following bullets explain these frameworks:

- **Hypervisor Framework:** Xilinx provides the Xen hypervisor, a critical item needed to support virtualization on APU of Zynq UltraScale+ MPSoC. The [Use of Hypervisors](#) section covers more details.
- **Authentication Framework:** The Zynq UltraScale+ MPSoC supports authentication and encryption features as a part of authentication framework. To understand more about the authentication framework, see [Boot Time Security](#).
- **TrustZone Framework:** The TrustZone technology allows and maintains isolation between secure and non-secure processes within the same system.

Xilinx provides the trustzone support through the Arm[®] Trusted Firmware (ATF) to maintain the isolation between secure and non-secure worlds. To understand more about ATF, see [Arm Trusted Firmware](#).

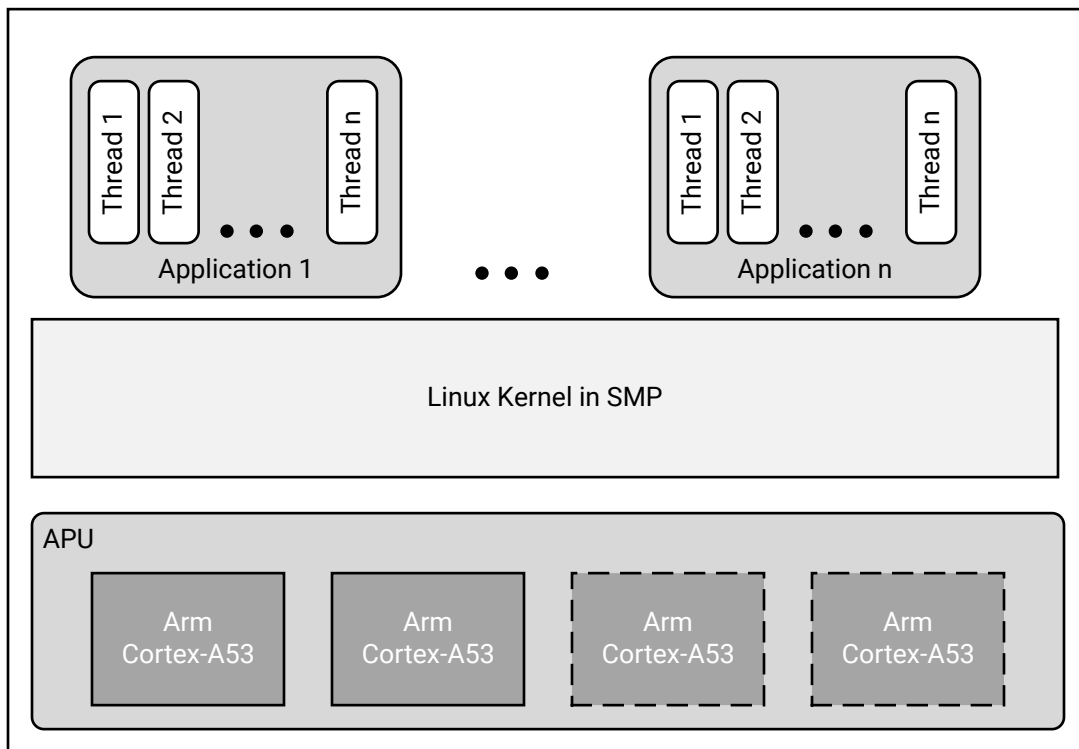
- **Multiprocessor Communication Framework:** Xilinx provides the OpenAMP framework for Zynq UltraScale+ MPSoCs to allow communication between the different processing units. For more details, see the *Xilinx Quick Emulator User Guide: QEMU* ([UG1169](#))
- **Power Management Framework:** The power management framework allows software components running across different processing units to communicate with the power management unit.

Symmetric Multiprocessing (SMP)

SMP enables the use of multiple processors via a single operating system instance. The operating system handles most of the complexity of managing multiple processors, caches, peripheral interrupts, and load balancing.

The APU in the Zynq UltraScale+ MPSoCs contains four homogeneous cache coherent Arm Cortex-A53 processors that support SMP mode of operation using an OS (Linux or VxWorks). Xilinx and its partners provide operating systems that make it easy to leverage SMP in the APU. The following diagram shows an example of Linux SMP with multiple applications running on a single OS.

Figure 12: Example SMP Using Linux



X14837-063017

This would not be the best mode of operation when there are hard, real-time requirements as it ignores Linux application core affinity which should be available to developers with the existing Xilinx software.

Asymmetric Multiprocessing (AMP)

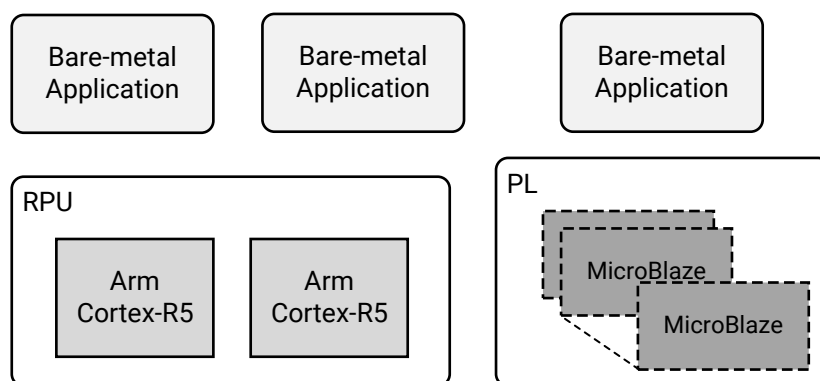
AMP uses multiple processors with precise control over what runs on each processor. Unlike SMP, there are many different ways to use AMP. This section describes two ways of using AMP with varying levels of complexity.

In AMP, a software developer must decide what code has to run on each processor before compiling and creating a boot image that includes the software executable for each CPU. Using AMP with the Arm Cortex-R5F processors (SMP is not supported in Cortex-R5F) in the RPU enables developers to meet highly demanding, hard real-time requirements as opposed to soft real-time requirements.

You can develop the applications independently, and program those applications to communicate with each other using inter-processing communication (IPC) options. See this [link](#) to the “Interrupts” chapter of the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)* for further description of this feature.

You can also apply this AMP method to applications running on MicroBlaze processors in the PL or even in the APU. The following diagram shows an AMP example with applications running on the RPU and the PL without any communication with each other.

Figure 13: AMP Example using Bare-Metal Applications Running on RPU and PL



X19225-071317

OpenAMP

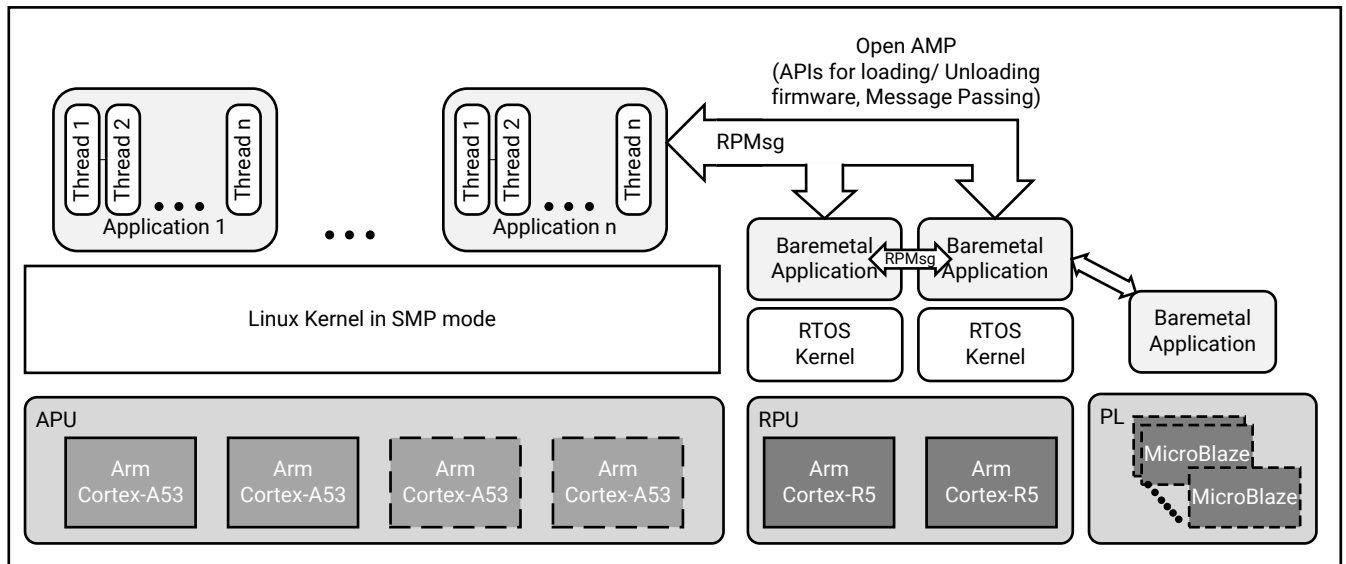
The OpenAMP framework provides mechanisms to do the following:

- Load and unload firmware
- Communicate between applications using a standard API

The following diagram shows an example of an OpenAMP and the hard real-time capabilities of the RPU using the OpenAMP framework.

In this case, Linux applications running on the APU perform the loading and unloading of RPU applications. This allows developers to load different processing dedicated algorithms to the RPU processing engines as needed with very deterministic performance.

Figure 14: Example with SMP and AMP using OpenAMP Framework



X14839-063017

See the *Libmetal and OpenAMP for Zynq Devices User Guide (UG1186)* for more information about the OpenAMP Framework.

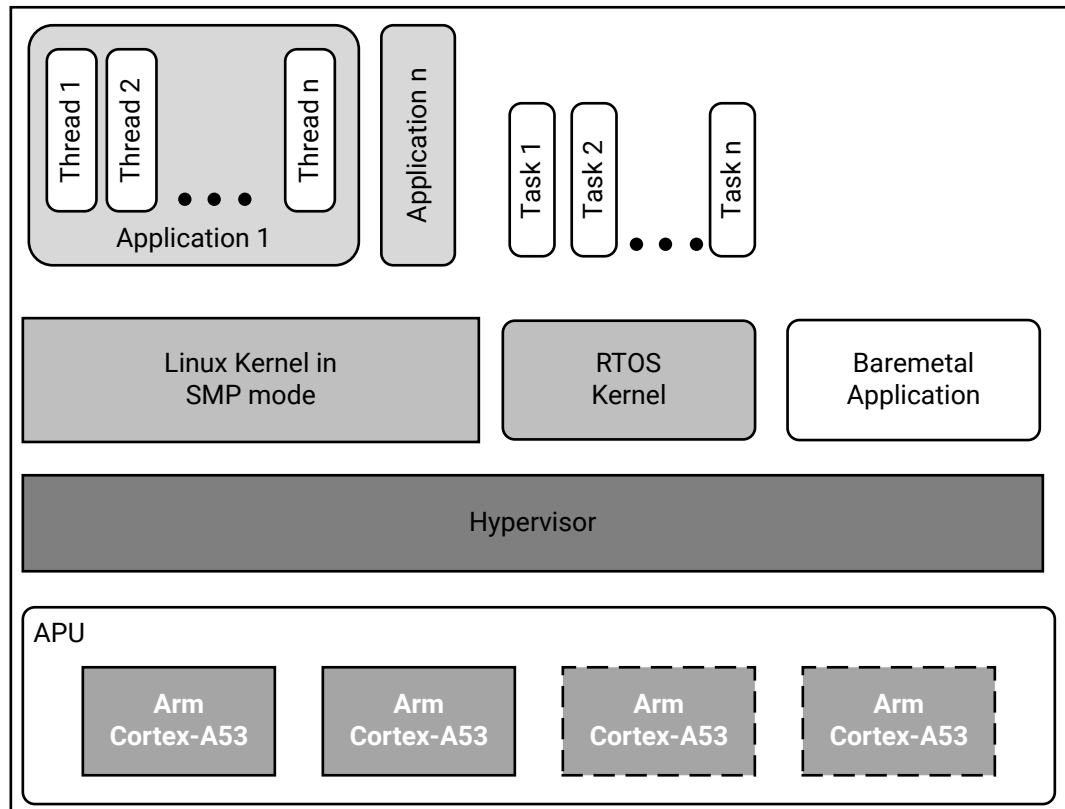
Virtualization with Hypervisor

The Zynq UltraScale+ MPSoCs include a hardware virtualization extension on the Arm Cortex-A53 processors, interrupt controller, and Arm System MMU (SMMU) that provides flexibility to combine various operating system combinations, including SMP and AMP, within the APU.

The following diagram shows an example of an SMP-capable OS, like Linux working along with Real-Time Operating System (RTOS) as well as a bare metal application using a single hypervisor.

This enables independent development of applications in their respective mode of operation.

Figure 15: Example with Hypervisor



X14840-063017

Although the hardware virtualization included within Zynq UltraScale+ MPSoC and its hypervisors allow the standard operating systems and their applications to function with low to moderate effort, the addition of a hypervisor does bring design complexity to low-level system functions such as power management, FPGA bitstream management,

OpenAMP software stack, and security accelerator access which must use additional underlying layers of system firmware. Hence, Xilinx recommends that the developers must initiate early effort into these aspects of system architecture and implementation.

For more details on using Hypervisor like the Xen Hypervisor, see the [MPSoC Xen Hypervisor website](#).

Use of Hypervisors

Xilinx distributes a port for the Xen open source hypervisor in the Xilinx Zynq UltraScale+ MPSoC. Xen hypervisor provides the ability to run multiple operating systems on the same computing platform. Xen hypervisor, which runs directly on the hardware, is responsible for managing CPU, memory, and interrupts. Multiple numbers of OS can run on top of the hypervisor. These operating systems are called domains (also called as virtual machines (VMs)).

The Xen hypervisor provides the ability to concurrently run multiple operating systems and their standard applications with relative ease. However, Xen does not provide a generic interface which gives the guest an operating system access to system functions. Hence, you need to follow the cautions mentioned in this section.

Xen hypervisor controls one domain, which is domain 0, and one or more guest domains. The control domain has special privileges, such as the following:

- Capability to access the hardware directly
- Ability to handle access to the I/O functions of the system
- Interaction with other virtual machines.

It also exposes a control interface to the outside world, through which the system is controlled. Each guest domain runs its own OS and application. Guest domains are completely isolated from the hardware.

Running multiple Operating Systems using Xen hypervisor involves setting up the host OS and adding one or more guest OS.

Note: Xen hypervisor is available as a selectable component within the PetaLinux tools; Xen hypervisor can also be downloaded from Xilinx GIT. With Linux and Xen software that is made available by Xilinx, it is possible to build custom Linux guest configurations. Guest OS other than Linux require additional software and effort from third-parties. See the [PetaLinux Product Page](#) for more information.

System Boot and Configuration

Zynq[®] UltraScale+[™] MPSoCs support the ability to boot from different devices such as a QSPI flash, an SD card, a host with Device Firmware Upgrade utility installed on it, or a NAND flash in place. This chapter details the booting process using different booting devices in both secure and non-secure modes.

Boot Process Overview

The platform management unit (PMU) and configuration security unit (CSU) manage and perform the multi-staged booting. You can boot the device in either secure (using authenticated boot image) or non-secure (using an unauthenticated boot image) mode. The boot stages are as follows:

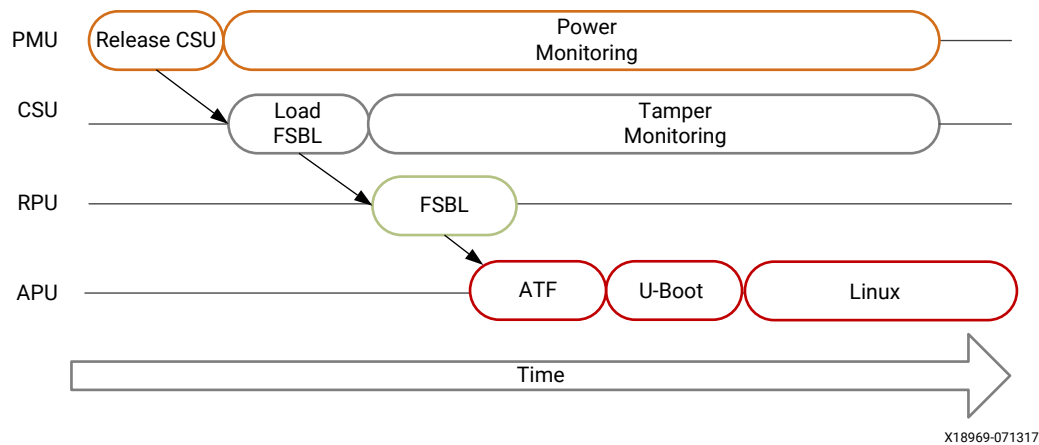
- **Pre-configuration stage:** The PMU primarily controls pre-configuration stage that executes PMU ROM to setup the system. The PMU handles all of the processes related to reset and wake-up.
- **Configuration stage:** This stage is responsible for loading the first-stage boot loader (FSBL) code for the PS into the on-chip RAM (OCM). It supports both secure and non-secure boot modes. Through the boot header, you can execute FSBL on the Cortex[™]-R5F-0 / R5-1 processor or the Cortex[™]-A53 processor. The Cortex-R5F-0 processor also supports lock step mode.
- **Post-configuration stage:** After FSBL execution starts, the Zynq UltraScale+ MPSoC enters the post configuration stage.

Boot Flow

There are two boot flows in the Zynq UltraScale+ MPSoC architecture: secure and non-secure. The following sections describe some of the example boot sequences in which you bring up various processors and execute the required boot tasks.

Note: The figures in these sections show the complete boot flow, including all mandatory and optional components.

Figure 16: Boot Flow Example



Non-Secure Boot Flow

In non-secure boot mode, the PMU releases the reset of the configuration security unit (CSU), and enters the PMU server mode where it monitors power. After the PMU releases the CSU from reset, it loads the FSBL into OCM. PMU firmware runs from PMU RAM in parallel to FSBL in OCM. FSBL is run on APU or RPU. FSBL runs from APU/RPU and ATF; U-Boot and Linux run on APU. Other boot configurations allow the RPU to start and operate wholly independent of the APU and vice-versa.

- On APU, ATF will be executed after the FSBL hands off to ATF. ATF hands off to a second stage boot loader like U-Boot which executes and loads an operating system such as Linux.
- On RPU, FSBL hands off to a software application.
- Linux, in turn, loads the executable software.

Note: The operating system manages the multiple Cortex-A53 processors in symmetric multi-processing (SMP) mode.

Secure Boot Flow

In the secure boot mode, the PMU releases the reset of the configuration security unit (CSU) and enters the PMU server mode where it monitors power. After the PMU releases the CSU from reset, the CSU checks to determine if authentication is required by the FSBL or the user application.

The CSU does the following:

- Performs an authentication check and proceeds only if the authentication check passes. Then checks the image for any encrypted partitions.
- If the CSU detects partitions that are encrypted, the CSU performs decryption and loads the FSBL into the OCM.

For more information on CSU, see the [Configuration Security Unit](#) section.

FSBL running on APU hands off to ATF. FSBL running on RPU loads ATF. In both the cases, ATF loads U-Boot which loads the OS. ATF then executes the U-Boot and loads an OS such as Linux. Then Linux, in turn, loads the executable software. Similarly, FSBL checks for authentication and encryption of each partition it tries to load. The partitions are only loaded by FSBL on successful authentication and decryption (if previously encrypted).

Note: In the secure boot mode, `psu_coresight_0` is not supported as a `stdout` port.

Boot Image Creation

Bootgen is a tool that lets you stitch binary files together and generate device boot images. Bootgen defines multiple properties, attributes and parameters that are input while creating boot images for use in a device.

The secure boot feature for devices uses public and private key cryptographic algorithms. Bootgen provides assignment of specific destination memory addresses and alignment requirements for each partition. It also supports encryption and authentication, described in the *Bootgen User Guide* (UG1283). More advanced authentication flows and key management options are discussed in the Using HSM Mode section of *Bootgen User Guide* (UG1283), where Bootgen can output intermediate hash files that can be signed offline using private keys to sign the authentication certificates included in the boot image. The program assembles a boot image by adding header blocks to a list of partitions.

Optionally, each partition can be encrypted and authenticated with Bootgen. The output is a single file that can be directly programmed into the boot flash memory of the system.

Various input files can be generated by the tool to support authentication and encryption as well.

Bootgen comes with both a GUI interface and a command line option. The tool is integrated into the software development toolkit, Integrated Development Environment (IDE), for generating basic boot images using a GUI, but the majority of Bootgen options are command line-driven. Command line options can be scripted. The Bootgen tool is driven by a boot image format (BIF) configuration file, with a file extension of `*.bif`. Along with SoC, Bootgen has the ability to encrypt and authenticate partitions for and later FPGAs, as described in FPGA Support. Along with SoC and ACAP devices, Bootgen has the ability to encrypt and authenticate partitions for and later FPGAs, as described in FPGA Support. In addition to the supported command and attributes that define the behavior of a Boot Image, there are utilities that help you work with Bootgen. Bootgen code is now available on Github.

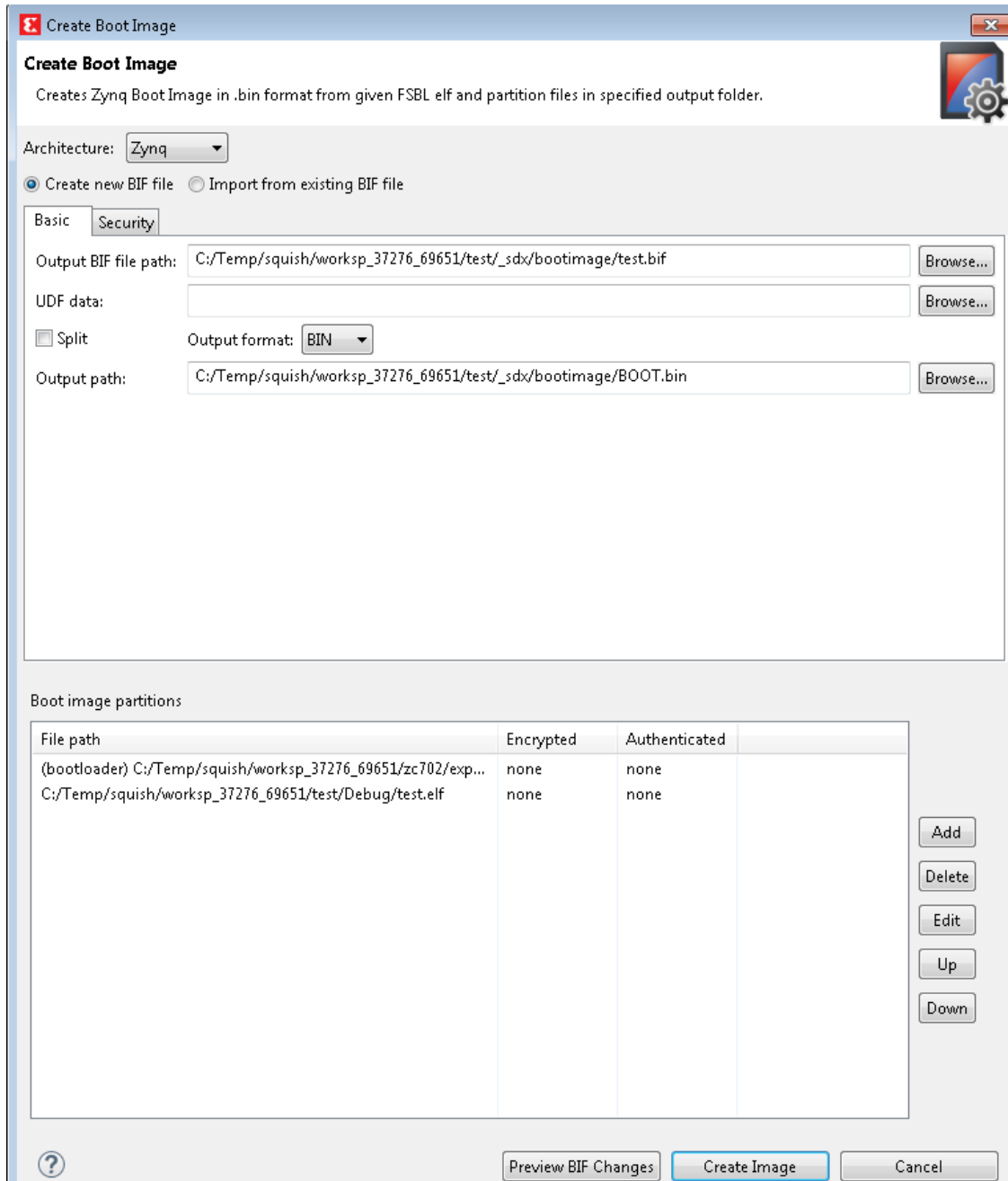
Creating a Bootable Image

When a system project is selected, by running build, the Vitis software platform builds all applications in the system project and creates a bootable image according to a pre-defined BIF or an auto-generated BIF.

You can create bootable images using Bootgen. In the Vitis IDE, the Create Boot Image menu option is used to create the boot image.

To create a bootable image, follow these steps:

1. Select the Application Project in the Project Explorer view.
2. Right-click the application and select **Create Boot Image** to open the Create Boot Image dialog box.
3. Specify the boot loader and the partitions.



4. Click **Create Image** to create the image and generate the `BOOT.bin` in the `<Application_project_name>/_ide/bootimage` folder.

Boot Modes

See Table 7-4 for a brief list of available boot modes. Refer to this link to the “Boot and Configuration” chapter of the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)* for a comprehensive table of the available boot modes.

QSPI24 and QSPI32 Boot Modes

The QSPI24 and QSPI32 boot modes support the following:

- x1, x2, and x4 read modes for single Quad SPI flash memory 24 (QSPI24) and single Quad SPI flash memory 32 (QSPI32)
- x8 read mode for dual QSPI.
- Image search for MultiBoot
- I/O mode for BSP drivers (no support in FSBL)

The bootROM searches the first 256 Mb in x8 mode. In QSPI24 and QSPI32 boot modes (where the QSPI24/32 device is < 128 Mb), to use MultiBoot, place the multiple images so that they fit in memory locations less than 128 Mb. The pin configuration for QSPI24 boot mode is 0x1.

Note: QSPI dual stacked (x8) boot is not supported. Only QSPI Single Transmission Rate (STR) is supported. Single Quad-SPI memory (x1, x2, and x4) is the only boot mode that supports execute-in-place (XIP).

To create a QSPI24/QSPI32 boot image, provide the following files to the Bootgen tool:

- An FSBL ELF
- A secondary boot loader (SBL), such as U-Boot, or a Cortex-R5F-0/R5-1 and/or a Cortex-A53 application ELF
- Authentication and encryption key (optional)

For more information on Authentication and Encryption, see [Chapter 8: Security Features](#).

Bootgen generates the `boot.mcs` and a `boot.bin` binary file that you can write into the QSPI flash using the flash writer. MCS is an Intel hex-formatted file that includes a checksum for reliability.

Note: The pin configuration for QSPI24 boot mode is 0x1 for qspi 24 and 0x2 for qspi32.

SD Boot Mode

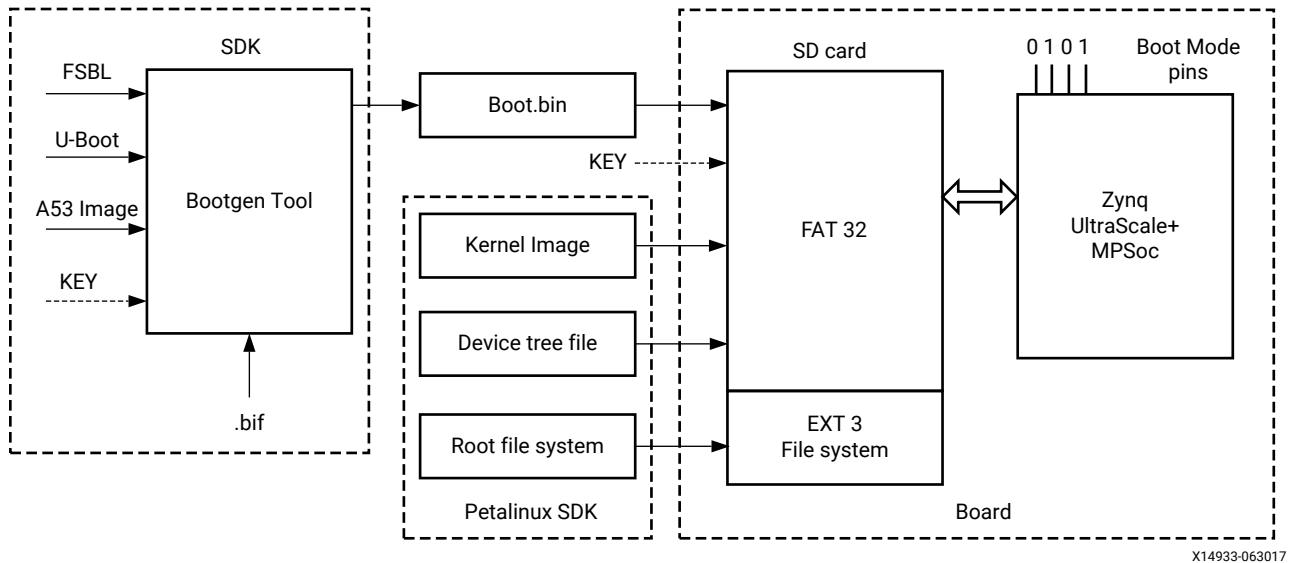
SD boot (version 3.0) supports the following:

- FAT 16/32 file systems for reading the boot images.

- Image search for MultiBoot with a maximum number 8,192 files are supported.

The following figure shows an example for booting Linux in SD mode.

Figure 17: Booting in SD Mode



To create an SD boot image, provide the following files to Bootgen:

- An FSBL ELF
- A Cortex-R5F-0/R5-1 and/or an Cortex-A53 application ELF
- Optional authentication and encryption keys

The Bootgen tool generates the `boot.bin` binary file. You can write the `boot.bin` file into an SD card using a SD card reader.

In PetaLinux, do the following:

1. Build the Linux kernel image, device tree file, and the root file system.
2. Copy the files into the SD card.

The formatted SD card then contains the `boot.bin`, the kernel image, and the device tree file in the FAT32 partition; the root file system resides in the EXT 3 partition.



IMPORTANT! To boot from SD1, configure the boot pins to `0x5`. To boot from SD0, configure the boot pins to `0x3`. To boot from SD with a level shifter, configure the boot pins to `0xE`.

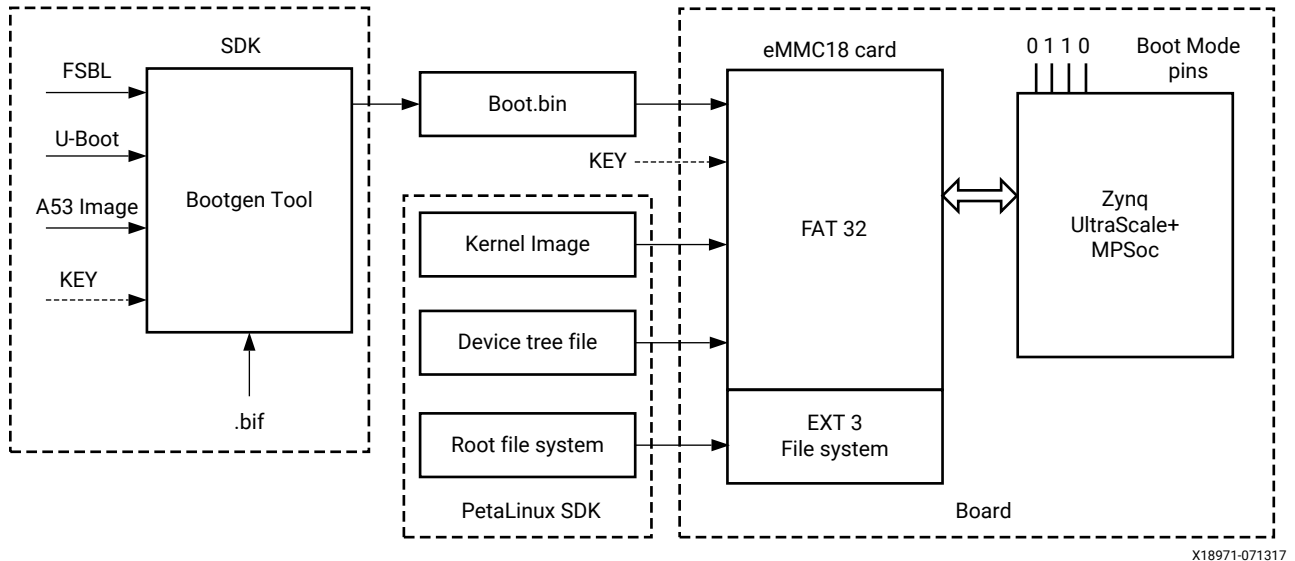
eMMC18 Boot Mode

eMMC18 boot (version 4.5) supports the following:

- FAT 16/32 file systems for reading the boot images.
- Image search for MultiBoot with a maximum number of 8,192 files being supported.

The following figure shows an example for booting Linux in eMMC18 mode.

Figure 18: Booting in eMMC18 Mode



To create an eMMC18 boot image, provide the following files to Bootgen:

- An FSBL ELF
- A Cortex-R5F-0/R5-1 and/or a Cortex-A53 application ELF
- Optional authentication and encryption keys

The Bootgen tool generates the `boot.bin` binary file. You can write the `boot.bin` file into an eMMC18 card using an eMMC18 card reader.

In PetaLinux, do the following:

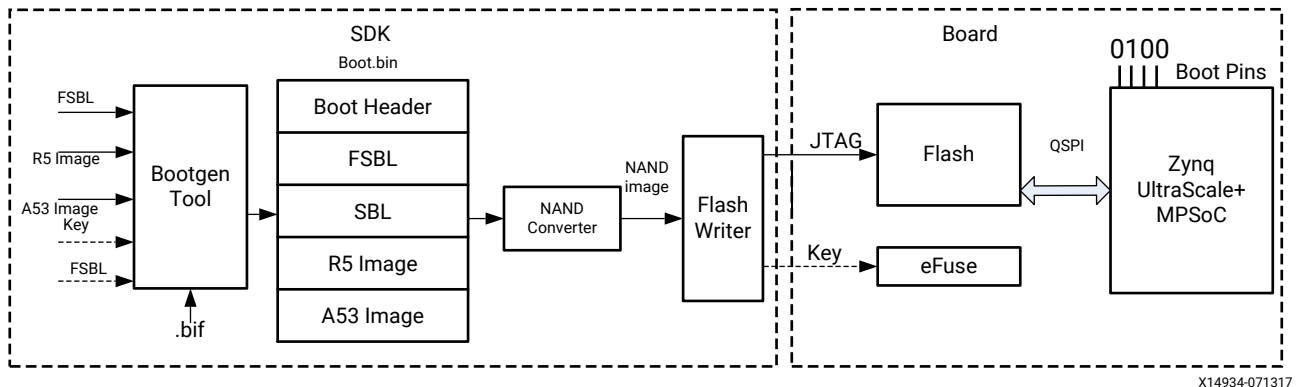
- Build the Linux kernel image, device tree file, and the root file system.
- Copy the files into the eMMC18 card.

The formatted eMMC18 card then contains the `boot.bin`, the kernel image, and the device tree files in the FAT32 partition; the root file system resides in the EXT3 partition.

NAND Boot Mode

The NAND boot only supports 8-bit widths for reading the boot images, and image search for MultiBoot. The following figure shows an example for booting Linux in NAND mode.

Figure 19: Booting in NAND Mode



To create a NAND boot image, provide the following files to Bootgen:

- An FSBL ELF
- A Cortex-R5F-0/R5-1 application ELF and/or an Cortex-A53 application ELF
- Optional authentication/encryption keys

The Bootgen tool generates the `boot.bin` binary file. You can then write the NAND bootable image into the NAND flash using the flash writer



IMPORTANT! To boot from NAND, configure boot pins to `0x4`.

JTAG Boot Mode

You can manually download any software image needed for the PS and any hardware image on the PL using JTAG. For JTAG boot mode settings, see this [link](#) in the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*.



IMPORTANT! Secure boot is not supported in the JTAG mode.

USB Boot Mode

The USB boot mode supports only USB 2.0. In USB boot mode, both the secure and non-secure boot modes are supported. USB boot mode is not supported for DDR-less systems. Features like Multiboot, fallback image, and XIP are not supported.

Note: USB boot mode is disabled by default in FSBL. To enable the USB boot mode, configure the `FSBL_USB_EXCLUDE_VAL` to 0 in `xfsbl_config.h` file.

Table 13: USB Boot Mode Details

Pin	Functionality
Mode pins	0x7
MIO pins	MIO[63:52]
Non-secure	Yes
Secure	Yes
Signed	Yes
Mode	Slave

USB boot mode requires a host PC with dfu-utils installed on it. The host and device need to be connected through a USB 2.0 or USB 3.0 cable. The host must contain one `boot.bin` to be loaded by bootROM, which contains only `fsbl.elf` and another `boot_all.bin` to be loaded by FSBL. On powering up the board in USB boot mode, issue the following commands:

- On Linux host:
 - **dfu-util -D boot.bin:** This downloads the file to the device, which is processed by bootROM.
 - **dfu-util -D boot_all.bin:** This downloads the file to the device, which is processed by FSBL.
- On Windows host:
 - **dfu-util.exe -D boot.bin:** This downloads the file to the device, which is processed by bootROM.
 - **dfu-util.exe -D boot_all.bin:** This downloads the file to the device, which is then processed by FSBL.

The size limit of `boot.bin` and `boot_all.bin` are the sizes of OCM and DDR. The size of OCM is 256 KB.

Secondary Boot Mode

There is a provision to have two boot devices in the Zynq UltraScale+ MPSoC architecture. The primary boot mode is the boot mode used by bootROM to load FSBL and optionally PMU FW. The secondary boot mode is the boot device used by FSBL to load all the other partitions. The supported secondary boot modes are QSPI24, QSPI32, SD0, eMMC, SD1, SD1-LS, NAND and USB.

When using PS-PCIe® on ZU+ in Endpoint mode, running FSBL is enough to set up the block for endpoint mode operation. FSBL should be able to program the PS/PS-PCIe® and GTR within 100 ms. However, this doesn't include PL-bitstream programming as including that would make this greater than 100 ms.



IMPORTANT! If secondary boot mode is specified, it should be different from the primary boot device. For example, if QSPI32 is the primary boot mode, QSPI24 cannot be the secondary boot mode. Instead, you can have SD0, eMMC, SD1, SD1-LS, NAND, USB as secondary boot modes. All combinations of boot devices are supported as primary and secondary boot devices.

Note: By default, the secondary boot mode is the same as primary boot mode and there will be only one boot image.

See [What is Secondary Boot Mode in FSBL wiki page](#) for more information.

Detailed Boot Flow

The platform management unit (PMU) in the Zynq UltraScale+ MPSoC is responsible for handling the primary pre-boot tasks.

PMU ROM will execute from a ROM during boot to configure a default power state for the device, initialize RAMs, and test memories and registers. After the PMU performs these tasks and relinquishes system control to the configuration security unit (CSU), it enters a service mode. In this mode, the PMU responds to interrupt requests made by system software through the register interface or by hardware through the dedicated I/O to perform platform management services.

Pre-Boot Sequence

The following table lists the tasks performed by the PMU in the pre-Boot sequence.

Table 14: Pre-Boot Sequence

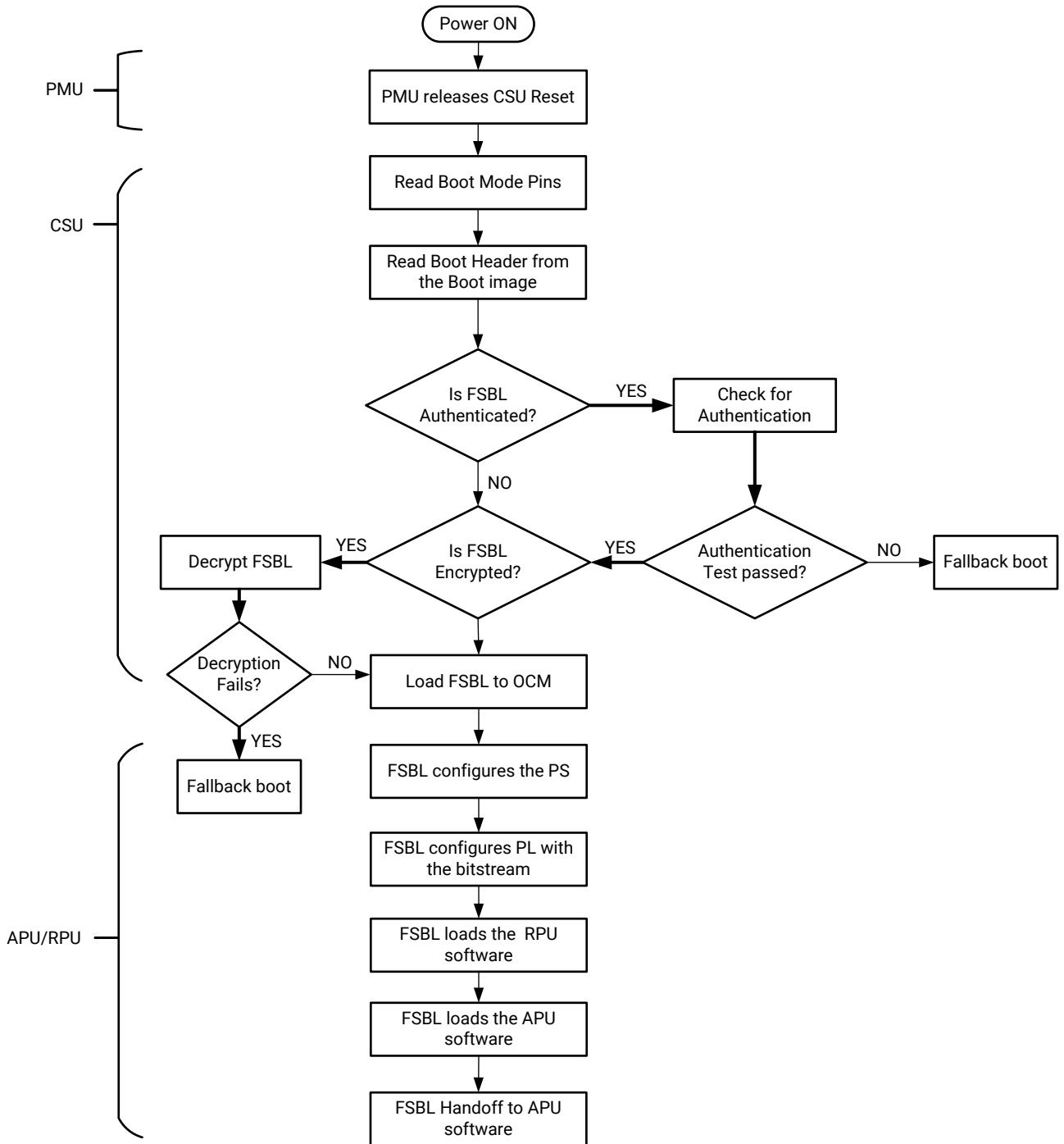
Pre-Boot Task	Description
0	Initialize MicroBlaze™ processor. Capture key states.
1	Scan, and clear LPD and FPD.
2	Initialize the System Monitor.
3	Initialize the PLL used for MBIST clocks.
4	Zero out the PMU RAM.
5	Validate the PLL. Configure the MBIST clock.
6	Validate the power supply.
7	Repair FPD memory (if required).
8	Zeroize the LPD and FPD and initialize memory self-test.
9	Power-down any disabled IPs.
10	Either release CSU or enter error state.
11	Enter service mode.

As soon as the CSU reset is released, it executes the CSU bootROM and performs the following sequence:

1. Initializes the OCM.
2. Determines the boot mode by reading the boot mode register, which captures the boot-mode pin strapping at the POR.
3. The CSU continues with the FSBL load and the optional PMU firmware load. PMU firmware is the software that can be executed by the PMU unit. The code executes from the RAM of the PMU. See [Chapter 9: Platform Management](#) for more information.

The following figure shows the detailed boot flow diagram.

Figure 20: Detailed Boot Flow Example



X14935-070717

Disabling FPD in Boot Sequence

Perform the following to avoid an FPD lockout, where FPD Power is applied momentarily:

- Apply the power until the completion of bootROM execution.
- To power down the FP during FSBL execution, set FPD bit '22' of PMU_GLOBAL_REQ_PWRDWN_STATUS register.
- To bring the FP domain up in a later stage of the boot process, set the PMU_GLOBAL_REQ_PWRUP_STATUS bit to '22'.

Perform the following in cases where the FPD power is not applied before the FSBL boots

1. Power up the R5.
2. A register is set indicating the FPD is locked pending POR as the reset or clear sequence cannot execute on the FPD.
3. R5 can read the FP locked status from PMU_GLOBAL_REQ_ISO_STATUS register bit '4'.
4. At this stage, PMU_GLOBAL_REQ_PWRUP_STATUS bit '22' will not be set.
5. To bring the FPD node back up, power must be supplied to the node and a POR needs to be issued.

Setting FSBL Compilation Flags

You can set compilation flags using the C/C++ settings in the Vitis FSBL project, as shown in the following figure:

Note: There is no need to change any of the FSBL source files or header files to include these flags.

Figure 21: FSBL Debug Flags

The following table lists the FSBL compilation flags.

Table 15: FSBL Compilation Flags

Flag	Description
FSBL_DEBUG	Prints basic information and error prints, if any.
FSBL_DEBUG_INFO	Enables debug information in addition to the basic information.
FSBL_DEBUG_DETAILED	Prints information with all data exchanged.
FSBL_NAND_EXCLUDE	Excludes NAND support code.
FSBL_QSPI_EXCLUDE	Excludes QSPI support code.
FSBL_SD_EXCLUDE	Excludes SD support code.

Table 15: FSBL Compilation Flags (cont'd)

Flag	Description
FSBL_RSA_EXCLUDE	Excludes authentication code.
FSBL_AES_EXCLUDE	Excludes decryption code.
FSBL_BS_EXCLUDE	Excludes bitstream code.
FSBL_WDT_EXCLUDE	Excludes WDT support code.
FSBL_USB_EXCLUDE	Excludes USB code. This is set to 1 by default. Set this value to 0 to enable USB boot mode.
FSBL_FORCE_ENC_EXCLUDE_VAL	Excludes forcing encryption of all partitions when ENC_ONLY fuse is programmed. By default, this is set to 0. FSBL forces to enable encryption for all the partitions when ENC_ONLY is programmed.

See I'm unable to build FSBL due to size issues, how can I reduce its footprint section in [FSBL wiki page](#) for more information.

Enabling Debug Prints

See [FSBL wiki page](#) for more information on debugging FSBL.

Fallback and MultiBoot Flow

In the Zynq® UltraScale+™ MPSoC, the CSU bootROM supports MultiBoot and fallback boot image search where the configuration security unit CSU ROM or bootROM searches through the boot device looking for a valid image to load. The sequence is as follows:

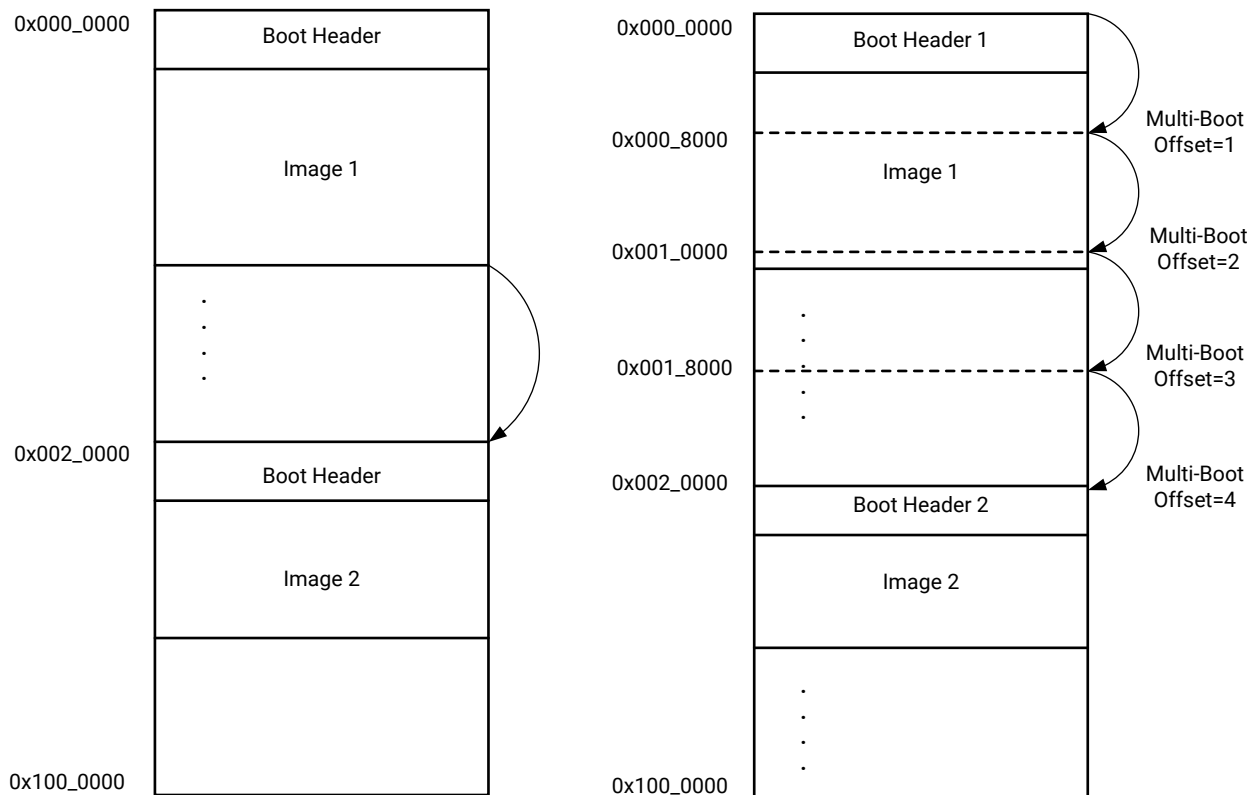
- BootROM searches for a valid image identification string (XLNX as image ID) at offsets of 32 KB in the flash.
- After finding a valid identification value, validates the checksum for the header.
- If the checksum is valid, the bootROM loads the image. This allows for more than one image in the flash.

In MultiBoot:

- CSU ROM or FSBL or the user application must initiate the boot image search to choose a different image from which to boot.
- To initiate this image search, CSU ROM or FSBL updates the MultiBoot offset to point to the intended boot image, and generates a soft reset by writing into the CRL_APB register.

The following figure shows an example of the fallback and MultiBoot flow.

Figure 22: MultiBoot Flow



X14936-071217

Note: The same flow is applicable to both Secure and Non-secure boot methods.

In the example fallback boot flow figure, the following sequence occurs:

- Initially, the CSU bootROM loads the boot image found at 0x000_0000.
- If this image is found to be corrupted or the decryption and authentication fails, CSU bootROM increments the MultiBoot offset by one and searches for a valid boot image at 0x000_8000 (32 KB offset).
- If the CSU bootROM does not find the valid identification value, it again increments the MultiBoot offset by 1, and searches for a valid boot image at the next 32 KB aligned address.
- The CSU bootROM repeats this until a valid boot image is found or the image search limit is reached. In this example flow, the next image is shown at 0x002_0000 corresponding to a MultiBoot offset value of four.
- In the example MultiBoot flow, to load the second image that is at the address 0x002_0000, MultiBoot offset is updated to four by FSBL/CSU-ROM. When the MultiBoot offset is updated, soft reset the system.

The following table shows the MultiBoot image search range for different booting devices.

Table 16: Boot Devices and MultiBoot Image Search Range

Boot Device	MultiBoot Image Search Range
QSPI Single (24-bit)	16 MB
QSPI Dual (24-bit)	32 MB
QSPI Single (32-bit)	256 MB
QSPI Dual (32-bit)	512 MB
NAND	128 MB
SD/EMMC	8,191 boot files
USB	Not applicable

FSBL Build Process

After authenticating and/or decrypting, the FSBL is loaded into OCM and handed off by the CSU bootROM. First Stage Boot Loader configures the FPGA with a bitstream (if it exists) and loads the Standalone (SA) Image or Second Stage Boot Loader image from the non-volatile memory (NAND/SD/eMMC/QSPI) to RAM(DDR/TCM/OCM). It takes the Cortex-R5F-0/R5F-1 processor or the Cortex-A53 processor unit out of reset. It supports multiple partitions. Each partition can be a code image or a bitstream. Each of these partitions, if required, will be authenticated and/or decrypted.

Note: If you are creating a custom FSBL, you should be aware that the OCM size is 256 KB and is available to CSU bootROM. The FSBL size is close to 170 KB and it would fit in the OCM. While using the USB boot mode, you should make sure that the PMU firmware is loaded by the FSBL and not by the CSU bootROM. This is because the size of boot.bin loaded by the CSU bootROM should be less than 256 KB.

Creating a New Zynq UltraScale+ MPSoC FSBL Application Project

To create a new Zynq UltraScale+ MPSoC FSBL application in the Vitis software platform, do the following:

1. Click **File** → **New** → **Application Project**.
The New Application Project dialog box appears.
2. In the Project Name field, type a name for the new project.
3. Select the location for the project. To use the default location as displayed in the Location field, leave the **Use default location** check box selected. Otherwise, click to deselect the check box, then type or browse to the directory location.
4. Select **Create a new platform from hardware (XSA)**. The Vitis IDE lists the all the available pre-defined hardware designs.

5. Select any one hardware design from the list and click **Next**.
6. From the CPU drop-down list, select the processor for which you want to build the application. This is an important step when there are multiple processors in your design. In this case you can either select **psu_cortexa53_0** or **psu_cortexr5_0**.
7. Select your preferred language: **C**.
8. Select an OS for the targeted application.
9. Click **Next**.
10. In the Templates dialog box, select the Zynq UltraScale+ MPSoC FSBL template.
11. Click **Finish** to create your application project and board support package (if it does not exist).

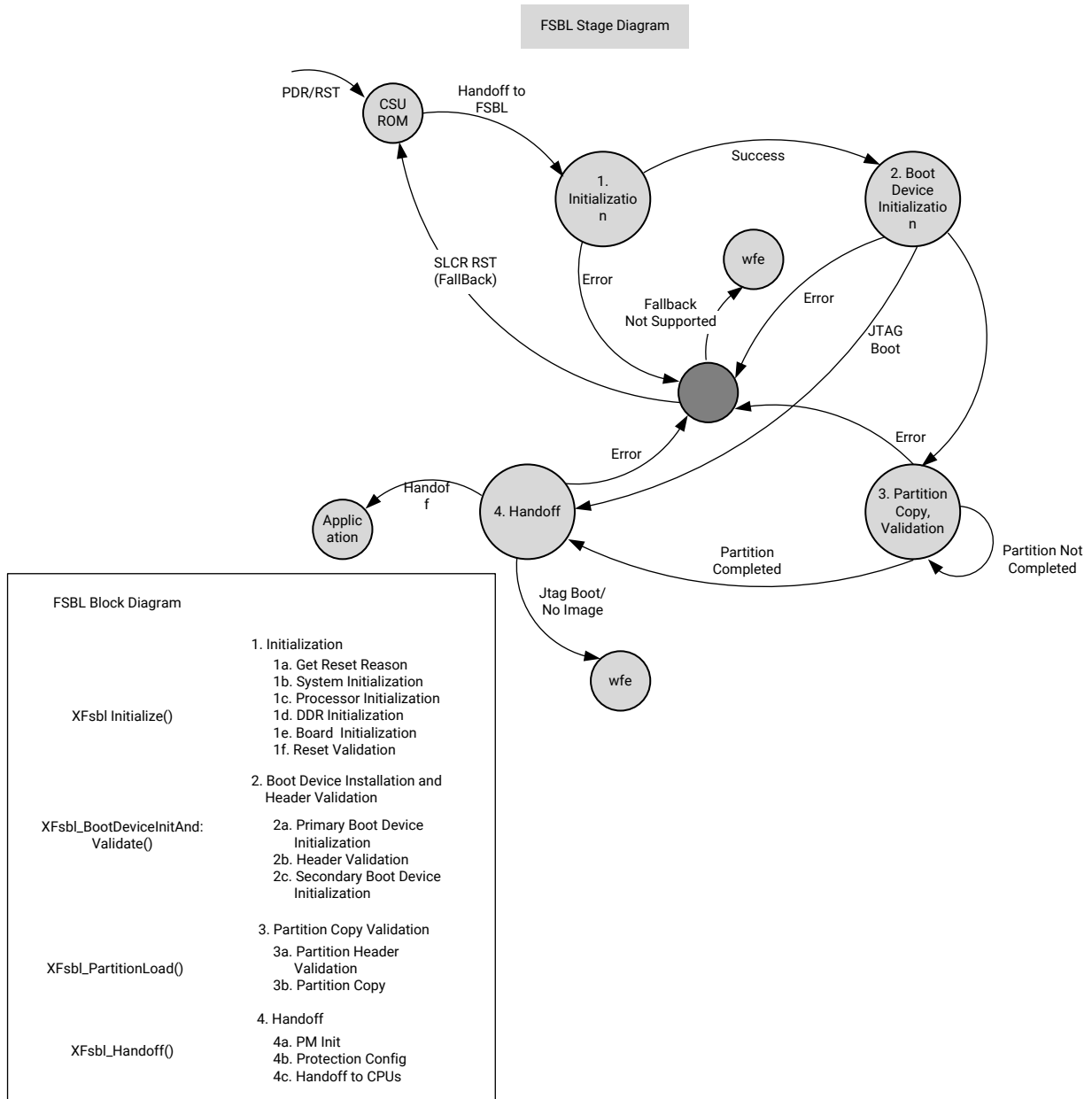
Phases of FSBL Operation

FSBL operation includes the following four stages:

- Initialization
- Boot device initialization
- Partition loading
- Handoff

The following figure shows the stages of FSBL operation:

Figure 23: Stages of FSBL



X19962-101917

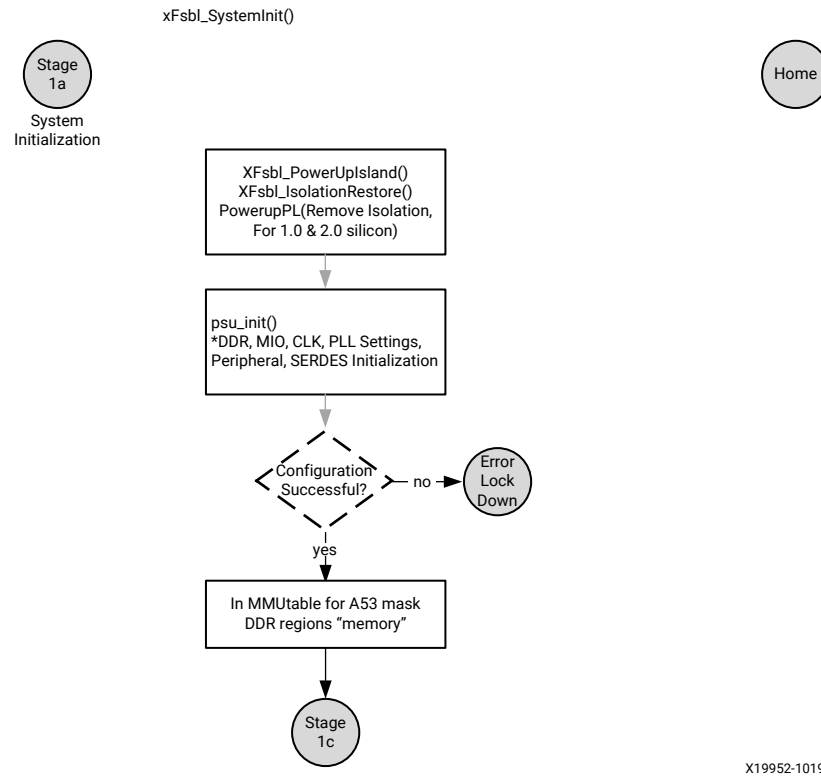
Initialization

Initialization consists of the following four internal stages:

XFsbI_SystemInit

This function powers up PL for 1.0 and 2.0 silicon and removes PS-PL isolation. It initializes clocks and peripherals as specified in psu-init. This function is not called in APU only reset.

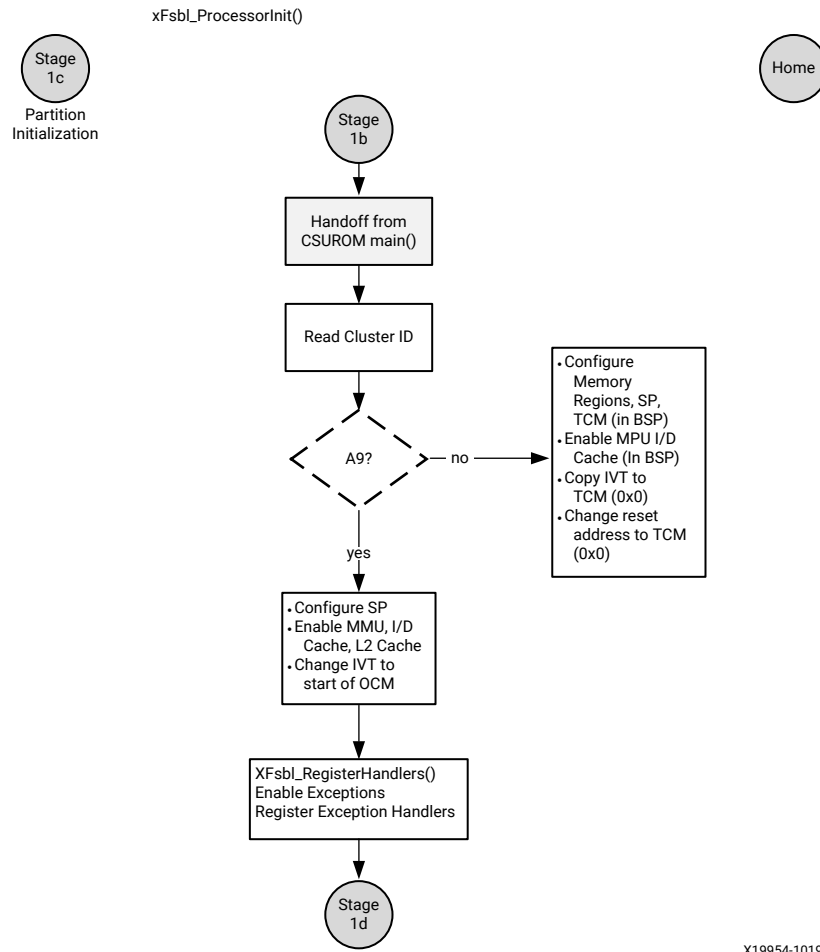
Figure 24: FSBL System Initialization



XFsbL_ProcessorInit

Processor initialization will start in this stage. It will set up the Instruction and Data caches, L2 caches, MMU settings, stack pointers in case of A53 and I/D caches, MPU settings, memory regions, stack pointers, and TCM settings for R5-0. Most of these settings will be performed in BSP code initialization. IVT vector is changed to the start of OCM for A53 and to start of TCM (0x0 in lowvec and 0xffff0000 in highvec) in case of R5-0.

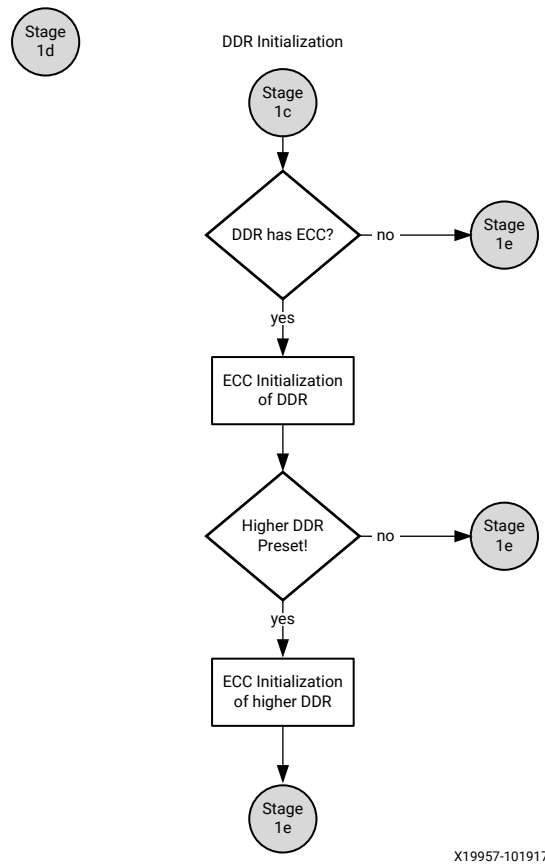
Figure 25: Processor Initialization



Initialize DDR

DDR would be initialized in this stage. This function is not called in Master only reset.

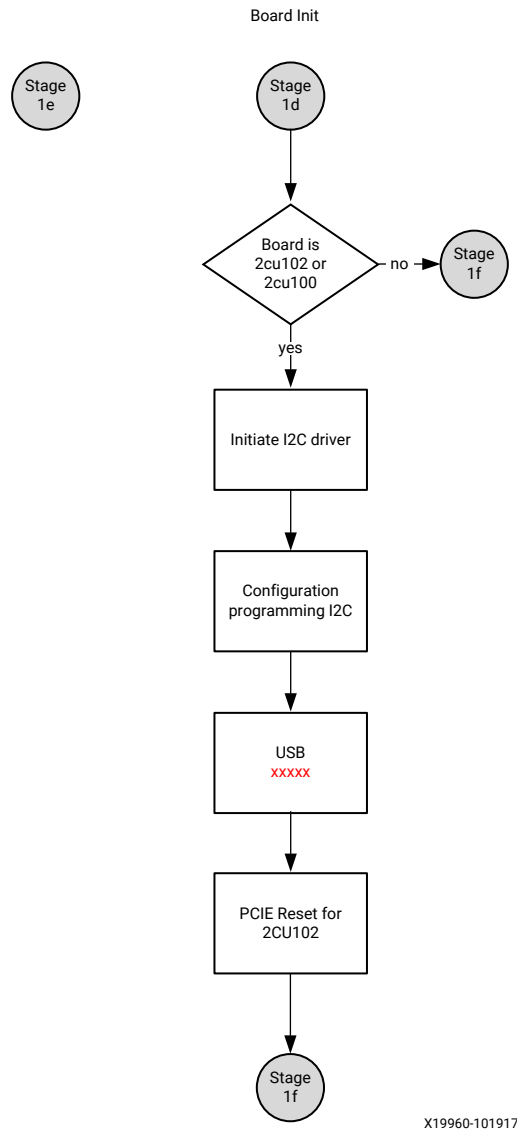
Figure 26: DDR Initialization



XFsbI_BoardInit

This function performs required board specific initializations. Most importantly, it configures GT lanes and IIC.

Figure 27: Board Initialization

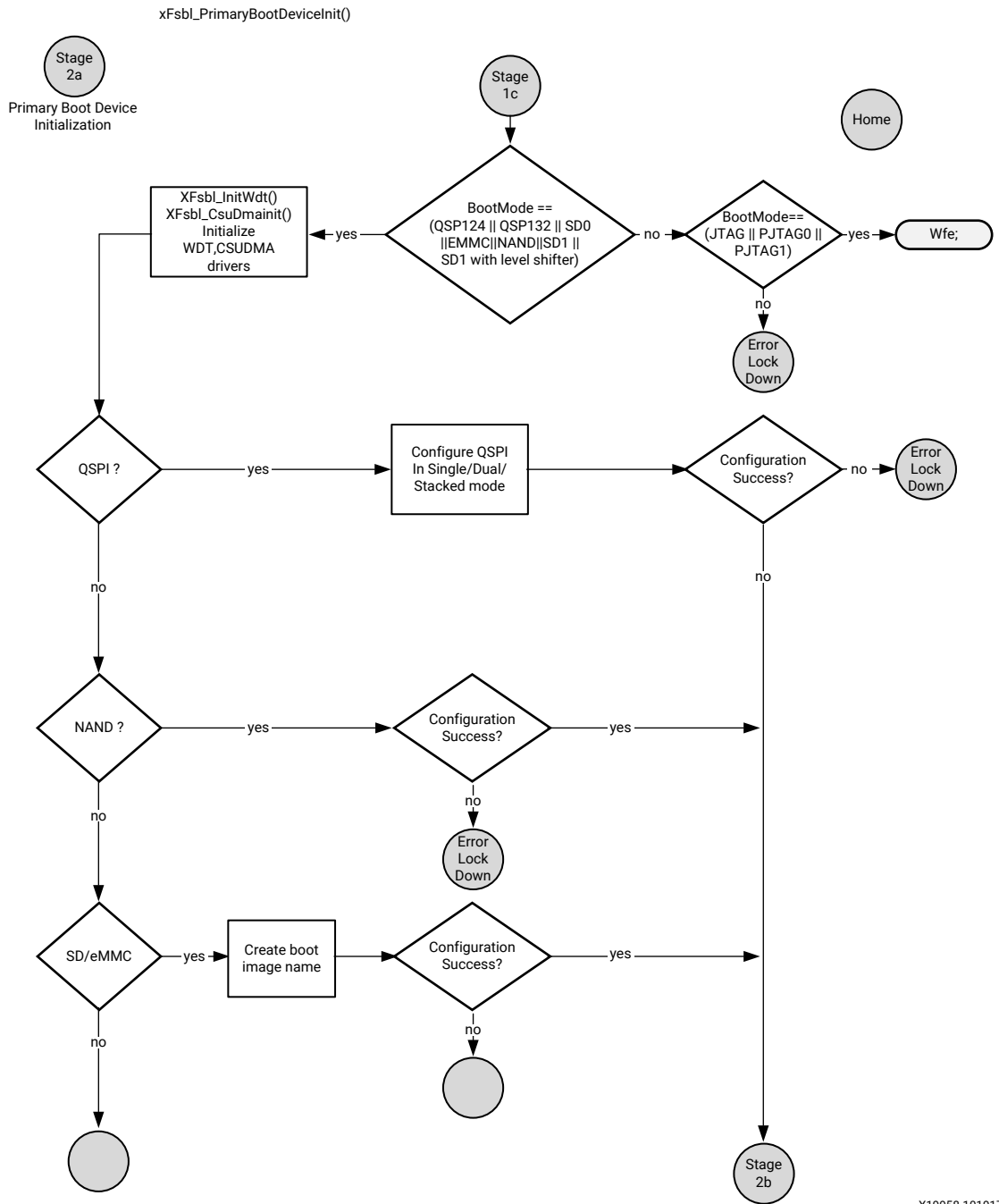


Boot Device Initialization

XFsbI_PrimaryBootDeviceInit

This stage involves reading boot mode register to identify the primary boot device and initialize the corresponding device. Each boot device driver provides init, copy and release functions which are initialized to DevOps function pointers in this stage.

Figure 28: Primary Boot Device Initialization

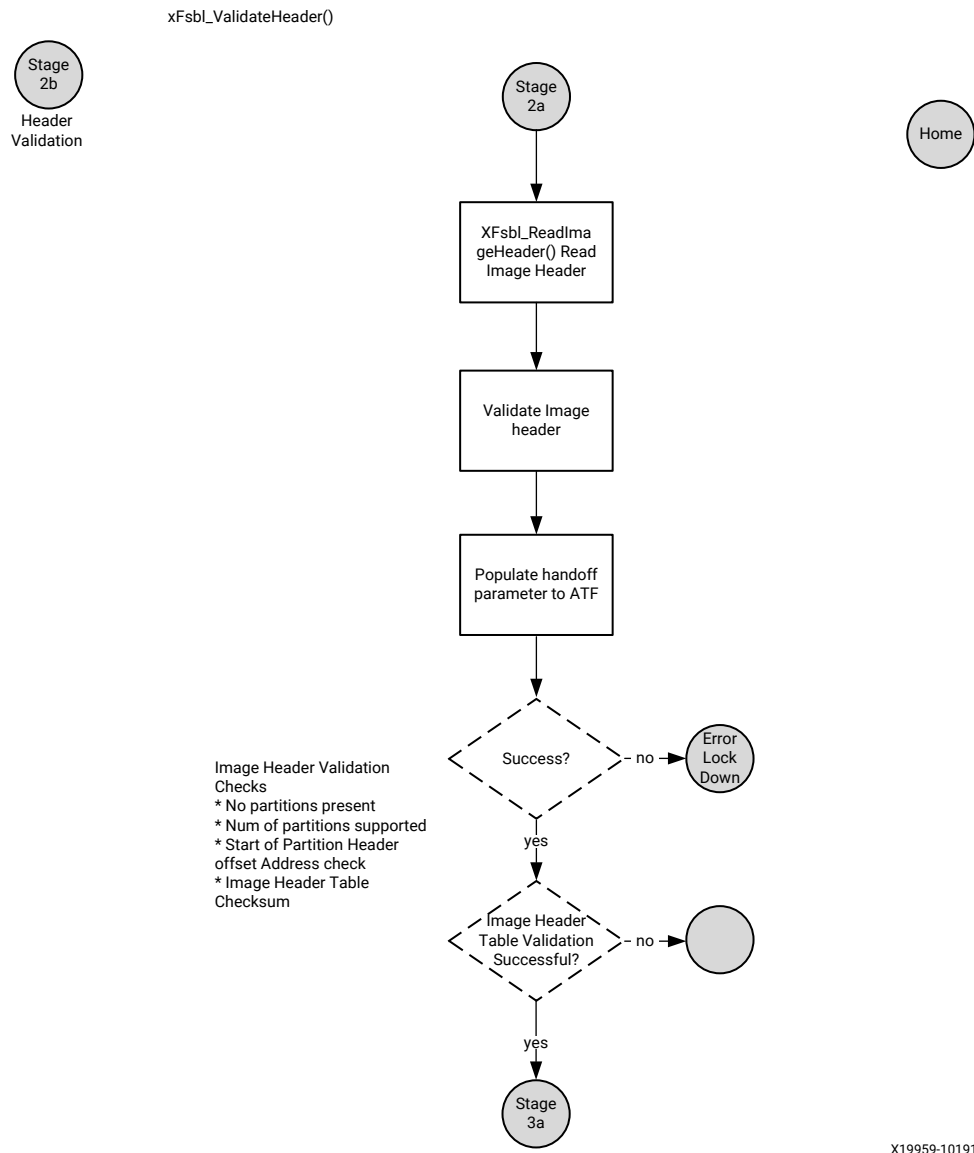


X19958-101917

XFsbI_ValidateHeader

Using the copy functions provided, the FSBL reads the boot header attributes and image offset address. It reads the EFUSE bit to check for authentication. It reads the image header and validates the image header table. It then reads the Partition Present Device attribute of image header. A non-zero value indicates a secondary boot device. A zero value indicates that the secondary boot device is the same as the primary boot device.

Figure 29: Validating Header



XFsbI_SecondaryBootDeviceInit

This function is called in case of a non-zero value of Partition Present Device attribute of image header table. It initializes the secondary boot device driver and the secondary boot device would be used to load all partitions by FSBL.

XFsbI_SetATFHandoffParams

ATF is assumed to be the next loadable partition after FSBL. It is capable of loading U-Boot and secure OS and hence, it is passed a handoff structure.

The first partition of an application will have a non-zero execution address. All the remaining partitions of that application will have 0 as execution address. Hence look for the non-zero execution address for partition which is not the first one and ensure the CPU is A53.

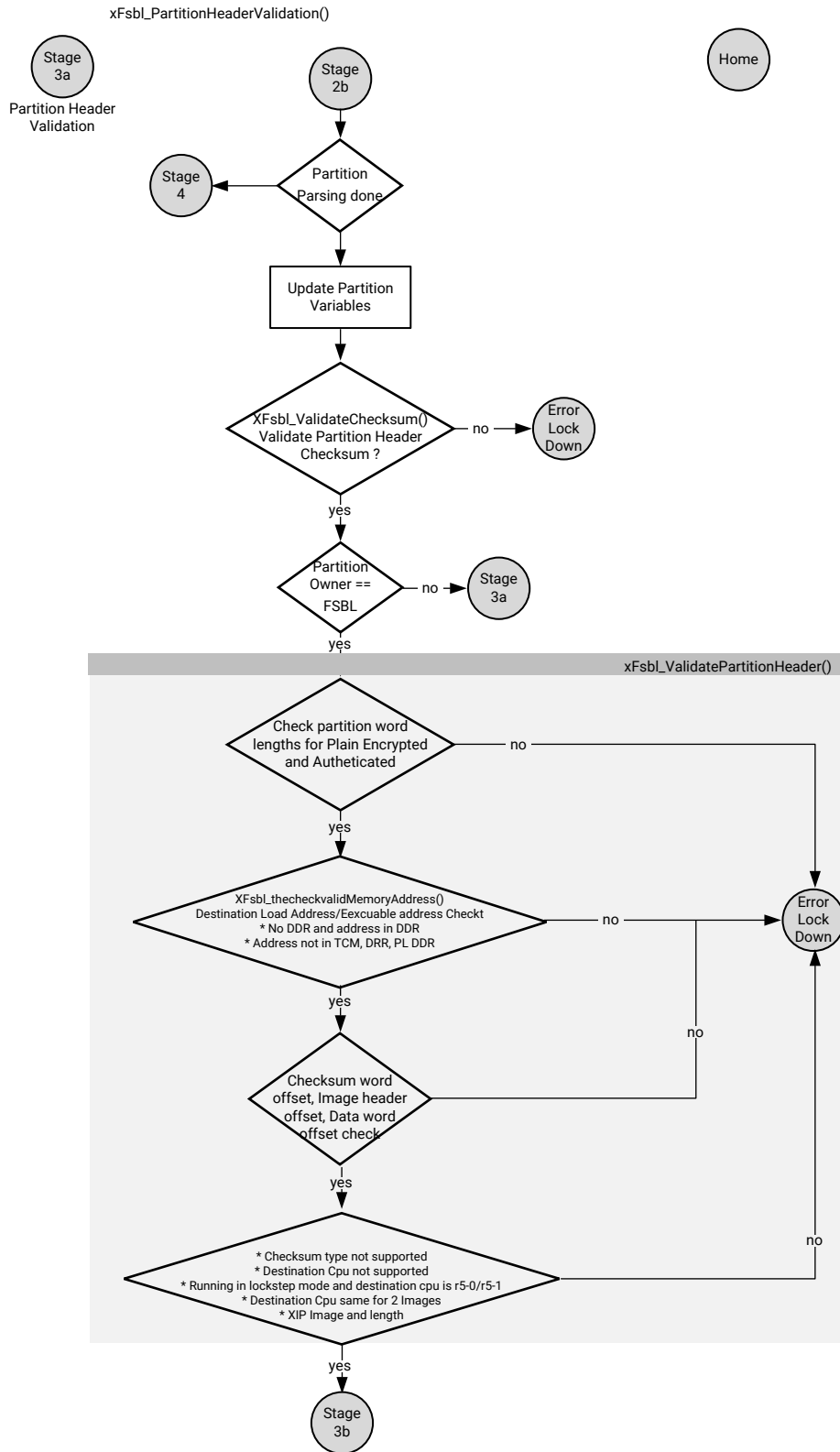
This function sets the handoff parameters to the Arm Trusted Firmware (ATF). The first argument is taken from the FSBL partition header. A pointer to the handoff structure containing these parameters is stored in the PMU_GLOBAL.GLOBAL_GEN_STORAGE6 register, which the ATF reads. The structure is filled with magic characters 'X', 'L', 'N', and 'X' followed by the total number of partitions and execution address of each partition.

Partition Loading

XFsbI_PartitionHeaderValidation

Partition header is validated against various checks. All the required partition variables are updated at this stage. If the partition owner is not FSBL, partition will be ignored and FSBL will continue loading the other partitions.

Figure 30: Partition Header Validation

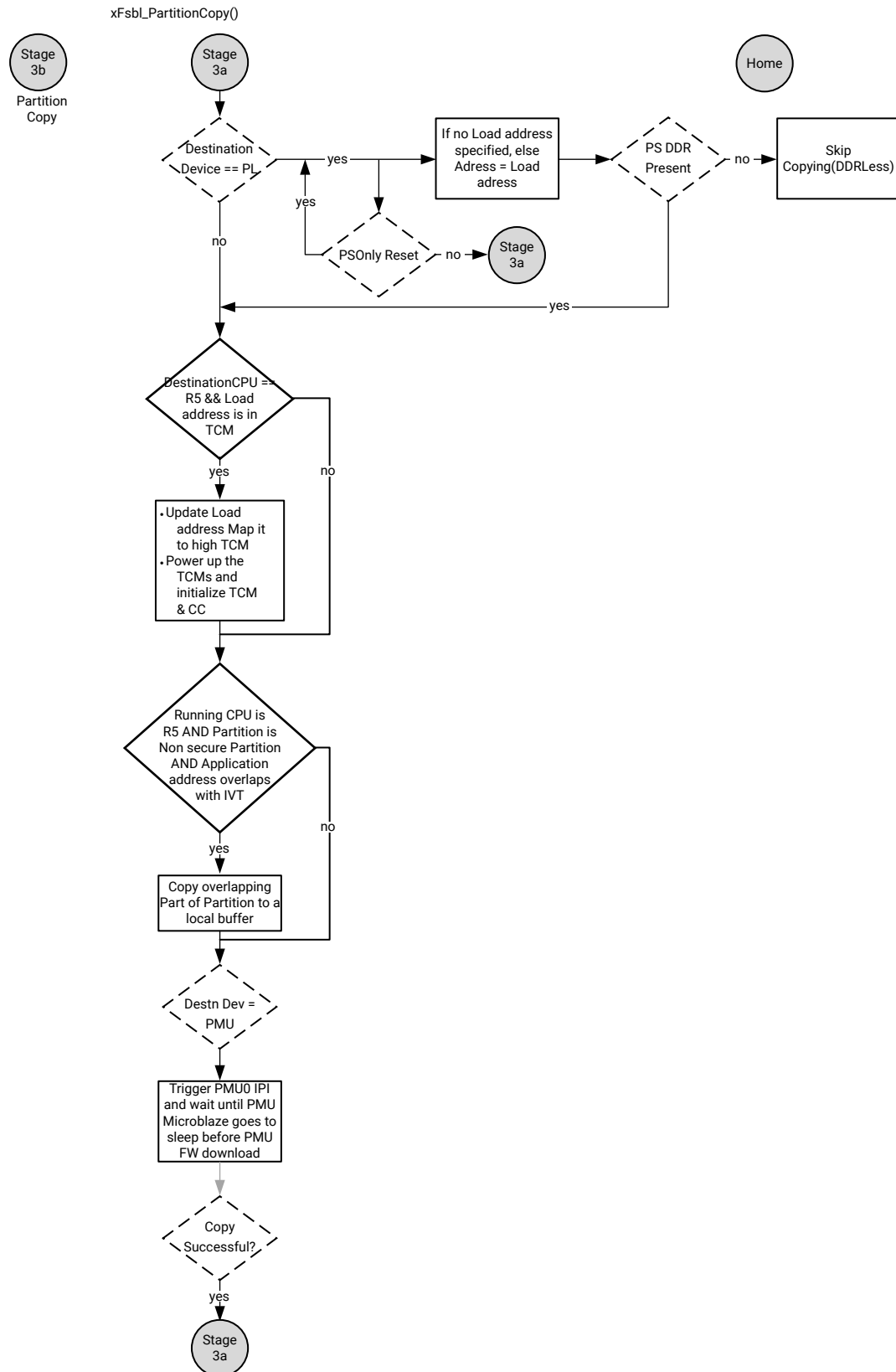


X19951-101917

XFsbI_PartitionCopy

Partition will be copied to the DDR or TCM or OCM or PMU RAM.

Figure 31: Partition Copy

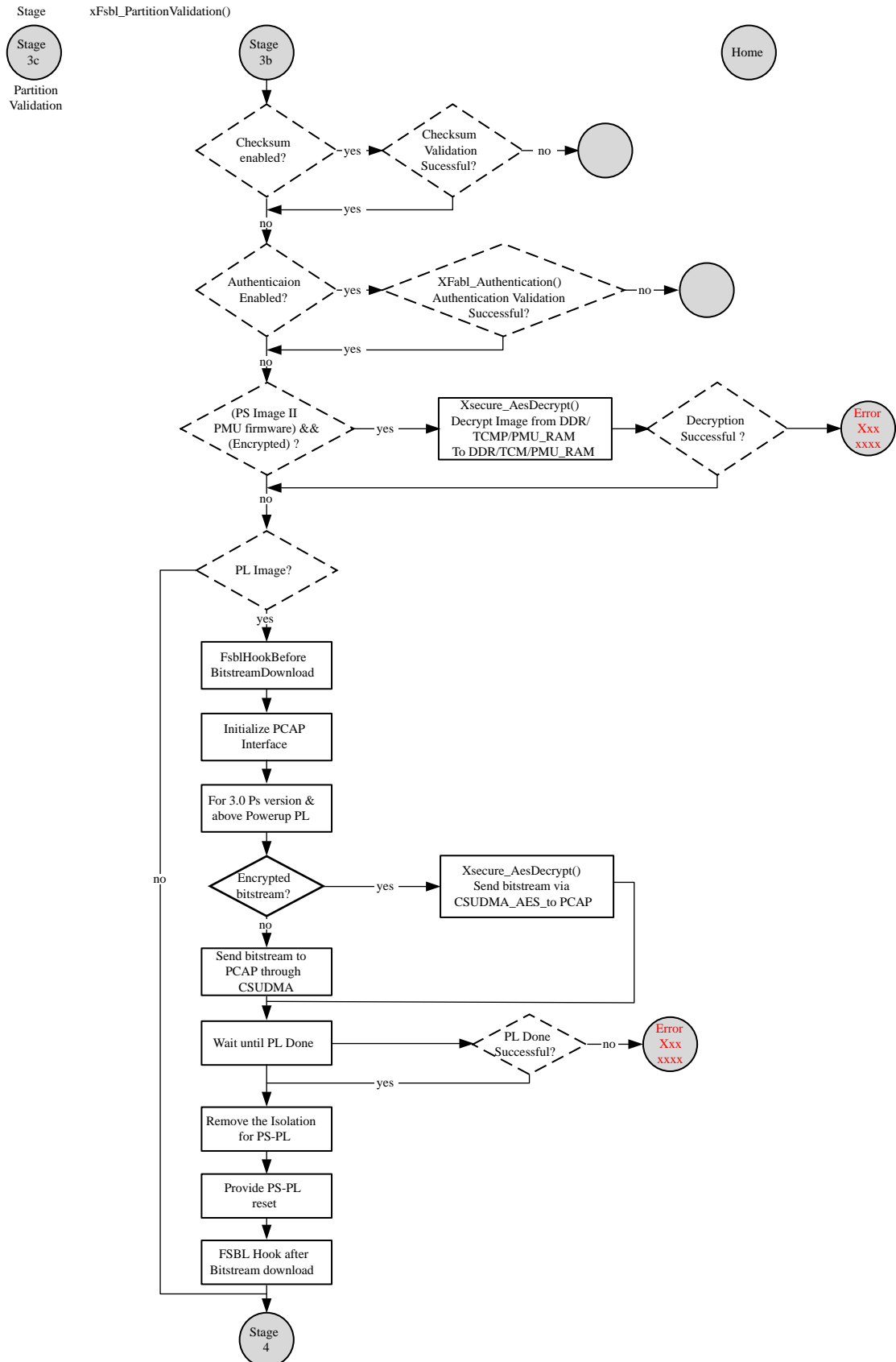


X19950-101917

XFsbI_PartitionValidation

Partition will be validated based on the partition attributes. If checksum bit is enabled, then the partition will be validated first for checksum correctness and then, based on the authentication flag, it would be authenticated. If encryption flag is set, then the partition will be decrypted and then copied to the destination.

Figure 32: Partition Validation Function



Handoff

In this stage, `protection_config` functions from `psu_init` will be executed and then, any handoff functionality is executed. Also PS-PL isolation is removed unconditionally. R5 will be brought out of reset if there is any partition supposed to run on its cores. R5-0/R5-1 will be configured to boot in lowvec mode or highvec mode as per the settings provided by you while building the boot image. The handoff address in lowvec mode is `0x0` and `0xffff0000` in highvec mode. Lowvec/Highvec information should be specified by you while building the boot image. After all the other PS images are done, then the running CPU will be handed off with an update of the PC value. If there is no image to hand off for the running the CPU, FSBL will be in wfe loop.

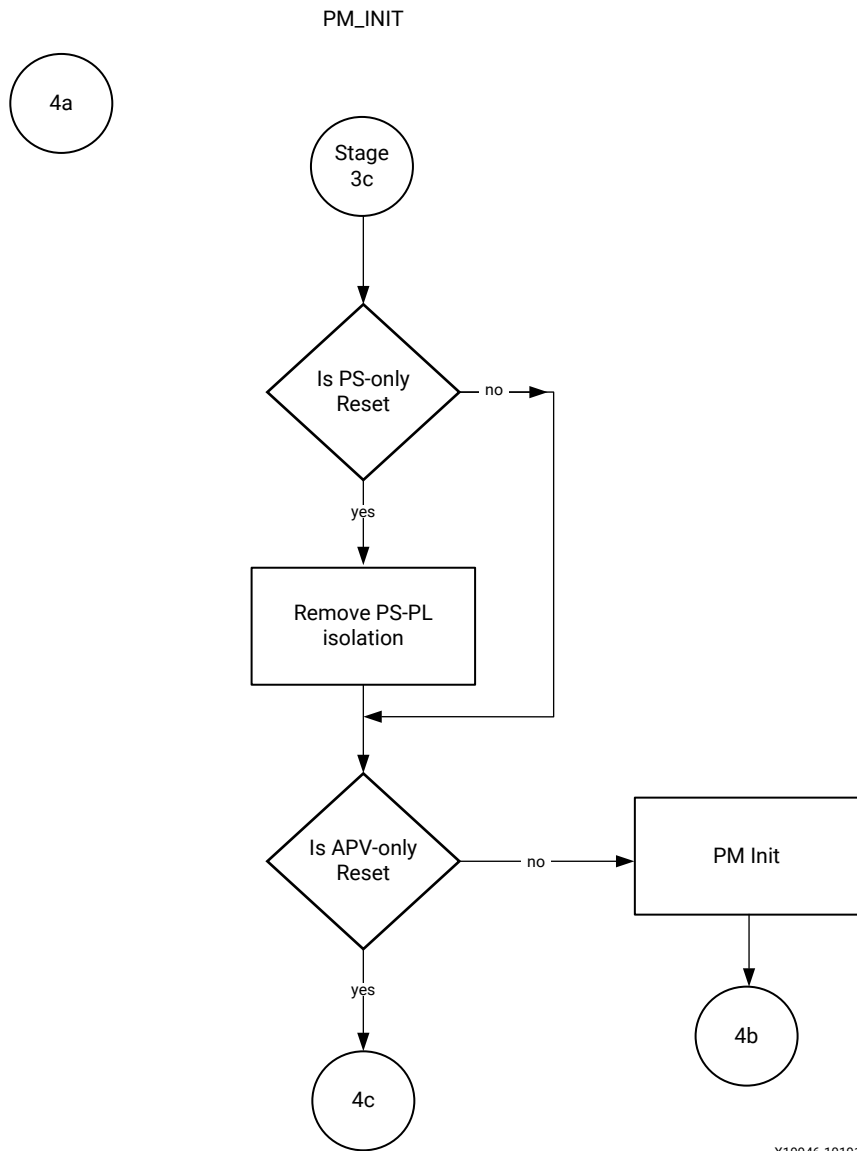
Any running processor cannot pass any parameters to any other processor. Any communication between various partitions can happen by reading from (or writing to) the PMU global registers.

Handoff on the running processor involves updating Program Counter (PC) of the running processor, as is done in the case of APU Reset. Handoff to other processors involves updating their PCs and bringing the processors out of reset.

XFsbI_PmInit

This function initializes and configures the Inter Processor Interrupts (IPI). It then writes the PM configuration object address to an IPI buffer and triggers an IPI to the target. The PMU firmware then reads and configures the device nodes as specified in the configuration object.

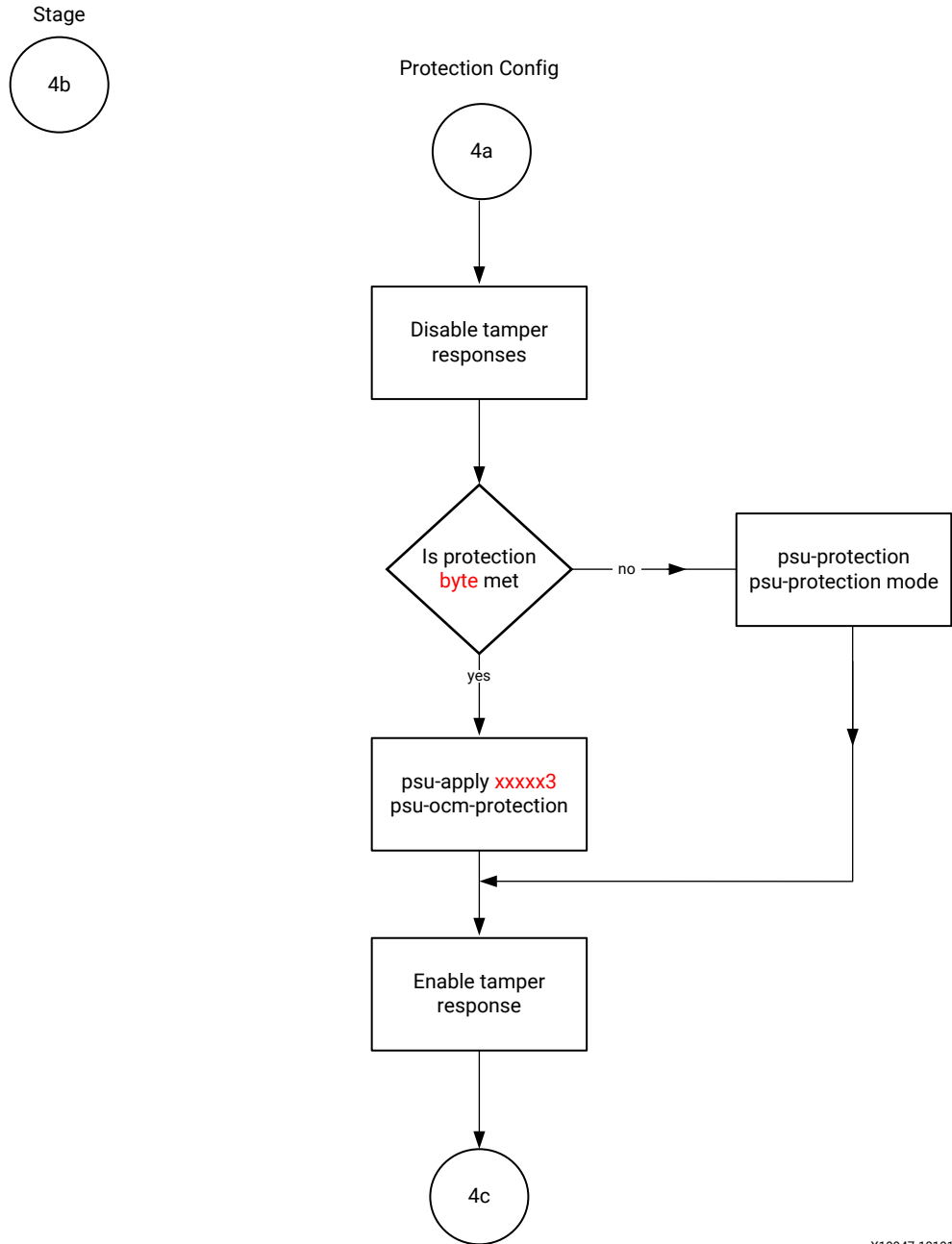
Figure 33: PM Initialization



Protection Configuration

In this stage, `protection_config` functions from `psu_init` will be executed. The application of protection happens in this stage.

Figure 34: Protection Configuration



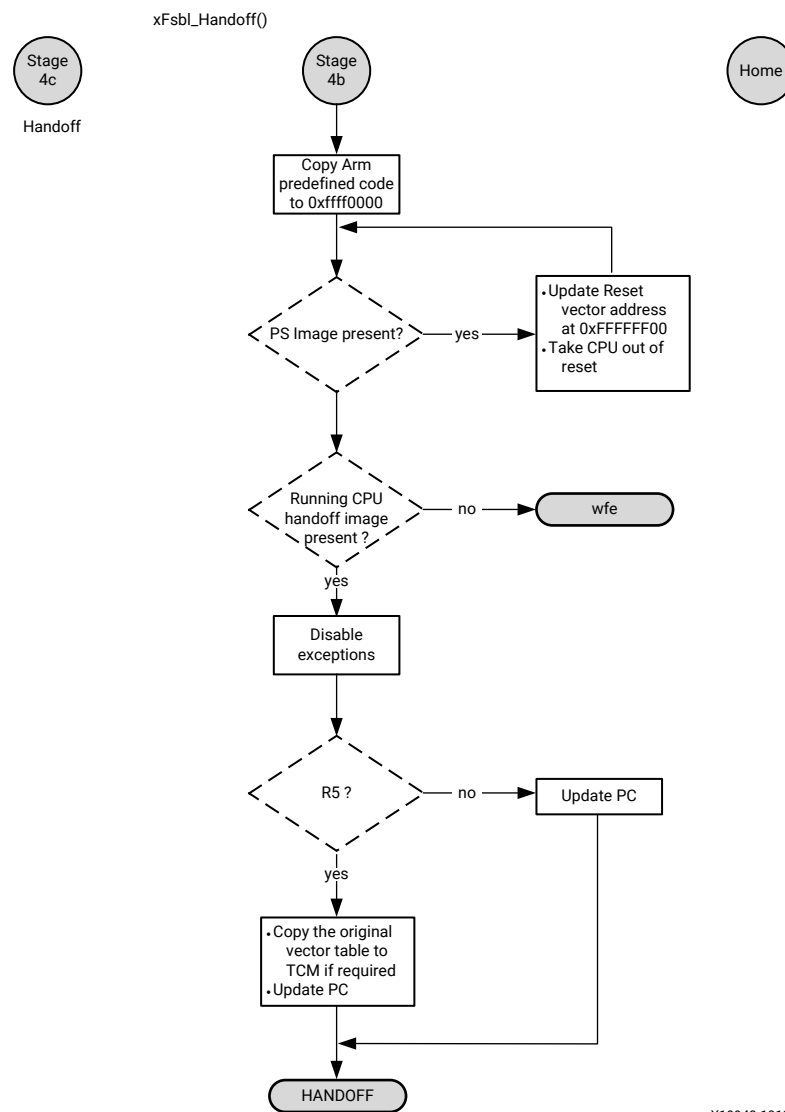
X19947-101917

Handoff

Handoff on the running processor involves updating Program Counter (PC) of the running processor, as is done in the case of APU Reset. Handoff to other processors involves updating their PCs and bringing the processors out of reset. A53 FSBL will bring R5 out of reset if there is any partition to run on it. R5 will be configured to boot in lowvec mode or highvec mode as per the settings provided by you while building the boot image. The handoff address in lowvec mode is `0x0` and `0xffff0000` in highvec mode.

You must specify Lowvec/Highvec information while building the boot image. After all the other PS images are done, then running the CPU image will be handed off to that cpu with an update on the PC value. If there no image for the running CPU, it will be in wfe loop.

Figure 35: Handoff



X19948-101917

Supported Handoffs

The following table shows the various combinations of handoffs that are supported in FSBL.

Table 17: Supported Handoffs

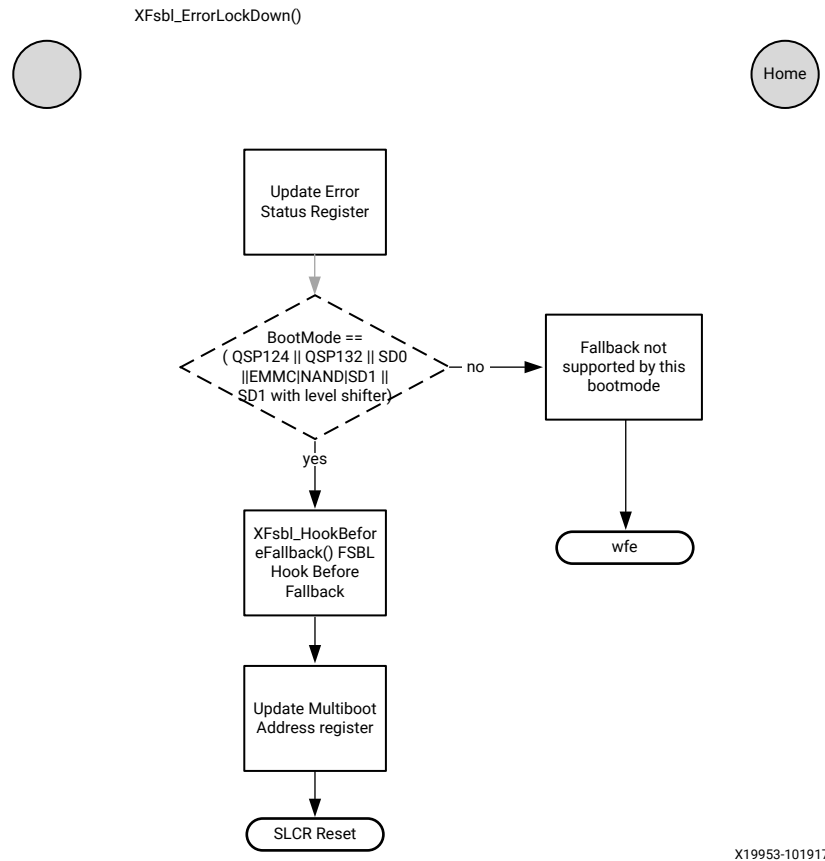
FSBL	Application	Processor Cores	Execution Address
64-bit	64-bit	All (i.e. A53-0, A53-1, A53-2, A53-3)	Any Address
64-bit	32-bit	A53-1, A53-2, A53-3	0x0
32-bit	32-bit	A53-0	Any Address
32-bit	32-bit	A53-1, A53-2, A53-3	0x0
32-bit	64-bit	A53-1, A53-2, A53-3	Any Address

Error Lock Down

`XFsb1_ErrorLockDown` function handles errors in FSBL. This function is called whenever the return value of a function is unsuccessful. This function updates error status register and then loops indefinitely, if fallback is not supported.

In case the boot mode supports fallback, MultiBoot offset register is updated and then waits for a WDT reset to occur. On reboot, bootROM and FSBL read the image from the new address calculated from MultiBoot offset, thus loading a new image.

Figure 36: Error Lock Down Function



X19953-101917

Miscellaneous Functions

The following functions are available in FSBL:

XFsbI_PrintArray

This function prints entire array in bytes as specified by the debug type.

```
void XFsbI_PrintArray (u32 DebugType, const u8 Buf[], u32 Len, const char *Str);
```

Table 18: XFsbI_PrintArray Parameters in FSBL

Parameters	Description
DebugType	Printing of the array is performed as defined by the debug type.
Buf	Pointer to the buffer to be printed
Len	Length of the bytes to be printed
Str	Pointer to the data that is printed

XFsb1_Strcpy

This function to copy the source string to the destination string.

```
char *XFsb1_Strcpy(char *DestPtr, const char *SrcPtr)
```

Table 19: XFsb1_Strcpy Parameters in FSBL

Parameters	Description
DestPtr	Pointer to the buffer to be printed
SrcPtr	Pointer to the buffer containing the source string

XFsb1_Strcat

This function to append the second string to the first string.

```
char* XFsb1_Strcat(char* Str1Ptr, const char* Str2Ptr)
```

Table 20: XFsb1_Strcat Parameters in FSBL

Parameters	Description
Str1Ptr	Pointer to the original string to which string pointed to by Str2Ptr would be appended
Str2Ptr	Pointer to the second string

XFsb1_Strcmp

This function compares strings.

```
s32 XFsb1_Strcmp( const char* Str1Ptr, const char* Str2Ptr)
```

Table 21: XFsb1_Strcmp Parameters in FSBL

Parameters	Description
Str1Ptr	Pointer to the first string
Str2Ptr	Pointer to the second string

XFsb1_MemCpy

This function copies the memory contents pointed to by SrcPtr to the memory pointed to by DestPtr. Len is number of bytes to be copied.

```
void* XFsb1_MemCpy(void * DestPtr, const void * SrcPtr, u32 Len)
```


Table 22: XFsbl_MemCpy Parameters in FSBL

Parameters	Description
SrcPtr	Pointer to the memory contents to be copied
DestPtr	Pointer to the destination
Len	Length of the bytes to be printed

XFsb1_PowerUpIsland

This function checks the power state of one or more power islands and powers them up if required.

```
u32 XFsbl_PowerUpIsland(u32 PwrIslandMask)
```

Table 23: XFsbl_PowerUpIsland Parameters in FSBL

Parameters	Description
PwrIslandMask	Mask of island that needs to be powered up

XFsb1_IsolationRestore

This function requests isolation restore through the PMU firmware.

```
u32 XFsbl_IsolationRestore(u32 IsolationMask);
```

Table 24: XFsbl_IsolationRestore Parameters in FSBL

Parameters	Description
IsolationMask	Mask of the entries for which isolation is to be restored

XFsb1_SetTlbAttributes

This function sets the memory attributes for a section in the translation table.

```
void XFsbl_SetTlbAttributes(INTPTR Addr, UINTPTR attrib);
```

Table 25: XFsbl_SetTlbAttributes Parameters in FSBL

Parameters	Description
Addr	Address for which the attributes are to be set
Attrib	Attributes for the memory region

XFsb1_GetSiliconIdName

This function reads the CSU_ID_CODE register and calculates the SvdId of the device. It returns the corresponding deviceId name.

```
const char *XFsb1_GetSiliconIdName(void);
```

XFsb1_GetProcEng

This function determines and returns the engine type. Currently only CG, EG, and EV engine types are supported.

```
const char *XFsb1_GetProcEng(void);
```

XFsb1_CheckSupportedCpu

This function checks if a given CPU is supported by this variant of Silicon. Currently it checks if it is CG part and disallows handoff to A53_2/3 cores.

```
u32 XFsb1_CheckSupportedCpu(u32 CpuId);
```

Table 26: XFsb1_CheckSupportedCpu Parameters in FSBL

Parameters	Description
CpuId	Checks if the processor is A53_2 or A53_3 or not.

XFsb1_AdmaCopy

This function copies data memory to memory using ADMA. You must take care of cache invalidation and flushing. ADMA also should be configured to simple DMA before calling this function.

```
u32 XFsb1_AdmaCopy(void * DestPtr, void * SrcPtr, u32 Size);
```

Table 27: XFsb1_AdmaCopy Parameters in FSBL

Parameters	Description
DestPtr	Pointer to the destination buffer to which data needs to be copied
SrcPtr	Pointer to the source buffer from which data needs to be copied
Size	Number of bytes of data that needs to be copied

XFsb1_GetDrvNumSD

This function is used to obtain drive number based on design and boot mode.

```
u32 XFsb1_GetDrvNumSD(u32 DeviceFlags);
```

Table 28: XFsbl_GetDrvNumSD Parameters in FSBL

Parameters	Description
Device flags	Contains the boot mode information, that is, one of SD0, SD1, eMMC, or SD1-LS boot modes

XFsbI_MakeSdFileName

This function returns the file name of the boot image. The name is deduced from the parameters.

```
void XFsbl_MakeSdFileName(char*XFsbI_SdEmmcFileName, u32 MultiBootReg, u32
DrvNum);
```

Table 29: XFsbl_MakeSdFileName Parameters in FSBL

Parameters	Description
XFsbI_SdEmmcFileName	Contains the final file name
Multiboot reg	The value of the MultiBoot register gets appended to the file name, if its value is non zero
DrvNum	Differentiates between SD0 and SD1 logical drives

Hooks in FSBL

Hooks are the functions that can be defined by you. FSBL provides blank functions and executes them from certain strategic locations. The following table shows the currently available hooks.

Table 30: Hooks in FSBL

Hook Purpose/Location	Hook Function Name
Before PL bitstream loading	XFsbI_HookBeforeBSDownload()
After PL bitstream loading	XFsbI_HookAfterBSDownload()
Before (the first) Handoff (to any application)	XFsbI_HookBeforeHandoff()
Before fallback	XFsbI_HookBeforeFallback()
To add more initialization code, in addition to that in psu_init or to replace psu_init with custom initialization	XFsbI_HookPsuInit()

See [FSBL wiki page](#) for more information on FSBL.

Security Features

This chapter details the Zynq[®] UltraScale+™ MPSoC features that you can leverage to address security during boot time and run time of an application. The Secure Boot mechanism is described in detail in this link to the Security chapter of the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)).

The system protection unit (SPU) provides the following hardware features for run-time security of an application running on Zynq UltraScale+ MPSoCs:

- [Xilinx Memory Protection Unit](#)
- [Xilinx Peripheral Protection Unit](#)
- [System Memory Management Unit](#)
- [A53 Memory Management Unit](#)
- [R5 Memory Protection Unit](#)

One of the runtime security features is access controls on the PMU and CSU global registers from Linux. These registers are classified into two lists: The white list (accessible all the time by default) and the black list (accessible only when a compile time flag is set). For more details, see CSU/PMU Register Access.

Boot Time Security

This section details the various boot image formats for authentication and encryption.

Encryption

Zynq UltraScale+ MPSoCs has AES-GCM hardware engine that supports confidentiality of your boot images, and can also be used by you post-boot to encrypt and decrypt your data.

The AES crypto engine has access to a diverse set of key sources. For more information on the key sources, see *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)).

The red key is used to encrypt the image. During the generation of the Boot file (`BOOT.bin`), the red key, and the initialization vector (IV) must be provided to the Bootgen tool `in.nky` file format.

PMU firmware can be loaded by CSU bootROM or FSBL. The CSUROM treats the FSBL and PMU firmware as separate partitions and hence, decrypts each of them individually. If both the FSBL and PMU firmware are encrypted, the AES Key/IV will be reused, which is a violation of the standard.



IMPORTANT! *If both the FSBL and PMU firmware are encrypted, the PMU firmware must be loaded by the FSBL (and not the CSU bootROM) to avoid reusing the AES Key/IV pair. For more information, see [Xilinx Answer 70622](#).*

The following BIF file is for encrypted image, where PMU firmware is loaded by FSBL:

```
the_ROM_image:
{
[aeskeyfile] bbram.nky [keysrc_encryption] bbram_red_key
[bootloader, encryption=aes, destination_cpu=a53-0] ZynqMP_Fsbl.elf
[destination_cpu = pmu, encryption=aes] pmufw.elf
}
```

BIF File with BBRAM Red Key

The following BIF file sample shows the red key stored in BBRAM:

```
the_ROM_image: { [aeskeyfile]      bbram.nky
[keysrc_encryption] bbram_red_key
[bootloader, encryption=aes, destination_cpu=a53-0]      ZynqMP_Fsbl.elf
[destination_cpu = a53-0, encryption=aes] App_A53_0.elf
}
```

BIF File with eFUSE Red Key

The following BIF file sample shows the red key stored in eFUSE.

```
the_ROM_image: { [aeskeyfile]      efuse.nky
[keysrc_encryption] efuse_red_key
[bootloader, encryption=aes, destination_cpu=a53-0]      fsbl.elf
[destination_cpu = a53-0, encryption=aes] App_A53_0.elf
}
```

BIF File with an Operational Key

For creating a boot image using Bootgen with an operational key, you must provide the tool with the operational key, along with the red key and IV in an `.nky` file. Bootgen places this operational key in a header and encrypts it with the device red key. The result is what is called an encrypted secure header. The main advantage of this is that it minimizes the use of the device key, thus limiting its exposure. For more details, refer to “Minimizing Use of the AES Boot Key (OP Key Option)” in the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)).

```
the_ROM_image :
{
[aeskeyfile]      bbram.nky [fsbl_config] opt_key [keysrc_encryption]
bbram_red_key
[bootloader, encryption=aes, destination_cpu=a53-0]      ZynqMP_Fsbl.elf
[destination_cpu = a53-0, encryption=aes] App_A53_0.elf
}
```

Using Op Key to Protect the Device Key in a Development Environment

The following steps provide a solution in a scenario where two development teams Team-A (secure team), which manages the secret red key and Team-B (not so secure team) work collaboratively to build an encrypted image without sharing the secret red key. Team-A manages the secret red key. Team-B builds encrypted images for development and test. However, it does not have access to the secret red key.

Team-A encrypts the boot loader with the device key (using the Op Key option) and delivers the encrypted boot loader to Team-B. Team-B encrypts all the other partitions using the Op Key.

Team-B takes the encrypted partitions that they created and the encrypted boot loader they received from the Team-A and uses Bootgen to ‘stitch’ everything together into a single `boot.bin`.

The following procedures describe the steps to build an image:

Procedure 1

In the initial step, Team-A encrypts the boot loader with the device Key using the `opt_key` option, delivers the encrypted boot loader to Team-B. Now, Team-B can create the complete image at a go with all the partitions and the encrypted boot loader using the operational key as device key.

1. Encrypt boot loader with device key:

```
bootgen -arch zynqmp -image stage1.bif -o fsbl_e.bin -w on -log error
```

Example `stage1.bif`:

```
stage1:
{
[aeskeyfile] aes.nky
[fsbl_config] opt_key
[keysrc_encryption] bbram_red_key
[bootloader,destination_cpu=a53-0,encryption=aes]fsbl.elf
}
```

Example `aes.nky` for stage1:

```
Device xc7z020clg484;
Key 0 AD00C023E238AC9039EA984D49AA8C819456A98C124AE890ACEF002100128932;
IV 0 F7F8FDE08674A28DC6ED8E37;
Key Opt 229C993D1310DD27B6713749B6D07FCF8D3DCA01EC9C64778CBAF457D613508F;
```

2. Attach the encrypted boot loader and rest of the partitions with the operational key as device key to form a complete image:

```
bootgen -arch zynqmp -image stage2a.bif -o final.bin -w on -log error
```

Example of `stage2.bif`:

```
stage2:
{
[aeskeyfile] aes-opt.nky
[bootimage]fsbl_e.bin
[destination_cpu=a53-0,encryption=aes]hello.elf
[destination_cpu=a53-1,encryption=aes]hello1.elf
}
```

Example `aes-opt.nky` for stage2:

```
Device xc7z020clg484;
Key 0 229C993D1310DD27B6713749B6D07FCF8D3DCA01EC9C64778CBAF457D613508F;
IV 0 F7F8FDE08674A28DC6ED8E37;
```

Procedure 2

In the initial step, Team-A encrypts the boot loader with the device key using the `opt_key` option and delivers the encrypted boot loader to Team-B. Now, Team-B can create encrypted images for each partition independently, using the operational key as the device key. Finally, Team-B can use Bootgen to stitch all the encrypted partitions and the encrypted boot loader, to get the complete image.

1. Encrypt boot loader with device key:

```
bootgen -arch zynqmp -image stage1.bif -o fsbl_e.bin -w on -log error
```

Example stage1.bif:

```
stage1:
{
[aeskeyfile] aes.nky
[fsbl_config] opt_key
[keysrc_encryption] bbram_red_key
[bootloader,destination_cpu=a53-0,encryption=aes]fsbl.elf
}
```

Example aes.nky for stage1:

```
Device xc7z020clg484;
Key 0 AD00C023E238AC9039EA984D49AA8C819456A98C124AE890ACEF002100128932;
IV 0 F7F8FDE08674A28DC6ED8E37;
Key Opt 229C993D1310DD27B6713749B6D07FCF8D3DCA01EC9C64778CBAF457D613508F;
```

2. Encrypt the rest of the partitions with operational key as device key:

```
bootgen -arch zynqmp -image stage2a.bif -o hello_e.bin -w on -log error
```

Example of stage2a.bif:

```
stage2a:
{
[aeskeyfile] aes-opt.nky
[destination_cpu=a53-0,encryption=aes]hello.elf
}
bootgen -arch zynqmp -image stage2b.bif -o hello1_e.bin -w on -log error
```

Example of stage2b.bif:

```
stage2b:
{
[aeskeyfile] aes-opt.nky
[destination_cpu=a53-1,encryption=aes]hello1.elf
}
```

Example of aes-opt.nky for stage2a and stage2b:

```
Device xc7z020clg484;
Key 0 229C993D1310DD27B6713749B6D07FCF8D3DCA01EC9C64778CBAF457D613508F;
IV 0 F7F8FDE08674A28DC6ED8E37;
```

3. Use Bootgen to stitch the above to form a complete image:

Example of stage3.bif:

```
stage3:
{
[bootimage]fsbl_e.bin [bootimage]hello_e.bin [bootimage]hello1_e.bin
}
```

Note: Key Opt of aes.nky is same as Key 0 in aes-opt.nky and IV 0 must be same in both nky files.

BIF File for Black Key Stored in eFUSE

For customers who would like to have the device key stored encrypted when not in use, the physical unclonable function (PUF) can be used. Here, the actual red key is encrypted with the PUF key encryption key (KEK), which is an encryption key that is generated by the PUF. The device will decrypt the black key to get the actual red key, so you need to provide the required inputs to Bootgen. The black key can be stored in either eFUSE or the Boot Header. Shutter value indicates the time for which the oscillator values can be captured for PUF. This value must always be 0x100005E.

For more details, refer to “Storing Keys in Encrypted Form (Black)” in the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)).

The following example shows storage of the black key in eFUSE.

```
the_ROM_image :
{
[pskfile]PSK.pem
[sskfile]SSK.pem
[aeskeyfile]red.nky
[keysrc_encryption] efuse_blk_key
[fsbl_config] shutter=0x0100005E
[auth_params] ppk_select=0
[bootloader, encryption = aes, authentication = rsa,
destination_cpu=a53-0]fsbl.elf
[bh_key_iv] black_key_iv.txt
}
```

BIF File for Black Key Stored in Boot Header

The following BIF file sample shows boot header black key encryption:

```
the_ROM_image :
{
[aeskeyfile] redkey.nky
[keysrc_encryption] bh_blk_key
[bh_keyfile] blackkey.txt
[bh_key_iv] black_key_iv.txt
[fsbl_config] pufhd_bh , puf4kmode , shutter=0x0100005E, bh_auth_enable
[pskfile] PSK.pem
[sskfile] SSK.pem
[bootloader, authentication=rsa , encryption=aes,
destination_cpu=a53-0]fsbl.elf
[puf_file]hlprdata4k.txt
}
```

Note: Authentication of boot image is compulsory for using black key encryption.

To generate or program eFUSE with black key, see Zynq eFUSE PS API in Appendix I, XiISKey Library v6.8.

BIF File for Obfuscated Form (Gray) Key Stored in eFUSE

If you would like to have the device key store in obfuscated form, you can encrypt the actual red key with the family key which is an encryption key. Device will decrypt the obfuscated key to get the actual red key. Hence, you need to provide the required inputs to Bootgen. The obfuscated key can be stored in either eFUSE or the Boot Header.

For more details, see *Storing Keys in Obfuscated Form (Gray)* section in the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*.

Note: The Family key is the same for all devices within a given ZynqMP SoC family. This solution allows you to distribute the Obfuscated key to contract manufacturer's without disclosing the actual user key.

The following example shows storage of the obfuscated key in eFUSE:

```
the_ROM_image :
{
[aeskeyfile]      red.nky
[keysrc_encryption] efuse_gry_key
[bh_key_iv] bhkeyiv.txt
[bootloader, encryption=aes, destination_cpu=a53-0]      fsbl.elf
}
```

The following example shows storage of the obfuscated form (gray) key in boot header:

```
the_ROM_image :
{
[aeskeyfile]      red.nky [keysrc_encryption] bh_gry_key [bh_key_iv]
bhkeyiv.txt
[bh_keyfile]      bhkey.txt
[bootloader, encryption=aes, destination_cpu=a53-0]      fsbl.elf
}
```

To Generate Obfuscated Key with Family Key:

Use Xilinx tools (Bootgen) to create the Obfuscated key. However, the family key is not distributed with the Xilinx development tools. It is provided separately. The family key received from Xilinx should be provided in the bif as shown in the example below.



IMPORTANT! To receive the family key, please contact secure.solutions@xilinx.com.

Sample bif to generate Obfuscated key:

```
all :
{
[aeskeyfile] aes.nky
[familykey] familyKey.cfg
[bh_key_iv] bhiv.txt
}
```

Using Bootgen to Generate Keys

If you are using Bootgen to create keys, NIST approved KDF is used, which is Counter Mode KDF with CMAC as the PRF.

With a Single Key/IV pair:

- If seed is specified - Key Generation is based on Seed.
- If seed is NOT specified - Key Generation is based on Key0.

If an empty file is mentioned, Bootgen generates a seed with time based randomization. This is not a standard like the KDF. This seed will in turn be the input for KDF to generate the Key/IV pairs.

BIF File with Multiple AESKEY Files

The following BIF file samples show the encryptions using aeskey files:

One AES Key / Partition

You may specify multiple nky files, one for each partition in the image. The partitions are encrypted using the key that is specified before the partition.

```
sample_bif:
{
[aeskeyfile] test1.nky
[bootloader, encryption=aes] fsb1.elf
[aeskeyfile] test2.nky
[encryption=aes] hello.elf
[aeskeyfile] test3.nky
[encryption=aes] app.elf
}
```

The `fsb1.elf` partition is encrypted using the keys from `test1.nky` file. If we assume that the `hello.elf` file has two partitions since it has two loadable sections, then both the partitions are encrypted using keys from `test2.nky` file. The `app.elf` partition is encrypted using keys from `test3.nky` file.

One AES Key / Each Partition (Multiple Loadable Sections Scenario)

You may specify multiple `nky` files, one for each partition in the image. The partitions are encrypted using the key that is specified before the partition. You are allowed to have unique key files for each of the partition created due to multiple loadable sections by having key file names appended with `'.1', '.2'...'n'` in the same directory of the key file meant for that partition.

```
sample_bif:
{
[aeskeyfile] test1.nky
[bootloader, encryption=aes] fsbl.elf
[aeskeyfile] test2.nky
[encryption=aes] hello.elf
[aeskeyfile] test3.nky
[encryption=aes] app.elf
}
```

The `fsbl.elf` partition is encrypted using the keys from `test1.nky` file. Assume that the `hello.elf` file has three partitions since it has three loadable sections, and `hello.elf.0` is encrypted using the keys from `test2.nky` file, `hello.elf.1` is encrypted using the keys from `test2.1.nky`, and `hello.elf.2` is encrypted using the keys from `test2.2.nky` file. The `app.elf` partition is encrypted using keys from `test3.nky` file.

Using the same `.nky` across multiple partitions, re uses the AES Key and AES Key/IV Pair in each partition. Using the AES key across multiple partitions increases the exposure of the key and may be a security vulnerability. Using the same AES Key/IV Pair across multiple partitions is a violation of the standard. To avoid the re-use of AES Key/IV pair, Bootgen increments the IV with the partition number. To avoid the re-use of both AES Key and AES Key/IV pair, Bootgen allows you to provide multiple `.nky` files, one for each partition.



IMPORTANT! *To avoid key re-use, support for single nky file across multiple partitions will be deprecated.*



CAUTION! *Using a single .nky file with multiple partitions means that the same key is being used in each partition - which can be a security vulnerability. A warning is issued in the current release with the plan to generate an error in future releases.*

Note: Key0/IV0 - should be the same in all the `nky` files.

If you specify multiple keys and if the number of keys are less than the number of blocks to be encrypted, it is ERRORED OUT.

If you need to specify multiple Key/IV pairs, you must specify (number-of-blocks+1) pairs. The extra Key/IV pair is for SH. Ex: If blocks=4;8;16 - you have to specify 4+1=5 Key/IV pairs.

Authentication

The SHA hardware accelerator included in the Zynq UltraScale+ MPSoC implements the SHA-3 algorithm and produces a 384-bit digest. It is used together with the RSA accelerator to provide image authentication and the AES-GCM is used to decrypt the image. These blocks (SHA-3/384, RSA and AES-GCM) are hardened and part of crypto interface block (CIB).

Authentication flow treats the FSBL as raw data, where it makes no difference whether the image is encrypted or not. There are two level of keys: primary key (PK) and secondary Key (SK).

Each key has two complementary parts: secret key and public key:

- PK contains primary public key (PPK) and primary secret key (PSK).
- SK contains secondary public key (SPK) and secondary secret key (SSK).

The hardened RSA block in the CIB is a Montgomery multiplier for acceleration of the big math required for RSA. The hardware accelerator can be used for signature generation or verification. The ROM code only supports signature verification. Secret keys are only used in the signature generation stage when the certificate is generated.



IMPORTANT! *Signature generation is not done on the device, but in software during preparation of the boot image.*

To better understand the format of the authentication certificate, see *Bootgen User Guide (UG1283)*.

The PPK and SPK keys authenticate a partition. PSK and SSK are used to sign the partition. The equations for each signature (SPK, boot header, and boot image) are listed here:

- SPK signature. The 512 bytes of the SPK signature is generated by the following calculation:

```
SPK signature = RSA(PSK, padding || SHA(SPK+ auth_header)).
```

- Boot header signature. The 512 bytes of the boot header signature is generated by the following calculation:

```
Boot header signature = RSA(SSK, padding || SHA(boot_header)).
```

- Boot image signature. The 512 bytes of the boot image signature is generated by the following calculation:

```
BI signature = RSA(SSK, padding || SHA(PFW + FSBL + authentication certificate)).
```

Note: For SHA-3 authentication, always use Keccak SHA3 to calculate hash on boot header, PPK hash and boot image. NIST-SHA3 is used for all other partitions which are not loaded by ROM.

Bootgen supports RSA signature generation only. The modulus, exponentiation and precalculated $R^2 \text{ Mod } N$ are required. Software is supported only for RSA public key encryption, for encrypting the signature RSA engine requires modulus, exponentiation and pre-calculated $R^2 \text{ Mod } N$, all these are extracted from keys.

BIF File with SHA-3 Boot Header Authentication and PPK0

The following BIF file sample supports the BH RSA option. This option supports integration and test prior to the system being fielded. For more details, see “Integration and Test Support (BH RSA Option)” in the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)).

The BIF file is for SHA-3 boot header authentication, where actual PPK hash is not compared with the eFUSE stored value.

```
the_ROM_image: {
[fsbl_config] bh_auth_enable
[auth_params] ppk_select=0; spk_id=0x00000000
[pskfile] primary_4096.pem
[sskfile] secondary_4096.pem
[bootloader, authentication=rsa, destination_cpu=a53-0] fsbl.elf
[pmufw_image, authentication=rsa] xpfw.elf
}
```

BIF File with SHA-3 eFUSE RSA Authentication and PPK0

The following BIF file sample shows eFUSE RSA authentication using PPK0 and SHA-3.

```
the_ROM_image:
{
[auth_params]ppk_select=0;spk_id=0x584C4E58
[pskfile]psk.pem
[sskfile]ssk.pem
[bootloader, authentication = rsa, destination_cpu=a53-0]zynqmp_fsbl.elf
[destination_cpu = a53-0, authentication = rsa]Application.elf
}
```

Enhanced RSA Key Revocation Support

The RSA key provides the ability to revoke the secondary keys of one partition without revoking them for all partitions.

Note: Primary key should be the same across all partitions.

This is achieved by using USER_FUSE0 to USER_FUSE7 eFuses (one can revoke up to 256 keys, if all are not required for their usage) with the new BIF parameter `spk_select`.

The following BIF file sample shows enhanced user fuse revocation:

Image header and FSBL uses different SSK's for authentication (`ssk1.pem` and `ssk2.pem` respectively) with the following bif input.

```
the_ROM_image:    {
[auth_params]ppk_select = 0
[pskfile]psk.pem
[sskfile]ssk1.pem
[bootloader, authentication = rsa, spk_select = spk-efuse, spk_id =
0x12345678, sskfile = ssk2.pem]zynqmp_fsbl.elf
[destination_cpu =a53-0, authentication = rsa, spk_select = user-efuse,
spk_id = 0x3, sskfile = ssk3.pem]Application1.elf
[destination_cpu =a53-0, authentication = rsa, spk_select = spk-efuse,
spk_id = 0x12345678, sskfile = ssk4.pem]Application2.elf
}
```

Same SSK will be used for both Image header and FSBL (`ssk2.pem`), if separate SSK is not mentioned.

```
the_ROM_image:    {
{
[auth_params]ppk_select = 0 [pskfile]psk.pem
[bootloader, authentication = rsa, spk_select = spk-efuse, spk_id =
0x12345678, sskfile = ssk2.pem]zynqmp_fsbl.elf
[destination_cpu =a53-0, authentication = rsa, spk_select = user-efuse,
spk_id = 0xF, sskfile = ssk3.pem]Application1.elf
[destination_cpu =a53-0, authentication = rsa, spk_select = spk-efuse,
spk_id = 0x12345678, sskfile = ssk4.pem]Application2.elf
}
```

`spk_select = spk-efuse` indicates that `spk_id` eFuse will be used for that partition.

`spk_select = user-efuse` indicates that user eFuse will be used for that partition.

Partitions loaded by CSU ROM will always use `spk_efuse`.

Note: The `spk_id` eFuse specifies which key is valid. Hence, the ROM checks the entire field of `spk_id` eFuse against the SPK ID to make sure it is a bit for bit match.

Valid range of `spk_id` for `spk_select` user-efuse is 0x1 to 0x100 (in decimal 1 to 256). The user eFuse specifies which key ID is not valid (has been revoked). Hence, the firmware (non-ROM) checks to see if a given user eFuse that represents the SPK ID has been programmed.

Bitstream Authentication Using External Memory

Authentication of bitstream is different from all other partitions. The FSBL can be wholly contained within the OCM, and therefore authenticated and decrypted inside of the device. For the bitstream, the size of the file is so large that it cannot be wholly contained inside the device and external memory must be used. The use of external memory creates a challenge to maintain security because an adversary may have access to this external memory.

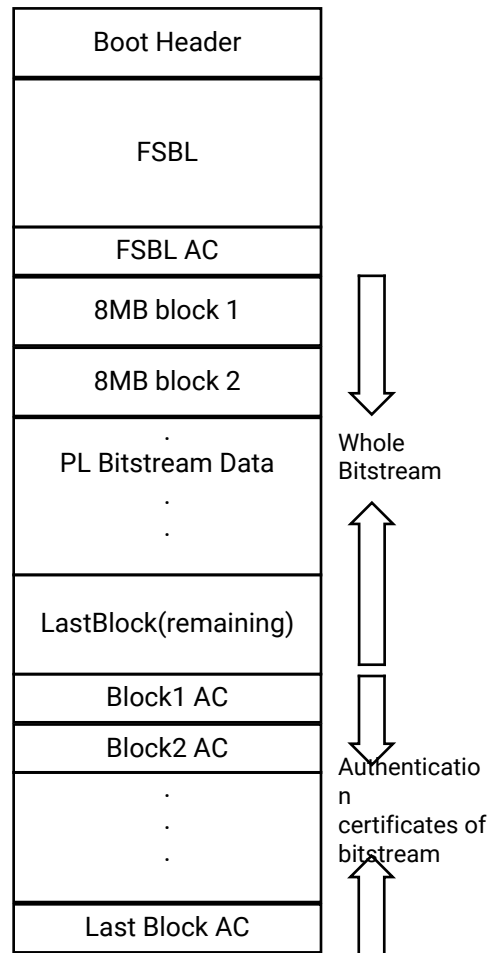
The following section describes how the bitstream is authenticated securely using external memory.

Bootgen

When bitstream is requested for authentication, Bootgen divides the whole bitstream into 8 MB blocks and has an authentication certificate for each block.

If a bitstream is not in multiples of 8 MB, the last block contains the remaining bitstream data.

Figure 37: **Bitstream Blocks**



X19220-071317

When authentication and encryption are both enabled, encryption is first done on the bitstream. Then Bootgen divides the encrypted data into blocks and places an authentication certificate for each block.

Software

To securely authenticate the bitstream partition, FSBL uses the ATF section's OCM memory to copy the bitstream in chunks from FLASH or DDR. Therefore, while creating a boot image, the bitstream partition should be before ATF partition. Otherwise ATF memory is

over-written while processing the bitstream partition.

The workflow for the DDR and DDR-less systems is nearly identical. The only difference is that for systems with the DDR, FSBL copies the entire bitstream partition (bitstream and authentication certificates) to the DDR from the FLASH devices, because DDR is faster to access. FSBL then, each time, copies a chunk of bitstream from the DDR. For the DDR-less systems, FSBL copies a chunk of bitstream directly from the FLASH devices.

The following is the software workflow for authenticating the bitstream:

1. FSBL identifies the availability of the DDR on the system based on the XFSBL_PS_DDR macro. FSBL has two buffers in OCM, ReadBuffer buffer of size 56 KB and HashsOfChunks[] to store intermediate hashes calculated for each 56 KB of 8 MB blocks.
2. FSBL copies a 56 KB chunk from the first 8 MB block to ReadBuffer.
3. FSBL calculates hash on 56 KB and stores in HashsOfChunks.
4. FSBL repeats the previous steps until the entire 8 MB of block is completed.
Note: 56 KB is taken for performance; it can be of any size.
5. FSBL authenticates the bitstream.
6. Once the authentication is successful, FSBL starts copying 56 KB starting from the first block which is located in DDR/FLASH to ReadBuffer, calculates the hash, and then compares it with the hash stored at HashsOfChunks.
7. If hash comparison is successful, FSBL transmits data to PCAP via DMA (for unencrypted bitstream) or AES (if encryption is enabled).
8. FSBL repeats the previous two steps until the entire 8 MB block is completed.
9. Repeats the entire process for all the blocks of bitstream.

Note: If there is any failure at any stage, PL is reset and FSBL is exited.

The bitstream is directly routed to PCAP via CSU DMA by configuring secure stream switch.

For a DDR system, the whole encrypted bitstream is copied to DDR. For DDR-less system, decryption is copied to OCM(ATF section) in chunks.

Note: Xilinx recommends that you have a bitstream partition immediately after the FSBL partition in the boot image.

Run-Time Security

Run-time security involves protecting the system against incorrectly programmed or malicious devices corrupting the system memory or causing a system failure.

To protect the system, it is important to secure memory and the peripherals during a software execution. The Zynq UltraScale+ MPSoCs provide memory and peripheral protection through the following blocks:

- [Arm Trusted Firmware](#)
- [Xilinx Memory Protection Unit](#)
- [Xilinx Peripheral Protection Unit](#)
- [System Memory Management Unit](#)
- [A53 Memory Management Unit](#)
- [R5 Memory Protection Unit](#)

One of the runtime security features is access controls on the PMU and CSU global registers from Linux. These registers are classified into two lists:

- The white list which is accessible all the time by default.
- The black list which is accessible only when a compile time flag is set.

Arm Trusted Firmware

The Zynq UltraScale+ MPSoC incorporates the standard execution model advocated for Armv8 cores. This model runs the normal operating system at a less privileged state, requiring it to request access to security-sensitive hardware or registers using a proxy software called as secure monitor code (SMC). The specific SMC provided by Xilinx for the Zynq UltraScale+ MPSoC device is a part of Linaro Arm Trusted Firmware (ATF). Xilinx neither requires nor provides a Trusted OS as secure boot functionality is available through the CSU and PMU as previously described. However, the ATF provided by Xilinx does include hooks which allow customers to add their own Trusted OS for incorporation of additional trusted applications. ATF includes a secure monitor for switching between the secure and the non-secure world.

The primary purpose of ATF is to ensure that the system modules (drivers, applications) do not have access to a resource unless absolutely necessary. For example, Linux should be prevented from accessing the region where the public key is stored in the SoC. Likewise, the driver for a crypto block does not need to know the current session key; the session key could be programmed by the key negotiation algorithm and stored in a secure location within the crypto block.

PSCI is the interface from non-secure software to firmware implementing power management use-cases (for example, secondary CPU boot, hotplug, and idle). It might be necessary for supervisory systems running at exception levels to perform actions, such as restoring context and switches to the power state of core. Non-secure software can access ATF runtime services using the Arm secure monitor call (SMC) instruction.

In the Arm architecture, synchronous control transfers between the non-secure state to a secure state through SMC exceptions, which are generated by the SMC instruction, and handled by the secure monitor. The operation of the secure monitor is determined by the parameters passed in through registers.

Two types of calls are defined:

- Fast calls to execute atomic secure operations
- Standard calls to start preemptive secure operations

Two calling conventions for the SMC instruction defines two function identifiers for the SMC instruction define two calling conventions:

- **SMC32:** A 32-bit interface that either 32-bit or 64-bit client code can use. SMC32 passes up to six 32-bit arguments.
- **SMC64:** A 64-bit interface used only by 64-bit client code that passes up to six 64-bit arguments.

You define the SMC function identifiers based upon the calling convention. When you define the SMC function identifier, you pass that identifier into every SMC call in register R0 or W0, which determines the following:

- Call type
- Calling convention
- Secure function to invoke

ATF implements a framework for configuring and managing interrupts generated in either security state. It implements a subset of the trusted board boot requirements (TBBR) and the platform design document (PDD) for Arm reference platforms.

The cold boot path is where the TBBR sequence starts when the platform is powered on, and runs up to the stage where it hands-off control to firmware running in the non-secure world in DRAM. The cold boot path starts when you physically turn on the platform.

- You chose one of the CPUs released from reset as the primary CPU, and the remaining CPUs are considered secondary CPUs.
- The primary CPU is chosen through platform-specific means. The cold boot path is mainly executed by the primary CPU, other than essential CPU initialization executed by all CPUs.
- The secondary CPUs are kept in a safe platform-specific state until the primary CPU has performed enough initialization to boot them.

For a warm boot, the CPU jumps to a platform-specific address in the same processor mode as it was when released from reset.

ATF Functions

The following table lists the ATF functions:

Table 31: ATF Functions

ATF Functions	Description
<code>bl31_arch_setup();</code>	Generic architectural setup from EL3.
<code>bl31_platform_setup();</code>	Platform setup in BL1.
<code>bl31_lib_init();</code>	Simple function to initialize all BL31 helper libraries.
<code>cm_init();</code>	Context management library initialization routine.
<code>dcs_w_op_all(DCCSW);</code>	Cleans caches before re-entering the non-secure software world.
<code>(*bl32_init)();</code>	Function pointer to initialize the BL32 image.
<code>runtime_svc_init();</code>	Calls the initialization routine in the descriptor exported by a runtime service. After a descriptor is validated, its start and end owning entity numbers and the call type are combined to form a unique oen. The unique oen is an index into the <code>rt_svc_descs_indices</code> array. This index stores the index of the runtime service descriptor.
<code>validate_rt_svc_desc();</code>	Simple routine to sanity check a runtime service descriptor before it is used.
<code>get_unique_oen();</code>	Gets a unique oen.
<code>bl31_prepare_next_image_entry();</code>	Programs EL3 registers and performs other setup to enable entry into the next image after BL31 at the next ERET.
<code>bl31_get_next_image_type();</code>	Returns the <code>next_image_type</code> .
<code>bl31_plat_get_next_image_ep_info (image_type);</code>	Returns a reference to the <code>entry_point_info</code> structure corresponding to the image that runs in the specified security state.
<code>get_security_state ()</code>	Gets the security state.
<code>cm_init_context()</code>	Initializes a <code>cpu_context</code> for the first use by the current CPU, and sets the initial entry point state as specified by the <code>entry_point_info</code> structure.
<code>get_scr_el3_from_routing_model()</code>	Returns the cached copy of the SCR_EL3 which contains the routing model (expressed through the IRQ and FIQ bits) for a security state that is stored through a previous call to <code>set_routing_model()</code> .
<code>cm_prepare_el3_exit()</code>	Prepares the CPU system registers for first entry into the secure or the non-secure software world. <ul style="list-style-type: none"> • If execution is requested to EL2 or hyp mode SCTLRL_EL2 is initialized. • If execution is requested to the non-secure EL1 or svc mode, and the CPU supports EL2; then EL2 is disabled by configuring all necessary EL2 registers. For all entries, the EL1 registers are initialized from the <code>cpu_context</code> .
<code>cm_get_context(security_state);</code>	Gets the context of the security state.
<code>el1_sysregs_context_restore</code>	Restores the context of the system registers.
<code>cm_set_next_context</code>	Programs the context used for exception return. This initializes the SP_EL3 to a pointer to a <code>cpu_context</code> set for the required security state.

Table 31: ATF Functions (cont'd)

ATF Functions	Description
bl31_register_bl32_init	Initializes the pointer to BL32 init function.
bl31_set_next_image_type	Accessor function to help runtime services determine which image to execute after BL31.

For more information about ATF, see [Arm Trusted Firmware documentation](#).

FPGA Manager Solution

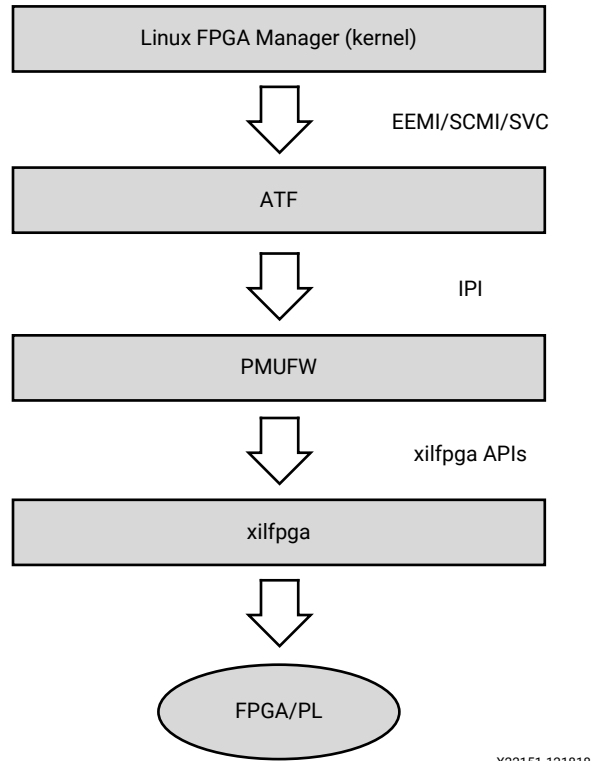
The FPGA Manager in the Zynq UltraScale+ MPSoC provides an interface to download different types of bitstreams (full, partial, authenticated, encrypted and so on) during runtime from Linux environment. The key features of the FPGA Manager are as follows:

- Full bitstream loading
- Partial Reconfiguration (partial bitstream loading)
- Encrypted full/partial bitstream loading
- Authenticated full/partial bitstream loading
- Authenticated and encrypted full/partial bitstream loading
- Readback of configuration registers
- Readback of bitstream (configuration data)

FPGA Manager Architecture

The following figure shows the architecture of the FPGA Manager.

Figure 38: FPGA Manager Architecture Block Diagram

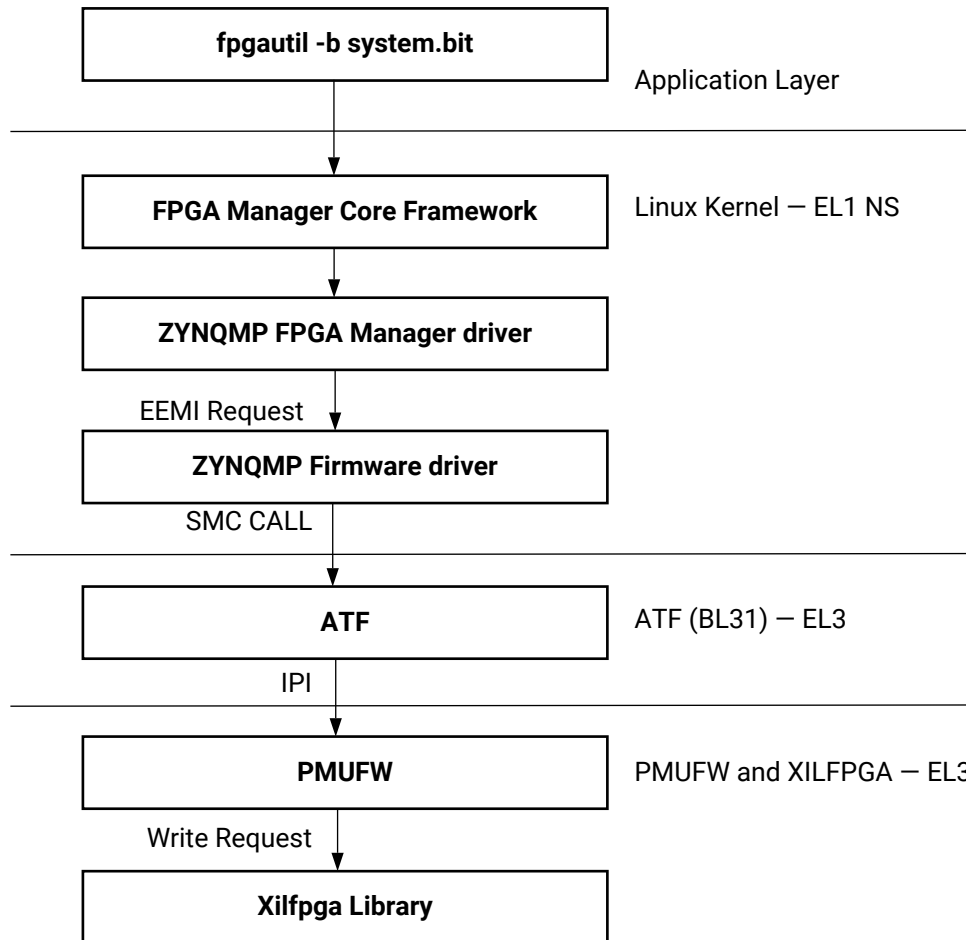


Execution Flow

FPGA manager provides an abstraction for the user to load bitstream using Linux. The the xilfpga library initializes the PCAP, CSUDMA and other hardware. For more details about xilfpga, see the Appendix K, XiIFPGA Library v5.0 section.

To load a bitstream, the FPGA manager allocates the required memory and invokes the EEMI API using the FPGA LOAD API ID. This request is a blocking call. The FPGA Manger waits for response from the ATF and response is provided to the fpga core layer which passes it to the application. This is described in the following figure:

Figure 39: FPGA Manager Flow



X22152-121818

Xilinx Memory Protection Unit

The Xilinx memory protection unit (XMPU) is a region-based memory protection unit. For more details, see “System Protection Unit” chapter in the *Zynq UltraScale+ Device Technical Reference Manual* (UG1085).

Protecting Memory with XMPU

To understand more about XMPU features and functionality, refer to “System Protection Unit” chapter in the *Zynq UltraScale+ Device Technical Reference Manual* (UG1085).

Configuring XMPU Registers

The XMPU is configurable either one-time or through trust-zone access from a secure master (PMU, APU TrustZone secure master, or RPU when configured as secure master). At boot time, XMPU can be configured and its configuration can be locked such that it can only be reconfigured at next power-on reset. If the configuration is not locked, then XMPU can be reconfigured any number of times by secure master accesses. If you choose to configure the XMPU dynamically, you must also consider many aspects including the idling of active devices and the AXI bus.

For more information on using the XMPU please see *Isolation Methods in Zynq UltraScale+ MPSoCs* ([XAPP1320](#)).

Xilinx Peripheral Protection Unit

To understand more about Xilinx peripheral protection unit (XPPU) features and functionality, see this [link](#) to the “Xilinx Peripheral Protection Unit” section of the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)).

For more information on using the XMPU please see *Isolation Methods in Zynq UltraScale+ MPSoCs* ([XAPP1320](#)).

System Memory Management Unit

The system memory management unit (SMMU) offers isolation services. The SMMU provides address translation for an I/O device to identify more than its actual addressing capability. In absence of memory isolation, I/O devices can corrupt system memory. The SMMU provides device isolation to prevent DMA attacks. To offer isolation and memory protection, it restricts device access for DMA-capable I/O to a pre-assigned physical space.

To understand more about SMMU features and functionality, see this [link](#) to the “System Memory Management Unit” section of the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)).

A53 Memory Management Unit

The memory management unit (MMU) controls table-walk hardware that accesses translation tables in main memory. The MMU translates virtual addresses to physical addresses. The MMU provides fine-grained memory system control through a set of virtual-to-physical address mappings and memory attributes held in page tables. These are loaded into the translation lookaside buffer (TLB) when a location is accessed.

To understand more about MMU features and functionality, see this [link](#) to the “Memory Management Unit” section of the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*.

R5 Memory Protection Unit

The memory protection unit (MPU) enables you to partition memory into regions and set individual protection attributes for each region. When the MPU is disabled, no access permission checks are performed, and memory attributes are assigned according to the default memory map. The MPU has a maximum of 16 regions.

To understand more about MPU features and functionality, see this [link](#) to the “Memory Protection Unit” section of the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*.

Platform Management

Zynq[®] UltraScale+[™] MPSoCs are designed for high performance and power-sensitive applications in a wide range of markets. The system power consumption depends on how intelligently software manages the various subsystems – turning them on and off only when they are needed and, also at a finer level, trading off performance for power. This chapter describes the features available to manage power consumption, and how to control the various power modes using software.

Platform Management in PS

To increase the scalability in the platform management unit (PMU), the Zynq UltraScale+ MPSoC supports multiple power domains such as:

- Full Power Domain
- Low Power Domain
- Battery Power Domain
- PL Power Domain

For details on the PMU and the optional PMU firmware (PMU firmware) functionality, see the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)).

For more information on dynamically changing the PS clocks, see [Chapter 14: Clock and Frequency Management](#).

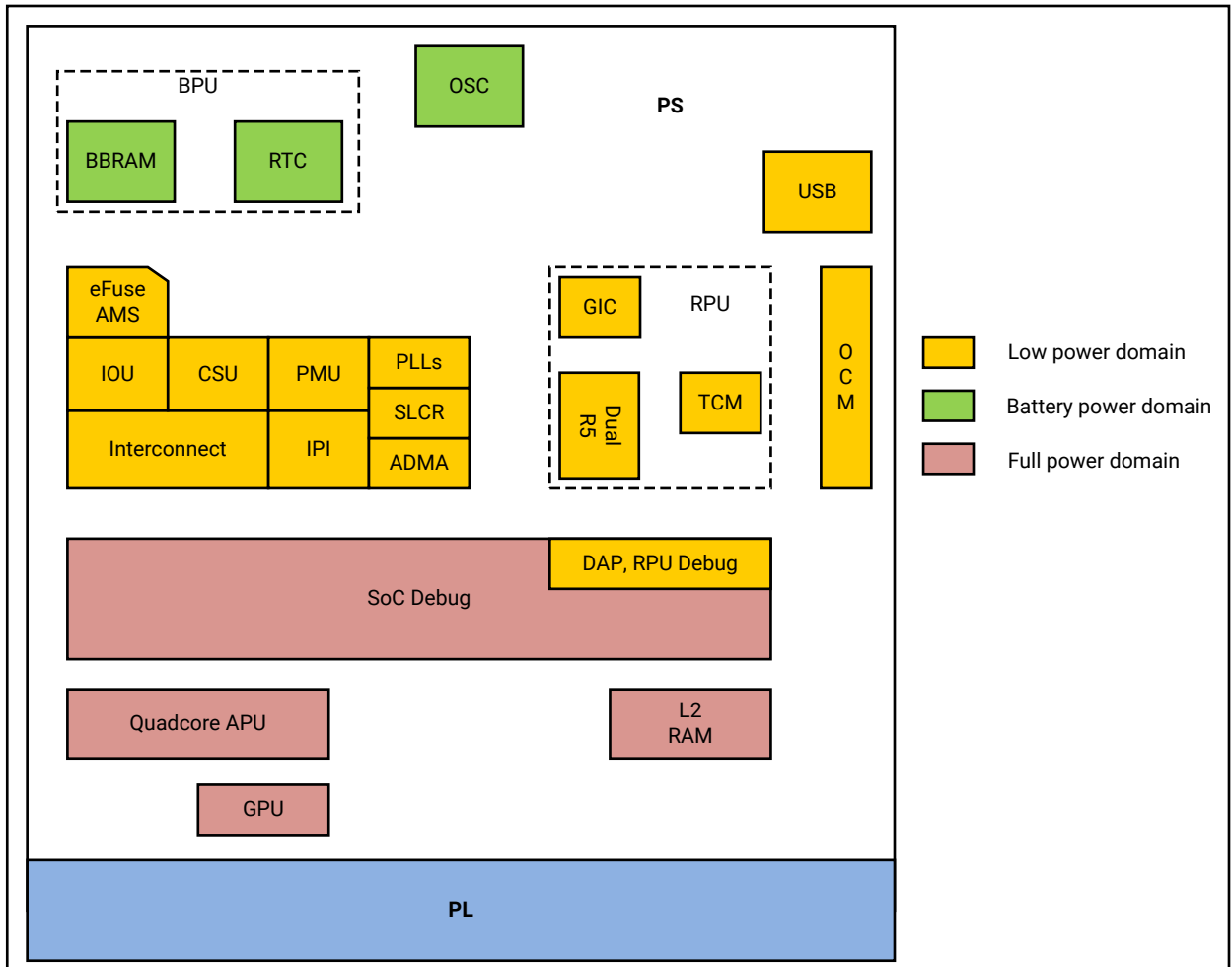
The PS block offers high levels of functionality and performance. At the same time, there is a strong need to optimize the power consumption of this block with respect to the functionality and performance that is necessary at each stage of the operation.

The Zynq UltraScale+ MPSoC has multiple power rails. Each rail can be turned off independently, or can use a different voltage. Many of the blocks on a specific power rail implement power-gating, which allows blocks to be gated off independently.

Examples of these power-gated domains are the: Arm[®] Cortex[™]-A53 and the Cortex[™]-R5F processors, GPU pixel processors (PP), large RAMs, and individual USBs.

The following figure shows a block diagram of the platform management at the PS level.

Figure 40: Platform Management at the PS Level



X19226-071317

From the power perspective, Zynq UltraScale+ MPSoCs offers the following modes of operation at the PS level:

- Full-power operation mode
- Low-power operation mode
- Deep-sleep mode
- Shutdown mode
- Battery-power mode

The following sections describe these modes.

Full-Power Operation Mode

In the full-power operation mode (shown as full power domain in the figure above), the entire system is up and running. Total power dissipation depends on the number of components that are running: their states and their frequencies. In this mode, dynamic power will likely dominate the total power dissipation.

To optimize static and dynamic power in full-power mode, all large modules have their own power islands to allow them to be shut down when they are not being used. To understand about full-power operation mode, see this [link](#) to the “Platform Management Unit” chapter in the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)).

Low-Power Operation Mode

In the low-power operation mode, a subset of the PS (shown as low-power domain in the figure above) is powered up that includes: the PMU, RPU, CSU, and the IOU.

In this mode, the ability to change system frequency allows power dissipation to be tuned. The CSU must be running continuously to monitor the system security against SEU and tampering. In this mode, the ability to change system frequency allows power dissipation to be tuned.

The low-power mode includes all lower-domain peripherals. Among the blocks within the low-power mode, PLLs, dual Cortex-R5F, USBs, and the TCM and OCM block RAMs offer power gating.

Note: SATA, PCIe®, and DisplayPort blocks are within the full power domain (FPD).

You can control power gating to different blocks through software by configuring the LPD_SLCR registers. See the [SLCR_Registers](#) link in the *Zynq UltraScale+ Device Register Reference* ([UG1087](#)) for more information on LPD_SLCR register.

Deep-Sleep Operation Mode

Deep-Sleep is a special mode in which the PS is suspended and waiting a wake-up signal. The wake can be triggered by the MIO, the USB, or the RTC.

Upon wake, the PS does not have to go through the boot process, and the security state of the system is preserved. The device consumes the lowest power during this mode while still maintaining its boot and security state.

In this mode, all the blocks outside the low-power domain, such as the system monitor and PLLs, are powered down. In LPD, Cortex-R5F is powered down. Because this mode has to preserve the context, TCM and OCM are in a retention state.

Shutdown Mode

Shutdown mode powers down the entire APU core. This mode is applicable to APU only. During shutdown, the entire processor state, including its caches, is completely lost; therefore, software is required to save all states before requesting the PMU to power down the APU core.

When a CPU is shutdown, it is expected that any interrupt from a peripheral that is associated with that CPU to initiate its power up; therefore, the interrupt lines to an APU core are also routed to the PMU interrupt controller, and are enabled when the APU core is powered down.

The *Embedded Energy Management Interface EEMI API Reference Guide* ([UG1200](#)) describe the APIs to invoke shutdown.

For more details, see this link to the “Platform Management Unit Programming Model” section in the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)).

Battery-Powered Mode

When the system is OFF, limited functionality within the PS must stay ON by operating on a battery. The following features operate within the battery-powered domain PS:

- Battery-backed RAM (BBRAM) to hold key for secure configuration
- Real-time clock (RTC) including the crystal I/O

The Zynq UltraScale+ MPSoC includes only one battery-powered domain and only the functions those are implemented in the PS can be battery backed-up. The required I/O for the battery-powered domain includes the battery power pads and the I/O pads for the RTC crystal.

Power Management Framework

The *Embedded Energy Management Interface EEMI API Reference Guide* ([UG1200](#)) describes how to use the power API functions.

Note: There is no difference between bare metal, FreeRTOS, or Linux-specific power management Xilinx EEMI API offerings.

Wake Up Mechanisms

To understand about wake up mechanisms, see this [link](#) to the “Platform Management Unit Operation” section of “Chapter 6, Platform Management Unit” of the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)).

Platform Management for Memory

The Zynq UltraScale+ MPSoCs include large RAMs like L2 cache, OCM, and TCM. These RAMs support various power management features such as: clock gating, power gating, and memory retention modes.

- TCM and OCM support independent power gating and retention modes.
- The L2 cache controller supports dynamic clock gating, retention, and shutdown modes to reduce power consumption at a finer granularity.

DDR Controller

The DDR controller implements the following mechanisms to reduce its power consumption:

- **Clock Stop:** When enabled, the DDR PHY can stop the clocks to the DRAM.
 - For DDR2 and DDR3, this feature is only effective in self-refresh mode.
 - For LPDDR2, this feature becomes effective during idle periods, power-down mode, self-refresh mode, and deep power-down mode.
- **Pre-Charge Power Down:** When enabled, the DDRC dynamically uses pre-charge power down mode to reduce power consumption during idle periods. Normal operation continues when a new request is received by the controller.
- **Self-Refresh:** The DDR controller can dynamically put the DRAM into self-refresh mode during idle periods. Normal operation continues when a new request is received by the controller.

In this mode, DRAM contents are maintained even when the DDRC core logic is fully powered down; this allows stopping the DDR3X clock and the DCI clock that controls the DDR termination.

Platform Management for Interconnects

The Interconnect lays across multiple power rails and power islands which can be on or off at different times. To ease the implementation, in most cases, the clocks for two power domains that communicate with one another must be asynchronous; consequently, requiring synchronizers on their interconnection.

To ease timing, the power domain is placed exactly at the clock crossing. The synchronizer must be implemented as two separate pieces with each placed in one of the two domains that are connected through the synchronizer, creating a bridge.

The bridge consists of a slave interface and a master interface with each lying entirely within a single power and clock domain. The clock frequencies at the interfaces can vary independent of each other, and each half can be reset independent of the other half.

Level shifters or clamping, or both, must be implemented between the two halves of the bridge for multi-voltage implementation or power-off.

Also, the bridge keeps track of open transactions, as follows:

- When the bridge receives a power-down request from the PMU, it logs that request.
- All new transactions return an error while the previously open transactions are being processed as usual until the transaction counter becomes 0. At that point, the bridge acknowledges to the PMU that it is safe to shut down the master or slave connected to the bridge.
- The entire Interconnect shuts down only when all bridges within that interconnect are idle.

For more details, see this [link](#) to the “PMU Interconnect” sub-section in the “Platform Management Unit” chapter of the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*.

PMU Firmware

Every system configuration that is supported by Xilinx includes PMU firmware in addition to the functions of power-up and sleep management. The PMU can execute user programs that implement advanced system monitoring and power management algorithms. In this mode, an application or a real-time processor copies the power management program into the PMU internal RAM through an inbound LPD switch. The PMU executes software that implements the required reset, power management, system monitoring, and interrupt controls within all Xilinx supported system configurations.

For more details, see this [link](#) to the “Platform Management Unit Programming Model” section in “Chapter 6” of the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*.

You can use the Vitis software platform to create custom PMU firmware. It provides the source code for the PMU firmware template and the necessary library support. For details on how to create a Vitis project, see [Chapter 5: Software Development Flow](#).

Platform Management Unit Firmware

The Platform Management Unit (PMU) in Zynq[®] UltraScale+[™] MPSoCs is located within the Low-power sub-system. The PMU consists of a MicroBlaze[™] processor which loads executable code from 32 KB ROM and 128 KB RAM into flat memory space. The PMU controls the power-up, reset, and monitoring of resources within the system including inter-processor interrupts and power management registers. The ROM is preloaded with PMU bootROM (PBR) which performs pre-boot tasks and enters a service mode. PMU_FW must be loaded to provide advanced system functionality for each of the Xilinx[®] supported use-cases. This chapter explains the features and functionality of PMU firmware developed for Zynq UltraScale+ MPSoC.

Features

The following are the key features of PMU firmware:

- Provides modular functionality: PMU firmware is designed to be modular. It enables you to add a new functionality in the form of a module
- Provides easy customization of modules
- Easily configurable to include only the required functionality for a user application
- Support communication with other components in the system over IPI (Inter-Processor Interrupt)
- Run time configurability for EM module
- Support for various Power Management features

PMU Firmware Architecture

The following figure shows the architecture block diagram of PMU firmware. PMU firmware is designed to be modular and enables adding new functionality in the form of modules. Each functionally distinct feature is designed as a module so that the PMU firmware can be configured to include only the required functionality for a user application. This type of modular design allows easy addition of new features and optimizes memory footprint by disabling unused modules.

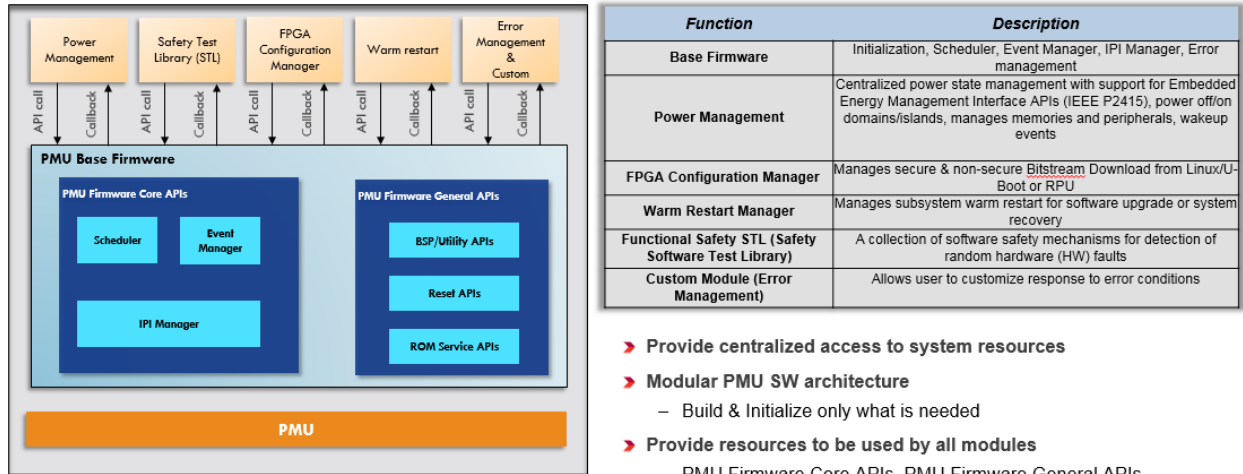
PMU firmware can be divided into base firmware and modules. PMU Base Firmware does initialization of modules, registering events for the modules, and provides all the common functions that may be required by the modules. These common functions can be categorized into the following APIs:

1. PMU firmware Core APIs
 - a. Scheduler
 - b. Event Manager
 - c. IPI Manager
2. PMU firmware General APIs
 - a. BSP/Utility APIs
 - b. Reset Services APIs
 - c. ROM Services APIs

These APIs can be used by the modules in PMU firmware to perform the specified actions as required.

Figure 41: PMU firmware Architecture Block diagram

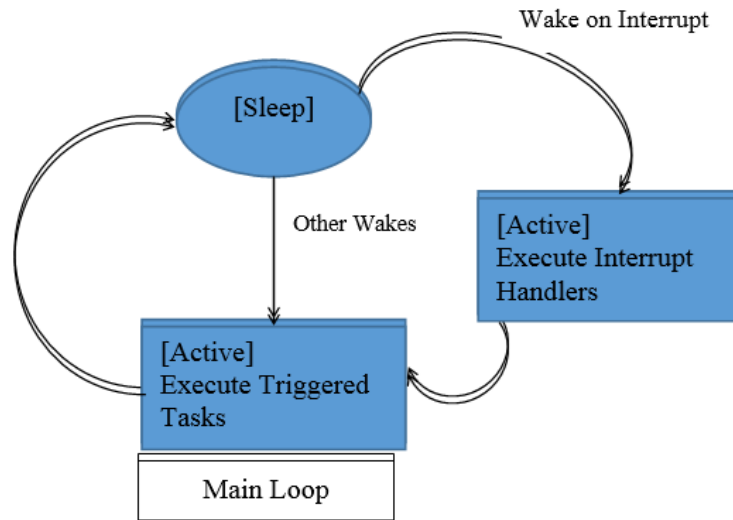
Platform Management Overview



Execution Flow

The initialization in PMU firmware takes place in a normal context. Interrupts are disabled to avoid unintended interruptions and prevent usage of the system resources before they are properly initialized. After initialization completes, interrupts are enabled and the required tasks are scheduled to be executed. The system enters in to a sleep state. The system wakes up only when an event occurs or the scheduled tasks are triggered and the corresponding handlers are executed. The following figure shows the state transitions for PMU firmware.

Figure 42: State Transitions for PMU firmware in Main Loop

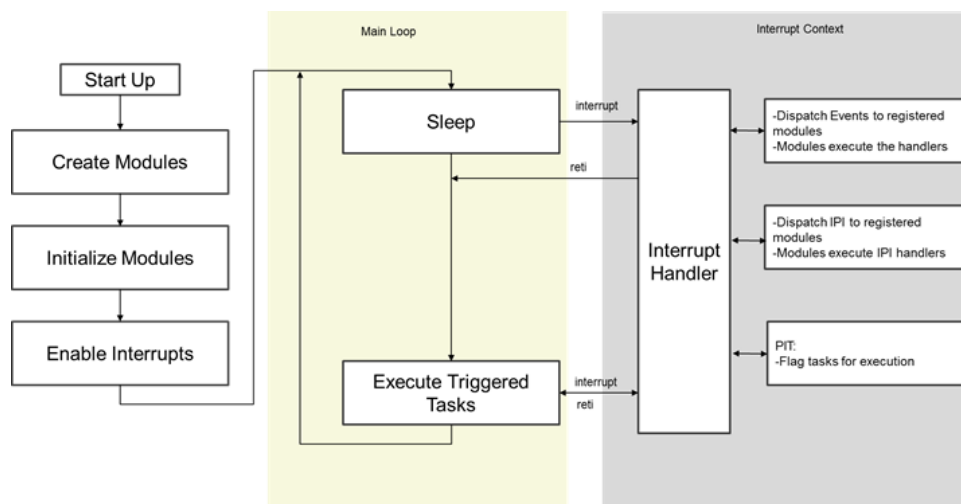


PMU firmware execution flow consists of the following three phases:

- **Initialization phase:** This phase consists of PMU firmware starting up, performing self-tests and validations, initializing the hardware, creating and initializing modules. Interrupts are disabled during this phase and are enabled at the end.
- **Post initialization:** In this phase, PMU firmware enters service mode, wherein it enters into sleep and waits for an interrupt.
- **Waking up:** PMU firmware enters the interrupt context and services the interrupt. After completing this task, it goes back to sleep.

The following figure shows the execution flow for PMU firmware.

Figure 43: Execution Context View for PMU firmware



Handling Inter-Process Interrupts in PMU firmware

IPI is a key interface between PMU firmware and non-PMU entities on the SoC. PMU includes four Inter-Processor Interrupts (IPI) assigned to it and one set of buffers. PMU firmware uses IPI-0 and associated buffers for communication by default, which is initiated by other masters on SoC to PMU. PMU firmware uses IPI-1 and associated buffers for callbacks from PMU to other masters and for communication initiated by PMU firmware.

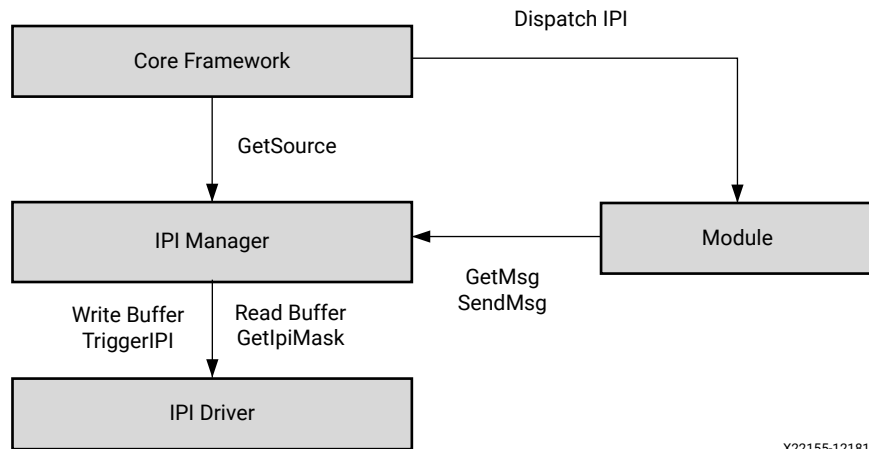
The following figure shows the IPI handling stack with interfaces between different components involved in this process. PMU firmware uses IPI driver to send and receive the messages. An IPI manager layer in Base Firmware is implemented over the driver and it takes care of dispatching the IPI message to the registered module handlers based on IPI ID in the first word of the message. The following table displays the message format for IPI.

Table 32: IPI Message Format

Word	Content	Description
0	Header	<target_module_id, api_id>
1	Payload	Module dependent payload
2		
3		
4		
5		
6	Reserved	Reserved - for future use
7	Checksum	

IPI-1 is used for the callbacks from PMU to other masters and for communication initiated by PMU firmware. Currently, PM and EM modules use IPIs and this can be taken as reference for implementing custom modules which require IPI messaging.

Figure 44: IPI Handler Stack with Interfaces



X22155-121818

PMU firmware provides wrapper APIs around IPI driver functions to send and receive IPI messages. During initialization, PMU firmware initializes the IPI driver and enables IPI interrupt from the masters which are IPI assigned.

Send IPI Message

`XPfw_IpiWriteMessage()` API is used to send IPI message to target. This function internally calls the IPI driver write API with buffer type as Message buffer.

Parameters

Table 33: Send IPI Message

Parameter	Description
ModPtr	Module pointer from where the IPI message is being sent. In IPI message, <code>target_module_id</code> field will be updated with the Module IPI ID information which is present in Module pointer.
DestCpuMask	Destination target IPI ID
MsgPtr	Message Pointer
MsgLen	Message Length

Return

`XST_SUCCESS`: If message is sent successfully.

`XST_FAILURE`: If message fails.

Send IPI Response

`XPfw_IpiWriteResponse()` API is used to send the response to the master which sent an IPI message. This function internally calls the IPI driver write API with buffer type as Response buffer.

Parameters

Table 34: Send IPI Response

Parameter	Description
ModPtr	Module pointer to check which module received this IPI response
SrcCpuMask	Source IPI ID to read IPI response
MsgPtr	Response Message Pointer
MsgLen	Response Message Length

Return

`XST_SUCCESS`: If IPI response is read successfully.

`XST_FAILURE`: If response fails.

Read IPI Message

`XPfw_IpiReadMessage()` is used to read the IPI message received when IPI interrupt comes. This function internally calls the IPI driver read API with buffer type as Message buffer.

Parameters

Table 35: Read IPI Message

Parameter	Description
SrcCpuMask	Source IPI ID to read the IPI message
MsgPtr	Message Pointer
MsgLen	Message Length

Return

`XST_SUCCESS`: If IPI message is read successfully.

`XST_FAILURE`: If message fails.

Read IPI Response

`XPfw_IpiReadResponse()` is used to read the IPI response for the message sent. This function internally calls the IPI driver read API with buffer type as response buffer.

Parameters

Table 36: Read IPI Response

Parameter	Description
ModPtr	Module pointer to check which module received this IPI response
SrcCpuMask	Source IPI ID to read IPI response
MsgPtr	Response Message Pointer
MsgLen	Response Message Length

Return

`XST_SUCCESS`: If IPI response is read successfully.

`XST_FAILURE`: If response fails.

Triggering an IPI

`XPfw_IpiTrigger()` is used to trigger an IPI to the destination. This function internally calls the IPI driver trigger. This function should be called after the IPI message writes IPI buffer.

Parameters

Table 37: Triggering an IPI

Parameter	Description
DestCpuMask	Destination target IPI ID

Return

`XST_SUCCESS`: If IPI is triggered successfully.

`XST_FAILURE`: If trigger fails.

Note: Vivado® allows you to enable or disable the IPI. To do so, select **MPSoC IP** → **Re-customize IP** → **Switch To Advanced Mode** → **Advanced Configuration** → **Inter Processor Interrupt (IPI) Configuration** → **IPI-Master Mapping**. However, it is not recommended that you disable IPI channels for APU or RPU for the PMU firmware PM module to work as expected because in the default configuration, PM assumes that both APU and RPU IPI channels are enabled.

PMU Firmware Modules

PMU firmware consists of the following modules:

1. Error Management (EM)
2. Power Management (PM)
3. Scheduler
4. Safety Test Library (STL)

PMU firmware has a module data structure (`XPfw_Module_t`) which contains the information about the module. This data structure is defined for each module when the module is created. The following table shows its members.

Table 38: Module Data Structure Members

Member	Range	Additional Information
ModId	0.. 31	
CfgInitHandler	Init handler function pointer	Default to NULL
IpiHandler	Handler for IPI manager	Default to NULL
EventHandler	Handler for registered events of the module	Default to NULL
IpiId	16-bit IPI ID	Unique to each module

PMU firmware also has a core data structure which contains the list and the details of all modules. The following table shows its members.

Table 39: Core Data Structure Members

Member	Range	Additional Information
ModList array	0.. 31	Module list array (of 32 elements) of Module structure
Scheduler	Scheduler structure	Scheduler task owned by the module
ModCount	0.. 31	
IsReady	Core is ready/dead	
Mode	Safety Diagnostics mode/Normal mode	

Base PMU firmware supports a few APIs that are used by these modules. Also, if you want to create a custom module, these APIs can be used from `xpfw_core.h`.

Creating a Module

`XPfw_CoreCreateMod()` API is called during the startup to create a module. PMU firmware can have maximum of 32 modules. This function checks if the module count reached the maximum count. If not, it fills in the details to core structure `ModList` and returns this module data structure to the caller. Otherwise, it returns `NULL`.

Setting up Handlers for the Module

Each module can be provided with three handlers which are called during the respective phases as described below:

Table 40: Module Handlers

Module Handler	Purpose	API for Registering the Handler	Execution context
Init	Called during the init of the core to configure the module, register for events or add scheduler tasks. This can be used to load the configuration data into the module if required.	<code>XPfw_CoreSetCfgHandler(const XPfw_Module_t *ModPtr, XPfwModCfgInitHandler_t CfgHandler);</code>	StartUp
Event Handler	Called when an event occurs (module should have registered for this event, preferably during the init phase)	<code>XPfw_CoreSetEventHandler(const XPfw_Module_t *ModPtr, XPfwModEventHandler_t EventHandler);</code>	Interrupt
IPI Handler	Called when an IPI message with respective module-id arrives	<code>XPfw_CoreSetIpiHandler(const XPfw_Module_t *ModPtr, XPfwModIpiHandler_t IpiHandler, u16 IpiId);</code>	Interrupt

PMU Firmware Build Flags

In PMU firmware, each module can be enabled/disabled based on your requirement. This is achieved by using build flags. The following table describes the important build flags in PMU firmware and its usage. Please see `xpfw_config.h` file in PMU firmware sources for a complete list of build flags.

Table 41: PMU Firmware Build Flags

Flag	Description	Prerequisites	Default Setting
<code>XPFW_DEBUG_DETAILED</code>	Enables detailed debug prints in PMU firmware. This feature is supported in 2017.3 release and above.		Disabled

Table 41: PMU Firmware Build Flags (cont'd)

Flag	Description	Prerequisites	Default Setting
PM_LOG_LEVEL	Enables print based debug functions for PM module. Possible values are: <ul style="list-style-type: none"> Alerts Errors Warnings Information Higher numbers include the debug scope of lower number, i.e. enabling 3 (warnings) also enables 1 (alerts) and 2 (errors).		Disabled
ENABLE_EM	Enables Error Management Module.	ENABLE_SCHEDULER	Disabled
ENABLE_ESCALATION	Enables escalation of sub-system restart to SRST/PS-Only if the first restart attempt fails.	ENABLE_RECOVERY, ENABLE_EM, ENABLE_SCHEDULER	Disabled
ENABLE_RECOVERY	Enables WDT based restart of APU sub-system.	ENABLE_EM, ENABLE_PM, ENABLE_SCHEDULER	Disabled
ENABLE_PM	Enables Power Management Module		Enabled
ENABLE_NODE_IDLING	Enables idling and reset of nodes before force shutdown of a sub-system.		Disabled
ENABLE_SCHEDULER	Enables Scheduler module		Enabled
ENABLE_WDT	Enables CSU WDT based restart of system used by PMU.	ENABLE_SCHEDULER, ENABLE_EM	Disabled
ENABLE_STL	Enables STL Module.	None	Disabled
ENABLE_RTC_TEST	Enables RTC event handler test module.	None	Disabled
ENABLE_SAFETY	Enables CRC calculation for IPI messages.	None	Disabled
ENABLE_FPGA_LOAD	Enables FPGA bit stream loading feature.	ENABLE PM	Enabled
ENABLE_SECURE	Enables security features.	ENABLE PM	Enabled
IDLE_PERIPHERALS	Enables idling peripherals before PS-only or System reset.	ENABLE PM	Disabled
ENABLE_POS	Enables Power Off Suspend feature.	ENABLE PM	Disabled
EFUSE_ACCESS	Enables efuse access feature.	ENABLE PM	Disabled
ENABLE_UNUSED_RPU_PWR_DWN	Powers down RPU(s) and slaves if they are not running after receiving PmInitFinalize.		Enabled
USE_DDR_FOR_APU_RESTART	Enables handling of APU restart gracefully by storing FSBL to DDR during boot and restoring it back to OCM before performing APU restart.	ENABLE_SECURE	Enabled

Error Management (EM) Module

Error Management Hardware

Zynq UltraScale+ MPSoC has a dedicated error handler to aggregate and handle fatal errors across the SoC. See the TRM/Arch Spec for more information.

All fatal errors routed to Error Manager can either set to be handled by HW (and trigger a SRST/PoR/PS error out) or trigger an interrupt to PMU.

Error Management in PMU firmware

Error management module initializes and handles the errors that are generated by hardware and provides an option for you to customize these handlers. In hardware, there are two error status registers which hold the type of error that occurred. Also any error can be enabled/disabled from interrupting the PMU MicroBlaze. For each of the errors, you can decide what action should be taken when the error occurs. The possible scenarios would be one or a combination of the following choices:

1. Asserting of `PS_ERROR_OUT` signal on the device
2. Generation of an interrupt to the PMU processor
3. Generation of a system reset (SRST)
4. Generation of a power-on-reset (POR)

PMU firmware provides APIs to register custom error handlers or assign a default (SRST/PoR/PS error out) action in response to an Error. When PMU firmware starts, it sets an error action as interrupt to PMU for some of the errors and PS error out for others as per the `ErrorTable[]` structure defined in `xpfw_error_manager.c`.

Error Management API Calls

This section describes the APIs supported by Error Management module in PMU firmware.

Setting up Error Action

`XPfw_EmSetAction()` API is used to setup an action for the specified error.

Parameters

Table 42: XPfw_EmSetAction

Parameter	Description
ErrorId	ErrorId is ID for error as defined in EM Error ID Table .
ActionId	ActionId is one of the actions defined in EM Error Action Table .
ErrorHandler	ErrorHandler is the handler to be called in case where action is interrupt to PMU

Return

XST_SUCCESS: If error action is set properly.

XST_FAILURE: If error action fails.

Removing Error Action

XPfw_EmDisable() API is used to remove error action for the specified error.

Parameters

Table 43: XPfw_EmDisable

Parameter	Description
ErrorId	ErrorId is ID for error to remove error action

Return

XST_SUCCESS: If successful.

XST_FAILURE: If action fails.

Processing an Error

XPfw_EmProcessError() API processes the errors that occur. If the respective error is registered with an error handler, then this function will call the respective handler to take appropriate action.

Parameters

Table 44: XPfw_EmProcessError

Parameter	Description
ErrorType	Type of error received (EM_ERR_TYPE_1: For errors in PMU GLOBAL ERROR_STATUS_1 EM_ERR_TYPE_2: For errors in PMU GLOBAL ERROR_STATUS_2)

Return

XST_SUCCESS: If successful.

XST_FAILURE: If action fails.

IPI Handling by EM Module

Along with the PM module, error management module also uses IPI-O channel for message exchange. APU and RPU 0/1 masters can communicate to this module using IPI. The `target_module_id` in IPI message differentiates which module needs to take an action based on the message received. The `target_module_id` for IPI handler registered for EM module is 0xE. Currently, PMU firmware supports only the messages shown in the following table using IPI.

Table 45: IPI Messages Supported by PMU firmware

S.No	IPI Message	IPI Message ID/API ID
1	Set error action	0x1
2	Remove error action	0x2
3	Send errors occurred	0x3

Set Error Action

When this IPI message is received from any target to PMU firmware, PMU firmware sets the error action for the error ID received in the message. If processing of the message is successful, it sends SUCCESS (0x0) response to the target. Otherwise FAILURE (0x1) response will be sent. The message format for the same is as below:

Table 46: Message Format for Error Action

Word	Description
0	<target_module_id, api_id>
1	Error ID. See EM Error ID Table for the Error IDs supported.
2	Error Action. See EM Error Action Table for the Error Actions supported.

Remove Error Action

When this IPI message is received from any target to PMU firmware, EM module IPI handler will remove the error action for the error ID received. And after processing the message, it will send SUCCESS/FAILURE response to the target respectively. The message format for the same is as below:

Table 47: Message Format for Removing Error Action

Word	Description
0	<target_module_id, api_id>
1	Error ID. See EM Error ID Table for the Error IDs supported.

Send Errors Occurred

PMU firmware saves the errors that occur in the system and sends to the target upon request. The message format is as below:

Table 48: Message Format for Sending Errors Occurred

Word	Description
0	<target_module_id, api_id>

The following table shows the response message sent by PMU firmware.

Table 49: Response Message by PMU Firmware

Word	Description
0	<target_module_id, Success/Failure>
1	Error_1 (Bit description is as ERROR_STATUS_1 register in PMU Global registers. If a bit is set to 1, then it means the respective error as described in ERROR_STATUS_1 has occurred)
2	Error_2 (Bit description is as ERROR_STATUS_2 register in PMU Global registers. If a bit is set to 1, then it means the respective error as described in ERROR_STATUS_2 has occurred)
3	PMU RAM Correctable ECC Count

EM Error ID Table

Error ID	Error Number	Error Description	Default Error Action
EM_ERR_ID_CSU_ROM	1	Errors logged by CSU bootROM (CBR)	PS Error Out
EM_ERR_ID_PMU_PB	2	Errors logged by PMU bootROM (PBR) in the pre-boot stage	PS Error Out
EM_ERR_ID_PMU_SERVICE	3	Errors logged by PBR in service mode	PS Error Out
EM_ERR_ID_PMU_FW	4	Errors logged by PMU firmware	PS Error Out

Error ID	Error Number	Error Description	Default Error Action
EM_ERR_ID_PMU_UC	5	Un-Correctable errors logged by PMU HW. This includes PMU ROM validation Error, PMU TMR Error, uncorrectable PMU RAM ECC Error, and PMU Local Register Address Error	PS Error Out
EM_ERR_ID_CSU	6	CSU HW related Errors	PS Error Out
EM_ERR_ID_PLL_LOCK	7	Errors set when a PLL loses lock (These need to be enabled only after the PLL locks-up)	PS Error Out
EM_ERR_ID_PL	8	PL Generic Errors passed to PS	PS Error Out
EM_ERR_ID_TO	9	All Time-out Errors [FPS_TO, LPS_TO]	PS Error Out
EM_ERR_ID_AUX3	10	Auxiliary Error 3	PS Error Out
EM_ERR_ID_AUX2	11	Auxiliary Error 2	PS Error Out
EM_ERR_ID_AUX1	12	Auxiliary Error 1	PS Error Out
EM_ERR_ID_AUX0	13	Auxiliary Error 0	PS Error Out
EM_ERR_ID_DFT	14	CSU System Watch-Dog Timer Error	System Reset
EM_ERR_ID_CLK_MON	15	Clock Monitor Error	PS Error Out
EM_ERR_ID_XMPU	16	XMPU Errors [LPS XMPU, FPS XMPU]	Interrupt to PMU
EM_ERR_ID_PWR_SUPPLY	17	Supply Detection Failure Errors	PS Error Out
EM_ERR_ID_FPD_SWDT	18	FPD System Watch-Dog Timer Error	Interrupt to PMU if ENABLE_RECO VERY flag is defined and FSBL runs on APU. Otherwise, System Reset
EM_ERR_ID_LPD_SWDT	19	LPD System Watch-Dog Timer Error	Interrupt to PMU if ENABLE_RECO VERY flag is defined and FSBL runs on RPU. Otherwise, System Reset
EM_ERR_ID_RPU_CCF	20	Asserted if any of the RPU CCF errors are generated	PS Error Out
EM_ERR_ID_RPU_LS	21	Asserted if any of the RPU CCF errors are generated	Interrupt to PMU
EM_ERR_ID_FPD_TEMP	22	FPD Temperature Shutdown Alert	PS Error Out
EM_ERR_ID_LPD_TEMP	23	LPD Temperature Shutdown Alert	PS Error Out
EM_ERR_ID_RPU1	24	RPU1 Error including both Correctable and Uncorrectable Errors	PS Error Out
EM_ERR_ID_RPU0	25	RPU0 Error including both Correctable and Uncorrectable Errors	PS Error Out
EM_ERR_ID_OCM_ECC	26	OCM Uncorrectable ECC Error	PS Error Out
EM_ERR_ID_DDR_ECC	27	DDR Uncorrectable ECC Error	PS Error Out

EM Error Action Table

Table 50: EM Error Action Table

Error Action	Error Action Number	Error Action Description
EM_ACTION_POR	1	Trigger a Power-On-Reset
EM_ACTION_SRST	2	Trigger a System Reset
EM_ACTION_CUSTOM	3	Call the custom handler registered as ErrorHandler parameter
EM_ACTION_PSERR	4	Trigger a PS-Error Out action

PMU Firmware Signals PLL Lock Errors on PS_ERROR_OUT

When EM module is enabled, it is recommended to enable SCHEDULER also. During FSBL execution of `psu_init`, it is expected to get the PLL lock errors. To avoid these errors during EM module initialization, PMU firmware will not enable PLL Lock errors. It waits for `psu_init` completion by FSBL using a scheduler task. After FSBL completes execution of `psu_init`, PMU firmware will enable all PLL Lock errors.

In `xpfw_error_management.c`, you can see the following default behavior of the PMU firmware for PLL Lock Errors:

```
[EM_ERR_ID_PLL_LOCK] = { .Type = EM_ERR_TYPE_2, .RegMask =
PMU_GLOBAL_ERROR_STATUS_2_PLL_LOCK_MASK, .Action = EM_ACTION_NONE, .Handler
=
NullHandler},
```

where, `PMU_GLOBAL_ERROR_STATUS_2_PLL_LOCK_MASK` is #defined with `0X00001F00` value, which means that all the PLL Lock Errors are enabled. Hence, if the design do not use any PLL/PLLs that are not locked, this triggers the `PS_ERROR_OUT` signal. It means that the `PMU_GLOBAL.ERROR_STATUS_2` register (bits [12:8]) signals that one or more PLLs are NOT locked and that triggers the `PS_ERROR_OUT` signal.

To analyze further and see if this is really an issue is to fully understand the status of the PLL in the design. For example, if the design only uses `IO_PLL` and `DDR_PLL` and `PMU_GLOBAL.ERROR_STATUS_2` register signals `0x1600` value, it means that the `RPU_PLL`, `APU_PLL`, and `Video_PLL` Lock errors have occurred. Looking at a few more registers, you can really understand the status of the PLLs.

PLL_STATUS

- `PLL_STATUS (CRL_APB) = FF5E0040: 00000019`
- `PLL_STATUS (CRF_APB) = FD1A0044: 0000003A`

Table 51: PLL_STATUS

PLL STATUS	ERROR_STATUS_2
IOPLL is locked and stable	Bit [8] is for IO_PLL = 0
RPLL is stabled and NOT locked (which means bypassed)	Bit [9] is for RPU_PLL = 1
APPL is stabled and NOT locked (which means bypassed)	Bit [10] is for APU_PLL = 1
DPLL is locked and stable	Bit [11] is for DDR_PLL = 0
VPLL is stabled and NOT locked (which means bypassed)	Bit [12] is for Video_PLL = 1

Hence, if the design only uses IO_PLL and DDR_PLL, then it is not really an error to have RPU_PLL, APU_PLL and Video_PLL in NOT locked status.

Xilinx recommends you to customize the PMU_GLOBAL_ERROR_STATUS_2_PLL_LOCK_MASK to cover only the PLL of interest so that you can have a meaningful PS_ERROR_OUT signal.

Example:

```
#define PMU_GLOBAL_ERROR_STATUS_2_PLL_LOCK_MASK ((u32)0X00000900U) will only
signal on PS_ERROR_OUT IO_PLL and DDR_PLL errors.
```

Power Management (PM) Module

Zynq UltraScale+ MPSoC Power Management framework is based on an implementation of the Embedded Energy Management Interface (EEMI). This framework allows software components running across different processing units (PUs) on a chip or device to issue or respond to requests for power management.

The Power Management module is implemented within the PMU firmware as an event-driven module. Events processed by the Power Management module are called power management events. All power management events are triggered via interrupts.

When handling an interrupt the PMU firmware determines whether the associated event shall be processed by the Power Management module. Accordingly, if the PMU firmware determines that an event is power management related and if the Power Management module is enabled, the PMU firmware triggers it to process the event.

For example, all the PS and PL interrupts can be routed to the PMU via the GIC Proxy. When the application processors (APU or RPU) are temporarily suspended, the PMU handles the GIC Proxy interrupt and wakes up the application processors to service the original interrupts. The PMU firmware does not actually service these interrupts, although you are free to customize the PMU firmware so that these interrupts are serviced by the PMU instead of by the application processors. For more information, see the 'Interrupts' chapter of the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)).

When processing a power management event the Power Management controller may exploit the PMU ROM handlers for particular operations regarding the state control of hardware resources. Warm restart and FPGA configuration manager are part of Power Management module. PMU firmware includes XilFPGA and XilSecure libraries to support the functionalities of PL FPGA configuration and to access secure features respectively. See [Chapter 11: Power Management Framework](#) for more information.

Note: Since the Power Management module uses base firmware APIs such as IPI manager/event manager, it is not possible to run standalone power management features without PMU firmware. See [PM Examples wiki page](#) for XilPM based design examples.

Scheduler

A scheduler is required by modules like STL in order to support periodic tasks like register coverage, scrubbing, etc. PMU firmware also uses scheduler for LPD WDT functionality. This will be explained in the following section. PMU MicroBlaze has 4 PITs (0-3) and Scheduler uses PIT1. The scheduler supports up to 10 tasks. Table shows the Scheduler's task list data structure with members.

Table 52: Scheduler Data Structure Members

Member	Values/Range	Additional information
Task ID 0	0.. 9	0 - Highest priority
Interval	Task interval in Milliseconds	
OwnerId	0.. 9	Modules that owns this task
Status	Enabled/Disabled	
Callback	Function pointer	Default to NULL

Note: By default, scheduler functionality is disabled. To enable the same, ENABLE_SCHEDULER build flag needs to be defined.

Safety Test Library

Safety Test Library (STL) is a collection of software safety mechanisms complementing hardware safety features for the detection of random hardware (HW) faults. PMU firmware has a placeholder for STL initialization during PMU firmware startup. This is enabled when ENABLE_STL build flag is defined. The software library and the safety documentation can be seen at the [Safety Lounge](#).

CSU/PMU Register Access

The following section discusses how to Read/Write the CSU and PMU global registers and provides a list of White and Black registers.

Register Write

```
$ echo > /sys/firmware/zynqmp/config_reg
```

Register Read

```
$ echo > /sys/firmware/zynqmp/config_reg  
$ cat /sys/firmware/zynqmp/config_reg
```

CSU and PMU global registers are categorized into two lists:

- By default, the White list registers can be accessed all the time. The following is a list of white registers.
 - CSU Module:
 - Csu_status
 - Csu_multi_boot
 - Csu_tamper_trig
 - Csu_ft_status
 - Jtag_chain_status
 - Idcode
 - Version
 - Csu_rom_digest(0:11)
 - Aes_status
 - Pcap_status
 - PMU Global Module:
 - Global_control
 - Global_Gen_Storage0 - 6
 - Pers_Glob_Gen_Storage0-6
 - Req_Iso_Status
 - Req_SwRst_Status
 - Csu_Br_Error

- Safety_Chk
- The Black list registers can accessed when a compile time flag is set.

Every other register in both the CSU Module and the PMU_GLOBAL Module that is not covered in the above white list will be a black register. RSA and RSA_CORE module registers are black registers.

The `#define` option (`SECURE_ACCESS_VAL`) provides access to the black list. To access black list registers, build the PMU firmware with `SECURE_ACCESS_VAL` flag set.

Timers

Zynq UltraScale+ MPSoCs have two system watchdog timers, one each for full-power domain (FPD) and low-power domain (LPD). Each of these WDT provides error condition information to the error manager. EM module can be configured to set a specific error action when FPD or LPD WDT expires. This section describes the usage of these watchdog timers and the PMU firmware functionality when these watchdog timers expire.

FPD WDT

FPD WDT can be used to reset the APU or the FPD. PMU firmware error management module can configure the error action to be taken when the FPD WDT error occurs. PMU firmware implemented a recovery mechanism for FPD WDT error. This mechanism is disabled by default. The same can be enabled by defining `ENABLE_RECOVERY` build flag.

The EM module in PMU firmware sets FPD WDT error action as 'system reset' when recovery mechanism is not enabled. In this case, PMU firmware doesn't initialize and configure the FPD WDT. It is left for Linux driver to initialize and start the WDT if required. When WDT expires, system restart happens.

When `ENABLE_RECOVERY` flag is defined and FSBL runs on APU, PMU firmware sets FPD WDT error action as 'interrupt to PMU' and registers a handler to be called when this error occurs. In this case, when PMU firmware comes up, it initializes and starts the WDT. It also initializes and sets the timer mode of TTC to interval mode.

PMU firmware configures FPD WDT expiry time to 60 seconds. And if WDT error occurs, PMU firmware gets an interrupt and it calls the registered handler. PMU firmware has a restart tracker structure to track the restart phase and other information for a master. APU and RPU are the masters currently using this structure. Following are its members:

Table 53: Restart Tracker Structure Members

Member	Description
Master	Master whose restart cycle is to be tracked
RestartState	Track different phases in restart cycle
RestartScope	Restart scope upon FPD WDT error interrupt
WdtBaseAddress	Base address for WDT assigned to this master
WdtTimeout	Timeout value for WDT
ErrorId	Error Id corresponding to the WDT
WdtPtr	Pointer to WDT for this master
WdtResetId	Wdt reset ID
TtcDeviceId	TTC timer device ID
TtcPtr	Pointer to TTC for this master
TtcTimeout	Timeout to notify master for event
TtcResetId	Reset line ID for TTC

When WDT error occurs, WDT error handler is called and PMU firmware performs the following:

1. It checks if master is APU and error ID is FPD WDT. Then, it checks if restart state is in progress or not. If restart state is not in progress, then it changes the restart state to in progress.
2. Later, it restarts the WDT so that the PMU firmware knows when the WDT error is not due to APU application.
3. Then, it idles APU by sending an IPI to ATF via timer interrupt TTC3_0.
Note: This is only true for Linux, and not for bare metal where there is no ATF.
4. If the first restart attempt fails, then PMU firmware escalates restart to either system-reset or PS-only reset if `ENABLE_ESCALATION` flag is defined. If `ENABLE_ESCALATION` is not defined, PMU firmware restarts the APU. Otherwise, PMU firmware performs the following:
 - First, PMU firmware checks if PL is configured or not.
 - If PL is configured, PMU firmware initiates PS-only restart. Otherwise, it initiates system-reset.

Note: Ensure that the WDT heartbeat application is running in Linux.

LPD WDT

LPD WDT can be used to reset the RPU. PMU firmware error management module can configure the error action to be taken when the LPD WDT error occurs. PMU firmware implements a recovery mechanism for LPD WDT error. This mechanism is disabled by default. The same can be enabled by defining the `ENABLE_RECOVERY` build flag.

The EM module in the PMU firmware sets LPD WDT error action as "system reset" when recovery mechanism is not enabled. In this case, PMU firmware doesn't initialize and configure the LPD WDT. It is left to the RPU user application to initialize and start the WDT, if required. When WDT expires, the system restarts.

When `ENABLE_RECOVERY` flag is defined and FSBL is running on RPU, PMU firmware sets FPD WDT error action as "interrupt to PMU" and registers a handler to be called when this error occurs. In this case, when PMU firmware comes up, it initializes and starts the WDT.

PMU firmware configures LPD WDT expiry time to 60s. And if WDT error occurs, PMU firmware gets an interrupt and it calls the registered handler. PMU firmware maintains a restart tracker structure for LPD WDT. Refer to Table 10-23 for more information.

When WDT error occurs, the WDT error handler is called and PMU firmware performs the following actions:

1. It checks if master is RPU and error ID is LPD WDT. Then, it checks if restart state is in progress or not. If restart state is not in progress, then it changes the restart state to in progress and restarts the WDT to track the next WDT expiry.
2. It applies AIB isolation for RPU and removes it.
3. If restart scope is set as a subsystem, then it will restart RPU subsystem.
4. If restart scope is set as PS only restart, then PMU firmware will restart PS subsystem.
5. If restart scope is set as system, then it will perform the system restart.

CSU WDT

The CSU WDT is configured to be used by PMU firmware that if PMU firmware application hangs for some reason, then the system would restart. This functionality is enabled only when `ENABLE_WDT` flag is defined.

EM modules sets CSU WDT error action as 'System Reset' Initialization of CSU WDT depends on bringing WDT out of reset which is performed by `psu_init` from FSBL. FSBL writes the status of `psu_init` completion to PMU Global general storage register 5, so that PMU firmware can check for its completion before initializing CSU WDT. When `ENABLE_WDT` flag is defined during PMU firmware initialization, it adds a task to scheduler to be triggered for every 100 milli-seconds until `psu_init` completion status is updated by FSBL. After `psu_init` is completed, this task will be removed from scheduler tasks list and PMU firmware initializes CSU WDT and configures it to 90 milli-seconds. It also starts a scheduler task to restart the WDT for every 50 milli-seconds. Whenever CSU WDT error occurs due to PMU firmware code hanging, this error is handled in hardware to trigger 'System Reset' and the system will restart.

Following are the dependencies to use this WDT functionality:

1. EM module needs to be enabled by defining `ENABLE_EM` flag.

2. `ENABLE_WDT` flag needs to be defined to use CSU WDT.
3. Scheduler module needs to be enabled by defining `ENABLE_SCHEDULER` to add a task to scheduler to check for `psu_init` completion and to restart WDT periodically.

Configuration Object

The configuration object is a binary data object used to allow updating data structures in the PMU firmware power management module at boot time. The configuration object must be copied into memory by a processing unit on the Zynq UltraScale+ MPSoC. The memory region containing the configuration object must be accessible by the PMU.

The PMU is triggered to load the configuration object via the following API call:

```
XPm_SetConfiguration(address);
```

The address argument represents the start address of the memory where the configuration object is located. The PMU determines the size of the configuration object based on its content.

Once the PMU loads the configuration object it updates its data structures which are used to manage the states of hardware resources (nodes). Partial configurations are not possible. If the configuration object does not provide information as defined in this document or provides partial information, the consistency of PMU firmware power management data cannot be guaranteed. The creator of the configuration object must ensure the consistency of the information provided in the configuration object. The PMU does not change the state of nodes once the configuration object is loaded. The PMU also does not check whether the information about current states of nodes provided in the configuration object really matches the current state of the hardware. Current state is a state of a hardware resource at the moment of processing the configuration object by the PMU.

The configuration object specifies the following:

- List of masters available in the system
- All the slave nodes the master is currently using and current requirement of the master for the slave configuration
- All the slave nodes the master is allowed to use and default requirement of the master for the slave configuration
- For each power node, which masters are allowed to request/release/power down
- For each reset line, which masters are allowed to request the change of a reset line value
- Which shutdown mode the master is allowed to request and shutdown timeout value for the master

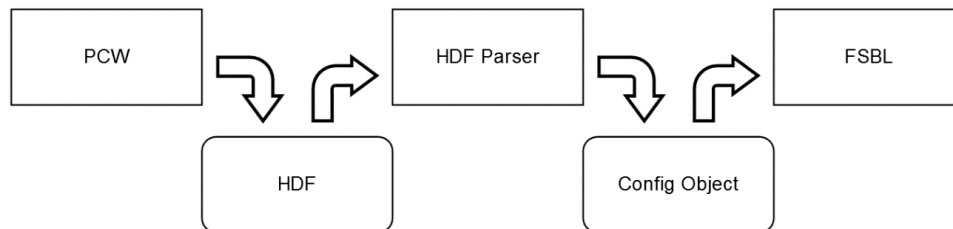
- Which masters are allowed to set configuration after the configuration is already set upon the system boot by the FSBL

PM Configuration Object Generation

PM Configuration Object is generated as follows:

1. Specify the custom PM framework Configuration using the PCW tool
2. PCW generates the HDF file
3. At build time, the HDF Parser parses the HDF file and insert the configuration object into the FSBL code

Figure 45: Configuration Object Generation



Initial Configuration at Boot

The configuration object shall be loaded prior to calling any EEMI API, except the following APIs:

- Get API version
- Set configuration
- Get Chip ID

Until the first configuration object is loaded the PM controller is configured to initially expect the EEMI API calls from the APU or RPU master, via IPI_APU or IPI_RPU_0 IPI channels, respectively. In other words, the first configuration object has to be loaded by APU or RPU.

After the first configuration object is loaded, the next loading of the configuration object can be triggered by a privileged master. Privileged masters are defined in the configuration object that was loaded the last.

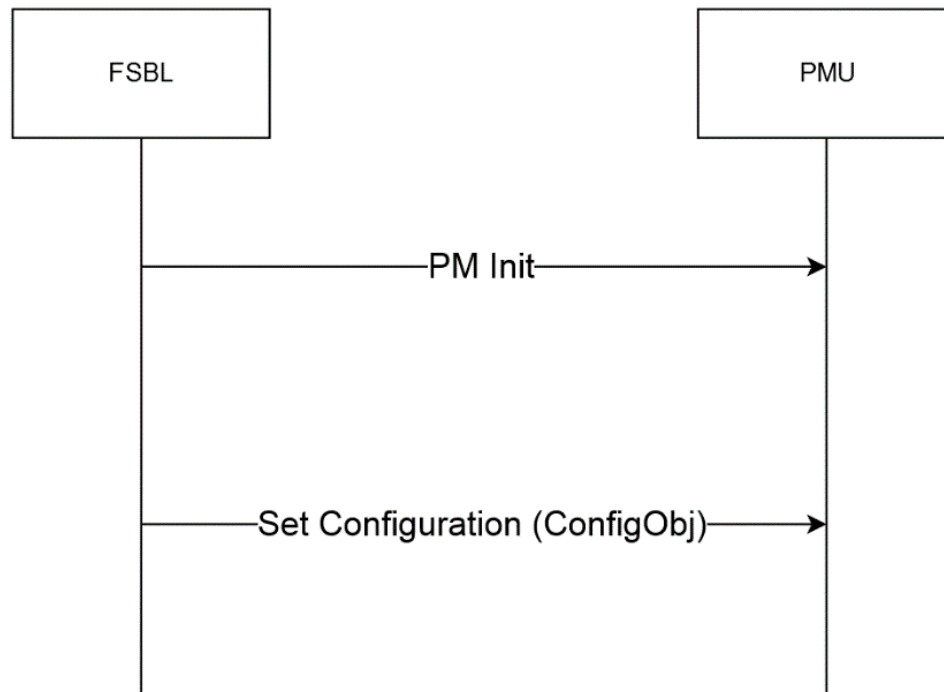
Following are the steps at boot level:

1. FSBL sends the configuration object to PMU with the Set Configuration API
2. PMU parses the configuration object and configures

3. PMU powers off all the nodes which are unused after all the masters have completed the initialization

All other requests prior to the first Set Configuration API call will be rejected by PMU firmware.

Figure 46: Initial Configuration at Boot



PMU Firmware Loading Options

PMU firmware can be loaded by either FSBL or CSU BootROM (CBR). Both these flows are supported by Xilinx. Loading PMU firmware using FSBL has the following benefits:

- Possible quick boot time, when PMU firmware is loaded after bitstream.
- In use cases where you want two BIN files - stable and upgradable, PMU firmware can be part of the upgradable (by FSBL) image.



IMPORTANT! CBR loads FSBL. If CBR also loads PMU firmware, it means that the secure headers for both FSBL and PMU firmware are decrypted with same Key-IV pair, which is a security vulnerability (security rule is: no two partitions should use the same Key-IV pair). This is addressed in FSBL, not in CBR. Hence, you should avoid CBR loading PMU firmware in secure (decryption) cases.

For DDR self-refresh over Warm restart, FSBL and PMU firmware must be loaded first (in any order) before all other images (e.g. bitstream).

For Power Off Suspend, PMU firmware must be loaded first (i.e. by CSU) before FSBL.

Loading PMU Firmware in JTAG Boot Mode

PM operations depend on the configuration object loaded by FSBL from 2017.1 release onwards. Hence, In JTAG boot mode, it is mandatory to load PMU FW before loading FSBL. In device boot modes, loading of configuration object to PMU firmware by FSBL is handled both in CBR loading PMU firmware and FSBL loading PMU firmware options. Use the following steps to boot in JTAG mode:

1. Disable security gates to view PMU MicroBlaze. PMU MicroBlaze is not visible in xsdb for Silicon v3.0 and above.
2. Load PMU firmware and run.
3. Load FSBL and run.
4. Continue with U-Boot/Linux/user specific application.

Following is a complete Tcl script:

```
#Disable Security gates to view PMU MB target
targets -set -filter {name =~ "PSU"}

#By default, JTAGsecurity gates are enabled
#This disables security gates for DAP, PLTAP and PMU.
mwr 0xffca0038 0x1ff
after 500

#Load and run PMU FW
targets -set -filter {name =~ "MicroBlaze PMU"}
dow xpfw.elf
con
after 500

#Reset A53, load and run FSBL
targets -set -filter {name =~ "Cortex-A53 #0"}
rst -processor
dow fsbl_a53.elf
con

#Give FSBL time to run
after 5000
stop

#Other SW...
dow u-boot.elf
dow bl31.elf
con

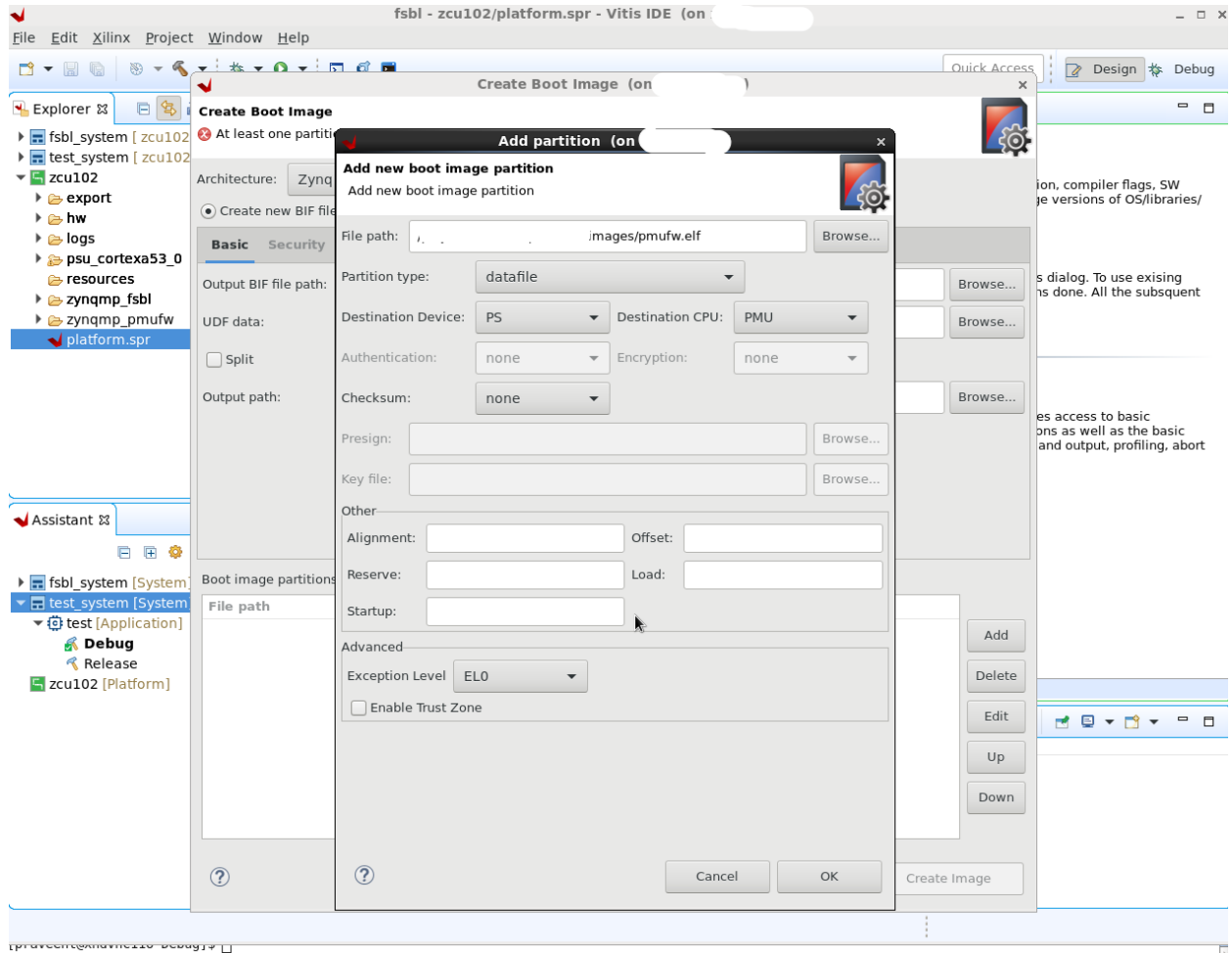
#Loading bitstream to PL
Targets -set -nocase -filter {name =~ "*PL*"}
fpga download.bit
```

Loading PMU Firmware in NON-JTAG Boot Mode

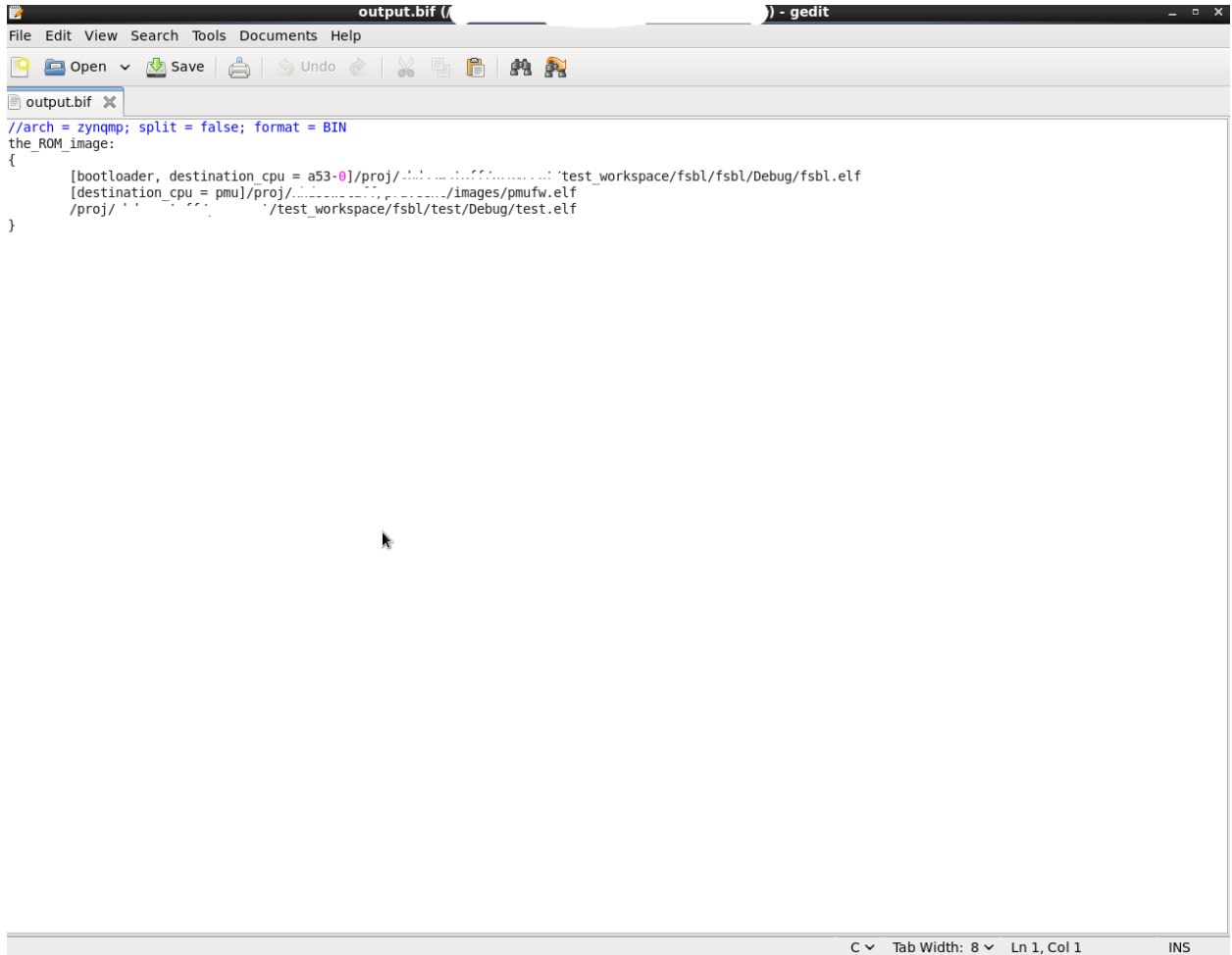
When PMU firmware is loaded in a non-JTAG Boot mode on a 1.0 Silicon, an error message 'Error: Unhandled IPI received' may be logged by PMU firmware at startup, which can be safely ignored. This is due to the IPI0 ISR not being cleared by PMU ROM. This is fixed in 2.0 and later versions of silicon.

Using FSBL to Load PMU Firmware

1. Build PMU firmware application in the Vitis IDE.
2. Build an FSBL in the Vitis IDE for A53. (R5F can also be used).
3. Create a hello_world example for A53.
4. Select **Xilinx → Create Boot Image**.
5. Create a new bif file. Choose:
 - a. Architecture: **ZynqMP**
 - b. You will see A53 fsbl and hello_world example by default in partitions. Also, we need PMU firmware.
 - c. Click on **Add**, then provide `pmufw.elf` path. Also select Partition type as **datafile**, Destination device as **PS**, and Destination CPU as **PMU**.
 - d. Click **OK**.



6. After adding pmufw as partition. Click on **pmufw partition** and then, click **UP**.
7. Make sure to select the following partition order:
 - a. A53 FSBL
 - b. PMU firmware
 - c. Hello World application
8. Click on **Create Image**. You will see `BOOT.bin` created in a new `bootimage` folder in your example project.
9. View the `.BIF` file to confirm the partition order.



```

output.bif (
) - gedit
File Edit View Search Tools Documents Help
Open Save Undo
output.bif x
//arch = zynqmp; split = false; format = BIN
the_ROM_image:
{
    [bootloader, destination_cpu = a53-0]/proj/.../test_workspace/fsbl/fsbl/Debug/fsbl.elf
    [destination_cpu = pmu]/proj/.../images/pmufw.elf
    /proj/.../test_workspace/fsbl/test/Debug/test.elf
}
    
```

10. Now copy this `BOOT.bin` into SD card.
11. Boot the ZCU102 board in SD boot mode. You can see the `fsbl` → `pmufw` → `hello_world` example prints in a sequence.

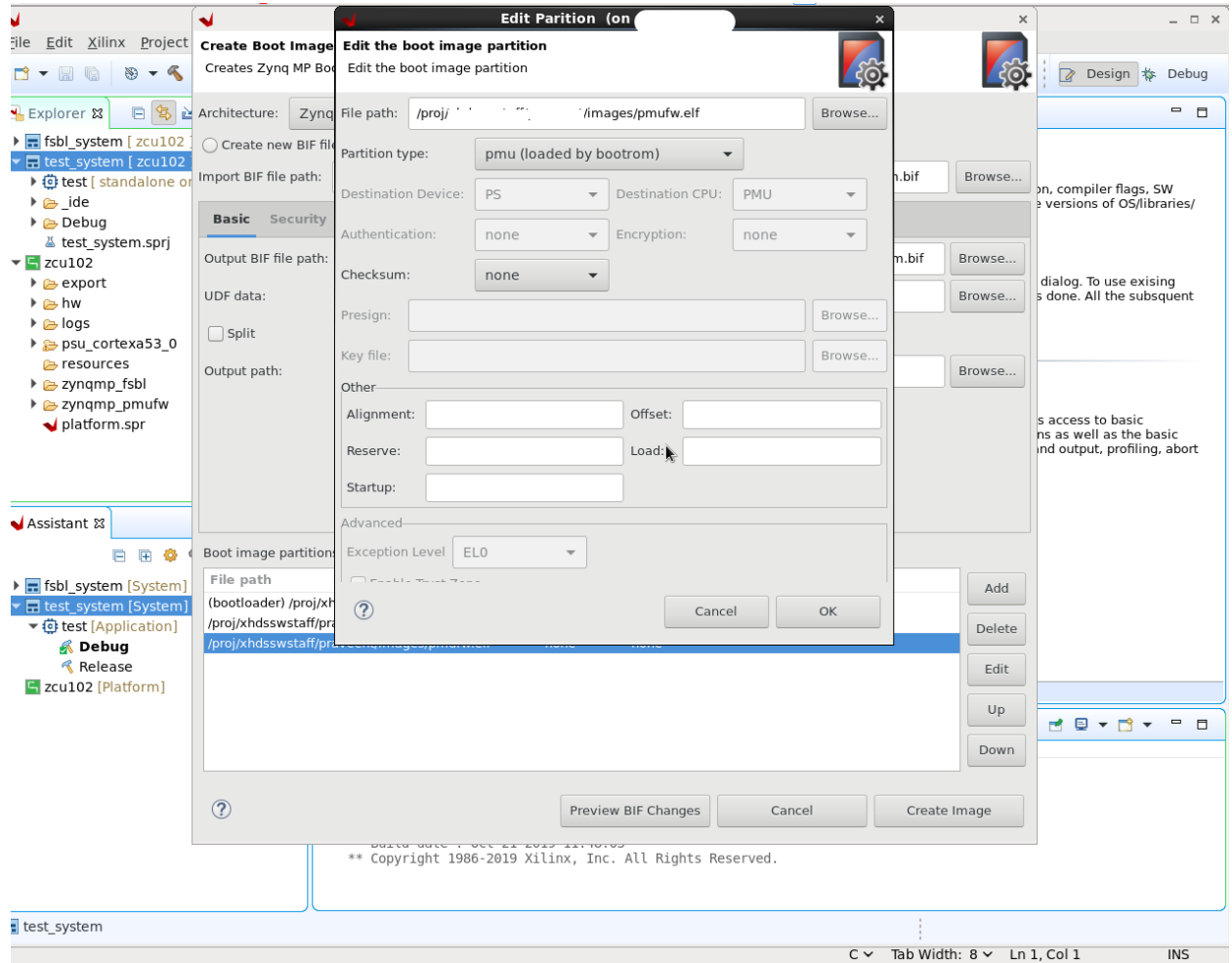
Using CBR to load PMU Firmware

When PMU firmware is loaded by CBR, it is executed prior to FSBL. So the MIOs, Clocks and other initializations are not done at this point. Consequently, the PMU firmware banner and other prints may not be seen prior to FSBL. Post FSBL execution, the PMU firmware prints can be seen as usual.

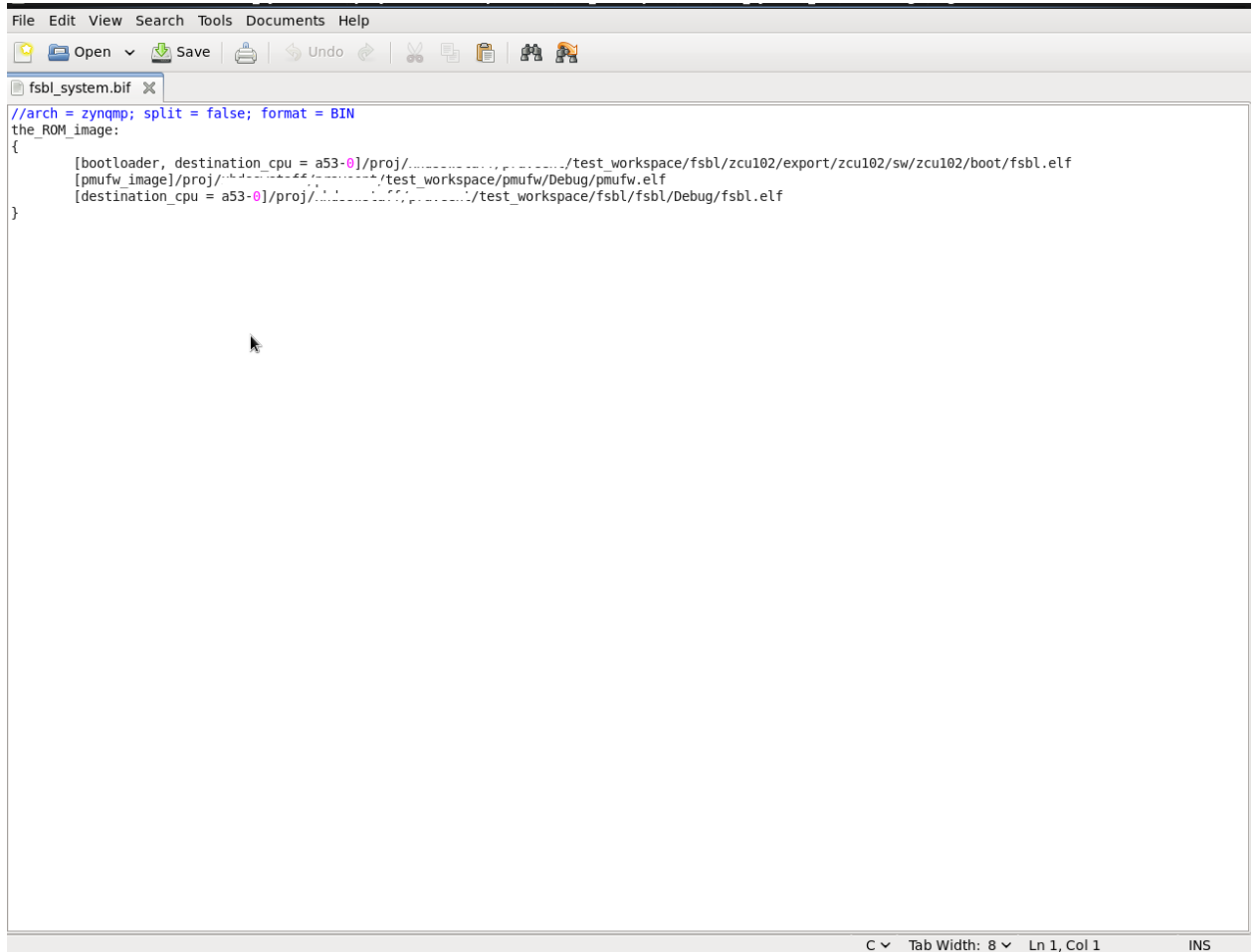
To make the CBR load PMU firmware, follow these steps:

1. Change the `BOOT.bin` boot partitions.
2. Perform the steps listed in [Loading PMU Firmware in NON-JTAG Boot Mode](#).
3. Create a new bif file. Choose the following:
 - a. Architecture: **ZynqMP**.

- b. You will see A53 fsbl and hello_world example by default in partitions. Also, we need pmufw.
- c. Click **Add** and then provide the `pmufw.elf` path. Select the Partition type as **pmu** (loaded by bootrom).
- d. Click **OK**.



- e. Click on **Create Image**. You will see `BOOT.bin` created in a new folder named `bootimage` in your example project.
- f. You can also view `.BIF` to confirm the partition order.



```

File Edit View Search Tools Documents Help
[Icons: Open, Save, Print, Undo, Redo, Copy, Paste, Find, Run, Stop]
fsbl_system.bif x
//arch = zynqmp; split = false; format = BIN
the_ROM_image:
{
    [bootloader, destination_cpu = a53-0]/proj/...../test_workspace/fsbl/zcu102/export/zcu102/sw/zcu102/boot/fsbl.elf
    [pmufw_image]/proj/...../test_workspace/pmufw/Debug/pmufw.elf
    [destination_cpu = a53-0]/proj/...../test_workspace/fsbl/fsbl/Debug/fsbl.elf
}
    
```

C Tab Width: 8 Ln 1, Col 1 INS

- g. Now copy this `BOOT.bin` into SD card.
- h. Boot the ZCU102 board in SD boot mode. You can see the `pmufw → fsbl → hello_world` example prints in a sequence.

PMU Firmware Usage

This section describes the usage of PMU firmware with examples.

Enable/Disable Modules

This section describes how to enable/disable PMU firmware build flags both in the Vitis software platform and PetaLinux.

In PetaLinux

1. Create a PetaLinux project.

2. Open `<plnx-project-root>/project-spec/meta-user/recipes-bsp/pmu/pmu-firmware_%.bbappend` file and add the following line:

```
YAML_COMPILER_FLAGS_append = -DENABLE_EM
```

The above line enables EM module. To enable any flag, it should be prefixed with '-D'.

3. After any change to the YAML compiler flags, force a clean state before rebuilding the application.

Custom Module Usage

Each set of user defined routines performing a specific functionality should be designed to be a module in PMU firmware. These files must be self-contained. However, these files can use declarations from `xpfw_core.h`. Each module can register with the following interfaces. If any of the handler is not needed by the module, it can be skipped from being registered.

- Config Handler: Called during initialization.
- Event Handler: Called when a registered event is triggered.
- IPI Handler: Called when an IPI message arrives with the registered IPI ID

Creating a Custom Module

To create a custom module, add the following code to PMU firmware:

```
/* IPI Handler */
static void CustomIpiHandler(const XPfw_Module_t *ModPtr, u32 IpiNum, u32
SrcMask,
const u32* Payload, u8 Len)
{
/**
* Code to handle the IPI message received
*/
}

/* CfgInit Handler */
static void CustomCfgInit(const XPfw_Module_t *ModPtr, const u32 *CfgData,
u32 Len)
{
/**
* Code to configure the module, register for events or add scheduler tasks
*/
}

/* Event Handler */
static void CustomEventHandler(const XPfw_Module_t *ModPtr, u32 EventId)
{
/**
* Code to handle the events received
*/
}

/*
* Create a Mod and assign the Handlers. We will call this function
```

```

* from XPfw_UserStartup()
*/
void ModCustomInit(void)
{
    const XPfw_Module_t *CustomModPtr = XPfw_CoreCreateMod();
    (void) XPfw_CoreSetCfgHandler(CustomModPtr, CustomCfgInit);
    (void) XPfw_CoreSetEventHandler(CustomModPtr, CustomEventHandler);
    (void) XPfw_CoreSetIpiHandler(CustomModPtr, CustomIpiHandler, (u16)IPI_ID);
}
    
```

Registering for an Event

All interrupts that come into PMU are exposed to user as Events with specific EVENTIDs defined in `xpfw_events.h`. Any module can register for an event (usually in `CfgHandler`) and the module's `EventHandler` will be called when an event is triggered.

To register for an RTC Event:

```
Status = XPfw_CoreRegisterEvent(ModPtr, XPFW_EV_RTC_SECONDS);
```

Example of an `EventHandler`:

```

void RtcEventHandler(const XPfw_Module_t *ModPtr, u32 EventId)
{
    xil_printf("MOD%d:EVENTID: %d\r\n", ModPtr->ModId, EventId);
    if(XPFW_EV_RTC_SECONDS == EventId){
        /* Ack the Int in RTC Module */
        Xil_Out32(RTC_RTC_INT_STATUS, 1U);
        xil_printf("RTC: %d \r\n", Xil_In32(RTC_CURRENT_TIME));
    }
}
    
```

Error Management Usage

This sections describes the usage of the EM module to configure the error action to be taken for the errors that comes to PMU firmware (the errors generated in the system which are mapped to PMU MB).

Example for Error Management (Custom Handler)

For this example, OCM uncorrectable error (`EM_ERR_ID_OCM_ECC`) is considered. The default error action for this error is set to PS Error Out. In the following example, a custom handler is registered for this error in PMU firmware and the handler in this case just prints out the error message. In a more realistic case, the corrupted memory may be reloaded, but this example is just limited to clearing the error and printing a message.

Adding the Error Handler for OCM Uncorrectable ECC in PMU firmware:

```

+++ b/lib/sw_apps/zynqmp_pmufw/src/xpfw_mod_em.c
@@ -140,6 +140,14 @@ void FpdSwdtHandler(u8 ErrorId)
    XPfw_RecoveryHandler(ErrorId);
    }
+/* OCM Uncorrectable Error Handler */
+static void OcmErrHandler(u8 ErrorId)
+{
+ XPfw_Printf(DEBUG_DETAILED, "EM: OCM ECC error detected\n");
+ /* Clear the Error Status in OCM registers */
+ XPfw_Write32(0xFF960004, 0x80);
+}
+/* CfgInit Handler */
+static void EmCfgInit(const XPfw_Module_t *ModPtr, const u32 *CfgData,
+u32 Len)
@@ -162,6 +170,8 @@ static void EmCfgInit(const XPfw_Module_t *ModPtr,
const u32
*CfgData,
    }
    }
+ XPfw_EmSetAction(EM_ERR_ID_OCM_ECC, EM_ACTION_CUSTOM, OcmErrHandler);
+
+ if (XPfw_RecoveryInit() == XST_SUCCESS) {
+ /* This is to enable FPD WDT and enable recovery mechanism when
    
```

To inject OCM Uncorrectable ECC error using debugger (xsdb):

```

;# Enable ECC UE interrupt in OCM_IEN
mwr -force 0xFF96000C [expr 1<<7]

;# Write to Fault Injection Data 0 Register OCM_FI_D0
;# Errors will be injected in the bits which are set, here its bit0, bit1
mwr -force 0xFF96004C 3

;# Enable ECC and Fault Injection
mwr -force 0xFF960014 1
;
;# Clear the Count Register : OCM_FI_CNTR
mwr -force 0xFF960074 0
;# Now write data to OCM for the fault to be injected
;# Since OCM does a RMW for 32-bit transactions, it should trigger error here
mwr -force 0xFFFFE0000 0x1234

;# Read back to trigger error again
mrd -force 0xFFFFE0000
    
```

Example for Error Management (PoR as a Response to Error)

Some error may be too fatal and the system recovery from those errors may not be feasible without doing a Reset of entire system. In such cases PoR or SRST can be used as actions. In this example we use PoR reset as a response to the OCM ECC double-bit error.

Here is the code that adds the PoR as action:

```
@@ -162,6 +162,8 @@ static void EmCfgInit(const XPfw_Module_t *ModPtr,
const u32
*CfgData,
}
}
+ XPfw_EmSetAction(EM_ERR_ID_OCM_ECC, EM_ACTION_POR, NULL);
+
if (XPfw_RecoveryInit() == XST_SUCCESS) {
/* This is to enable FPD WDT and enable recovery mechanism when
```

The Tcl script to inject OCM ECC error is same as the one for above example. Once you trigger the error, a PoR occurs and you may see that all processors are in reset state similar to how they would be in a fresh power-on state. PMU RAM also gets cleared off during a PoR. Hence, PMU firmware needs to be reloaded.

Example for Error Management (PS Error out as a Response to Error)

If you need to communicate outside of system when any error occurs, PS ERROR OUT response can be set for that respective error. So, when that error occurs, error will be propagated outside and PS_ERROUT signal LED will glow. In this example we use PS ERROR OUT as a response to the OCM ECC double-bit error.

Following is the code that adds the PS ERROR OUT as action:

```
@@ -162,6 +162,8 @@ static void EmCfgInit(const XPfw_Module_t *ModPtr,
const u32
*CfgData,
}
}
+ XPfw_EmSetAction(EM_ERR_ID_OCM_ECC, EM_ACTION_PSERR, NULL);
+
if (XPfw_RecoveryInit() == XST_SUCCESS) {
/* This is to enable FPD WDT and enable recovery mechanism when
```

The Tcl script to inject OCM ECC error is same as the one for above example. Once you trigger the error, a PS_ERROUT LED will glow on board.

IPI Messaging Usage

This section describes the usage of IPI messaging from PMU firmware to RPU0 and RPU0 to PMU firmware. PMU firmware, while initializing IPI driver, also enables IPI interrupt from the IPI channel assigned master.

From PMU Firmware to RPU0

See [Zynq UltraScale Plus MPSoC - IPI Messaging Example](#) for more information.

Note: You need to enable EM module in PMU firmware to run this example.

From RPU0 to PMU Firmware

See [Zynq UltraScale Plus MPSoC - IPI Messaging Example](#) for IPI messaging example from RPU to PMU.



IMPORTANT! Since the example in the wiki page shows how to trigger IPI from PMU to RPU0 and vice versa, to trigger an IPI to/from APU or RPU1, you need to change the destination CPU mask to the intended master.

Adding a Task to Scheduler

Tasks are functions which take void arguments and return void. Currently PMU firmware has no way to check that the task returns in a pre-determined time, so this needs to be ensured by the task design. Let us consider a task which prints out a message:

```
void TaskPrintMsg(void)
{
    xil_printf("Task has been triggered\r\n");
}
```

If we want to schedule the above task to occur every 500ms, the following code can be used. The TaskModPtr is a pointer for module which is scheduling the task.

```
Status = XPfw_CoreScheduleTask(TaskModPtr, 500U, TaskPrintMsg);
if(XST_SUCCESS == Status) {
    xil_printf("Task has been added successfully !\r\n");
}
else {
    xil_printf("Error: Failed to add Task !\r\n");
}
```

Reading FPD Locked Status from RPU

Register 0xFFD600F0 is a local register to PMU firmware, in which bit 31 displays whether FPD is locked or not locked. (If bit 31 is set to 1, then FPD is locked. It remains isolated until POR is asserted). You can verify the FPD locked status by reading this register through PMU firmware. This can be achieved by an MMIO read call to PMU firmware. Use the following steps to read FPD locked status from R5:

1. Create an empty application for R5 processor. Enable xilpm library in BSP settings.
2. Create a new.c file in the project and add the following code:

```
#include "xipipsu.h"
#include "pm_api_sys.h"
#define IPI_DEVICE_IDXPAR_XIPIPSU_0_DEVICE_ID
#define IPI_PMU_PM_INT_MASKXPAR_XIPIPS_TARGET_PSU_PMU_0_CH0_MASK

#define MMIO_READ_API_ID20U
#define FPD_LOCK_STATUS_REG0xFFD600F0
```

```

int main(void)
{
XIpiPsu IpiInstance; XIpiPsu_Config *Config; s32 Status;
u32 Value;

/* Initialize IPI peripheral */
Config = XIpiPsu_LookupConfig(IPI_DEVICE_ID); if (Config == NULL) {
xil_printf("Config Null\r\n"); goto END;
}

Status = XIpiPsu_CfgInitialize(&IpiInstance, Config, Config-
>BaseAddress);
if (0x0U != Status) { xil_printf("Config init failed\r\n"); goto END;
}

/* Initialize the XilPM library */ Status = XPm_InitXilpm(&IpiInstance);
if (0x0U != Status) {
xil_printf("XilPM init failed\r\n"); goto END;
}
/* Read using XPm_MmioRead() */
Status = XPm_MmioRead(FPD_LOCK_STATUS_REG, &Value); if (0x0U != Status)
{
xil_printf("XilPM MMIO Read failed\r\n"); goto END;
}
xil_printf("Value read from 0x%x: 0x%x\r\n", FPD_LOCK_STATUS_REG, Value);

END:
xil_printf("Exit from main\r\n");
}
    
```

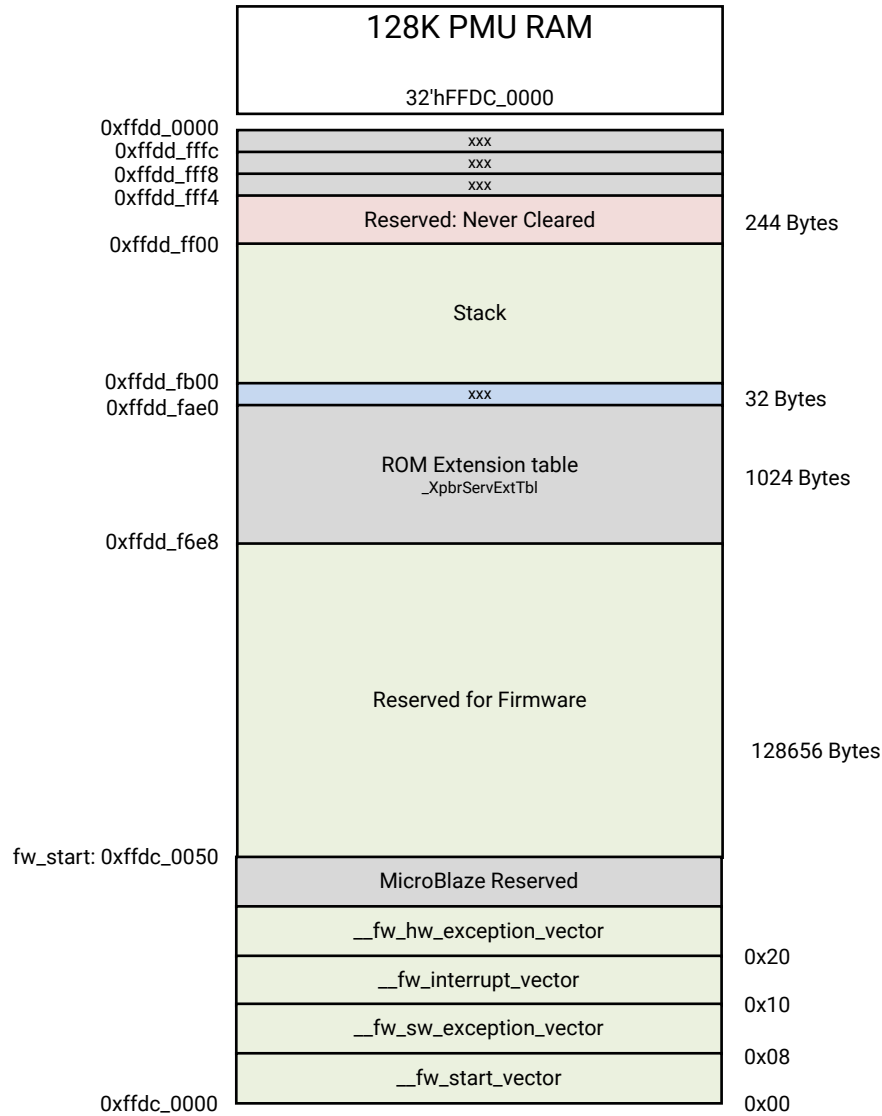
Note: This application must be run after FSBL is successfully executed. This application cannot run successfully, if FSBL fails to send configuration object to PMU firmware.

PMU Firmware Memory Layout and Footprint

This section contains the approximate details of PMU firmware Memory Layout and also the Memory Footprint with various modules enabled.

In PMU RAM, some part is reserved for PBR leaving around 125.7 KB for PMU firmware. The following figure shows the memory layout of PMU RAM.

Figure 47: PMU Firmware Memory Layout



In PMU firmware, only PM module is enabled by default along with Base Firmware and all the other modules are disabled. See the [PMU Firmware Build Flags](#) section to know about the default setting of a module.

Note: All the metrics are with compilation optimized for size -Os. This optimization setting is enabled by default in the Vitis IDE. To disable the same, follow the steps mentioned in [Enable/Disable Modules](#) section.

Table 54: PMU Firmware Metrics

S.No	Feature/Component	Size Occupied (KB)	Free Space (KB)	Additional Notes	Remarks
1	PMU firmware without detailed debug prints enabled	110.6	17.4	This is with base PMU firmware and PM module.	
2	PMU firmware with detailed debug prints enabled	114.5	13.5	Detailed debug prints are enabled when XPFW_DEBUG_DETAILED flag is defined.	This estimation is with combination of (1) and (2)
3	PMU firmware with Error Management Module enabled	113.6	14.4	Error Management module is enabled when ENABLE_EM and ENABLE_SCHEDULER flags are defined.	This estimation is with combination of (1) and (3)
4	PMU firmware with Restart functionality enabled	115.8	12.2	Restart functionality is enabled when ENABLE_RECOVERY, ENABLE_ESCALATION and CHECK_HEALTHY_BOOT flags are defined along with EMABLE_EM and ENABLE_SCHEDULER flags.	This estimation is with combination of (1) and (4)

Dependencies



RECOMMENDED: It is recommended to have all the software components (FSBL, PMU firmware, ATF, U-Boot and Linux) of the same release tag (e.g.: 2017.3).

Power Management Framework

Introduction

The Zynq[®] UltraScale+[™] MPSoC is the industry's first heterogeneous multiprocessor SoC (MPSoC) that combines multiple user programmable processors, FPGA, and advanced power management capabilities.

Modern power efficient designs requires usage of complex system architectures with several hardware options to reduce power consumption as well as usage of a specialized CPU to handle all power management requests coming from multiple masters to power on, power off resources and handle power state transitions. The challenge is to provide an intelligent software framework that complies to industry standard (IEEE P2415) and is able to handle all requests coming from multiple CPUs running different operating systems.

Xilinx has created the Power Management Framework (PMF) to support a flexible power management control through the platform management unit (PMU).

This Power Management Framework handles several use case scenarios. For example, Linux provides basic power management capabilities such as idle, hotplug, suspend, resume, and wakeup. The kernel relies on the underlying APIs to execute power management decisions, but most RTOSes do not have this capability. Therefore they rely on user implementation, which is made easier with use of the Power Management Framework.

Industrial applications such as embedded vision, Advanced Driver Assistance, surveillance, portable medical, and Internet of Things (IoT) are ramping up their demand for

high-performance heterogeneous SoCs, but they have a tight power budget. Some of the applications are battery operated, and battery life is a concern. Some others such as cloud and data center have demanding cooling and energy cost, not including their need to reduce environmental cost. All of these applications benefit from a flexible power management solution.

Key Features

The following are the key features of the Power Management Framework.

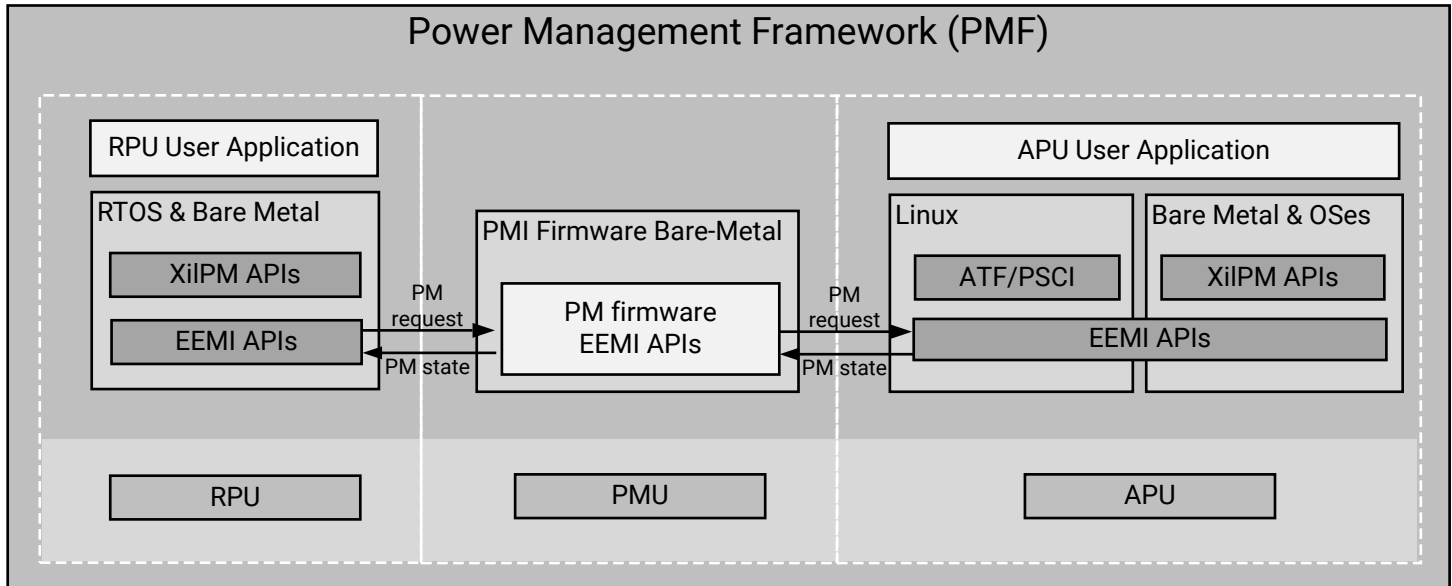
- Provides centralized power state information through use of a Power Management Unit (PMU)

- Supports Embedded Energy Management Interface (EEMI) APIs (IEEE P2415)
- Manages power state of all devices
- Provides support for Linux power management, including:
 - Linux device tree power management
 - ATF/PSCI power management support
 - Idle
 - Hotplug
 - Suspend
 - Resume
 - Wakeup process management
- Provides direct control of the following power management features with more than 24 APIs:
 - Processor unit suspend and wake up management
 - Memories and peripherals management

Power Management Software Architecture

The Zynq UltraScale+ MPSoC architecture employs a dedicated programmable unit (PMU) that controls the power-up, power-down, monitor, and wakeup mechanisms of all system resources. The customer benefits from a system that is better equipped on handling power management administration for a multiprocessor heterogeneous system. However, it is inherently more complex. The goal of the Power Management Framework is to abstract this complexity, exposing only the APIs you need to be aware of to meet your power budget goal.

Figure 48: Power Management Framework



X19504-071317

The intention of the EEMI is to provide a common API that allows all software components to power manage cores and peripherals. At a high level, EEMI allows you to specify a high-level power management goal such as suspending a complex processor cluster or just a single core. The underlying implementation is then free to autonomously implement an optimal power-saving approach.

The Linux device tree provides a common description format for each device and its power characteristics. Linux also provides basic power management capabilities such as idle, hotplug, suspend, resume, and wakeup. The kernel relies on the underlying APIs to execute power management decisions.

You can also create your own power management applications using the XilPM library, which provides access to more than 24 APIs.

Zynq UltraScale+ MPSoC Power Management Overview

The Zynq UltraScale+ MPSoC power management framework is a set of power management options, based upon an implementation of the Embedded Energy Management Interface (EEMI). The power management framework allows software components running across different processing units (PUs) on a chip or device to issue or respond to requests for power management.

Zynq UltraScale+ MPSoC Power Management Hardware Architecture

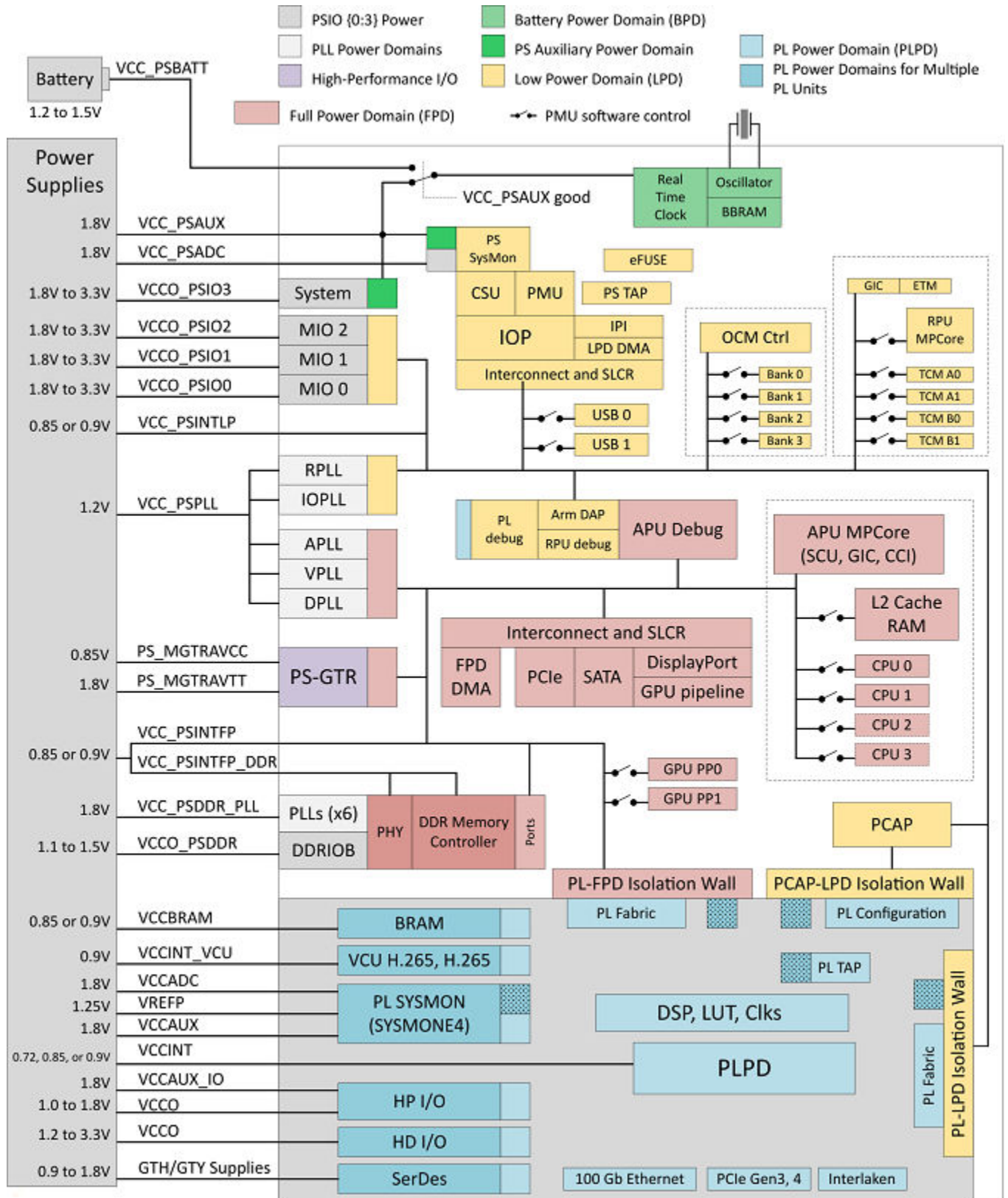
The Zynq UltraScale+ MPSoC is divided into four major power domains:

- Full power domain (FPD): Contains the Arm® Cortex™-A53 application processor unit (APU) as well as a number of peripherals typically used by the APU.
- Low power domain (LPD): Contains the Arm Cortex™-R5F real-time processor unit (RPU), the platform management unit (PMU), and the configuration security unit (CSU), as well as the remaining on-chip peripherals.
- Programmable logic (PL) power domain: Contains the PL.
- Battery-power domain: Contains the real-time clock (RTC) as well as battery-backed RAM (BBRAM).

Other power domains listed in the following figure are not actively managed by the power framework. Designs that want to take advantage of the Power Management switching of power domains must keep some power rails discrete. This allows individual rails to be powered off with the power domain switching logic. For more details, see the “PCB Power Distribution and Migration in UltraScale+ FPGAs” in the *UltraScale Architecture PCB Design User Guide* ([UG583](#)).

The following diagram illustrates the Zynq UltraScale+ MPSoC power domains and islands.

Figure 49: Zynq UltraScale+ MPSoC Power Domain and Islands



X19927-120418

Because of the heterogeneous multi-core architecture of the Zynq UltraScale+ MPSoC, no single processor can make autonomous decisions about power states of individual components or subsystems.

Instead, a collaborative approach is taken, where a power management API delegates all power management control to the platform management unit (PMU). It is the key component coordinating the power management requests received from the other processing units (PUs), such as the APU or the RPU, and the coordination and execution from other processing units through the power management API.



IMPORTANT! *In the EEMI implementation for Zynq UltraScale+ MPSoC, the platform management unit (PMU) serves as the power management controller for the different processor units (PUs), such as the APU and the RPU. These APU/RPU act as a power management (PM) master node and make power management requests. Based on those requests, the PMU controls the power states of all PM slave nodes as well as the PM masters. Unless otherwise specified, the terms "PMU" and "power management controller" are interchangeable.*

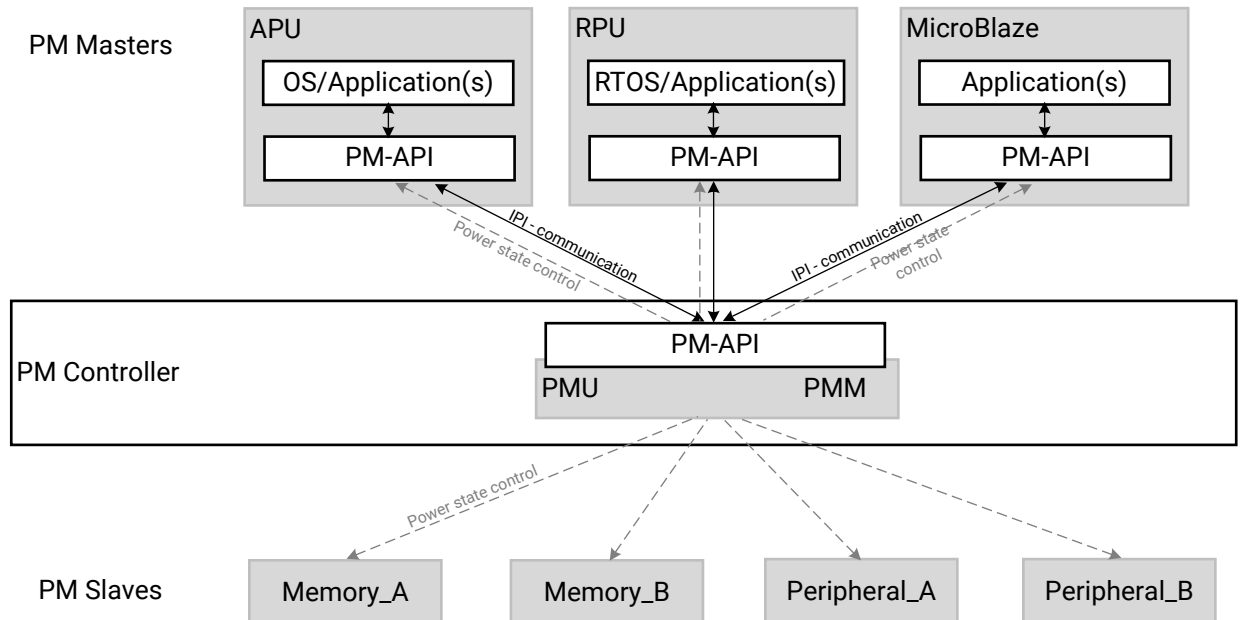
The Zynq UltraScale+ MPSoC also supports inter-processor interrupts (IPIs), which are used as the basis for power-management related communication between the different processors. See this [link](#) to the “Interrupts” chapter of the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)* for more detail on this topic.

Zynq UltraScale+ MPSoC Power Management Software Architecture

To enable multiple processing units to cooperate in terms of power management, the software framework for the Zynq UltraScale+ MPSoC provides an implementation of the power management API for managing heterogeneous multiprocessing systems.

The following figure illustrates the API-based power management software architecture.

Figure 50: API-Based Power Management Software Architecture



X19503-071317

Power Management Framework Overview

The Zynq UltraScale+ MPSoC power management framework (PMF) is based on an implementation of EEMI, see the *Embedded Energy Management Interface EEMI API Reference Guide* (UG1200). It includes APIs that consist of functions available to the processor units (PUs) to send messages to the power management controller, as well as callback functions in for the power management controller to send messages to the PUs. The APIs can be grouped into the following functional categories:

- Suspending and waking up PUs
- Slave device power management, such as memories and peripherals
- Miscellaneous
- Direct-access

API Calls and Responses

Power Management Communication using IPIs

In the Zynq UltraScale+ MPSoC, the power management communication layer is implemented using inter-processor interrupts (IPIs), provided by the IPI block. See this [link](#) to the “Interrupts” chapter of the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)* for more details on IPIs.

Each PU has a dedicated IPI channel with the power management controller, consisting of an interrupt and a payload buffer. The buffer passes the API ID and up to five arguments. The IPI interrupt to the target triggers the processing of the API, as follows:

- When calling an API function, a PU generates an IPI to the power management unit (PMU), prompting the execution of necessary power management action.
- The PMU performs each PM action atomically, meaning that the action cannot be interrupted.
- To support PM callbacks, which are used for notifications from the PMU to a PU, each PU implements handling of these callback IPIs.

Acknowledge Mechanism

The Zynq UltraScale+ MPSoC power management framework (PMF) supports blocking and non-blocking acknowledges. In most API calls that offer an acknowledge argument, the caller can choose one of the following three acknowledge options:

- REQUEST_ACK_NO: No acknowledge requested
- REQUEST_ACK_BLOCKING: Blocking acknowledge requested
- REQUEST_ACK_NON_BLOCKING: Non-blocking acknowledge using callback requested

Multiple power management API calls are serialized because each processor unit (PU) uses a single IPI channel for the API calls. After one request is sent to the power management controller, the next one can be issued only after the power management controller has completed servicing the first one. Therefore, no matter which acknowledge mechanism is used, the caller can be blocked when issuing subsequent requests.

No Acknowledge

If no acknowledge is requested (REQUEST_ACK_NO), the power management controller processes the request without returning an acknowledge to the caller, otherwise an acknowledgment is sent.

Blocking Acknowledge

After initiating a PM request with the (REQUEST_ACK_BLOCKING) specified, a caller remains blocked as long as the power management controller does not provide the acknowledgment.

The platform management unit (PMU) writes the acknowledge values into the response portion of the IPI buffer before it clears the IPI interrupt. The caller reads the acknowledge values from the IPI buffer after the IPI observation register shows that the interrupt is cleared, which is when PMU has completed servicing the issued IPI. The IPI for the PU is disabled until the PMU is ready to handle the next request.

Non-Blocking Acknowledge

After initiating a PM request with the (REQUEST_ACK_NON_BLOCKING) specified, a caller does not wait for the platform management unit (PMU) to process that request. Moreover, the caller is free to perform some other activities while waiting for the acknowledge from the PMU.

After the PMU completes servicing the request, it writes the acknowledge values into the IPI buffer. Next, the PMU triggers the IPI to the caller PU to interrupt its activities, and to inform it about the sent acknowledge.

Non-blocking acknowledges are implemented using a callback function that is implemented by the calling PU, see `XPm_NotifyCb` Callback.

For more information about `XPm_NotifyCb`, see Appendix J, XilPM Library v3.0.

Power Management Framework Layers

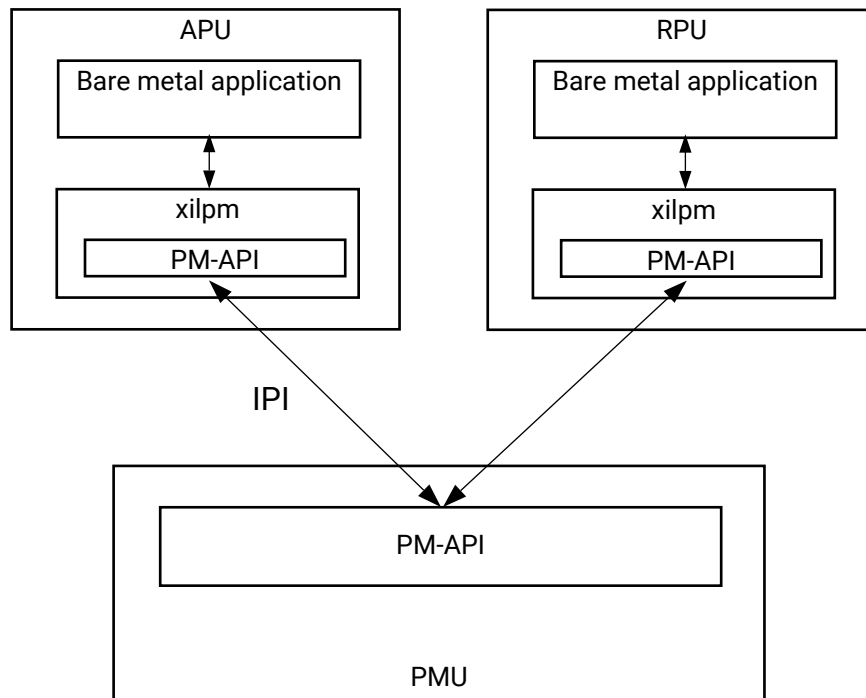
There are different API layers in the power management framework (PMF) implementation for Zynq UltraScale+ MPSoCs, which are, as follows:

- **Xilpm:** This is a library layer used for standalone applications in the different processing units, such as the APU and RPU.
- **ATF:** The Arm Trusted Firmware (ATF) contains its own implementation of the client-side PM framework. It is currently used by Linux operating systems.
- **PMU firmware:** The power management unit firmware (PMUFW) runs on the power management unit (PMU) and implements of the power management API.

For more details, see this [link](#) in the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*.

The following figure shows the interaction between the APU, the RPU, and the PMF APIs.

Figure 51: API Layers Used with Bare-Metal Applications Only

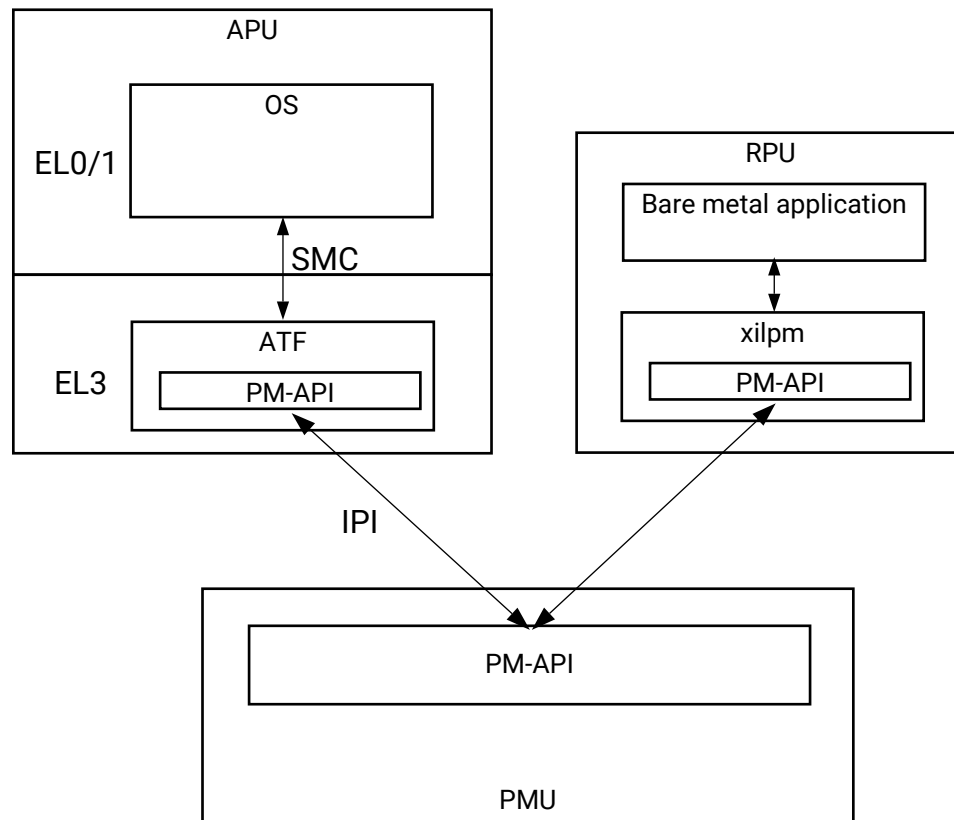


X19094-071317

If the APU is running a complete software stack with an operating system, the Xilpm library is not used. Instead, the ATF running on EL3 implements the client-side power management API, and provides a secure monitor call (SMC)-based interface to the upper layers.

The following figure illustrates this behavior. See the Armv8 manuals for more details on the Armv8 architecture and its different execution modes. It illustrates the PMF layers that are involved when running a full software stack on the APU.

Figure 52: PM Framework Layers Involved When Running a Full Software Stack on the APU



X19093-071317

Typical Power Management API Call Flow

Any entity involved in power management is referred to as a *node*. The following sections describe how the power management framework (PMF) works with slave nodes allocated to the APU and the RPU.

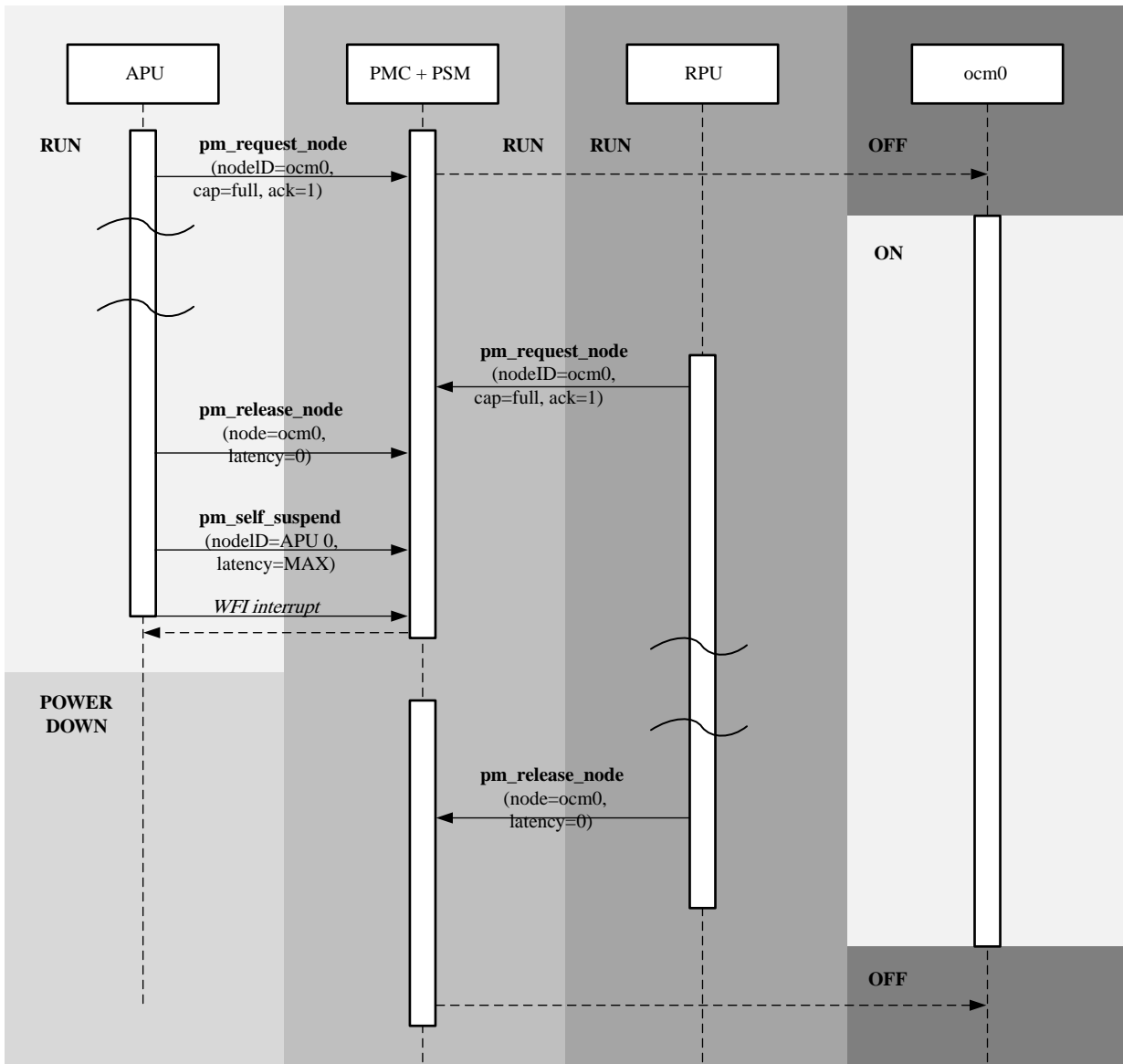
Generally, the APU or the RPU inform the power management controller about their usage of a slave node, by requesting for it. They then inform the power management controller about the capability requirement needed from the slave node. At this point, the power management controller powers up the slave node so that it can be initialized by the APU or the RPU.

Requesting and Releasing Slave Nodes

When a PU requires a slave node, either peripheral or memory, it must request that slave node using the power management API. After the slave node has performed its function and is no longer required, it may be released, allowing the slave node to be powered off.

The following figure shows the call flow for a use-case in which the APU and the RPU are sharing an OCM memory bank, ocm0.

Figure 53: PM Framework Call Sequence for APU and RPU Sharing an OCM Memory Bank



X20022-062420

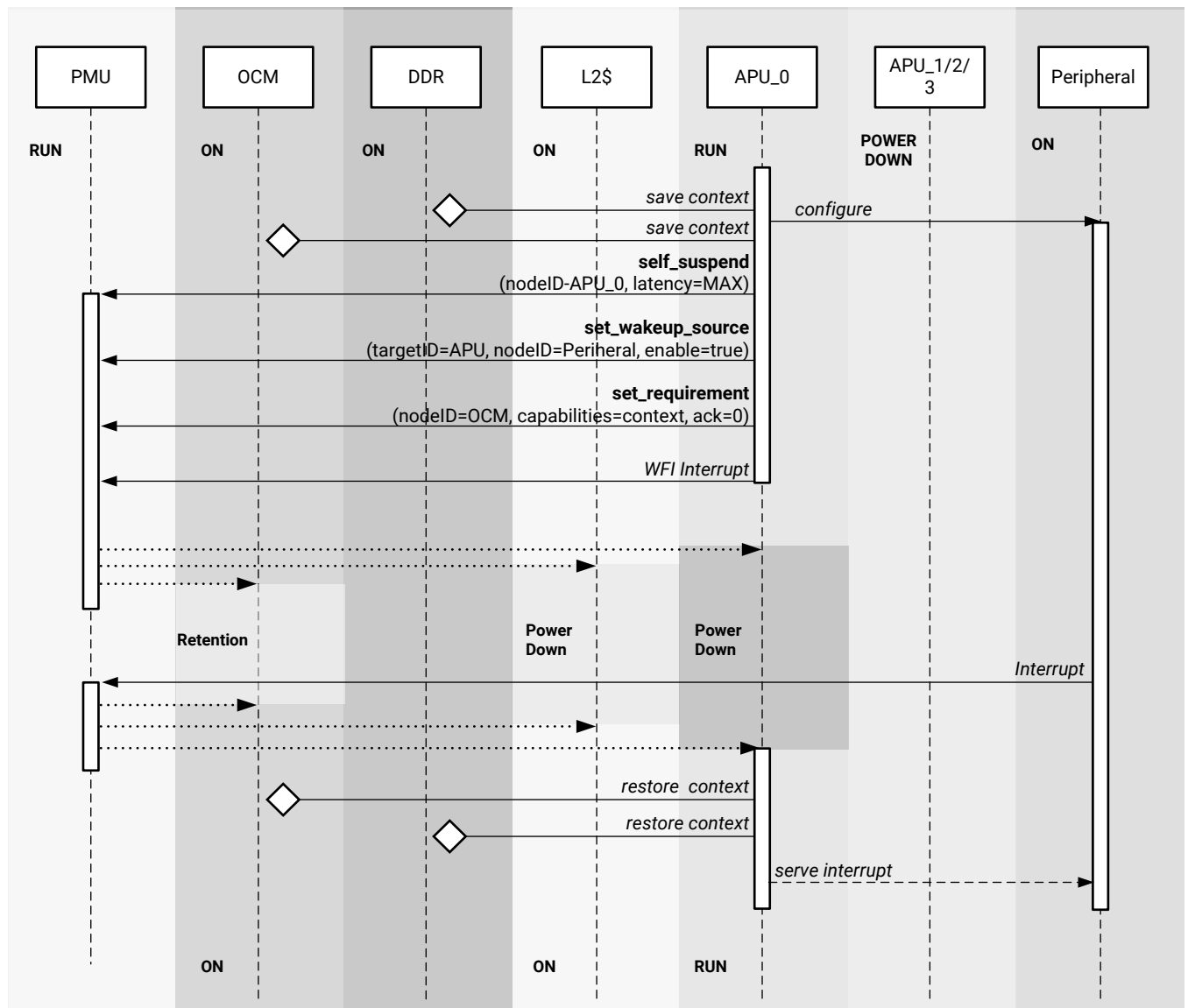
Note: The ocm0 memory remains powered on after the APU calls `XStatus XPm_ReleaseNode`, because the RPU has also requested the same slave node. It is after the RPU also releases the ocm0 node that the PMU powers off the ocm0 memory.

Processor Unit Suspend and Resume

To allow a processor unit (PU) to be powered off, as opposed to just entering an idle state, an external entity is required to take care of the power-down and power-up transitions. For the Zynq UltraScale+ MPSoC, the platform management unit (PMU) is the responsible entity for performing all power state changes.

The processor unit (PU) notifies the PMU that a power state transition is being requested. The following figure illustrates the process.

Figure 54: APU Suspend and Resume Procedure



X20023-110217

The Self-Suspending a CPU/PU section in [Implementing Power Management on a Processor Unit](#) provides more details on the suspend or resume procedure. Each PU usually depends on a number of slave nodes to be able to operate.

Sub-system Power Management

Isolation Configuration

The Zynq UltraScale+ MPSoC can be partitioned into sub-systems, so that they can be managed independently by the power management framework. For example, you can define a Linux sub-system and a Real-time sub-system. The Linux sub-system may include the APU (as the PM master) and a number of peripherals (as the PM slaves). The Real-time sub-system may include the RPU and a number of other peripherals. Each sub-system can be powered up, powered down, restarted or suspended without affecting the other sub-systems. A sub-system has only one PM Master, and may include both FPD and LPD peripherals.

You can create your own sub-systems using the Vivado PCW tool. The following figure shows the PCW screen shots of a valid configuration, which contains only an APU sub-system and no RPU sub-systems.

Figure 55: PCW Configuration

Please review **Known Limitations** under the **Isolation Configuration** Section of **PG201**.

Enable Isolation
 Enable Secure Debug
 Lock Unused Memory

Search:

Name	Start Address	Size	Unit	TZ Settings	Access Settl...	End Address	Type
LINUX							
Masters							
SD1				Secure			
GEM3				NonSecure			
APU							
PCIe				NonSecure			
DP				NonSecure			
GPU				NonSecure			
Coresight							
SATA0				NonSecure			
SATA1				NonSecure			
USB0				NonSecure			
FPD_DMA				NonSecure			
DAP							
QSPI				NonSecure			
Slaves							
Memory							
DDR_LOW	0x0	2	GB	NonSecure	Read/Write	0x7FFFFFFF	DDR
QSPI_Linear...	0xC0000000	524288	KB	NonSecure	Read/Write	0xDFFFFFFF	LPD
Peripherals							
CAN1	0xFF070000	64	KB	NonSecure	Read/Write	0xFF07FFFF	LPD
GEM3	0xFF0E0000	64	KB	NonSecure	Read/Write	0xFF0EFFFF	LPD
GPIO	0xFF0A0000	64	KB	NonSecure	Read/Write	0xFF0AFFFF	LPD
I2C0	0xFF020000	64	KB	NonSecure	Read/Write	0xFF02FFFF	LPD
I2C1	0xFF030000	64	KB	NonSecure	Read/Write	0xFF03FFFF	LPD
SWDT0	0xFF150000	64	KB	NonSecure	Read/Write	0xFF15FFFF	LPD
TTC0	0xFF110000	64	KB	NonSecure	Read/Write	0xFF11FFFF	LPD
UART0	0xFF000000	64	KB	NonSecure	Read/Write	0xFF00FFFF	LPD
UART1	0xFF010000	64	KB	NonSecure	Read/Write	0xFF01FFFF	LPD
TTC1	0xFF120000	64	KB	NonSecure	Read/Write	0xFF12FFFF	LPD
TTC2	0xFF130000	64	KB	NonSecure	Read/Write	0xFF13FFFF	LPD
TTC3	0xFF140000	64	KB	NonSecure	Read/Write	0xFF14FFFF	LPD
Control and Status Registers							

Figure 56: PCW Configuration Contd

Name	Start Address	Size	Unit	TZ Settings	Access Setti...	End Address	Type
LINUX							
> Masters							
> Slaves							
> Memory							
> Peripherals							
> Control and Status Registers							
USB3_0	0xFF9D0000	64	KB	NonSecure	Read/Write	0xFF9DFFFF	LPD
USB3_0_XHCI	0xFE200000	1024	KB	NonSecure	Read/Write	0xFE2FFFFFFF	LPD
Coresight	0xFE800000	8192	KB	NonSecure	Read/Write	0xFEFFFFFFF	LPD
LPD_DMA_0	0xFFA80000	64	KB	NonSecure	Read/Write	0xFFA8FFFF	LPD
LPD_DMA_1	0xFFA90000	64	KB	NonSecure	Read/Write	0xFFA9FFFF	LPD
LPD_DMA_2	0xFFAA0000	64	KB	NonSecure	Read/Write	0xFFAAFFFF	LPD
LPD_DMA_3	0xFFAB0000	64	KB	NonSecure	Read/Write	0xFFABFFFF	LPD
LPD_DMA_4	0xFFAC0000	64	KB	NonSecure	Read/Write	0xFFACFFFF	LPD
LPD_DMA_5	0xFFAD0000	64	KB	NonSecure	Read/Write	0xFFADFFFF	LPD
LPD_DMA_6	0xFFAE0000	64	KB	NonSecure	Read/Write	0xFFAEFFFF	LPD
LPD_DMA_7	0xFFAF0000	64	KB	NonSecure	Read/Write	0xFFAFFFFF	LPD
QSPI	0xFF0F0000	64	KB	NonSecure	Read/Write	0xFF0FFFFF	LPD
SD1	0xFF170000	64	KB	NonSecure	Read/Write	0xFF17FFFF	LPD
AMS	0xFFA50000	64	KB	NonSecure	Read/Write	0xFFA5FFFF	LPD
APM1	0xFFA00000	64	KB	NonSecure	Read/Write	0xFFA0FFFF	LPD
APM2	0xFFA10000	64	KB	NonSecure	Read/Write	0xFFA1FFFF	LPD
APM_FPD_LPD	0xFFA30000	64	KB	NonSecure	Read/Write	0xFFA3FFFF	LPD
APM_INTC_IQU	0xFFA20000	64	KB	NonSecure	Read/Write	0xFFA2FFFF	LPD
IOU_GPV	0xFE000000	1024	KB	NonSecure	Read/Write	0xFE0FFFFF	LPD
IPI_CTRL	0xFF380000	512	KB	NonSecure	Read/Write	0xFF3FFFFFFF	LPD
LPD_GPV	0xFE100000	1024	KB	NonSecure	Read/Write	0xFE1FFFFFFF	LPD
RTC	0xFFA60000	64	KB	NonSecure	Read/Write	0xFFA6FFFF	LPD

Figure 57: PCW Configuration Contd

v APU_secure								
v Masters								
SD1					Secure			
APU								
v Slaves								
v Memory								
OCM	0xFFC0000	256	KB		Secure	Read/Write	0xFFFFFFFF	OCM
v Control and Status Registers								
CRF_APB	0xFD1A0000	1280	KB		Secure	Read/Write	0xFD2DFFFF	FPD
CRL_APB	0xFF5E0000	2560	KB		Secure	Read/Write	0xFF85FFFF	LPD
EFUSE	0xFFCC0000	64	KB		Secure	Read/Write	0xFFCCFFFF	LPD
IOU_SLCR	0xFF180000	768	KB		Secure	Read/Write	0xFF23FFFF	LPD
v PMU Firmware								
v Masters								
PMU								
v Slaves								
v Peripherals								
UART0	0xFF000000	64	KB		NonSecure	Read/Write	0xFF00FFFF	LPD
v Control and Status Registers								
CRF_APB	0xFD1A0000	1280	KB		Secure	Read/Write	0xFD2DFFFF	FPD
DDR_XMPU0...	0xFD000000	64	KB		Secure	Read/Write	0xFD00FFFF	FPD
DDR_XMPU1...	0xFD010000	64	KB		Secure	Read/Write	0xFD01FFFF	FPD
DDR_XMPU2...	0xFD020000	64	KB		Secure	Read/Write	0xFD02FFFF	FPD
DDR_XMPU3...	0xFD030000	64	KB		Secure	Read/Write	0xFD03FFFF	FPD
DDR_XMPU4...	0xFD040000	64	KB		Secure	Read/Write	0xFD04FFFF	FPD
DDR_XMPU5...	0xFD050000	64	KB		Secure	Read/Write	0xFD05FFFF	FPD
FPD_SLCR	0xFD610000	512	KB		Secure	Read/Write	0xFD68FFFF	FPD
FPD_XMPU...	0xFD5D0000	64	KB		Secure	Read/Write	0xFD5DFFFF	FPD
LPD_XPPU	0xFF980000	64	KB		Secure	Read/Write	0xFF98FFFF	LPD
CRL_APB	0xFF5E0000	2560	KB		Secure	Read/Write	0xFF85FFFF	LPD
EFUSE	0xFFCC0000	64	KB		Secure	Read/Write	0xFFCCFFFF	LPD
IOU_SLCR	0xFF180000	768	KB		Secure	Read/Write	0xFF23FFFF	LPD
LPD_SLCR	0xFF410000	640	KB		Secure	Read/Write	0xFF4AFFFF	LPD
OCM_XMPU...	0xFFA70000	64	KB		Secure	Read/Write	0xFFA7FFFF	LPD
RPU	0xFF9A0000	64	KB		Secure	Read/Write	0xFF9AFFFF	LPD

Note: The PCW tool is also used to isolate some peripherals from each other for security purposes. See *Zynq UltraScale+ MPSoC: Embedded Design Tutorial (UG1209)* and *Zynq UltraScale+ MPSoC Processing System LogiCORE IP Product Guide (PG201)* for details on how to set up isolation between peripherals.

Configuration Object

The sub-system configuration is captured in a Configuration Object, which is generated by the Vivado and PetaLinux toolchain. The Configuration Object contains:

- The PM Masters that are present in the system (APU and/or RPU). Any PM Master not specified in the Configuration Object will be powered down by the PMU.
- Configurable permissions for each PM Master, such as:
 - Which PM Master can use which PM Slave (A PM Master can use all the PM Slaves that belong in the same sub-system.)
 - Access to MMIO address regions.
 - Access to peripheral reset lines.

- **Pre-allocated PM Slaves.** The PM Master can use these PM Slaves without requesting for them first. These PM Slaves are needed by the PM Master in order to boot. The toolchain makes sure that the APU can access the L2 cache and DDR banks without first requesting for them. The same is true for the RPU accessing all the TCM banks.

During boot, the Configuration Object is passed from the FSBL to the PMU firmware. For more details, see the [Configuration Object](#).

Note: Isolation is not required for the Configuration Object to be created. You can create subsystems to customize the Configuration Object and then uncheck the isolation checkbox.

Power Management Initialization

Power management is disabled during boot and all the peripherals are powered up at this time. That is because it is often necessary to allow for possible, and temporary, inter-dependencies between peripherals during boot and initialization. When FSBL is finished with initializing the peripherals and loading the application binaries, it passes the Configuration Object to the PMU. The PMU is now aware of all the sub-systems and their associated PM Masters and PM Slaves. PM Masters and PM Slaves that are not included in the Configuration Object are never used, and are powered down by the PMU.

A PM Master is not likely to use all the PM Slaves at all times. Therefore, a PM Slave should be powered up only when it is being used. The PM Master must notify the PMU before and after using a PM Slave. This functionality is implemented in the PetaLinux kernel. This requirement hinders developers starting with a new RPU application, when the focus is on functionality and not power optimization. Therefore, it is convenient for the PMU to also support PM-incapable Masters that do not provide notifications when they are using the PM Slaves. This is done by keeping all the PM Slaves in the sub-system powered up until the PM Master sends the `PmInitFinalize` request to the PMU. A PM-incapable Master will never send this request, which means that its PM Slaves will remain powered up at all times or until this PM Master itself is powered down.

A PM-capable Master sends this request after initializing the sub-system. The PMU then begins powering down the PM Slaves in this sub-system whenever they are not being used.

As a result, when there is an RPU master present in the system but it is not running any application, the PMU firmware will consider it as a PM incapable master and hence will never power down the RPU and its slaves. From the 2018.3 release and onwards, this behavior is fixed and allows you to power down unused RPUs. This change is protected by the compilation flag `ENABLE_UNUSED_RPU_PWR_DWN` and is enabled by default. When this flag is enabled, the unused RPU and allocated slaves will be powered down if not in use.

Note: If you do not want to power down RPU by default, set the `ENABLE_UNUSED_RPU_PWR_DWN` flag to 0 while compiling the PMU firmware. For the JTAG boot mode there is no impact on behavior change even though `ENABLE_UNUSED_RPU_PWR_DWN` flag is 1.

Note: Sub-systems may overlap each other. This means that some PM Slaves may belong to more than one sub-system (for example, DDR, OCM, and so on). If a PM Slave is in more than one sub-system, the PMU does not power down this PM Slave until it has been released by all its PM Masters, or until all these PM Masters have powered down themselves.

Default Configuration

By default, Isolation Configuration is disabled, and the tool chain generates a configuration with three sub-systems. Each has a PM Master: APU, R5-0 and R5-1. All three sub-systems contain all the PM Slaves (meaning that the sub-systems completely overlap each other.) This is the default configuration generated by PCW when the “Enable Isolation” box is unchecked. The default PetaLinux kernel configuration is PM-capable, but R5-0 and R5-1 must be also running “PM-capable” applications, or be powered down. Otherwise, the PMU will not power down any PM Slaves.

Note: You can create a configuration that does not allow the processors to boot and run. If you are a beginner, use the APU-only configuration as described in Isolation Configuration section and customize it as necessary.

RPU Lock-step vs. Split Mode

The toolchain infers the RPU run modes from the PCW Isolation Configuration as follows:

- No RPU present in any subsystem: Configuration Object contains no RPU.
- Only R5-0 present in subsystem(s): Configuration Object contains R5-0 running in lock-step mode.
- Both R5-0 and R5-1 in subsystems: Configuration Object contains R5-0 and R5-1 running in split mode.
- Only R5-1 present in subsystem(s): Configuration Object contains R5-1 running in split mode.

The default Configuration Object contains two RPU PM Masters: R5-0 and R5-1, and the PMU assumes that the R5-0 and R5-1 are running in split mode. However, the boot image actually determines whether the RPU runs in lock-step or split mode at boot time. The RPU run mode from the boot image must match the number of RPU PM Masters in the Configuration Object. Otherwise, the power management framework will not work properly.

Note: If you intend to use the R5 in lock-step mode, you need to ensure that the Isolation Configuration is enabled in PCW, and only R5-0 (not R5-1) is present in a subsystem.

Sharing Devices

Sharing access to devices between APU and RPU is possible but must always be done with great care. The access and operation of a device depend on its clock (if applicable), its configuration and its power state (on, off, retention, and so on.) The PMU makes sure the device is in the lowest power state that will satisfy the requirement of all the PM Masters, but it is up to the APU and RPU to set up the clock and configuration of the device.

Extra care must be taken when a device is shared between the APU running Linux and the RPU. Linux is not aware that another entity might be using one of its devices, and will clock-gate, power-gate and disable the device whenever it is not being used. The options available are:

- Disable Linux runtime power management of the device. See <https://www.kernel.org/doc/Documentation/ABI/testing/sysfs-devices-power>. This will keep the device running even when Linux is not using it, but the device will still be clock-gated and disabled when Linux goes to sleep.
- Implement a special driver for the device.

Any devices not used by the APU should be removed from the device tree.

Using the API for Power Management

This chapter contains detailed instructions on how to use the Xilinx® power management framework (PMF) APIs to carry out common power management tasks.

Implementing Power Management on a Processor Unit

The Xilpm library provides the functions that the standalone applications executing on a processor can use to initiate the power management API calls.

See the *SDK Online Help* ([UG782](#)) for information on how to include the Xilpm library in a project.

Initializing the Xilpm Library

Before initiating any power management API calls, you must initialize the Xilpm library by calling `XPm_InitXilpm`, and passing a pointer to a properly initialized inter-processor interrupt (IPI) driver instance.

See this [link](#) to the “Interrupts” chapter of the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)). for more information regarding IPIs.

For more information about `XPm_InitXilpm`, see Appendix J, XilPM Library v3.0.

Working with Slave Devices

The Zynq UltraScale+ MPSoC power management framework (PMF) contains functions dedicated to managing slave devices (also referred to as PM slaves), such as memories and peripherals. Processor units (PUs) use these functions to inform the power management controller about the requirements (such as capabilities and wake-up latencies) for those devices. The power management controller manages the system so that each device resides in the lowest possible power state, meeting the requirements from all eligible PUs.

Requesting and Releasing a Node

A PU uses the `XPm_RequestNode` API to request the access to a slave device and assert its requirements on that device. The power management controller manages the requested device's power-on and active state, provided the PU and the slave belong to the same sub-system.

After a device is no longer used, the PU typically calls the `XPm_ReleaseNode` function to allow the PM controller to re-evaluate the power state of that device, and potentially place it into a low-power state. It also then allows other PUs to request that device.

For more information about `XPm_ReleaseNode`, see Appendix J, XilPM Library v3.0.

Changing Requirements

When a PU is using a PM slave, its requirement on the slave's capability may change. For example, an interface port may go into a low power state, or even be completely powered off, if the interface is not being used. The PU may use `XPm_SetRequirement` to change the capability requirement of the PM slave. Typically, the PU would not release the PM slave if it will be changing the requirement again in the future.

The following example call changes the requirement for the node argument to require wake-interrupts only:

```
XPm_SetRequirement(node, PM_CAP_WAKEUP, 0, REQUEST_ACK_NO);
```



IMPORTANT! *Setting requirements of a node to zero is not equivalent to releasing the PM slave. By releasing the PM slave, a PU may be allowing other PUs to use the device exclusively.*

When multiple PUs share a PM slave (this applies mostly to memories), the power management controller selects a power state of the PM slave that satisfies all requirements of the requesting PUs.

The requirements on a PM slave include capability as well as latency requirements. Capability requirements may include a top capability state, some intermediate capability states, an inactive state (but with the configuration retained), and the off state. Latency requirement specifies the maximum time allowed for the PM slave to switch to the top capability state from any other state. If this time limit cannot be met, the power management controller will leave the PM slave in the top capability state regardless of other capability requirements.

For more information about `XPM_SetRequirement`, see Appendix J, XilPM Library v3.0.

Self-Suspending a CPU/PU

A PU can be a cluster of CPUs. The APU is a PU, that has four CPUs. An RPU has two CPUs, but it is considered as two PUs when running in the split mode, and one PU when it is running in the lock-step mode.

To suspend itself, a CPU must inform the power management controller about its intent by calling the `XPM_SelfSuspend` function. The following actions then occur:

- After the `XPm_SelfSuspend()` call is processed, none of the future interrupts can prevent the CPU from entering a sleep state. To manage such behavior in the case of the APU and RPU, after the `XPm_SelfSuspend()` call has completed, all of the interrupts to a CPU are directed to the power management controller as GIC wake interrupts.
- The power management controller then waits for the CPU to finalize the suspend procedure. The PU informs the power management controller that it is ready to enter a sleep state by calling `XPm_SuspendFinalize`.
- The `XPm_SuspendFinalize()` function is architecture-dependent. It ensures that any outstanding power management API call is processed, then executes the architecture-specific suspend sequence, which also signals the suspend completion to the power management controller.
- For Arm® processors such as the APU and RPU, the `XPm_SuspendFinalize()` function uses the wait for interrupt (WFI) instruction, which suspends the CPU and triggers an interrupt to the power management controller.
- When the suspend completion is signaled to the power management controller, the power management controller places the CPU into reset, and may power down the power island of the CPU, provided that no other component within the island is currently active.
- Interrupts enabled through the GIC interface of the CPU are redirected to the power management controller (PMC) as a GIC wake interrupt assigned to that particular CPU. Because the interrupts are redirected, the CPU can only be woken up using the power management controller.
- Suspending a PU requires suspending all of its CPUs individually.

For more information about `XPM_SelfSuspend` and `XPm_SuspendFinalize`, see Appendix J, XilPM Library v3.0.

Resuming Execution

A CPU can be woken up either by a wake interrupt triggered by a hardware resource or by an explicit wake request using the `XPM_RequestWakeup` API.

The CPU starts executing from the resume address provided with the `XPm_SelfSuspend` call.

For more information about `XPM_RequestWakeup` and `XPm_SelfSuspend`, see Appendix J, XilPM Library v3.0.

Setting up a Wake-up Source

The power management controller can power down the entire FPD if none of the FPD devices are in use and existing latency requirements allow this action. If the FPD is powered off and the APU is to be woken up by an interrupt triggered by a device in the LPD, the GIC Proxy must be configured to allow propagation of FPD wake events. The APU can ensure this by calling `XPM_SetWakeUpSource` for all devices that might need to issue wake interrupts.

Hence, prior to suspending, the APU must call `XPm_SetWakeUpSource(NODE_APU, node, 1)` to add the required slaves as a wake-up source. The APU can then set the requirements to zero for all slaves it is using. After the APU finalizes its suspend procedure, and provided that no other PU is using any resource in the FPD, the PM controller powers off the entire FPD and configures the GIC proxy to enable propagation of the wake event of the LPD slaves.

For more information about `XPm_SetWakeUpSource`, see Appendix J, XilPM Library v3.0.

Aborting a Suspend Procedure

If a PU decides to abort the suspend procedure after calling the `XPm_SetSelfSuspend` function, it must inform the power management controller about the aborted suspend by calling the `XPm_AbortSuspend` function.

For more information about `XPm_SetSelfSuspend` and `XPm_AbortSuspend`, see Appendix J, XilPM Library v3.0.

Handling PM Slaves During the Suspend Procedure

A PU that suspends itself must inform the power management controller about its changed requirements on the peripherals and memories in use. If a PU fails inform the power management controller, all of the used devices remain powered on. Typically, for memories you must ensure that their context is preserved by using the following function:

```
XPm_SetRequirement(node, PM_CAP_CONTEXT, 0, REQUEST_ACK_NO);
```

When setting requirements for a PM slave during the suspend procedure; after calling `XPm_SelfSuspend`, the setting is deferred until the CPU finishes the suspend. This deference ensures that devices that are needed for completing the suspend procedure can enter a low power state after the calling CPU finishes suspend.

A common example is instruction memory, which a CPU can access until the end of a suspend. After the CPU suspends a memory, that memory can be placed into retention. All deferred requirements reverse automatically before the respective CPU is woken up.

When an entire PU suspends, the last awake CPU within the PU must manage the changes to the devices.

For more information about `XPm_SelfSuspend`, see Appendix J, XilPM Library v3.0.

Example Code for Suspending an APU/RPU

There the following is an example of source code for suspending the APU or RPU:

```

/* Base address of vector table (reset-vector) */ extern void
*_vector_table;
/* Inform PM controller that APU_0 intends to suspend */
XPm_SelfSuspend(NODE_APU_0, MAX_LATENCY, 0, (u64)&_vector_table);
/**
 *      Set requirements for OCM banks to preserve their context.
 *      The PM controller will defer putting OCMs into retention until the
 *      suspend is finalized
 */
XPm_SetRequirement(NODE_OCM_BANK_0, PM_CAP_CONTEXT, 0, REQUEST_ACK_NO);
XPm_SetRequirement(NODE_OCM_BANK_1, PM_CAP_CONTEXT, 0, REQUEST_ACK_NO);
XPm_SetRequirement(NODE_OCM_BANK_2, PM_CAP_CONTEXT, 0, REQUEST_ACK_NO);
XPm_SetRequirement(NODE_OCM_BANK_3, PM_CAP_CONTEXT, 0, REQUEST_ACK_NO);

/* Flush data cache */ Xil_DCacheFlush();
/* Inform PM controller that suspend procedure is completed */
XPm_SuspendFinalize();
    
```

Suspending the Entire FPD Domain

To power-down the entire full power domain, the power management controller must suspend the APU at a time when none of the FPD devices is in use. After this condition is met, the power management controller can power-down the FPD automatically. The power management controller powers down the FPD if no latency requirements constrain this action, otherwise the FPD remains powered on.

Forcefully Powering Down the FPD

There is the option to force the FPD to power-down by calling the function `XPm_ForcePowerdown`. This requires that the requesting PU has proper privileges configured in the power management controller. The power management controller releases all PM Slaves used by the APU automatically.

Note: This force method is typically not recommended, especially when running complex operating systems on the APU because it could result in loss of data or system corruption, due to the OS not suspending itself gracefully.



IMPORTANT! Use the `XPm_RequestSuspend` API.

For more information about `XPm_ForcePowerdown`, see Appendix J, XilPM Library v3.0.

Interacting with Other Processing Units

Suspending a PU

A PU can request that another PU be suspended by calling `XPm_RequestSuspend`, and passing the targeted node name as an argument.

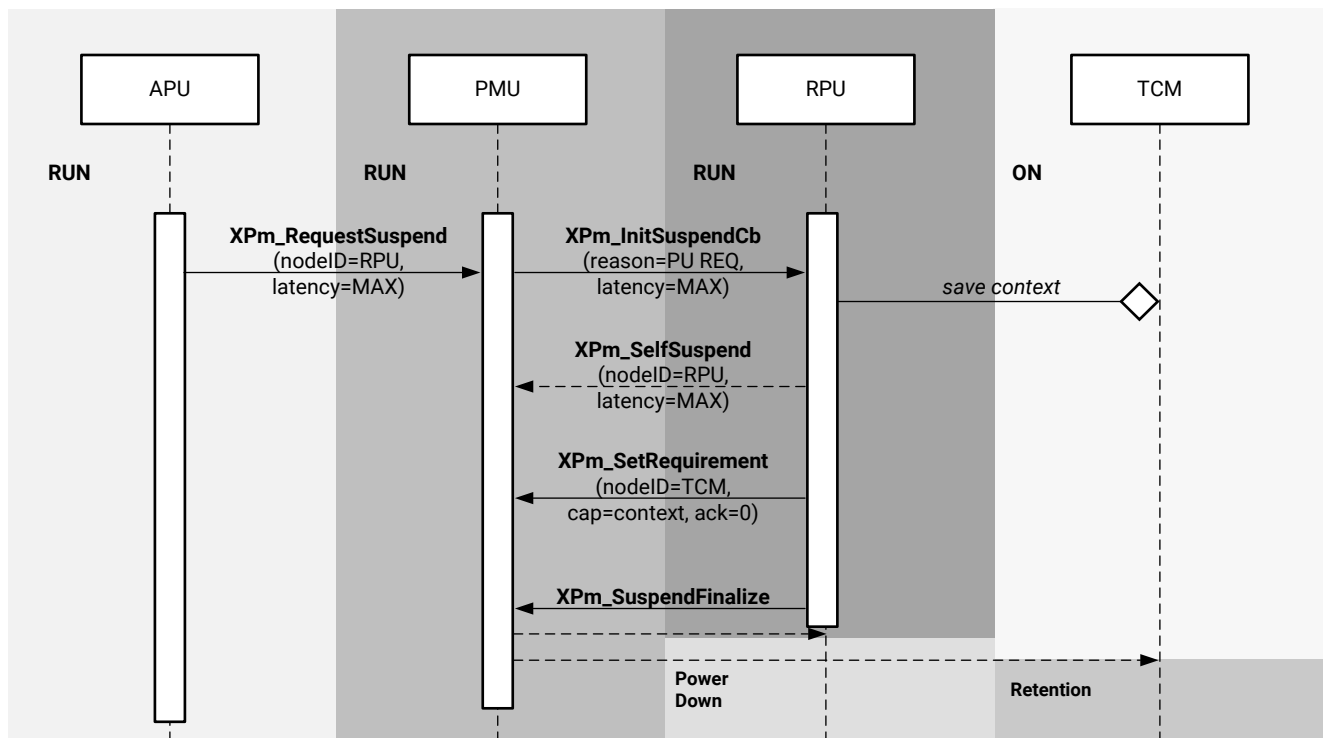
This causes the power management controller to call `XPm_InitSuspendCb()`, which is a callback function implemented in the target PU. The target PU then initiates its own suspend procedure, or call `XPm_AbortSuspend` and specify the abort reason. For example, you can request an APU to suspend with the following command:

```
XPm_RequestSuspend(NODE_APU, REQUEST_ACK_NON_BLOCKING, MAX_LATENCY, 0);
```

The following diagram shows the general sequence triggered by a call to the `XPm_RequestSuspend`.

For more information about `XPm_RequestSuspend`, `XPm_InitSuspendCb`, and `XPm_AbortSuspend`, see Appendix J, XilPM Library v3.0.

Figure 58: APU initiating suspend for the RPU by calling `XPm_RequestSuspend`



X20024-110217

Waking a PU

Additionally, a PU can request the wake-up of one of its CPUs or of another PU by calling `XPm_RequestWakeup`.

- When processing the call, the power management controller causes a target CPU or PU to be awakened.
- If a PU is the target, only one of its CPUs is woken-up by this request.
- The CPU chosen by the power management controller is considered the primary CPU within the PU.

The following is an example of a wake-up request:

```
XPm_RequestWakeup(NODE_APU_1, REQUEST_ACK_NO);
```

For more information about `XPm_RequestWakeup`, see Appendix J, XilPM Library v3.0.

XilPM Implementation Details

The system layer of the PM framework is implemented on the Zynq UltraScale+ MPSoC using inter-processor interrupts (IPIs). To issue an EEMI API call, a PU will write the API data (API ID and arguments) into the IPI request buffer and then trigger the IPI to the PMU.

After the PM controller processes the request it will send the acknowledge depending on the particular EEMI API and provided arguments.

Payload Mapping for API Calls to PMU

Each EEMI API call is uniquely identified by the following data:

- EEMI API identifier (ID)
- EEMI API arguments

Please see Appendix A for a list of all API identifiers as well as API argument values.

Prior to initiating an IPI to the PMU, the PU shall write the information about the call into the IPI request buffer. Each data written into the IPI buffer is a 32-bit word. Total size of the payload is six 32-bit words - one word is reserved for the EEMI API identifier, while the remaining words are used for the arguments. Writing to the IPI buffer starts from offset zero. The information is mapped as follows:

- Word [0] EEMI API ID
- Word [1:5] EEMI API arguments

The IPI response buffer is used to return the status of the operation as well as up to 3 values.

- Word [0] success or error code
- Word [1:3] value 1..3

Payload Mapping for API Callbacks from the PMU

The EEMI API includes callback functions, invoked by the PM controller, sent to a PU.

- Word [0]EEMI API Callback ID
- Word [1:5]EEMI API arguments

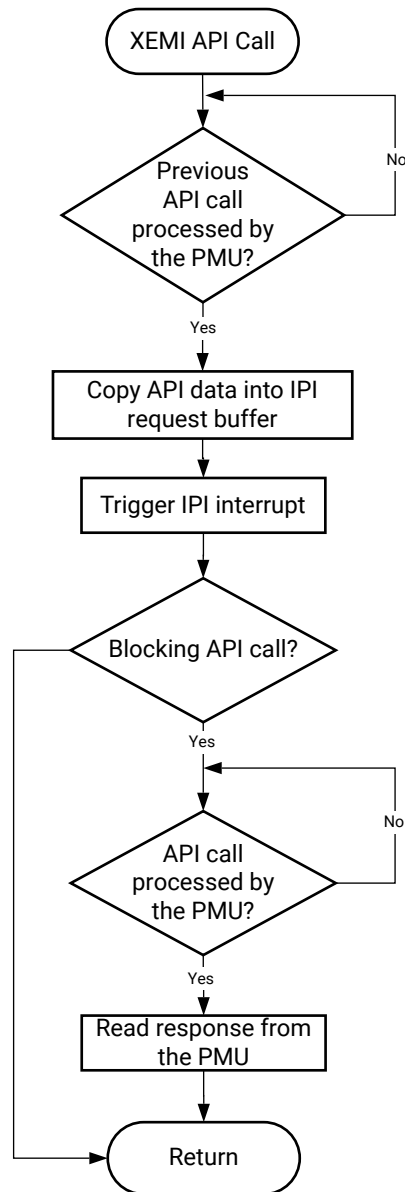
Refer to Appendix J, XiIPM Library v3.0 for a list of all API identifiers as well as API argument values.

Issuing EEMI API calls to the PMU

Before issuing an API call to the PMU, a PU must wait until its previous API call is processed by the PMU. A check for completion of a PMU action can be implemented by reading the corresponding IPI observation register.

An API call is issued by populating the IPI payload buffer with API data and triggering an IPI interrupt to the PMU. In case of a blocking API call, the PMU will respond by populating the response buffer with the status of the operation and up to 3 values. See Appendix B for a list of all errors that can be sent by the PMU if a PM operation was unsuccessful. The PU must wait until the PMU has finished processing the API call prior to reading the response buffer, to ensure that the data in the response buffer is valid.

Figure 59: Example Flow of Issuing API Call to the PMU



X19506-071017

Handling API callbacks from the PMU

The PMU invokes callback functions to the PU by populating the IPI buffers with the API callback data and triggering an IPI interrupt to the PU. In order to receive such interrupts, the PU must properly initialize the IPI block and interrupt controller. A single interrupt is dedicated to all callbacks. For this reason, element 0 of the payload buffer contains the API ID, which the PU should use to identify the API callback. The PU should then call the respective API callback function, passing in the arguments obtained from locations 1 to 4 of the IPI request buffer.

An implementation of this behavior can be found in the XilPM library.

Linux

Linux executes on the EL1 level, and the communication between Linux and the ATF software layer is realized using SMC calls.

Power management features based on the EEMI API have been ported to the Linux kernel, ensuring that the Linux-centric power management features utilize the EEMI services provided by the PMU.

Additionally, the EEMI API can be access directly via debugfs for debugging purposes. Note that direct access to the EEMI API through debugfs will interfere with the kernel power management operations and may cause unexpected problems.

All the Linux power management features presented in this chapter are available in the PetaLinux default configuration.

User Space PM Interface

System Power States

You may request to change the power state of a system or the entire system. The PMU facilitates the switching of the system or sub-system to the new power state.

Shutdown

You may shutdown the APU sub-system with the standard 'shutdown' command.

To shut down the entire system, the user must shut down all the other sub-systems prior to shutting down the APU sub-system. For example, use the following command to power down the PL.

```
echo pm_release_node 69 > /sys/kernel/debug/zynqmp-firmware/pm
```

Use this command to power up the PL again:

```
echo pm_request_node 69 > /sys/kernel/debug/zynqmp-firmware/pm
```

For information about how to shut down the PL sub-system, see the *Libmetal and OpenAMP for Zynq Devices User Guide* ([UG1186](#)).

Reboot

You can use the reboot command to reset the APU, the PS or the System. By default, the reboot command resets the system. You can change the scope of the reboot command to APU or PS if required. To change the reboot scope to APU:

```
echo subsystem > /sys/firmware/zynqmp/shutdown_scope
```

To change the reboot scope to PS:

```
echo ps_only > /sys/firmware/zynqmp/shutdown_scope
```

To change the reboot scope to System:

```
echo system > /sys/firmware/zynqmp/shutdown_scope
```

The reboot scope is set to System again after the reset.

Suspend

The kernel is suspended when the CPU and most of the peripherals are powered down. The system run states needed to resume from suspend is stored in the DRAM, which is put into self-refresh mode.

Kernel configurations required:

- Power management options
 - [*] Suspend to RAM and standby
 - [*] User space wakeup sources interface
 - [*] Device power management core functionality
- Device Drivers
 - SoC (System On Chip) specific Drivers
 - Xilinx SoC drivers
 - Zynq MPSoC SoC
 - [*] Enable Xilinx Zynq MPSoC Power Management driver
 - [*] Enable Zynq MPSoC generic PM domains
- Firmware Drivers
 - Zynq MPSoC Firmware Drivers
 - -* Enable Xilinx Zynq MPSoC firmware interface

Note: Any device can prevent the kernel from suspending.

See also https://wiki.archlinux.org/index.php/Power_management/Suspend_and_hibernate.

To suspend the kernel:

```
$ echo mem > /sys/power/state
```

Wake-up Source

The kernel resumes from the suspend mode when a wake-up event occurs. The following wake-up sources can be used:

- UART

If enabled as a wake-up source, a UART input will trigger the kernel to resume from the suspend mode.

Kernel configurations required:

- Same as Suspend.

For example, to wake up the APU on UART input:

```
$ echo enabled > /sys/devices/platform/amba/ff000000.serial/tty/ttyPS0/power/wakeup
```

- RTC

If enabled as a wake-up source, the kernel will resume from the suspend mode when the RTC timer expires. Note that the RTC wake-up source is enabled by default.

Kernel configurations required:

- Same as Suspend.

For example, to set RTC to wake up the APU after 10 seconds:

```
$ echo +10 > /sys/class/rtc/rtc0/wakealarm
```

- GPIO

If enabled as a wake-up source, a GPIO event will trigger the kernel to resume from the suspend mode.

Kernel configurations required:

- Device Drivers

- Input device support, [*]

Generic input layer (needed for keyboard, mouse, ...) (INPUT [=y]) [*] Keyboards (INPUT_KEYBOARD [=y])

[*] GPIO Buttons (CONFIG_KEYBOARD_GPIO=y)

[*] Polled GPIO buttons

For example, to wake up the APU on the GPIO pin:

```
$ echo enabled > /sys/devices/platform/gpio-keys/power/wakeup
```

Power Management for the CPU

CPU Hotplug

The user may take one or more APU cores on-line and off-line as needed via the CPU Hotplug control interface.

Kernel configurations required:

- Kernel Features
 - [*] Support for hot-pluggable CPUs

See also:

- <https://www.kernel.org/doc/Documentation/cpu-hotplug.txt>
- <http://lxr.free-electrons.com/source/Documentation/devicetree/bindings/arm/idle-states.txt>

For example, to take CPU3 off-line:

```
$ echo 0 > /sys/devices/system/cpu/cpu3/online
```

CPU Idle

If enabled, the kernel may cut power to individual APU cores when they are idling. The kernel configurations required are:

- CPU Power Management
 - CPU Idle
 - [*] CPU idle PM support
 - Arm CPU Idle Drivers
 - [*] Generic Arm/Arm64 CPU idle Driver

See also:

- <https://www.kernel.org/doc/Documentation/cpuidle/core.txt>
- <https://www.kernel.org/doc/Documentation/cpuidle/driver.txt>
- <https://www.kernel.org/doc/Documentation/cpuidle/governor.txt>
- <https://www.kernel.org/doc/Documentation/cpuidle/sysfs.txt>

Below is the sysfs interface for cpuidle.

```
$ ls -lR /sys/devices/system/cpu/cpu0/cpuidle/

/sys/devices/system/cpu/cpu0/cpuidle/:
drwxr-xr-x  2 root  root  0  Jun  10  21:55  state0
drwxr-xr-x  2 root  root  0  Jun  10  21:55  state1

/sys/devices/system/cpu/cpu0/cpuidle/state0:
-r--r--r--  1  root  root  4096  Jun  10  21:55  desc
-rw-r--r--  1  root  root  4096  Jun  10  21:55  disable
-r--r--r--  1  root  root  4096  Jun  10  21:55  latency
-r--r--r--  1  root  root  4096  Jun  10  21:55  name
-r--r--r--  1  root  root  4096  Jun  10  21:55  power
-r--r--r--  1  root  root  4096  Jun  10  21:55  residency
-r--r--r--  1  root  root  4096  Jun  10  21:55  time
-r--r--r--  1  root  root  4096  Jun  10  21:55  usage

/sys/devices/system/cpu/cpu0/cpuidle/state1:
-r--r--r--  1  root  root  4096  Jun  10  21:55  desc
-rw-r--r--  1  root  root  4096  Jun  10  21:55  disable
-r--r--r--  1  root  root  4096  Jun  10  21:55  latency
-r--r--r--  1  root  root  4096  Jun  10  21:55  name
-r--r--r--  1  root  root  4096  Jun  10  21:55  power
-r--r--r--  1  root  root  4096  Jun  10  21:55  residency
-r--r--r--  1  root  root  4096  Jun  10  21:55  time
-r--r--r--  1  root  root  4096  Jun  10  21:55  usage
```

where:

- desc: Small description about the idle state (string)
- disable: Option to disable this idle state (bool)
- latency: Latency to exit out of this idle state (in microseconds)
- name: Name of the idle state (string)
- power: Power consumed while in this idle state (in milliwatts)
- time: Total time spent in this idle state (in microseconds)
- usage: Number of times this state was entered (count)

Below is the sysfs interface for cpuidle governors.

```
$ ls -lR /sys/devices/system/cpu/cpuidle/

/sys/devices/system/cpu/cpuidle/:
-r--r--r--  1 root  root  4096 Jun 10 21:55 current_driver
-r--r--r--  1 root  root  4096 Jun 10 21:55 current_governor_ro
```

CPU Frequency

If enabled, the CPU cores may switch between different operation clock frequencies. The kernel configurations required are:

- CPU Frequency scaling
 - [*] CPU Frequency scaling
 - Default CPUFreq governor
 - Userspace
- CPU Power Management
 - [*] CPU Frequency scaling
 - Default CPUFreq governor
 - Userspace
 - <*> Generic DT based cpufreq driver

Look up the available CPU speeds:

```
$ cat /sys/devices/system/cpu/cpu*/cpufreq/scaling_cpu_freq
```

Select the 'userspace' governor for CPU frequency control:

```
$ echo userspace > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
```

Look up the current CPU speed (same for all cores):

```
$ cat /sys/devices/system/cpu/cpu*/cpufreq/scaling_cpu_freq
```

Change the CPU speed (same for all cores):

```
$ echo <freq> > /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed
```

For details on adding and changing CPU frequencies, see the [Linux kernel documentation on Generic Operating Points](#).

Power Management for the Devices

Clock Gating

Stop device clocks when they are not being used (also called Common Clock Framework.) The kernel configurations required are:

- Common Clock Framework
 - [*] Support for Xilinx ZynqMP Ultrascale+ clock controllers

Runtime PM

Power off devices when they are not being used. Note that individual drivers may or may not support run-time power management. The kernel configurations required are:

- Power management options
 - [*] Suspend to RAM and standby
- Device Drivers
 - SoC (System-on-a-chip) specific drivers
 - [*] Xilinx Zynq MPSoC driver support

Global General Storage Registers

Four 32-bit storage registers are available for general use. Their values are not preserved across after software reboots. The following table lists the global general storage registers.

Table 55: Global General Storage Registers

Device Node	MMIO Register	MMIO Address	Valid Value Range
/sys/firmware/zynqmp/ggs0	GLOBAL_GEN_STORAGE0	0xFFD80030	0x00000000 - 0xFFFFFFFF
/sys/firmware/zynqmp/ggs1	GLOBAL_GEN_STORAGE1	0xFFD80034	0x00000000 - 0xFFFFFFFF
/sys/firmware/zynqmp/ggs2	GLOBAL_GEN_STORAGE2	0xFFD80038	0x00000000 - 0xFFFFFFFF
/sys/firmware/zynqmp/ggs3	GLOBAL_GEN_STORAGE3	0xFFD8003C	0x00000000 - 0xFFFFFFFF

Read the value of a global storage register:

```
$cat /sys/firmware/zynqmp/ggs0
```

Write the mask and value of a global storage register:

```
$echo 0xFFFFFFFF 0x1234ABCD > /sys/firmware/zynqmp/ggs0
```

Persistent Global General Storage Registers

Four 32-bit persistent global storage registers are available for general use. Their values are preserved across after software reboots. The lists the persistent global general storage registers.

Table 56: Persistent Global General Storage Registers

Device Node	MMIO Register	MMIO Address	Valid Value Range
/sys/firmware/zynqmp/pggs0	PERS_GLOB_GEN_STORAGE0	0xFFD80050	0x00000000 - 0xFFFFFFFF
/sys/firmware/zynqmp/pggs1	PERS_GLOB_GEN_STORAGE1	0xFFD80054	0x00000000 - 0xFFFFFFFF
/sys/firmware/zynqmp/pggs2	PERS_GLOB_GEN_STORAGE2	0xFFD80058	0x00000000 - 0xFFFFFFFF
/sys/firmware/zynqmp/pggs3	PERS_GLOB_GEN_STORAGE3	0xFFD8005C	0x00000000 - 0xFFFFFFFF

Read the value of a persistent global storage register:

```
$cat /sys/firmware/zynqmp/pggs0
```

Write the mask and value of a persistent global storage register:

```
$echo 0xFFFFFFFF 0x1234ABCD > /sys/firmware/zynqmp/pggs0
```

Demo

A demo script is included with the PetaLinux pre-built images, which performs a few simple power management tasks:

- System Suspend
- CPU Hotplug
- CPU Freq
- System Reboot
- System Shutdown

To start the demo, type the following command:

```
$ hellopm
```

Debug Interface

The PM platform driver exports a standard debugfs interface to access all EEMI services. The interface is intended for testing only and does not contain any checking regarding improper usage, and the number, type and valid ranges of the arguments. The user should be aware that invoking EEMI services directly via this interface can very easily interfere with the kernel power management operations, resulting in unexpected behavior or system crash. Zynq MPSoC debugfs interface is disabled by default in defconfig. It needs to be enabled explicitly as mentioned below.

Kernel configurations required (in this order):

- Kernel hacking
 - Compile-time checks and compiler options
 - [*] Debug File system
- Firmware Drivers
 - Zynq MPSoC Firmware Drivers
 - [*] Enable Xilinx Zynq MPSoC firmware interface
 - [*] Enable Xilinx Zynq MPSoC firmware debug APIs

You may invoke any EEMI API except for:

- Self Suspend

- System Shutdown
- Force Power Down the APU
- Request Wake-up the APU

Command-line Input

The user may invoke an EEMI service by writing the EEMI API ID, followed by up to four arguments, to the debugfs interface node.

API ID

Function ID can be EEMI API function name or ID number, type string or type integer, respectively.

Arguments

The number and type of the arguments directly depend on the selected API function. All arguments must be provided as integer types and represent the ordinal number for that specific argument type from the EEMI argument list. For more information about function descriptions, type and number of arguments see the EEMI API Specification.

Example

The following example shows how to invoke a request_node API call for NODE_USB_0.

```
$ echo "pm_request_node 22 1 100 1" > /sys/kernel/debug/zynqmp-firmware/pm
```

Command List

Get API Version

Get the API version.

```
$ echo pm_get_api_version > /sys/kernel/debug/zynqmp-firmware/pm
```

Request Suspend

Request another PU to suspend itself.

```
$ echo pm_request_suspend <node> > /sys/kernel/debug/zynqmp-firmware/pm
```

Self Suspend

Notify PMU that this PU is about to suspend itself.

```
$ echo pm_self_suspend <node> > /sys/kernel/debug/zynqmp-firmware/pm
```

Force Power Down

Force another PU to power down.

```
$ echo pm_force_powerdown <node> > /sys/kernel/debug/zynqmp-firmware/pm
```

Abort Suspend

Notify PMU that the attempt to suspend has been aborted.

```
$ echo pm_abort_suspend > /sys/kernel/debug/zynqmp-firmware/pm
```

Request Wake-up

Request another PU to wake up from suspend state.

```
$ echo pm_request_wakeup <node> <set_address> <address> > /sys/kernel/debug/zynqmp-firmware/pm
```

Set Wake-up Source

Set up a node as the wake-up source.

```
$ echo pm_set_wakeup_source <target> <wkup_node> <enable> > /sys/kernel/debug/zynqmp-firmware/pm
```

Request Node

Request to use a node.

```
$ echo pm_request_node <node> > /sys/kernel/debug/zynqmp-firmware/pm
```

Release Node

Free a node that is no longer being used.

```
$ echo pm_release_node <node> > /sys/kernel/debug/zynqmp-firmware/pm
```

Set Requirement

Set the power requirement on the node.

```
$ echo pm_set_requirement <node> <capabilities> > /sys/kernel/debug/zynqmp-firmware/pm
```

Set Max Latency

Set the maximum wake-up latency requirement for a node.

```
$ echo pm_set_max_latency <node> <latency> > /sys/kernel/debug/zynqmp-firmware/pm
```

Get Node Status

Get status information of a node. (Any PU can check the status of any node, regardless of the node assignment.)

```
$ echo pm_get_node_status <node> > /sys/kernel/debug/zynqmp-firmware/pm
```

Get Operating Characteristic

Get operating characteristic information of a node.

```
$ echo pm_get_operating_characteristic <node> > /sys/kernel/debug/zynqmp-firmware/pm
```

Reset Assert

Assert/de-assert on specific reset lines.

```
$ echo pm_reset_assert <reset> <action> > /sys/kernel/debug/zynqmp-firmware/pm
```

Reset Get Status

Get the status of the reset line.

```
$ echo pm_reset_get_status <reset> > /sys/kernel/debug/zynqmp-firmware/pm
```

Get Chip ID

Get the chip ID.

```
$ echo pm_get_chipid > /sys/kernel/debug/zynqmp-firmware/pm
```

Get Pin Control Functions

Get current selected function for given pin.

```
$ echo pm_pinctrl_get_function <pin-number> > /sys/kernel/debug/zynqmp-firmware/pm
```

Set Pin Control Functions

Set requested function for given pin.

```
$ echo pm_pinctrl_set_function <pin-number> <function-id> > /sys/kernel/
debug/zynqmp-firmware/pm
```

Get Configuration Parameters for the Pin

Get value of requested configuration parameter for given pin.

```
$ echo pm_pinctrl_config_param_get <pin-number> <parameter to get> > /sys/
kernel/debug/zynqmp-firmware/pm
```

Set Configuration Parameters for the Pin

Set value of requested configuration parameter for given pin.

```
$ echo pm_pinctrl_config_param_set <pin-number> <parameter to set> <param
value> > /sys/kernel/debug/zynqmp-firmware/pm
```

Control Device and Configurations

Control device and configurations and get configurations values.

```
$ echo pm_ioctl <node id> <ioctl id> <arg1> <arg2> > /sys/kernel/debug/
zynqmp-firmware/pm
```

Table 57: IOCTLS in SDG

IOCTL_ID	Name	Description
0	IOCTL_GET_RPU_OPER_MODE	returns current RPU operating mode (lockstep/split).
1	IOCTL_SET_RPU_OPER_MODE	configures RPU operating mode (lockstep/split).
2	IOCTL_RPU_BOOT_ADDR_CONFIG	configures RPU boot address
3	IOCTL_TCM_COMB_CONFIG	configures TCM to be in split mode or combined mode
4	IOCTL_SET_TAPDELAY_BYPASS	enable/disable tap delay bypass
5	IOCTL_SET_SGMII_MODE	enable/disable SGMII mode for the GEM device
6	IOCTL_SD_DLL_RESET	resets DLL logic for the SD device
7	IOCTL_SET_SD_TAPDELAY	sets input/output tap delay for the SD device
8	IOCTL_SET_PLL_FRAC_MODE	sets PLL mode
9	IOCTL_GET_PLL_FRAC_MODE	returns current PLL mode
10	IOCTL_SET_PLL_FRAC_DATA	sets PLL fraction data
11	IOCTL_GET_PLL_FRAC_DATA	returns PLL fraction data value
12	IOCTL_WRITE_GGS	writes value to GGS register
13	IOCTL_READ_GGS	returns GGS register value
14	IOCTL_WRITE_PGGS	writes value to PGGS register

Table 57: IOCTLs in SDG (cont'd)

IOCTL_ID	Name	Description
15	IOCTL_READ_PGGS	returns PGGS register value
16	IOCTL_ULPI_RESET	performs the ULPI reset sequence for resetting the ULPI transceiver
17	IOCTL_SET_BOOT_HEALTH_STATUS	sets healthy bit value to indicate boot health status to firmware.
18	IOCTL_AFI	writes the afi values at given index

Table 58: Description of IOCTLs

IOCTL_ID	Name	Description	Arguments			
			Node ID	Arg1	Arg2	Return Value
0	IOCTL_GET_RPU_OPER_MODE	returns current RPU operating mode (lockstep/split)	unused	unused	unused	Operating mode 0: LOCKSTEP 1: SPLIT
1	IOCTL_SET_RPU_OPER_MODE	configures RPU operating mode (lockstep/split)	unused	Value of operating mode 0: LOCKSTEP 1: SPLIT	unused	unused
2	IOCTL_RPU_BOOT_ADDRESS_CONFIG	configures RPU boot address	NODE_RPU_0 NODE_RPU_1	Value to set for boot address 0: LOVEC/TCM 1: HIVEC/OCM	unused	unused
3	IOCTL_TCM_COMB_CONFIG	configures TCM to be in split mode or combined mode	unused	Value to set (Split/Combined) 0: SPLIT 1: COMB	unused	unused
4	IOCTL_SET_TAPDELAY_BYPASS	enables/disables tap delay bypass	unused	Type of tap delay 0: NAND_DQS_IN 1: NAND_DQS_OUTPUT - 2: QSPI	Tap-delay Enable/ Disable 0: DISABLE 1: ENABLE	unused
5	IOCTL_SET_SGMII_MODE	enables/disables SGMII mode for the GEM device	NODE_ETH_0, NODE_ETH_1, NODE_ETH_2, NODE_ETH_3	"GMII mode Enable/ Disable 0: DISABLE 1: ENABLE	unused	unused

Table 58: Description of IOCTLs (cont'd)

IOCTL_ID	Name	Description	Arguments			
			Node ID	Arg1	Arg2	Return Value
6	IOCTL_SD_DLL_RESET	resets DLL logic for the SD device	NODE_SD_0 , NODE_SD_1	SD DLL Reset type 0: ASSERT 1: RELEASE 2: PULSE	unused	unused
7	IOCTL_SET_SD_TAPDELAY	sets input/output tap delay for the SD device	NODE_SD_0 , NODE_SD_1	Type of tap delay to set 0: INPUT 1: OUTPUT	Value to set for the tap delay	unused
8	IOCTL_SET_PLL_FRAC_MODE	sets PLL mode	unused	PLL clock ID	PLL Mode 0: FRAC_MODE 1: INT_MODE	unused
9	IOCTL_GET_PLL_FRAC_MODE	returns current PLL mode	unused	PLL clock ID	unused	PLL Mode 0: FRAC_MODE 1: INT_MODE
10	IOCTL_SET_PLL_FRAC_DATA	sets PLL fraction data	unused	PLL clock ID	PLL fraction data	unused
11	IOCTL_GET_PLL_FRAC_DATA	returns PLL fraction data value	unused	PLL clock ID	unused	PLL fraction data
12	IOCTL_WRITE_GGS	writes value to GGS register	unused	GGS register index (0/1/2/3)	Register value to be written	unused
13	IOCTL_READ_GGS	returns GGS register value	unused	GGS register index (0/1/2/3)	unused	Register value
14	IOCTL_WRITE_PGGS	writes value to PGGS register	unused	PGGS register index (0/1/2/3)	Register value to be written	unused
15	IOCTL_READ_PGGS	returns PGGS register value	unused	PGGS register index (0/1/2/3)	unused	Register value

Table 58: Description of IOCTLs (cont'd)

IOCTL_ID	Name	Description	Arguments			
			Node ID	Arg1	Arg2	Return Value
16	IOCTL_ULPI_RESET	performs the ULPI reset sequence for resetting the ULPI transceiver	unused	unused	unused	unused
17	IOCTL_SET_BOOT_HEALTH_STATUS	sets healthy bit value to indicate boot health status to firmware	unused	healthy bit value	unused	unused
18	IOCTL_AFI	writes the afi values at given index	unused	AFI register index (0 to 15)	Register value to be written	unused

Query Data

Request data from firmware.

```
$ echo pm_query_data <query id> <arg1> <arg2> <arg3> > /sys/kernel/debug/zynqmp-firmware/pm
```

Enable Clock

Enable the clock for a given clock node id.

```
$ echo pm_clock_enable <clock id> > /sys/kernel/debug/zynqmp-firmware/pm
```

Disable Clock

Disable the clock for a given clock node id.

```
$ echo pm_clock_disable <clock id> > /sys/kernel/debug/zynqmp-firmware/pm
```

Get Clock State

Get the state of clock for a given clock node id.

```
$ echo pm_clock_getstate <clock id> > /sys/kernel/debug/zynqmp-firmware/pm
```

Set Clock Divider

Set the divider value of clock for a given clock node id.

```
$ echo pm_clock_setdivider <clock id> <divider value> > /sys/kernel/debug/zynqmp-firmware/pm
```

Get Clock Divider

Get the divider value of clock for a given clock node id.

```
$ echo pm_clock_getdivider <clock id> > /sys/kernel/debug/zynqmp-firmware/pm
```

Set Clock Rate

Set the clock rate for a given clock node id.

```
$ echo pm_clock_setrate <clock id> <clock rate> > /sys/kernel/debug/zynqmp-firmware/pm
```

Get Clock Rate

Get the clock rate for a given clock node id.

```
$ echo pm_clock_getrate <clock id> > /sys/kernel/debug/zynqmp-firmware/pm
```

Set Clock Parent

Set the parent clock for a given clock node id.

```
$ echo pm_clock_setparent <clock id> <parent clock id> > /sys/kernel/debug/zynqmp-firmware/pm
```

Get Clock Parent

Get the parent clock for a given clock node id.

```
$ echo pm_clock_getparent <clock id> > /sys/kernel/debug/zynqmp-firmware/pm
```

Note: Clock id definitions are available in the following txt file of the clock bindings documentation:

Documentation/devicetree/bindings/clock/xlnx,zynqmp-clk.txt

PM Platform Driver

The Zynq UltraScale+ MPSoC power management for Linux is encapsulated in a power management driver, power domain driver and platform firmware driver. The system-level API functions are exported and as such, can be called by other Linux modules with GPL compatible license. The function declarations are available in the following location:

```
include/linux/firmware/xilinx/zynqmp/firmware.h
```

The function implementations are available in the following location:

```
drivers/firmware/xilinx/zynqmp/firmware*.c
```

Provide the correct node in the Linux device tree for proper driver initialization. The firmware driver relies on the 'firmware' node to detect the presence of PMU firmware, determine the calling method (either 'smc' or 'hvc') to the PM-Framework firmware layer and to register the callback interrupt number.

The 'firmware' node contains following properties:

- **Compatible:** Must contain 'xlnx,zynqmp-firmware'.
- **Method:** The method of calling the PM framework firmware. It should be 'smc'.

Note: Additional information is available in the following txt file of Linux Documentation:

```
Documentation/devicetree/bindings/firmware/xilinx/xlnx,zynqmp-firmware.txt.
```

Example:

```
firmware {
    zynqmp_firmware: zynqmp-firmware { compatible = "xlnx,zynqmp-firmware";
    method = "smc";
    };
};
```

Note: Power domain driver and power management driver binding details are available in the following files of Linux Documentation:

- `Documentation/devicetree/bindings/soc/xilinx/xlnx,zynqmp-power.txt`
- `Documentation/devicetree/bindings/power/zynqmp-genpd.txt`

Note: xilPM do not support the following EEMI APIs. For current release, they are only supported for Linux through ATF.

- `query_data`
- `ioctl`
- `clock_enable`
- `clock_disable`
- `clock_getstate`
- `clock_setdivider`
- `clock_getdivider`
- `clock_setrate`
- `clock_getrate`

- clock_setparent
- clock_getparent
- pinctrl_request
- pinctrl_release
- pinctrl_set_function
- pinctrl_get_function
- pinctrl_set_config
- pinctrl_get_config

Arm Trusted Firmware (ATF)

The Arm Trusted Firmware (ATF) executes in EL3. It supports the EEMI API for managing the power state of the slave nodes, by sending PM requests through the IPI-based communication to the PMU.

ATF Application Binary Interface

All APU executable layers below EL3 may indirectly communicate with the PMU via the ATF. The ATF receives all calls made from the lower ELs, consolidates all requests and send the requests to the PMU.

Following Arm's SMC Calling Convention, the PM communication from the non-secure world to the ATF is organized as SiP Service Calls, using a predefined SMC function identifier and SMC sub-range ownership as specified by the calling convention.

Note that the EEMI API implementation for the APU is compliant with the SMC64 calling convention only.

EEMI API calls made from the OS or hypervisor software level pass the 32-bit API ID as the SMC Function Identifier, and up to four 32-bit arguments as well. As all PM arguments are 32-bit values, pairs of two are combined into one 64-bit value.

The ATF returns up to five 32-bit return values:

- Return status, either success or error and reason
- Additional information from the PM controller

Checking the API Version

Before using the EEMI API to manage the slave nodes, the user must check that EEMI API version implemented in the ATF matches the version implemented in the PMU firmware. EEMI API version is a 32-bit value separated in higher 16 bits of MAJOR and lower 16 bits of MINOR part. Both fields must be the same between the ATF and the PMU firmware.

The EEMI version implemented in the ATF is defined in the local EEMI_API_VERSION flag. The rich OS may invoke the PM_GET_API_VERSION function to retrieve the EEMI API version from the PMU. If the versions are different, this call will report an error.

Note: This EEMI API call is version independent; every EEMI version implements it.

Checking the Chip ID

Linux or other rich OS can invoke the PM_GET_CHIPID function via SMC to retrieve the chip ID information from the PMU.

The return values are:

- CSU idcode register (see TRM).
- CSU version register (see TRM).

For more details, see the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)).

Power State Coordination Interface (PSCI)

Power State Coordination Interface is a standard interface for controlling the system power state of Arm processors, such as suspend, shutdown, and reboot. For the PSCI specifications, see <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0022c/index.html>.

ATF handles the PSCI requests from Linux. ATF supports PSCI v0.2 only (with no backward compatible support for v0.1).

The Linux kernel comes with standard support for PSCI. For information regarding the binding between the kernel and the ATF/PSCI, see <https://www.kernel.org/doc/Documentation/devicetree/bindings/arm/psci.txt>.

Table 59: PSCI v0.2 Functions Supported by the ATF

Functions	Description	Supported
PSCI Version	Return the version of PSCI implemented.	Yes
CPU Suspend	Suspend execution on a core or higher level topology node. This call is intended for use in idle subsystems where the core is expected to return to execution through a wakeup event.	Yes

Table 59: PSCI v0.2 Functions Supported by the ATF (cont'd)

Functions	Description	Supported
CPU On	Power up a core. This call is used to power up cores that either: <ul style="list-style-type: none"> • Have not yet been booted into the calling supervisory software. • Have been previously powered down with a CPU_OFF call. 	Yes
CPU Off	Power down the calling core. This call is intended for use in hotplug. A core that is powered down by CPU_OFF can only be powered up again in response to a CPU_ON.	Yes
Affinity Info	Enable the caller to request status of an affinity instance.	Yes
Migrate (Optional)	This is used to ask a uniprocessor Trusted OS to migrate its context to a specific core.	Yes
Migrate Info Type (Optional)	This function allows a caller to identify the level of multicore support present in the Trusted OS.	Yes
Migrate Info Up CPU (Optional)	For a uniprocessor Trusted OS, this function returns the current resident core.	Yes
System Off	Shut down the system.	Yes
System Reset	Reset the system.	Yes
PSCI Features	Introduced in PSCI v1.0. Query API that allows discovering whether a specific PSCI function is implemented and its features.	Yes
CPU Freeze (Optional)	Introduced in PSCI v1.0. Places the core into an IMPLEMENTATION DEFINED low-power state. Unlike CPU_OFF it is still valid for interrupts to be targeted to the core. However, the core must remain in the low power state until it a CPU_ON command is issued for it.	No
CPU Default Suspend (Optional)	Introduced in PSCI v1.0. Will place a core into an IMPLEMENTATION DEFINED low-power state. Unlike CPU_SUSPEND the caller need not specify a power state parameter.	No
Node HW State (Optional)	Introduced in PSCI v1.0. This function is intended to return the true HW state of a node in the power domain topology of the system.	Yes
System Suspend (Optional)	Introduced in PSCI v1.0. Used to implement suspend to RAM. The semantics are equivalent to a CPU_SUSPEND to the deepest low-power state.	Yes
PSCI Set Suspend Mode (Optional)	Introduced in PSCI v1.0. This function allows setting the mode used by CPU_SUSPEND to coordinate power states.	No
PSCI Stat Residency (Optional)	Introduced in PSCI v1.0. Returns the amount of time the platform has spent in the given power state since cold boot.	Yes
PSCI Stat Count (Optional)	Introduced in PSCI v1.0. Return the number of times the platform has used the given power state since cold boot.	Yes

PMU Firmware

The EEMI service handlers are implemented in the PMU firmware, as one of the modules called PM Controller (There are other modules running in the PMU firmware to handle other types of services). For more details, see the [Chapter 10: Platform Management Unit Firmware](#).

Power Management Events

The PM Controller is event-driven, and all of the operations are triggered by one of the following events:

- EEMI API events triggered via IPI0 interrupt.
- Wake events triggered via GPI1 interrupt.
- Sleep events triggered via GPI2 interrupt.
- Timer event triggered via PIT2 interrupt.

EEMI API Events

EEMI API events are software-generated events. The events are triggered via IPI interrupt when a PM master initiates an EEMI API call to the PMU. The PM Controller handles the EEMI request and may send back an acknowledgment (if one is requested.) An EEMI request often triggers a change in the power state of a node or a master, with some exceptions.

Wake Events

Wake events are hardware-generated events. They are triggered by a peripheral signaling that a PM master should be woken-up. All wake events are triggered via the GPI1 interrupt.

The following wake events are supported by the PM controller:

- GIC wake events which signal that a CPU shall be woken up due to an interrupt triggered by a hardware resource to the associated GIC interface. The following GIC wake events are supported:
 - APU[3:0]An event for each APU processor
 - RPU[1:0]An event for each RPU processor
- FPD wake event directed by the GIC Proxy. This wake event is triggered when any of the wake sources enabled prior to suspending. The purpose of this event is to trigger a wake-up of APU master when FPD is powered down. If FPD is not powered down, none of the wake signals would propagate through FPD wake. Instead, the wake would propagate through GIC wake if the associated interrupt at the GIC is properly enabled. All wake events targeted to the RPU propagate via the associated GIC wake.

Sleep Events

Sleep events are software-generated events. The events are triggered by a CPU after it finalizes the suspend procedure with the aim to signal to the PMU that it is ready to be put in a low power state. All sleep events are triggered via GPI2 interrupt.

The following sleep events are supported:

- APU[3:0]An event for each APU processor
- RPU[1:0]An event for each RPU processor

When the PM controller PM Controller receives the sleep event for a particular CPU, the CPU is put into a low power state.

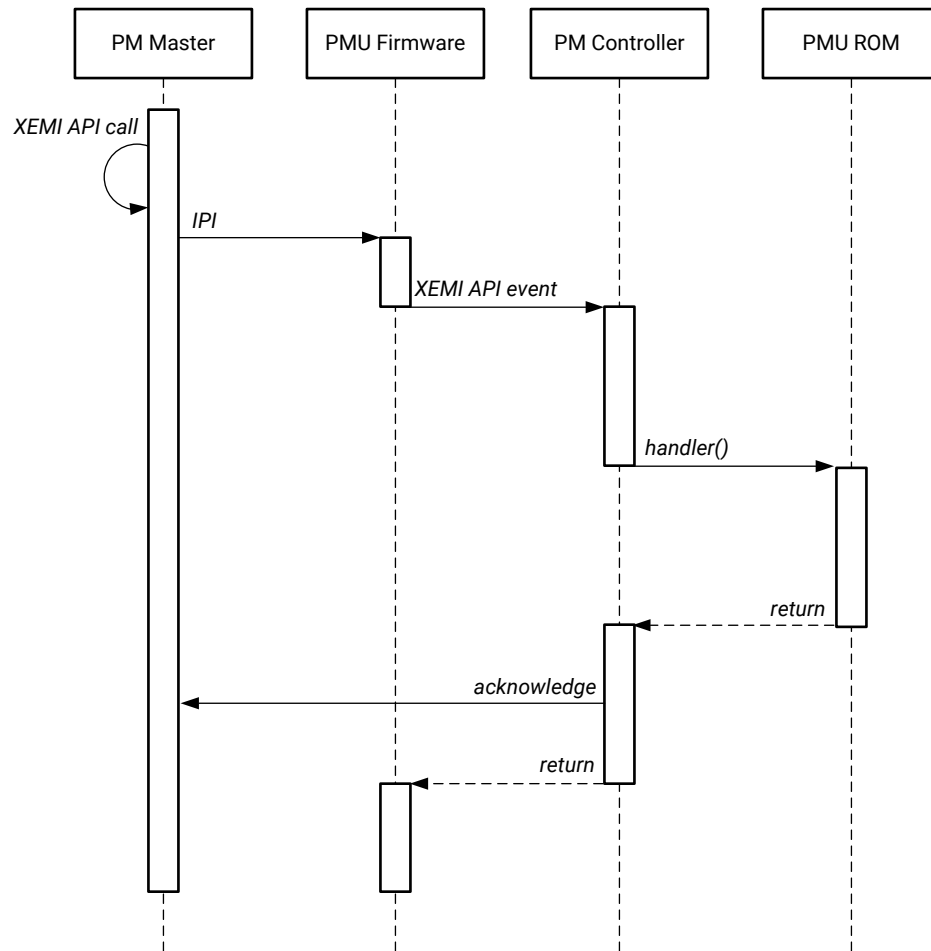
Timer Event

Timer event is hardware-generated event. It is triggered by a hardware timer when a period of time expires. The event is used for power management timeout accounting and it is triggered via PIT2 interrupt.

General flow of an EEMI API Call

The following diagram illustrates the sequence diagram of a typical API call, starting with the call initiated by a PM Master (such as another PU):

Figure 60: EEMI API Call Sequence Diagram



X20021-110217

The previous diagram shows four actors, where the first one represents the PM Master, i.e. either the RPU, APU, or a MicroBlaze™ processor core. The remaining 3 actors are the different software layers of the PMU.

First the PMU firmware receives the IPI interrupt. Once the interrupt has been identified as a power management related interrupt, the IPI arguments are passed to the Power Management Module. The PM controller then processes the API call. If necessary it may call the PMU ROM in order to perform power management actions, such as power on or off a power island, or a power domain.

Reset

The Zynq[®] UltraScale+™ MPSoC reset block is responsible for handling both internal and external reset inputs to the system, and to meet the reset requirements for all the peripherals and the APU and RPU. The reset block generates resets for the programmable logic part of the device, and allows independent reset assertion for PS and PL blocks.

This chapter explains the reset mechanisms involved in the system reset and the individual module resets.

System-Level Reset

The Zynq UltraScale+ MPSoCs let you reset individual blocks such as the APU, RPU, or even individual power domains like the FPD and LPD. There are multiple, system-level reset options, as follows:

- Power-on reset (POR)
- System reset (SRST_B)
- Debug system reset

For more details on the system-level reset flow, see this [link](#) to the “Reset System” chapter in the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*.

Block-Level Resets

The PS-only reset can be implemented as a subset of system-reset; however, the user must provide software that ensures PS-to-PS AXI transactions are gracefully terminated before initiating a PS-only reset.

PS-Only Reset

The PS-only reset re-boots the PS while that PL remains active. You can trigger the PS-only reset by hardware error signal(s) or a software register write. If the PS-only reset is due to an error signal, then the error can be indicated to the PL also, so that the PL can prepare for the PR restart.

The PS-only reset sequence can be implemented as follows:

- [ErrorLogic] Error interrupt is asserted whose action requires PS-only reset. This request is sent to PMU as an interrupt.
- [PMU-FW] Set PMU Error (=>PS-only reset) to indicate to PL.

See the PS Only Reset section in the “Reset System” chapter of the *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)) describes the PS-only reset sequence.

Note: PS-only reset is not supported in qspi24 mode on systems with a flash size that is greater than 16 MB.

Application Processing Unit Reset

You can independently reset each of the APU CPU core in the software.

The APU MPCore reset can be triggered by FPD, WDT, or a software register write; however, APU MPCore is reset without gracefully terminating requests to and from the APU. The intent is that you use the FPD in case of catastrophic failures in the FPD. The APU reset is primarily for software debug.

The *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)) describes the APU reset sequence.

APU-Only Reset

APU-only reset is supported in qspi24, qspi32, sd0, sd1, sd-ls boot modes. However, APU-only reset is not supported in qspi24 mode on systems with a flash size that is greater than 16 MB.

Real Time Processing Unit Reset

Each Cortex™-R5F core can be independently reset. In lockstep mode, only the Cortex-R5F_0 needs to be reset to reset both Cortex-R5F cores. It can be triggered by errors or a software register write. The Cortex-R5F reset can be triggered due to a lockstep error to be able to reset and restart the RPU. It needs to gracefully terminate Cortex-R5F ingress and egress transactions before initiating reset of corresponding Cortex-R5F.

Full Power Domain Reset

The FPD-reset resets all of the FPD power domain and can be triggered by errors or a software register write. If the FPD reset is due to error signal, then the error must be indicated to both the LPD and the PL.

The FPD reset can be implemented by leveraging the FPD power-up sequence; however, it needs to gracefully terminate FPD ingress and egress AXI transactions before initiating reset of FPD. FPD reset sequence can be PL Reset.

The Zynq UltraScale+ MPSoCs has general-purpose output pins from the PMU block that can be used to reset the blocks in PL. Additionally, GPIO using the EMIO interface can also be used to reset PL logic blocks. For a detailed description of the reset flow, see the [this link](#) to the “Reset System” chapter in the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*.

For more information on the software APIs for reset, see the PMU firmware in [Chapter 9: Platform Management](#).

Warm Restart

The Zynq UltraScale+ MPSoC is a highly complex piece of silicon, capable of running multiple subsystems on the chip simultaneously. As such, Zynq UltraScale+ supports various types of reset. This varies from the simplest system reset to the much more complicated subsystem restart. In any system or subsystem that has a processor component and a programmable logic component, reset must entail both reset to the hardware as well as software. Reset to the hardware includes the following:

- Resetting of the processor and all peripherals associated with the system/subsystem
- Cleaning up of the memory as needed
- Making sure that the interconnect is in a clean state that is capable of routing traffic.

Reset to the software results in the processor starting from the reset vector. However, designer must make sure that a valid and clean code for the system/subsystem is located at the reset vector in order to bring the system back to a clean running state.

Resets for Zynq UltraScale+ are broadly divided into two categories. They are:

- Full system resets
- Subsystem restarts

Full system resets include the following:

- Power-On-Reset (POR)
- System-reset
- PS-only-reset

Subsystem restarts include APU subsystems and RPU subsystem restarts.

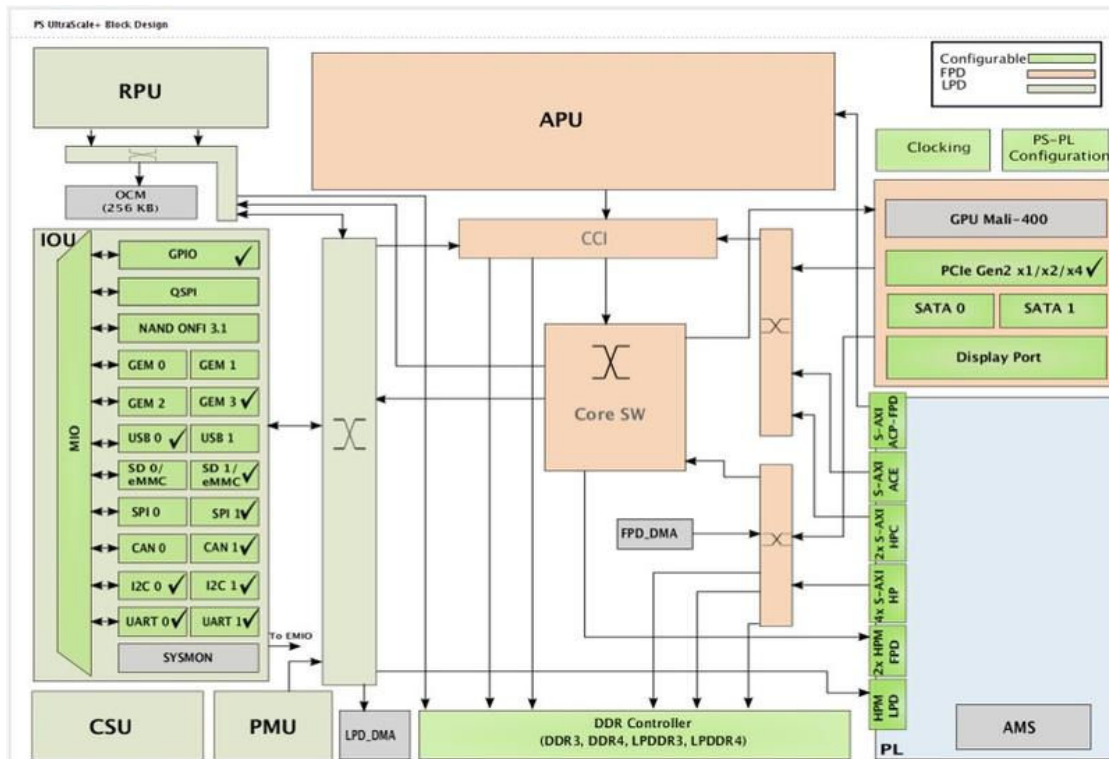
Full system resets are quite straight forward. Hardware is brought back to the reset state and software starts executing ROM code, with a minor behavior difference between the reset types. There are subtleties to PS-only reset which will be discussed in later sections.

Subsystem restart is more complicated. A subsystem in Zynq UltraScale+ is composed of all the components of a particular operating system. The following figure shows both Vivado's view of the PS as well as example subsystems as defined by the OS. The default IP configuration menu in Vivado provides a flattened view, consisting of all available PS components. In the example, these components are partitioned into three separate subsystems, each running an independent operating system. Each subsystem consists of a processor, list of peripherals and memory. The example shows the following subsystems:

- RPU based subsystem running uC/OS-II
- RPU based subsystem running FreeRTOS
- APU based subsystem running Linux

Subsystems can be configured in the Isolation Configuration view that is inside the Vivado PCW (PS Configuration Wizard), when the Advanced Mode check box is enabled.

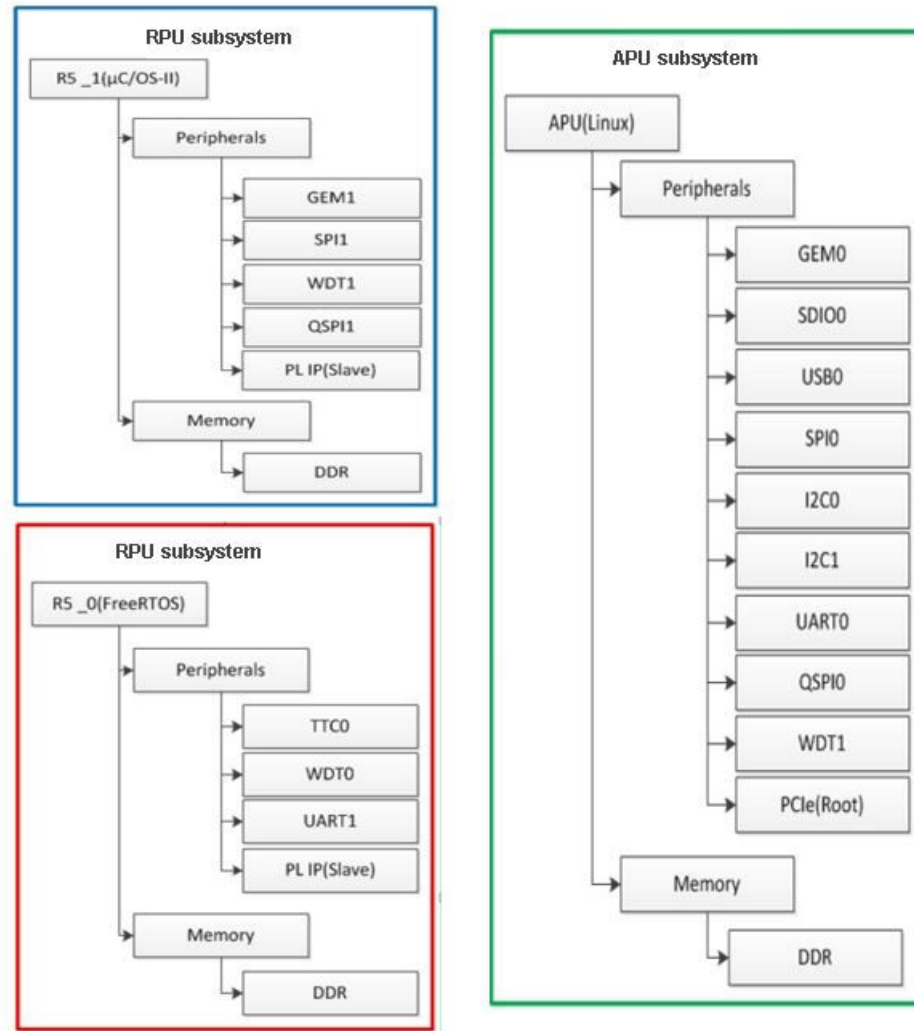
Figure 61: Vivado IP Configuration Menu



During subsystem restart, the entire subsystem is restarted from a clean state without affecting the running of the other active subsystems defined in MPSoC. For example, during an APU subsystem restart, an APU subsystem running Linux is restarted as far back as FSBL, while the RPU subsystem running FreeRTOS and uC/OS-II continues to function undisturbed. Similarly for a RPU subsystem restart, an APU subsystem continues to function undisturbed.

Subsystem restarts are managed by the platform management unit (PMU). To restart each subsystem, PMU must first ensure that all on-going AXI-transactions are terminated and that no new transactions are issued. In the subsystems shown in the following figure, the interconnects that connects the components of the subsystem, are not explicitly shown. However, each subsystem includes multiple interconnects and the same interconnects are used by all three subsystems. If the PMU firmware resets all the components in a subsystem while leaving unfinished transactions in the interconnect, the AXI master and slave might both be in the reset state. However, the unfinished AXI transactions will remain in the interconnect, thus blocking all subsequent traffic. Stuck transactions in the interconnect causes the system to freeze as these connections are shared. It is therefore imperative that the PMU ensures all transactions are completely finished before resetting each and every components in the subsystem, including the processor.

Figure 62: Subsystem Components for Various Operating Systems



Before releasing the processor from reset, the PMU must ensure that the code in the reset vector will result in a clean system restart. In the case of the RPU subsystem running standalone applications, this means either loading a clean copy of the application elf or making sure that the application code is re-entrant. In the case of the APU subsystem running Linux, this means starting from a re-entrant copy of FSBL.

Note: The on-chip memory (OCM) module contains 256 KB of RAM starting at `0xFFFC0000`. The OCM is mainly used by the FSBL and ATF components. The FSBL uses the OCM region from `0xFFFC0000` to `0xFFFE9FFF`. The last 512B of this region is used by the FSBL to share the handoff parameters corresponding to applications that the ATF hands off. The ATF uses the rest of the OCM i.e. from `0xFFFEA000` to `0xFFFFFFFF`.

The current implementation of a warm reset requires the FSBL to be in the OCM to support the PMU firmware hand off to (already existing) the FSBL without actually restarting. Hence, the OCM is completely used and no other application is allowed to use it when a warm restart is enabled.

Supported Use Cases

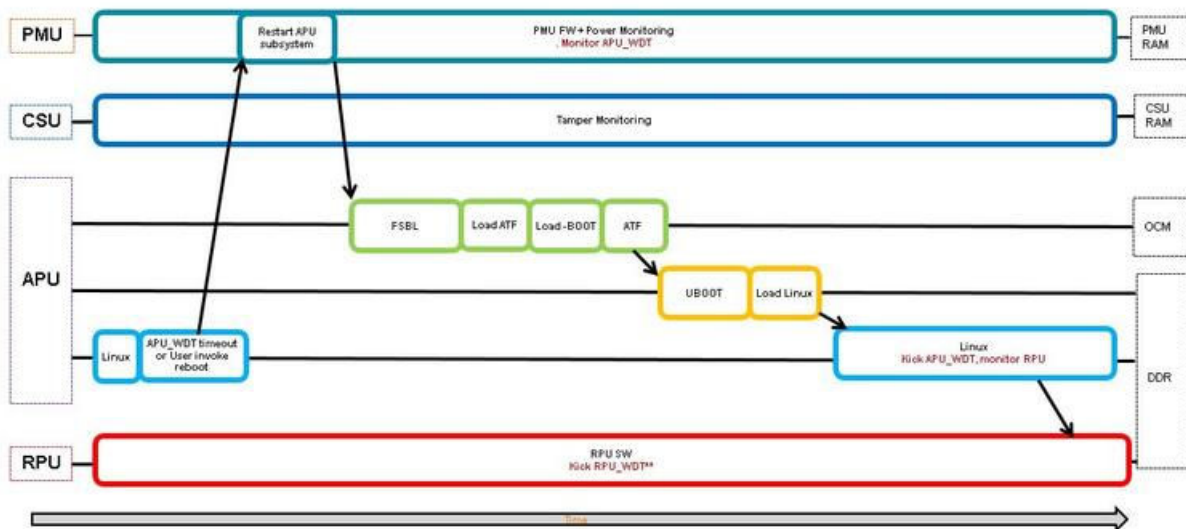
APU Subsystem Restart

For an APU subsystem only restart, you must define the APU subsystem using PCW in the Vivado design tools. The PMU executes the function to restart the APU subsystem. First, the PMU idles all components in the APU subsystem. When all is quiet, the PMU will reset each component, including the APU processors. When the reset is released, it will re-execute the FSBL code in the OCM. The task carried out by the FSBL for restart differs only slightly than that of the POR.

Note: The FSBL is re-entrant. Hence, the APU can simply re-execute the FSBL without having to reload a clean copy.

The following figure shows the APU subsystem restart process.

Figure 63: APU Subsystem Restart Process



The start of this flow diagram represents a clean running state. Linux, RPU, PMU, and CSU subsystems are in running status. The health of the APU subsystem is monitored by an APU WDT (watchdog timer). Linux runs a background application which periodically boosts the watchdog to prevent it from timing out. If an APU subsystem hangs, the WDT times out. The timeout interrupts the PMU and results in an APU subsystem restart. Alternatively, you can invoke the APU subsystem restart by directly calling for it in Linux.

Implementation

To support any subsystem restart, a subsystem must first be defined in the Vivado design tools using the Isolation Configuration view. For an APU subsystem running Linux, the following APU subsystem are required in addition to the default PMU subsystem:

- A secure APU system for running the FSBL and ATF
- A non-secure APU subsystem for running Linux.

See [Sub-system Power Management](#) for more information on subsystem configuration and an example of the APU only subsystem.



IMPORTANT! While APU subsystem consists solely of PS components, it is often the case that APU subsystem also includes IP peripherals implemented in PL. Unfortunately, isolation configuration menu does not include features to assign PL IPs to different subsystems. As a result, all IPs instantiated in Vivado are added to the generated device tree source (DTS) file. In order to properly define the APU subsystem, all PL IPs that do not belong in the APU subsystem need to be manually removed from the DTS file. Otherwise, drivers for all the soft IPs will be enabled for Linux, and APU will attempt to manage all the soft IPs even when the APU is going through a warm restart.



IMPORTANT! During a subsystem restart, all components in the subsystem must be in the idle state, followed by reset. This is implemented for supported components in the PS. For all IPs in PL of a subsystem that are AXI slaves, no additional tasks are required to idle them. You may supply code to reset these slaves if desired. For PL IPs that are AXI masters, you must provide the necessary code to stop and complete all AXI transactions from the master as well as to reset it. See *Idle and Reset of Peripherals* for details on adding the idle and reset code.

See GPIO Reset to PL for design issue and guidelines pertaining to using `resetn` signal from PS to PL (`ps_resetn`). You can optionally enable the recovery and escalation features as desired. Building Software for detailed instructions on building the software.

RPU Subsystem Restart

RPU as Master

For an RPU subsystem only restart, you must define the RPU subsystem using PCW in the Xilinx Vivado® Design Suite. The PMU executes the function to restart the RPU subsystem. First, the PMU checks if master is RPU and FSBL was initially running on RPU. Then PMU will idle all components in the RPU subsystem. When all is quiet, the PMU will reset each component, including the RPU processors. When the reset is released, it will re-execute the FSBL code. FSBL for subsystem restart loads only RPU partitions without interrupting other subsystems.

Note: RPU only subsystem restart is supported only with FSBL running on RPU just as APU only restart. Here the FSBL is re-entrant. Hence, the RPU can simply re-execute the FSBL without having to reload a clean copy.

Once all the subsystems have started and represent a clean running state, the health of the RPU subsystem can be monitored using an LPD WDT (watchdog timer) by an application running on RPU. This application must take care of boosting the watchdog to prevent it from timing out. If an RPU subsystem hangs, this WDT times out and interrupts the PMU which results in RPU subsystem restart. For more information, see the [LPD WDT](#) section.

Alternatively, you can invoke the RPU subsystem restart by directly calling for it in RPU application.

Implementation

The implementation is same as APU only subsystem restart except that RPU subsystem must be defined in the Vivado® Design Suite using the Isolation Configuration view.

Note: To support any subsystem restart, a subsystem must first be defined in the Vivado design tools using the Isolation Configuration view.

The RPU subsystem requires RPU running an FSBL and RPU application in addition to PMU subsystem. See [Sub-system Power Management](#) for more information on subsystem configuration and an example of the APU only subsystem.



IMPORTANT! *During a subsystem restart, all components in the subsystem must be in the idle state, followed by reset. This is implemented for supported components in the PS. For all IPs in PL of a subsystem that are AXI slaves, no additional tasks are required to idle them. You may supply code to reset these slaves if desired. For PL IPs that are AXI masters, you must provide the necessary code to stop and complete all AXI transactions from the master as well as to reset it. See [Idle and Reset of Peripherals](#) for details on adding the idle and reset code.*

See [GPIO Reset to PL](#) for design issue and guidelines pertaining to using `resetn` signal from PS to PL (`ps_resetn`). You can optionally enable the recovery and escalation features as desired. See [Building Software](#) for detailed instructions on building the software.

APU as Master

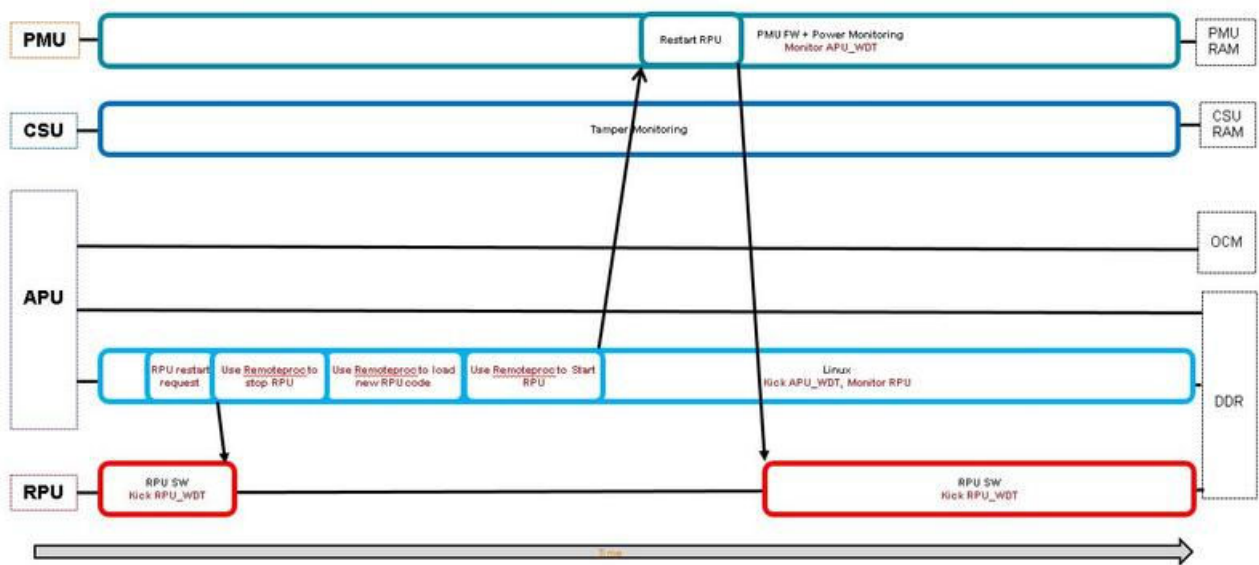
RPU subsystem restart requires the APU subsystem and one or more RPU subsystems running in lock-step or split mode. The APU subsystem running Linux is the master of the RPU subsystems and manages the life cycle of the subsystem using the remoteproc feature of OpenAMP. APU uses remoteproc to load, start, and stop the RPU application. It also re-syncs the APU subsystem with RPU subsystem after the restart. APU subsystem can trigger a RPU restart by following sequence:

1. First, it stops the RPU
2. Loads the new firmware
3. Then, it starts the RPU again.

Many events including user command, RPU watchdog timeout or message from the RPU to APU via message pipe may trigger the RPU subsystem restart. Then, APU issues remoteproc command to PMU to start or stop the RPU, and the PMU changes the state of the RPU subsystem.

The following figure shows the RPU subsystem restart process.

Figure 64: RPU Subsystem Restart



The start of the above diagram represents a clean running state for all subsystems, Linux, RPU, PMU and CSU. In the flowchart, APU receives a RPU subsystem restart request. When APU receives the restart request, it uses remoteproc features to stop the RPU subsystem, load new firmware code, and then starts the RPU subsystem again. The flow chart shows the use of a RPU WDT. The RPU periodically boosts the watch dog. If the RPU hangs, WDT times out. Linux will receive the timeout and restarts the RPU subsystem.

Implementation

You must define the RPU subsystem using the Isolation Configuration view in Vivado PCW, and both PMU and APU subsystems are required. In addition, two configurations are possible for the RPU subsystem: RPUs in lock step mode or in split mode. See the [Isolation Configuration Consideration wiki page](#) for more information on subsystem configuration. Sharing of peripherals between subsystems are not supported. Make sure that the peripherals in all subsystems are mutually exclusive.



IMPORTANT! In the process of subsystem restart, all components in the subsystem must be in the idle state, followed by reset. This is implemented for supported components in the PS. For all IPs in PL of a subsystem that are AXI slaves, no additional tasks are required to idle them. User may supply code to reset the slaves if desired. For PL IPs that are AXI masters, user must provide the necessary code to stop and complete all AXI transactions from the master as well as to reset it. See *Idle and Reset of Peripherals* for details on adding the idle and reset code.

RPU subsystem restart is supported with Linux kernel implementation of remoteproc on APU in conjunction with OpenAMP library on RPU. It is currently not supported with Linux userspace OpenAMP library on APU. RPU application must be written in accordance with the OpenAMP application requirements. See *Libmetal and OpenAMP for Zynq Devices User Guide (UG1186)* for more information. Note that the `rpmsg` is not required for remoteproc. You can employ `rpmsg` feature to provide a communication pipe between the two processors. However, remoteproc is independent of `rpmsg`. To make remoteproc function properly with subsystem restart, RPU application needs to include a resource table with static shared memory allocation. Dynamic shared memory allocation is not supported for subsystem restart. You must implement the steps outlined in *How to Write a Simple OpenAMP Application in Libmetal and OpenAMP for Zynq Devices User Guide (UG1186)* to satisfy the remoteproc requirement, but not beyond that. After initialization, the RPU application needs to signal to the PMU that it is Power Management (PM) aware by calling `XPm_InitFinalize()`.

Note: If you call `XPm_InitFinalize()` too early, then the slaves that are not yet initialized are powered off. They will be powered up again when the RPU application comes around to initialize them, which will incur some additional power-up latency. See [ZU+ Example - PM Hello World wiki page](#) for more information on how to write a PM aware RPU application.

Finally, you must ensure that the address of the reserved memory for RPU code is synchronized across all layers. It must be defined under memory for both APU and RPU subsystems in the isolation configuration of Vivado. The same address region should be used in the DTS file for OpenAMP overlay in Linux and again, in resource table and linker script for the RPU application.

See *GPIO Reset to PL* for design issue and guidelines pertaining to using `resetn` signal from PS to PL (`ps_resetn`). You can optionally enable the recovery and escalation features as desired. Building Software for detailed instructions on building the software.

PS-Only Reset

For a PS-only restart, the entire processor system is reset while PL continues to function. Prior to invoking PS-only reset, PMU turns on isolation between PS and PL, thus clamping the signals between them in well-defined states. After PS-only reset is released, PS executes the standard boot process starting from the PMU ROM, followed by CSU ROM, then FSBL and so on. During FSBL, the isolation between PS and PL is removed.



IMPORTANT! As the software has gone through a reset cycle, the state of the hardware IPs in PL which continue to run during the PS-only reset may become out of sync with the state of the software which interfaces or controls the IPs. It is your responsibility to make sure that the software and hardware states are properly re-synchronized. In a PS-only reset, you cannot download the bitstream again.

PS-only reset can be initiated by Linux command or watchdog timeout or PMU error management block. If you are interested in PS-only reset without APU/RPU subsystem restart, subsystem/isolation configuration is not required. Linux commands for setting reboot type and reboot will work without additional modifications.

System Reset

In a system-reset, the entire hardware, both PS and PL are reset. After system reset is released, PS executes the standard boot process starting from the PMU ROM, followed by CSU ROM, then FSBL and so on. The following table shows the differences between system reset and POR:

Table 60: Differences between POR and System Reset

POR	System Reset
Reset persistent registers	Preserves persistent registers
Resamples boot mode pins	Does not resample boot mode pins
Reset debug states	Preserves debug states
Resample eFuse values	Requires explicit software action to refresh
Security state determined	Security state locked
Clear tamper response	Preserves tamper response
Select security key source	Security key source locked
Optional LBIST and/or SCAN/CLEAR	Does not run LBIST or SCAN/CLEAR
Run MBIST	Explicit software action needed to run MBIST

System reset can be initiated by Linux command or watchdog timeout or PMU error management block. If you are interested in only System reset without APU/RPU subsystem restart, subsystem/isolation configuration is not required.

Note: System reset is not supported in qspi24 mode on systems with a flash size that is greater than 16 MB.

Idle and Reset of Peripherals

It is necessary to stop/complete any ongoing transaction by any IP or processor of the subsystem before resetting them. Otherwise, it may lead to hanging of the interconnect and eventually hanging of the entire system. Also, to ensure proper operation by the IP after reboot, it is best to reset them and bring them to post bootROM state.

PMU firmware implements peripheral idling and resetting for the PS IPs that can be idled / reset during the subsystem reset. The IPs that will be attempted to idled/reset is based on isolation configuration of the Vivado.

Build PMU firmware with the following idling flags to enable subsystem node idling and resetting:

- ENABLE_NODE_IDLING
- IDLE_PERIPHERALS

Node Reset and Idle

During a subsystem restart, the PMU firmware makes sure that the associated PS peripheral nodes are idled and brought to reset state. Following is the list of currently supported PS peripherals that will undergo idle/reset, if they are part of the subsystem that is undergoing reset:

- TTC
- Ethernet/EMAC
- I2C
- SD
- eMMC
- QSPI
- USB
- DP
- SATA

See GPIO reset to PL to understand the implication of GPIO reset.

Note: PS peripherals are idled prior to invoking resets for user invoked reboot of PS-only and system-reset command.

Custom Hooks

PMU firmware does not keep track of PL peripherals. Hence, there is no idle/reset function implementation available in the PMU firmware. However, it is necessary to treat those peripherals in the same the PS peripherals are treated. You can add a custom hook in the `idle_hooks.c` file to idle the PL peripherals and reset them. These hooks can be called from the `PmMasterIdleSlaves` function in the `pm_master.c` file of the PMU firmware.

```
lib/sw_apps/zynqmp_pmufw/src/pm_master.c
:dir:dir -769,6 +769,12 :dir:dir static void PmMasterIdleSlaves(PmMaster*
const master)

PmDbg(DEBUG_DETAILED, "%s\r\n", PmStrNode(master->nid));
```



```

+    /*
+    * Custom hook to idle PL peripheral before PS peripheral idle
+    */
+
+    Xpfw_PL_Idle_HookBeforeSlaveIdle(master);
+
+    while (NULL != req) {
+        u32 usage = PmSlaveGetUsageStatus(req->slave, master); Node = &req->slave->node;
+        :dir:dir -783,6 +789,11 :dir:dir static void PmMasterIdleSlaves(PmMaster*
+        const master)
+    }
+    req = req->nextSlave;
+    }
+
+    /*
+    * Custom hook to idle PL peripheral after PS peripheral idle
+    */
+    Xpfw_PL_Idle_HookAfterSlaveIdle(master);
+    #endif
+    }
    
```

The `Xpfw_PL_Idle_HookBeforeSlaveIdle` and `Xpfw_PL_Idle_HookAfterSlaveIdle` can contain the code to idle the PL peripherals and reset them if necessary. The implementation can be either of the following:

- Write AXI registers of PL IPs to bring them to idle state and reset. This is the preferred and a graceful way to idle PL peripherals.
- Implement a signal based handshake where PMU firmware signals PL to idle all PL IPs. This implementation should be used when there is no direct control to gracefully stop traffic. For example, you can use this implementation if there are non DMA PL IPs, which does not have reset control but are connected through a firewall IP. This implementation also allows stopping all traffic passing through it unlike the other where each IP needs to be idled individually.

Note: Implementation for these custom hooks is not provided by Xilinx.

GPIO Reset to PL

Vivado configuration allows you to enable fabric resets from PS to PL. The following figure shows that the Zynq UltraScale+ block outputs `pl_resetn0` and `pl_resetn1` signals with Fabric Reset Enabled and the Number of Fabric Resets set to 2, can be used to drive reset pins of PL components.

Figure 65: Resets from PS to PL

The `pl_resetn` signals are implemented with PS GPIOs. `Pl_resetn` pins are released after bitstream configuration in software using the `psu_ps_pl_reset_config_data` function. In the case where a subsystem also uses GPIO for purpose other than reset, the GPIO block is included in the subsystem definition. The image below shows an example of an APU subsystem with GPIO as a slave peripheral.

Figure 66: APU Subsystem with GPIO

Isolation Configuration

Please review **Known Limitations** under the **Isolation Configuration** Section of PG201.

Enable Isolation
 Enable Secure Debug
 Lock Unused Memory

Search:

Name	Start Address	Size	Unit	TZ Settings
APU				
> Masters				
> Slaves				
> Memory				
> Peripherals				
CAN1	0xFF070000	64	KB	NonSecure
GEM3	0xFF0E0000	64	KB	NonSecure
I2C0	0xFF020000	64	KB	NonSecure
UART0	0xFF000000	64	KB	NonSecure
TTC0	0xFF110000	64	KB	NonSecure
TTC1	0xFF120000	64	KB	NonSecure
SWDT0	0xFF150000	64	KB	NonSecure
GPIO	0xFF0A0000	64	KB	NonSecure

In the case where GPIO is a subsystem slave peripheral, the entire GPIO component will be reset as part of the restart process when the subsystem is being restarted. Since `pl_resetn` are implemented with GPIOs, `pl_resetn` will be forced low during subsystem restart. This behavior may be undesirable if the `pl_resetn` signals are being used to drive PL IPs in subsystems other than the one being reset. For example, if `pl_resetn0` drives resets to PL IP for APU subsystem and `pl_resetn1` drives PL IPs for RPU subsystem.

During APU subsystem restart, both `pl_resetn0` and `pl_resetn1` will be forced into the reset state. Consequently, PL IPs in RPU subsystem will be reset. This is the wrong behavior since APU-restart should not affect the RPU subsystem as the GPIO is implicitly shared between the APU and RPU subsystem via `pl_resetn` signals. Since sharing of peripherals is not supported for subsystem restart, `pl_resetn` causes problems during subsystem reset. The work-around is to skip idling and resetting GPIO peripheral during any subsystem restart even if the component is assigned in the subsystem/isolation configuration.

To skip the GPIO reset during the node idling and reset, build the PMU firmware with following flag:

`REMOVE_GPIO_FROM_NODE_RESET_INFO`

Note: GPIO component goes through a reset cycle also during PS-only reset. PMU firmware enables PS-PL isolation prior to calling PS only reset which locks `pl_resetn` to High. However, as soon as FSBL removes the PS-PL isolation, the reset goes Low. FSBL then calls `psu_ps_pl_reset_config_data` to reconfigure `pl_resetn` back to High. This is needed since resetting the PL components allows proper synchronization of software and hardware states after reset.

Recovering from a Hang System

In an event of system hang, as indicated by FPT WDT timeout, PMU can be used to carry out a sequence of events to try and recover from the unresponsive condition. By default, when FPD WDT times out, PMU firmware will not invoke any type of restart. This is so that user can specify the exact desired behavior. However, Xilinx provides a typical recovery scheme in which PMU firmware monitors the state of APU subsystem using FPD WDT and restart APU (Linux) subsystem if the timer expires, indicating problem with Linux.

Since RPU subsystem is managed by Linux using remoteproc, the life-cycle of the RPU subsystem is completely up to Linux. PMU is not involved in deciding when to restart RPU subsystem(s). RPU hang recovery can also be implemented with help of either software or hardware watchdog between APU and RPU subsystems. In that case, the watchdog is configured and handled by Linux but the heartbeats is provided by RPU application(s). The exact method of deciding when to restart RPU is up to the user, watchdog is simply one of many possibilities. To enable recovery, PMU firmware should be built with enabling error management and recovery. Following macros enable the Recovery feature:

- ENABLE_EM
- ENABLE_RECOVERY

It is also necessary to build ATF with following flags (see APU Idling for details):

```
ZYNQMP_WARM_RESTART=1
```



IMPORTANT! *One TTC timer (timer 9) will be reserved for PMU's use when these compile flags are enabled.*

Watchdog Management

The FPD WDT is used for monitoring APU state. Software running on APU periodically touch FPD WDT to keep it from timing out. The occurrence of WDT timeout indicates an unexpected condition on the APU which prevents the software from running properly and an APU restart is invoked. FPD WDT is configured by PMU firmware at initialization stage, but is periodically serviced by software running on APU.

The default timeout configured for WDT is 60 seconds and can be changed by RECOVERY_TIMEOUT flag in PMU firmware. When APU subsystem goes into a restart cycle, FPD WDT is kept running to ensure that the restart lands in a clean running state where software running on APU is able to touch the WDT again. Therefore, the timeout for the WDT must be long enough to cover the entire APU subsystem restart cycle to prevent the WDT from timing out in the middle of restart process. It is advisable to start providing the heartbeat as soon as is

feasible in Linux. PetaLinux BSP includes recipe to add the watchdog management service in `init.d`. As FPD WDT is owned by PMU firmware, it would be unsafe to use full fledged Linux driver for handling WDT. It is advisable to just pump the heartbeats by writing restart key (0x1999) to restart register (WDT base + 0x8) of the WDT. It can be done through C program daemon or it can be part of bash script daemon.

It is recommended to be part of idle thread or similar low priority thread, which if hangs we should consider the subsystem hang.

The following is the snippet of the single heartbeat stroke to the FPD WDT from command prompt. This can be included in the bash script which runs periodically.

```
# devmem 0xFD4D0008 32 0x1999
```

The following `wdt-heartbeat` application periodically provides the heartbeat to FPD WDT. For demo purpose this application is launched as daemon. The code from this application can be implemented in appropriate location such as an idle thread of Linux.

```
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>

#define WDT_BASE    0xFD4D0000
#define WDT_RESET_OFFSET    0x8
#define WDT_RESET_KEY    0x1999

#define REG_WRITE(addr, off, val) (*(volatile unsigned int*)(addr+off)=(val))
#define REG_READ(addr,off) (*(volatile unsigned int*)(addr+off))

void wdt_heartbeat(void)
{
    char *virt_addr; int fd;
    int map_len = getpagesize();
    fd = open("/dev/mem", (O_RDWR | O_SYNC)); virt_addr = mmap(NULL,
    map_len, PROT_READ|PROT_WRITE,
    MAP_SHARED,
    fd, WDT_BASE);

    if (virt_addr == MAP_FAILED) perror("mmap failed");

    close(fd);

    REG_WRITE(virt_addr,WDT_RESET_OFFSET, WDT_RESET_KEY);

    munmap((void *)virt_addr, map_len);
}
int main()
{
    while(1)
    {
        wdt_heartbeat(); sleep(2);
    }
    return 0;
}
```

On the expiry of watchdog, PMU firmware receives and handles the WDT interrupt. PMU firmware idles the subsystem's master CPU i.e., all A53 cores (see APU Idling), and then carries out APU only restart flow which includes CPU reset and idling and resetting peripherals (see Peripheral Idling) associated to the subsystem reset.

Note: If ESCALATION is enabled PMU firmware will trigger the appropriate restart flow (which can be other than APU only restart) as explained in Escalation section.

APU Idling

Each A53 is idled by taking them to the WFI state. This is done through Arm Trusted Firmware (ATF). For idling CPU, the PMU firmware raises TTC interrupt (timer 9) to ATF, which issues software interrupt to each alive A53 core. The respective cores then clears the pending SGI on itself and put itself into WFI.

The last core just before going into WFI issues `pm_system_shutdown` (PMU firmware API) to PMU firmware, which then performs APU only restart flow.

This feature must be enabled in ATF for recovery to work properly. It can be enabled by building ATF with `ZYNQMP_WARM_RESTART=1` flag.

Modifying Recovery Scheme

When `ENABLE_RECOVERY` is turned on, Xilinx provides a recovery implementation in which a FPD WDT timeout results in the invocation of APU subsystem restart. You can easily modify the recovery behavior by modifying the code. Alternatively, an example of PMU firmware invoking system-reset on FPD WDT timeout is detailed in Xilinx Answer: [69423](#).

Escalation

If current recovery cannot bring the system back to the working state, the system must escalate to a more severe type of reset on the next WDT expiry in order to try and recover fully. It is up to you to decide on the escalation scheme. A commonly used scheme starts with APU-restart on the first watchdog expiration, followed by PS-only reset on the next watchdog expiration, then finally system-reset.

To enable escalation, PMU firmware must be built with following flags:

```
ENABLE_ESCALATION
Escalation Scheme
```

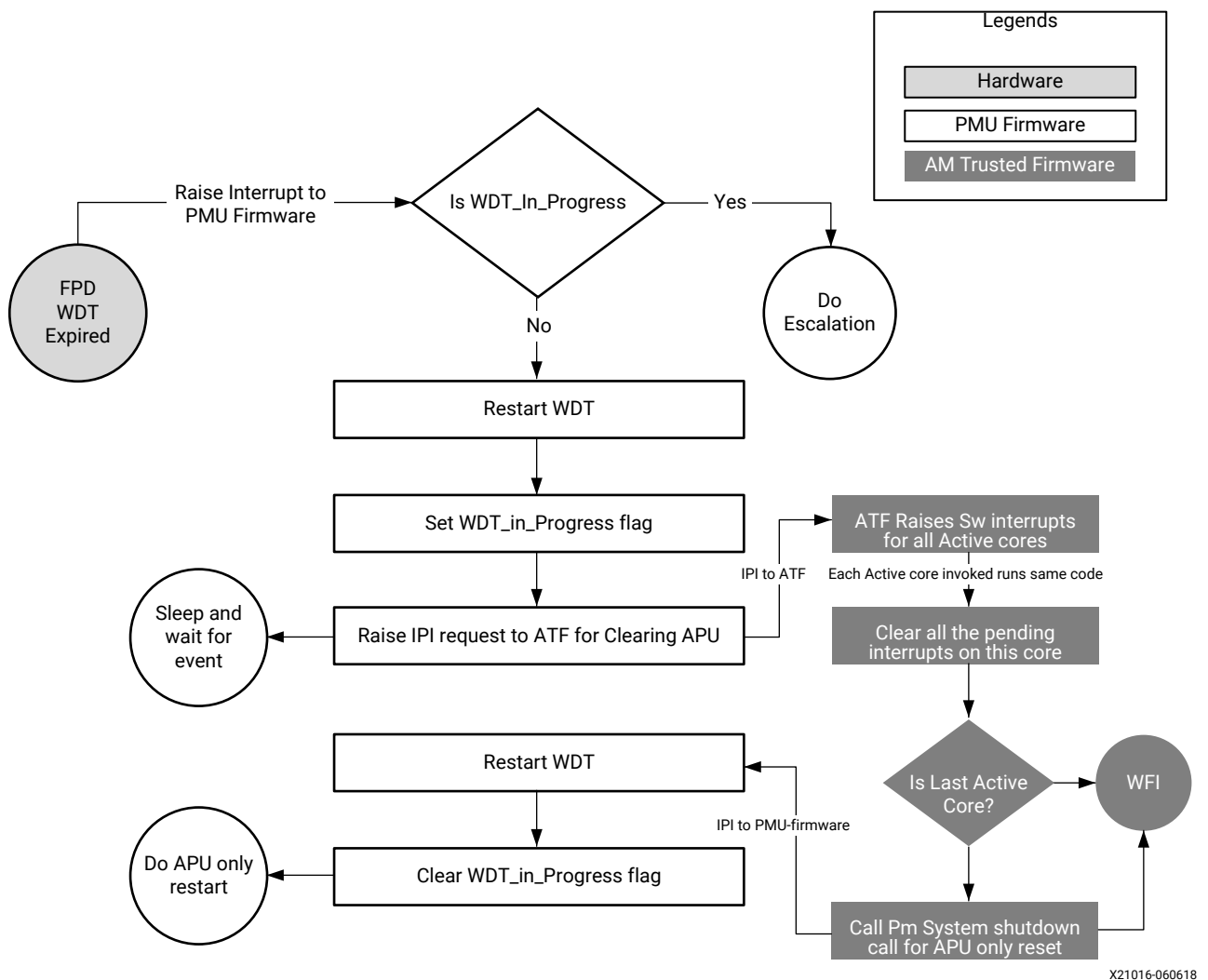
Default Scheme

Default escalation scheme checks for the successful `pm_system_shutdown` call from ATF for APU-only restart which happens when the ATF is able to successfully idle all active CPUs. If ATF is not successful in idling the active cores, WDT will time out again with the `WDT_in_Progress` flag set, resulting in do escalation.

Escalation will trigger System level reset. System level reset is defined as PS only reset if PL is present or System restart if PL is not present.

The following figure shows the flow of the control in case of default escalation scheme.

Figure 67: name



Healthy Bit Scheme

Default scheme for escalation does not guarantee the successful reboot of the system. It only guarantees the successful role of ATF to idle the CPU during the recovery. Consider the scenario in which the FPD_WDT has timed out and APU subsystem restart is called in which ATF is able to successfully make the `pm_system_shutdown` call. However, APU subsystem restart is far from finished after `pm_system_shutdown` is called. The restart process can be stuck elsewhere, such as `fsbl`, `u-boot` or Linux init state. If the restart process is stuck in one of the aforementioned tasks, FPD_WDT will expire again, causing the same cycle to be repeated as long as ATF is loaded and functioning. This cycle can continue indefinitely without the system booting back into a clean running state.

The Healthy Bit scheme solves this problem. In addition to default scheme, the PMU firmware checks for a Healthy Bit, which is set by Linux on successful booting. On WDT expiry, if Healthy Bit is set, it indicates that Linux is able to boot into a clean running state, then no escalation is needed. However, if Healthy Bit is not set, that means the last restart attempt did not successfully boot into Linux and escalation is needed. There is no need to repeat the same type of restart. PMU firmware will escalate and call a system level reset.

Healthy Bit scheme is implemented using the bit-29 of PMU global general storage register (PMU_GLOBAL_GLOBAL_GEN_STORAGE0[29]). PMU firmware clears the bit before starting the recovery or normal reboot and Linux must set this bit to flag a healthy boot.

PMU global registers are accessed through `sysfs` interface from Linux. Hence, to set the healthy bit from the Linux, execute the following command (or include in the code):

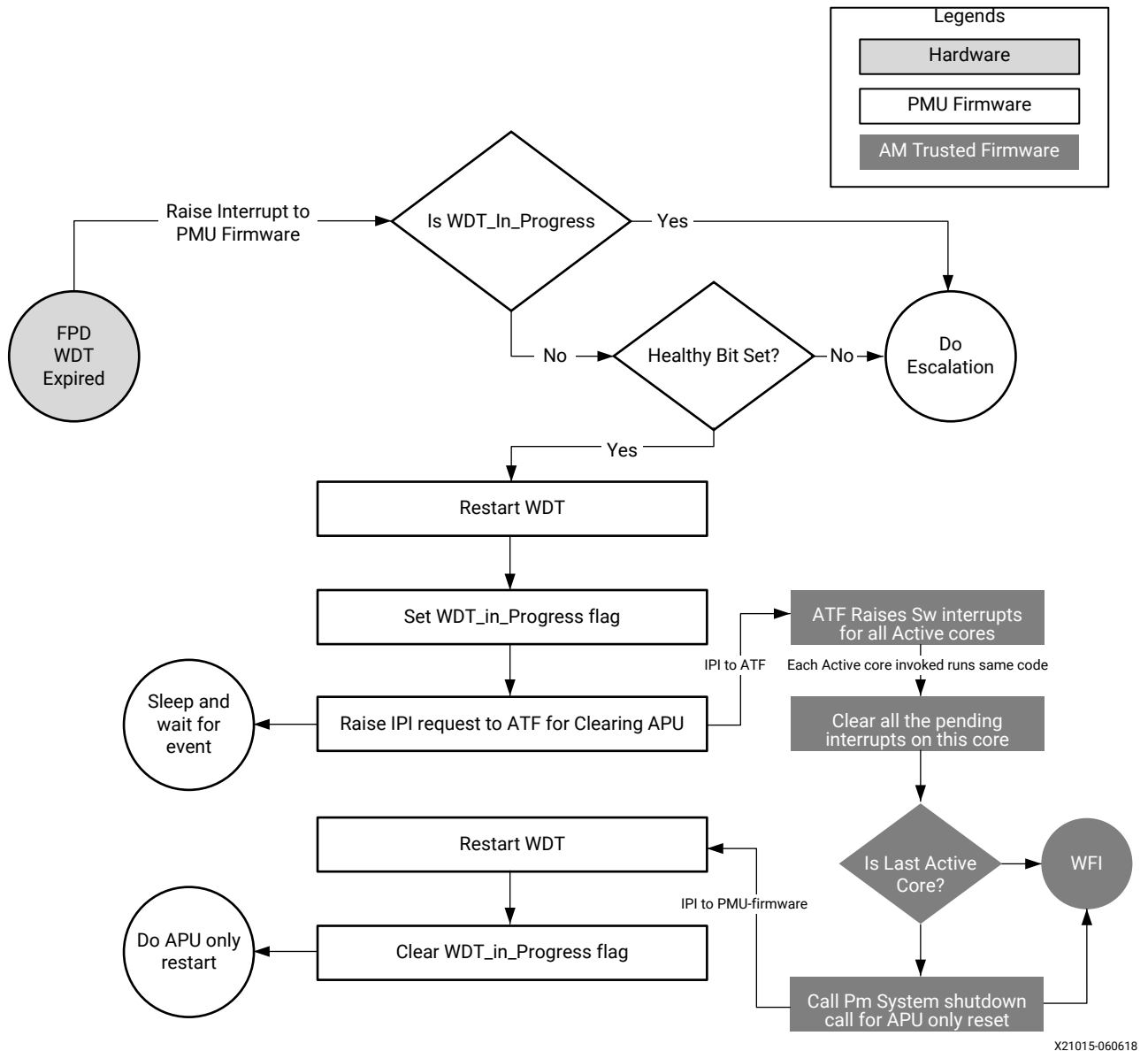
```
# echo "0x20000000 0x20000000" > "/sys/devices/platform/firmware/ggs0"
```

To enable the healthy bit based escalation scheme, build the PMU firmware with the following flag:

```
CHECK_HEALTHY_BOOT
```

The following figure shows the flow of the control in case of the healthy bit escalation scheme.

Figure 68: Healthy Bit Escalation Scheme



Customizing Recovery and Escalation Scheme

By default, when FPD WDT times out, PMU FW will not invoke any type of restart. While Xilinx has provided predefined RECOVERY and ESCALATION behaviors, users can easily customize different desired schemes.

When FPD_WDT times out, it calls `FpdSwdtHandler`. If `ENABLE_EM` is defined, `FpdSwdtHandler` calls `XPfw_recoveryHandler`. It is otherwise an empty function.

In `xp_fw_mod_em.c`,

```
#ifdef ENABLE_EM
oid FpdSwdtHandler(u8 ErrorId)
{
XPfw_Printf(DEBUG_ERROR, "EM: FPD Watchdog Timer Error (Error ID: %d)\r\n",
ErrorId);
XPfw_RecoveryHandler(ErrorId);
}

#else
void FpdSwdtHandler(u8 ErrorId) { }
```

Without `ENABLE_EM`, you can simply update `FpdSwdtHandler` which will be called at FPD Timeout. With `ENABLE_EM` turned on, you need to update `XPfw_recoveryHandler`.

Similarly, turning on `RECOVERY` defines the `XPfw_RecoveryHandler` (see `xp_fw_restart.c`). Unless `RECOVERY` is turned on, `XPfw_RecoveryHandler` is an empty function and nothing will happen when `FPD_WDT` times out.

`RecoveryHandler` basically follows the flow chart detailed in the Escalation Scheme section. When `FPD_WDT` times out, the code follows the progression of orange boxes. If `WDT` is not already in progress, Restart `WDT`, Set `WDT_In_Progress` flag, Raise `TTC` (timer 9) interrupt to `ATF`. Then `ATF` takes over. It Raises `SW` interrupt for all active cores, clear pending interrupts, etc. (see blue boxes). Essentially, `PMU` restarts and boosts the `WDT`, then sends a request to `ATF`. `ATF` cleanly idles all four `APUs` and when they all get to `WFI` (`Last Active Core is true`), `ATF` issues `PMU System Shutdown` with `APU subsystem` as argument back to `PMU`. When `PMU` gets this command, it invokes `APU subsystem restart`.

If `ENABLE_ESCALATION` is not set, the code never takes the Do Escalation path. If the `RecoveryHandler` hangs for some reason (for example, something went wrong and `APU` cannot put all four `CPU` cores to `WFI`), it keeps retrying `APU restart` or hang forever. When `ENABLE_ESCLATION` is on and if anything goes wrong during execution of the flowchart, it will look like `WDT` is still in progress (since `clear WDT_in_progress` flag happens only as the last step), Do Escalation will call `SYSTEM_RESET` instead of trying `APU-restart` again and again.

To customize recovery and escalation behavior, use the provided `XPfw_recoveryHandler` as a template to provide a customized `XPfw_recoveryHandler` function.

Building Software

All the software components are built and packaged by Xilinx PetaLinux tool. See [PetaLinux wiki page](#) for more information on how to build and package software components.

Build Flag for Restart Solution

Following build time flags are not set by default and can alter the behavior of the restart in Zynq UltraScale+ MPSoC:

Table 61: Build Time Flags

Component	Flag Name	Description	Dependency
PMU firmware	ENABLE_EM	Enable error management and provide WDT interrupt handling. This is not directly related to restart solution but needed for recovery.	
	ENABLE_RECOVERY	Enable Recovery during WDT expiry	
	ENABLE_ESCALATION	Allow escalation on failure of boot or recovery	
	CHECK_HEALTHY_BOOT	Use Healthy bit to determine escalation	
	IDLE_PERIPHERALS ENABLE_NODE_IDLING	Both the flags must be used together to allow PMU firmware to attempt peripherals node idling (and reset).	
	REMOVE_GPIO_FROM_NODE_RESET_INFO	Skips GPIO from the node idling and resetting list. This is needed when the system is using GPIO to provide reset (or similar) signals to PL or other peripherals outside current subsystem. If this flag is set, GPIO is not reset.	
ATF	ZYNQMP_WARM_RESTART=1	Enable WARM RESTART recovery feature in ATF that allow the CPU idling triggered from PMU firmware.	
FSBL	FSBL_PROT_BYPASS	Skip XMPU/XPPU based configuration for system except for DDR and OCM.	
Linux	CONFIG_SRAM	Needed for Remoteproc to work for load RPU images in the TCM.	

Modifying Component Recipes

Each component's recipe can be changed to either include the build time compilation flags or to include patches for custom code modification/addition. PetaLinux provides meta-user Yocto based layer for user specific modifications. The layer can be found in project directory `project-spec/meta-user/` location.

PMU Firmware

User specific recipe for PMU firmware can be found in the following location:

`dir:project-spec/meta-user/recipes-bsp/pmu/pmu-firmware_%.bbappend` (if doesn't exist please create this file at this path).

The PMU firmware code can be modified by patches against embeddedsw GitHub repo. Location for the source code is `embeddedsw/tree/master/lib/sw_apps/zynqmp_pmufw`. The patches should be copied to `project-spec/meta-user/recipes-bsp/pmu/files` directory and the same patch names should be added `pmu-firmware_%.bbappend` file.

Example:

If `my_changes.patch` (against PMU firmware source) is to be added and all the flags explained in the Build Time Flags in [Building Software](#) are to be enabled (set), then `project-spec/meta-user/recipes-bsp/pmu/pmu-firmware_%.bbappend` may look like the following file:

```
YAML_COMPILER_FLAGS_append = " -O2 -DENABLE_EM -DENABLE_RECOVERY
-DENABLE_ESCALATION -DENABLE_NODE_IDLING -DREMOVE_GPIO_FROM_NODE_RESET_INFO
-DCHECK_HEALTHY_BOOT -DIDLE_PERIPHERALS "

FILESEXTRAPATHS_prepend := "${THISDIR}/files:" SRC_URI_append = " file://
my_changes.patch"
```

FSBL

User specific recipe for the FSBL can be found in the following location:

`dir:project-spec/meta-user/recipes-bsp/fsbl/fsbl_%.bbappend` (if does not exist, please create this file at this path). The FSBL code can be modified by patches against [embeddedsw GitHub repo](#). Location for the source code is as follows:

```
embeddedsw/tree/master/lib/sw_apps/zynqmp_fsbl
```

The patches should be copied to `project-spec/meta-user/recipes-bsp/fsbl/files` directory and the same patch names should be added to `fsbl_%.bbappend` file.

Example:

If `my_changes.patch` (against the FSBL source) is to be added and all the flags explained in the Build Time Flags in [Building Software](#) are to be enabled (set), then the modified `project-spec/meta-user/recipes-bsp/fsbl/fsbl_%.bbappend` file will look like the following file (XPS_BOARD_ZCU102 flag was already existing):

```
YAML_COMPILER_FLAGS_append = " -DXPS_BOARD_ZCU102 -DFSBL_PROT_BYPASS "
FILESEXTRAPATHS_prepend := "${THISDIR}/files:"
SRC_URI_append = " file://my_changes.patch"
```

ATF

User specific recipe for ATF can be found in the following location:

dirproject-spec/meta-user/recipes-bsp/arm-trusted-firmware/arm-trusted-firmware_%.bbappend file (if it doesn't exist, create this file in this path). You can find the ATF files in [Git repository for arm trusted firmware](#).

Example:

To add warm restart flag to ATF, project-spec/meta-user/recipes-bsp/arm-trusted-firmware/arm-trusted-firmware_%.bbappend will look like the following file:

```
#
# Enabling warm restart feature
#
EXTRA_OEMAKE_append = " ZYNQMP_WARM_RESTART=1"
```

Linux

There are many ways to add /modify Linux configuration. See *PetaLinux Tools Documentation: Reference Guide (UG1144)* for the same.

User specific recipe for Linux kernel can be found in the following location:

project-spec/meta-user/recipes-kernel/linux/linux-xlnx_%.bbappend (if it doesn't exist, create this file at this path).

You can find the Linux files at [Git Repository for Linux Example](#):

To add SRAM config to Linux, create the following bsp.cfg file:

```
project-spec/meta-user/recipes-kernel/linux/linux-xlnx/bsp.cfg

CONFIG_SRAM=y
```

Add this file in the following bbappend file of Linux:

```
project-spec/meta-user/recipes-kernel/linux/linux-xlnx_%.bbappend

SRC_URI += "file://bsp.cfg"
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
```

Modifying Device Tree

User specific recipe for device tree can be found in the following location:

project-spec/meta-user/recipes-bsp/device-tree/device-tree-generation_%.bbappend. This file contains the following contents:

```
SRC_URI_append = "\ file://system-user.dtsi \
"
FILESEXTRAPATHS_prepend := "${THISDIR}/files:"
```

The content of `system-user.dtsi` in `project-spec/meta-user/recipes-bsp/device-tree/files` directory is as follows:

```
/include/ "system-conf.dtsi"
/ {
};
```

This file can be modified to extend the device tree functionality by adding, removing, or modifying the DTS nodes.

Example: Adding DT node(s) [remoteproc RPU split mode]

The overlay dtsi(s) can be added in `files/` directory (remember to update `bbappend` file accordingly) and included in `system-user.dtsi`. For adding remoteproc related entries to enable RPU subsystem to load, unload, or restart, add a new overlay file called `remoteproc.dtsi`.

Note: This is for split mode. Check open amp documentation for lockstep and other possible configurations.

File: `remoteproc.dtsi`

```
/ {
    reserved-memory {
        #address-cells = <2>;
        #size-cells = <2>; ranges;
        rproc_0_reserved: rproc:dir3ed000000 { no-map;
            reg = <0x0 0x3ed00000 0x0 0x1000000>;
        };
    };

    power-domains {

        pd_r5_0: pd_r5_0 {
            #power-domain-cells = <0x0>; pd-id = <0x7>;
        };

        pd_r5_1: pd_r5_1 {
            #power-domain-cells = <0x0>; pd-id = <0x8>;
        };

        pd_tcm_0_a: pd_tcm_0_a {
            #power-domain-cells = <0x0>; pd-id = <0xf>;
        };

        pd_tcm_0_b: pd_tcm_0_b {
            #power-domain-cells = <0x0>; pd-id = <0x10>;
        };
    };
};
```

```

};

pd_tcm_1_a: pd_tcm_1_a {
#power-domain-cells = <0x0>;

pd-id = <0x11>;
};

pd_tcm_1_b: pd_tcm_1_b {
#power-domain-cells = <0x0>; pd-id = <0x12>;
};
amba {

r5_0_tcm_a: tcm:dirffe00000 { compatible = "mmio-sram";
reg = <0x0 0xFFE00000 0x0 0x10000>;

pd-handle = <&pd_tcm_0_a>;

};

r5_0_tcm_b: tcm:dirffe20000 { compatible = "mmio-sram";
reg = <0x0 0xFFE20000 0x0 0x10000>;

pd-handle = <&pd_tcm_0_b>;

};

r5_1_tcm_a: tcm:dirffe90000 { compatible = "mmio-sram";
reg = <0x0 0xFFE90000 0x0 0x10000>;

pd-handle = <&pd_tcm_1_a>;

};

r5_1_tcm_b: tcm:dirffeb0000 { compatible = "mmio-sram";
reg = <0x0 0xFFEB0000 0x0 0x10000>;

pd-handle = <&pd_tcm_1_b>;

};

elf_dds_0: ddr:dir3ed00000 { compatible = "mmio-sram";
reg = <0x0 0x3ed00000 0x0 0x40000>;

};

elf_dds_1: ddr:dir3ed40000 { compatible = "mmio-sram";
reg = <0x0 0x3ed40000 0x0 0x40000>;

};

test_r50: zynqmp_r5_rproc:dir0 {

compatible = "xlnx,zynqmp-r5-remoteproc-1.0";

reg = <0x0 0xff9a0100 0x0 0x100>, <0x0 0xff340000 0x0 0x100>, <0x0
0xff9a0000 0x0 0x100>;
    
```

```

reg-names = "rpu_base", "ipi", "rpu_glbl_base"; dma-ranges;

core_conf = "split0"; sram_0 = <&r5_0_tcm_a>; sram_1 = <&r5_0_tcm_b>;
sram_2 = <&elf_dds_0>; pd-handle = <&pd_r5_0>;
interrupt-parent = <&gic>; interrupts = <0 29 4>;
} ;
test_r51: zynqmp_r5_rproc:dir1 {
compatible = "xlnx,zynqmp-r5-remoteproc-1.0";
reg = <0x0 0xff9a0200 0x0 0x100>, <0x0 0xff340000 0x0 0x100>, <0x0
0xff9a0000 0x0 0x100>;

reg-names = "rpu_base", "ipi", "rpu_glbl_base"; dma-ranges;
core_conf = "split1"; sram_0 = <&r5_1_tcm_a>; sram_1 = <&r5_1_tcm_b>;
sram_2 = <&elf_dds_1>; pd-handle = <&pd_r5_1>;
interrupt-parent = <&gic>; interrupts = <0 29 4>;
} ;
};
};

```

Now include this node in `system-user.dtsi`:

```

/include/ "system-conf.dtsi"
/include/ "remoteproc.dtsi"
/ {
};

```

For information on OpenAMP and remoteproc, see the [OpenAmp wiki page](#).

Example: Removing DT node(s) [PL node]

It is necessary to remove PL nodes, which are not accessed or dependent on APU subsystem, from the device tree. Again, you can modify `system-user.dtsi` in `project-spec/meta-user/recipes-bsp/device-tree/files` to remove specific node or property.

For example, you can modify the `system-user.dtsi` as following, if you are willing to remove AXI DMA node from the dts:

```

/include/ "system-conf.dtsi"
/include/ "remoteproc.dtsi"
/ {
/delete-node/axi-dma;
};

```

High-Speed Bus Interfaces

The Zynq[®] UltraScale+™ MPSoC has a serial input/output unit (SIOU) for a high-speed serial interface. It supports protocols such as PCIe[®], USD 3.0, DisplayPort, SATA, and Ethernet protocols.

- The SIOU block is part of the full-power domain (FPD) in the PS.
- The USB and Ethernet controller blocks that are part of the low-power domain (LPD) in the Zynq UltraScale+ MPSoC also share the PS-GTR transceivers.
- The interconnect matrix enables multiplexing of four PS-GTR transceivers in various combinations across multiple controller blocks.
- A register block controls or monitors signals within the SIOU.

This chapter explains the configuration flow of the high-speed interface protocols.

See this [link](#) to the “High-Speed PS-GTR Transceiver Interface” of the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)* for more information.

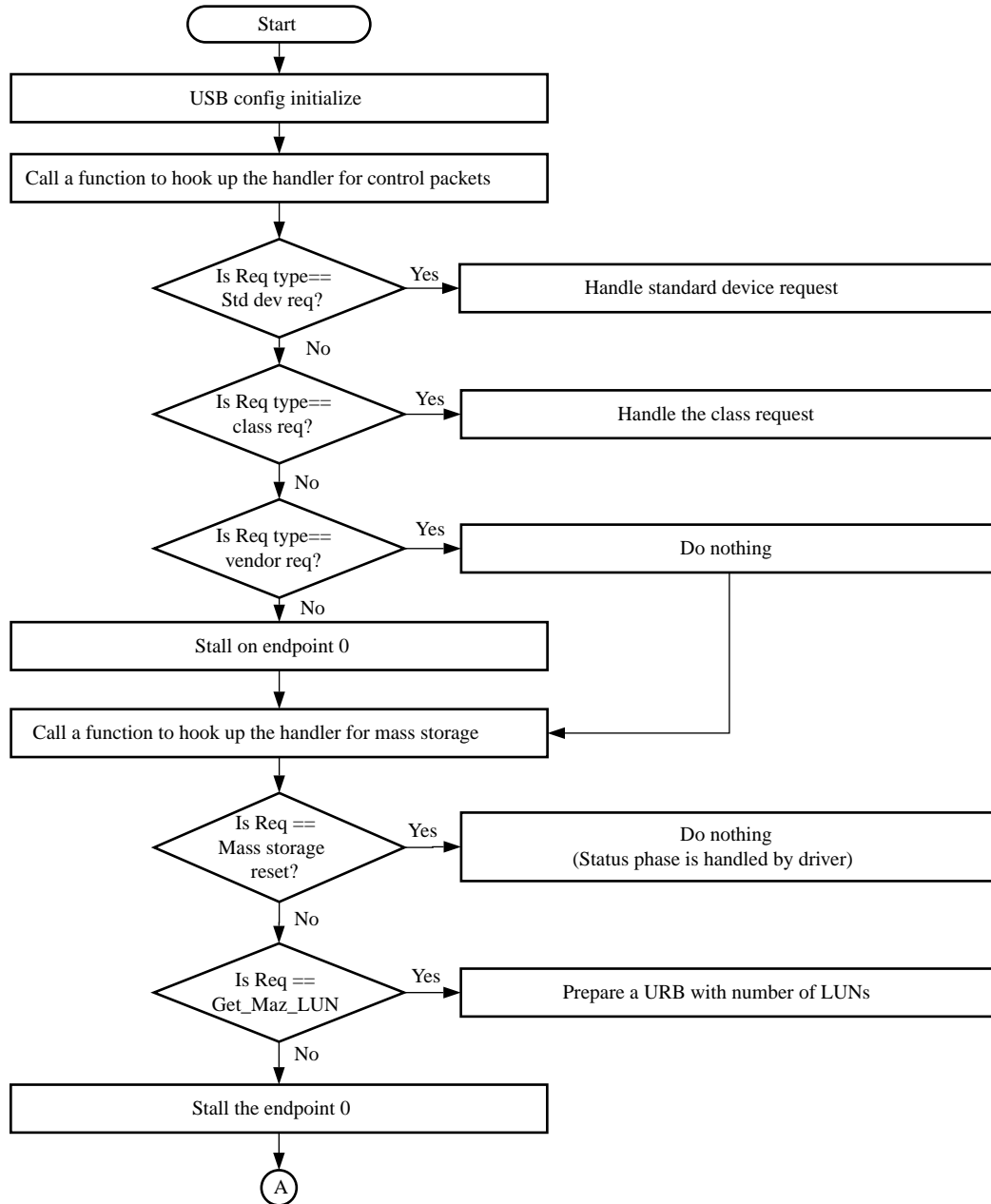
USB 3.0

The Zynq UltraScale+ MPSoC USB 3.0 controller consists of two independent dual-role device (DRD) controllers. Both can be individually configured to work as host or device at any given time. The USB 3.0 DRD controller provides an eXtensible host controller interface (xHCI) to the system software through the advanced eXtensible interface (AXI) slave interface.

- An internal DMA engine is present in the controller and it uses the AXI master interface to transfer data.
- The three dual-port RAM configurations implement the RX data FIFO, TX data FIFO, and the descriptor/register cache.

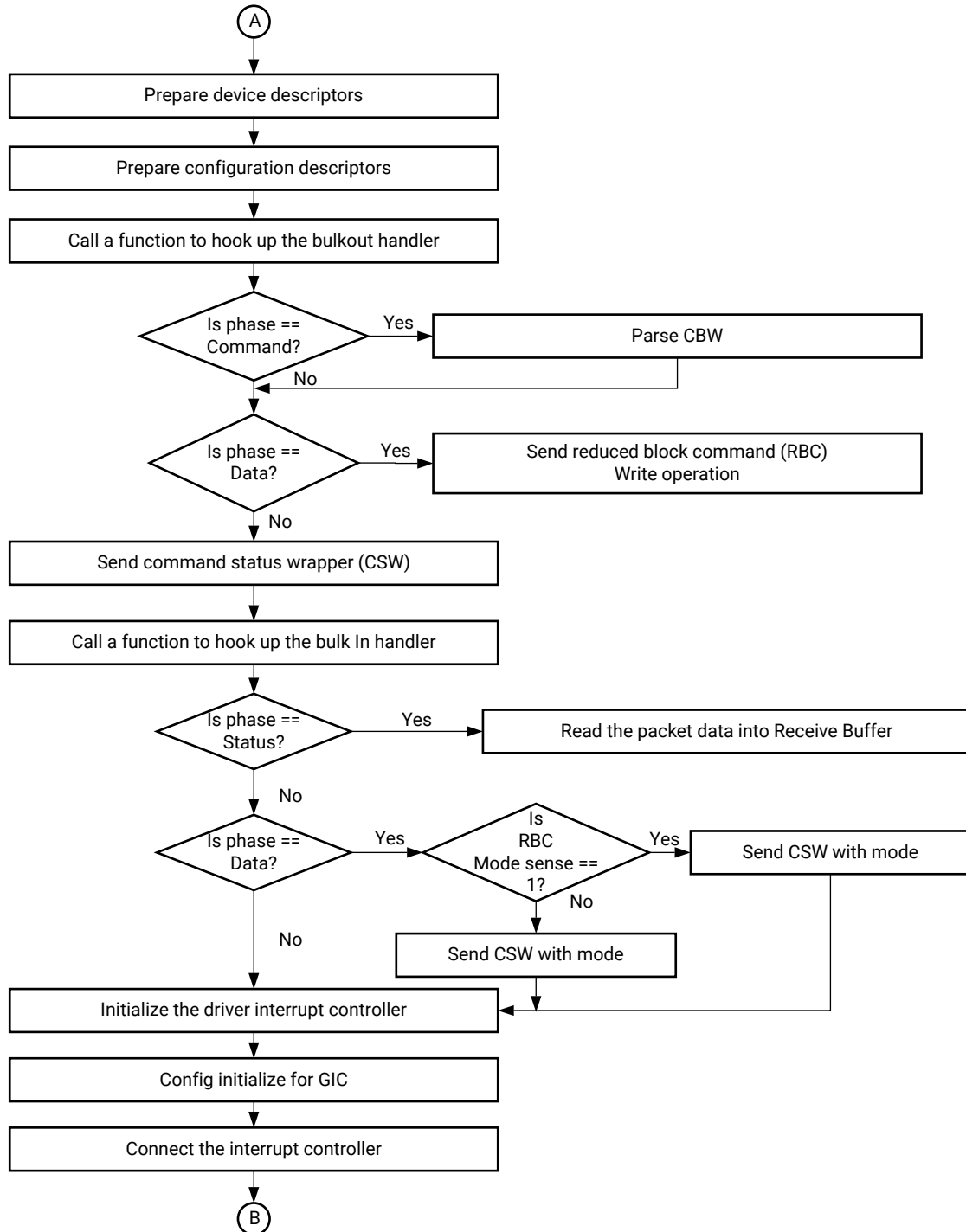
The following flow diagrams illustrate how to configure USB as mass storage device.

Figure 69: USB Example Flow: USB Initialization



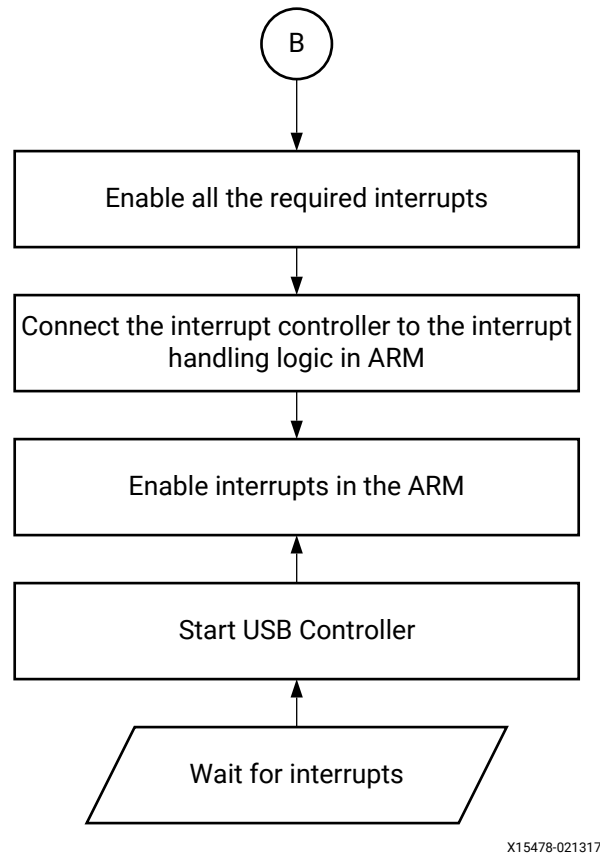
XI5463-071017

Figure 70: Example USB Flow: Hookup Bulk in and Bulk out Handlers and Initialize Interrupt Controller



X15477-071017

Figure 71: Enable Interrupts and Start the USB Controller



For more information on USB controller, see this [link](#) to the “USB 2.0/3.0 Host, Device, and Controller,” chapter of the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*.

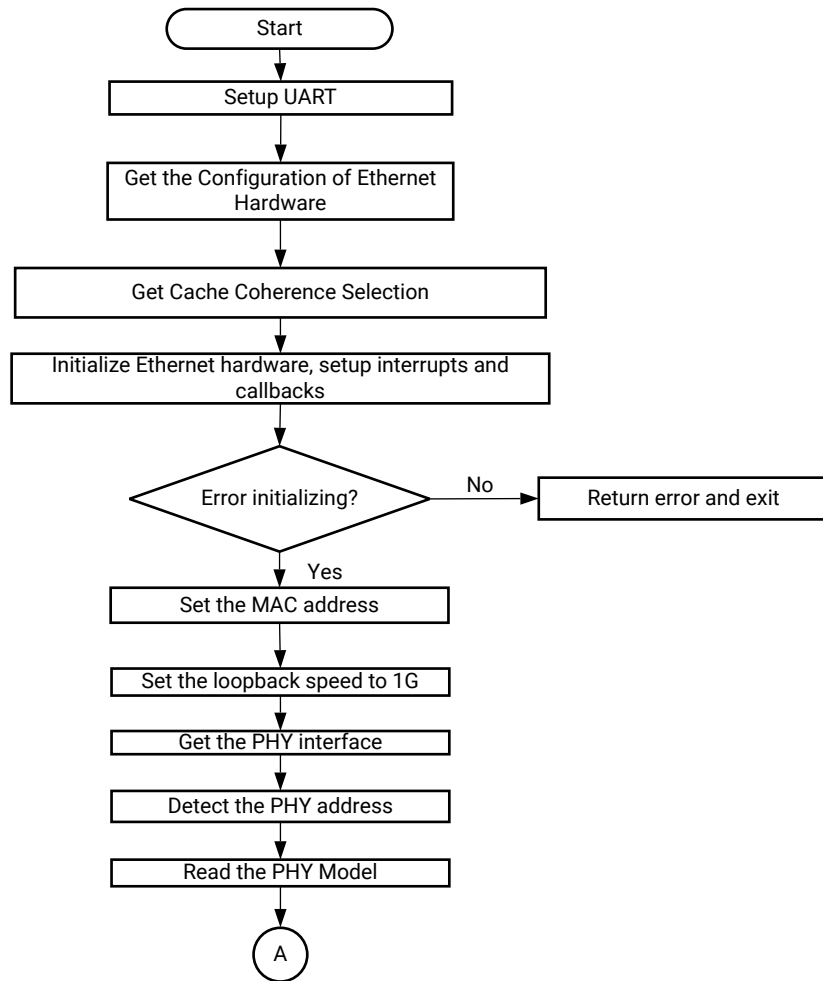
Gigabit Ethernet Controller

The gigabit Ethernet controller (GEM) implements a 10/100/1000 Mb/s Ethernet MAC compatible with IEEE Standard for Ethernet (IEEE Std 802.3-2008) and is capable of operating in either half or full-duplex mode in 10/100 mode and full-duplex in 1000 mode.

The processor system (PS) is equipped with four gigabit Ethernet controllers. Registers are used to configure the features of the MAC, and select different modes of operation. The DMA controller connects to memory through the advanced eXtensible interface (AXI). It is attached to the FIFO interface of the controller of the MAC to provide a scatter-gather type capability for packet data storage in an embedded processing system.

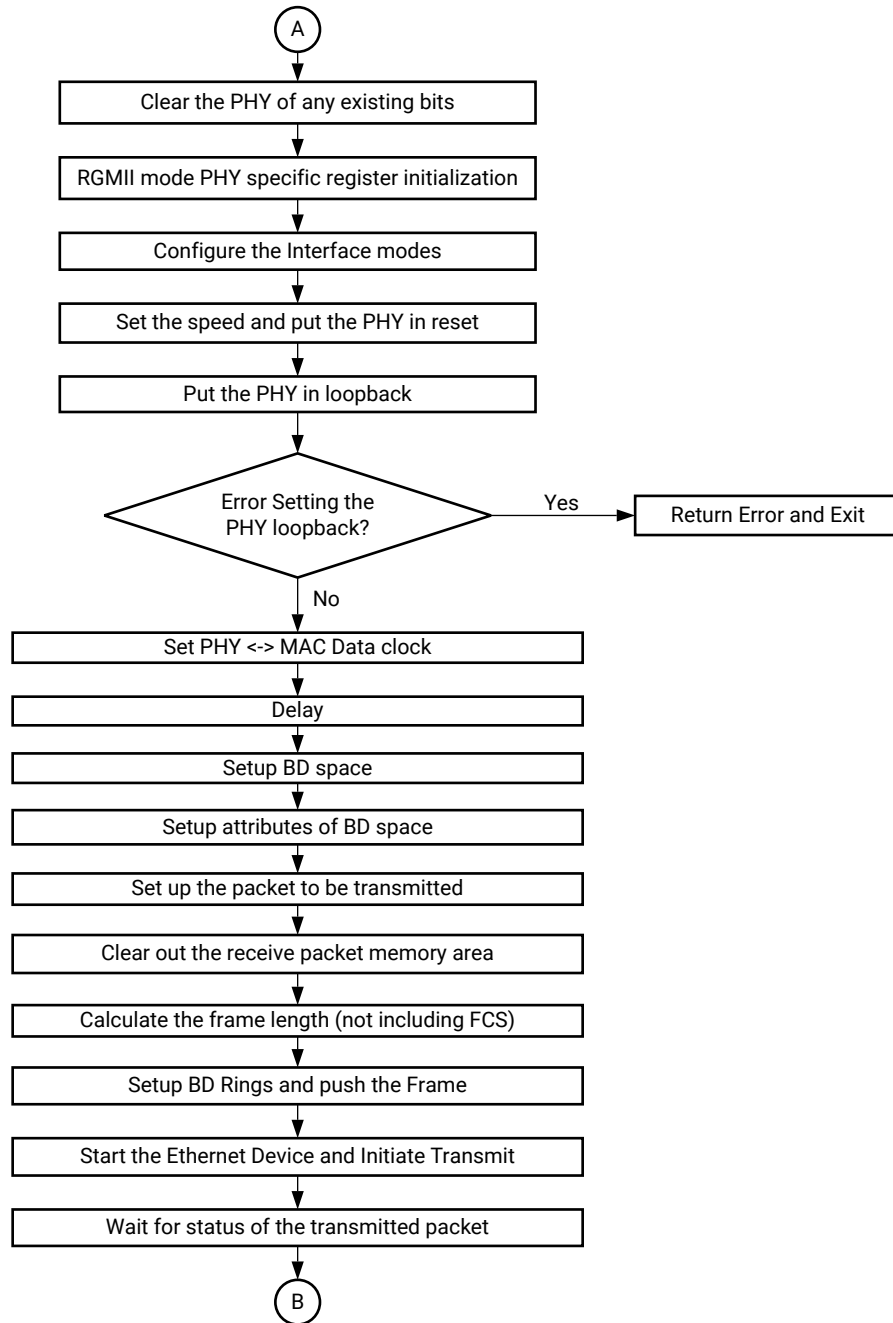
The following figures illustrate an example for configuring an Ethernet controller to send a single packet of data in RGMII mode.

Figure 72: Example Ethernet Flow: Initialize Ethernet Controller



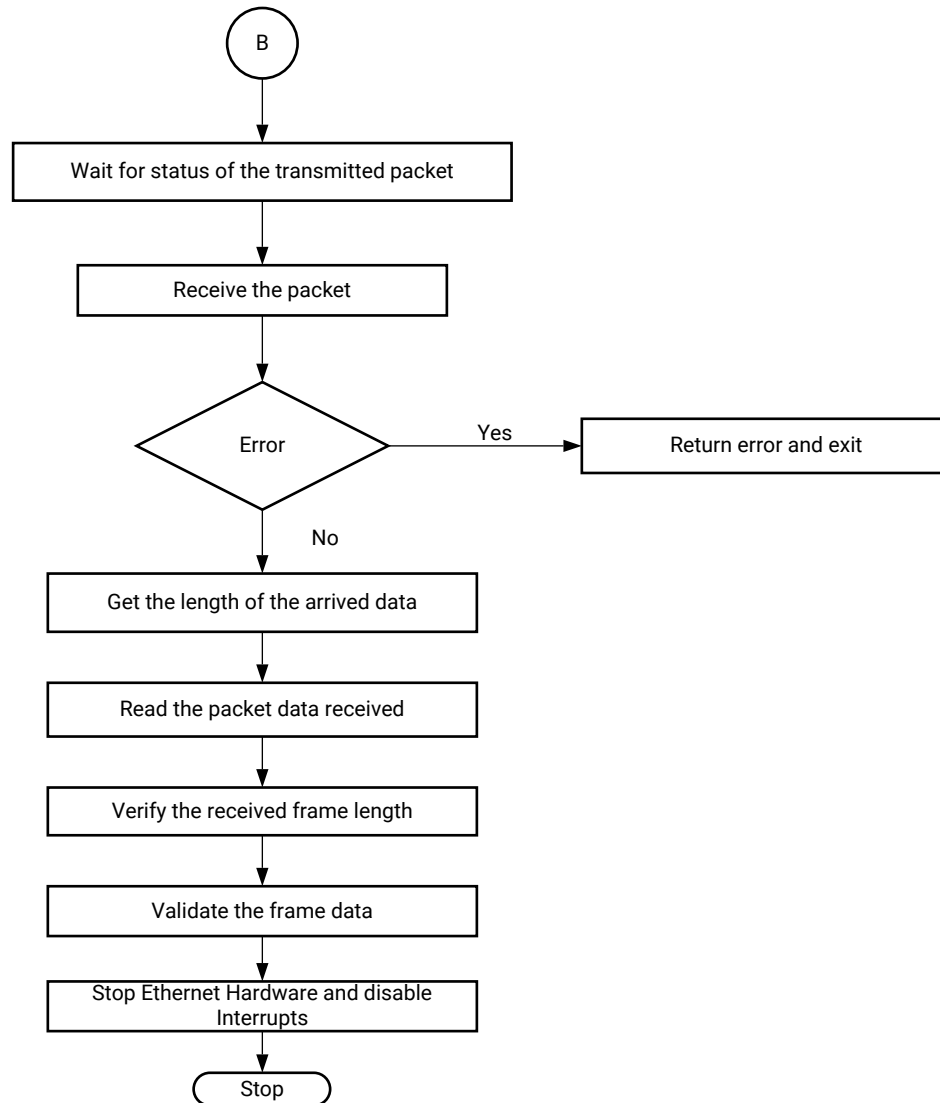
X15462-071017

Figure 73: Example Ethernet Flow: Configure the Ethernet Parameters & Initiate the Transmit



X15479-071017

Figure 74: Example Ethernet Flow: Receive and Validate the Data



X15480-021317

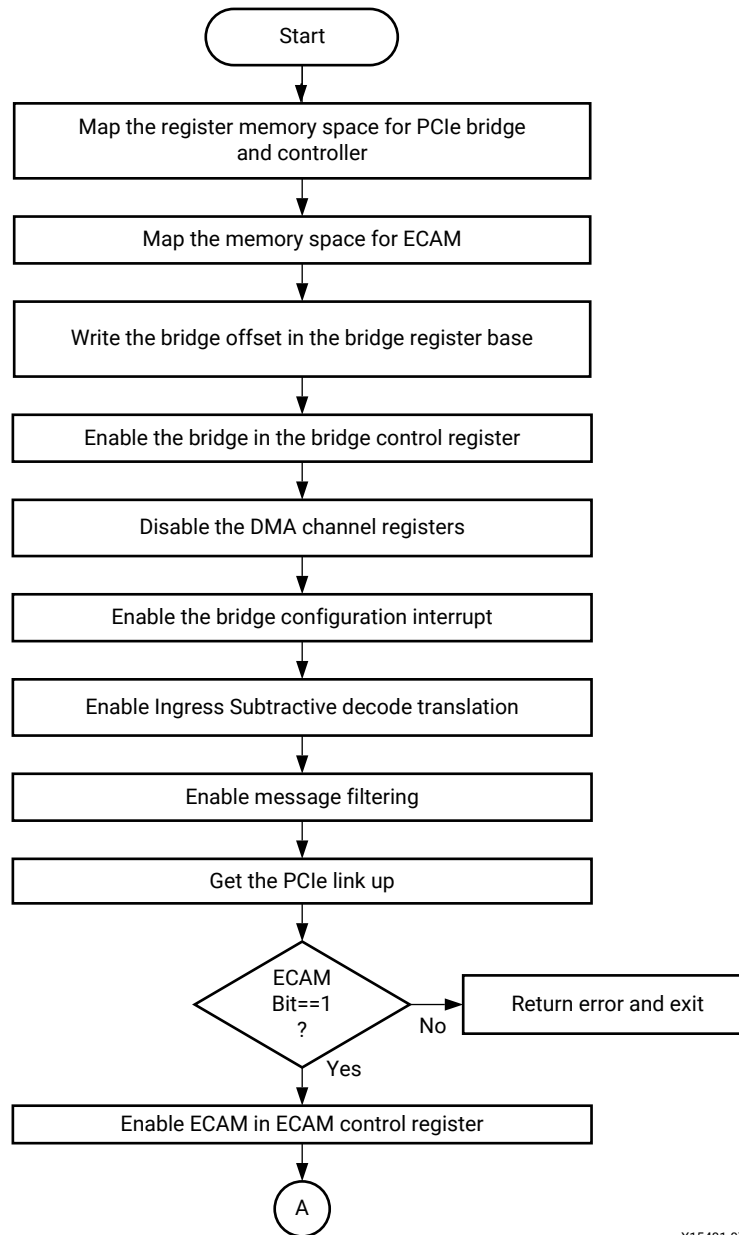
For more information on Ethernet Controller, see this [link](#) to the “Gigabit Ethernet Controller” chapter in the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*.

PCI Express

The Zynq UltraScale+ MPSoC provides a controller for the integrated block for PCI™ Express v2.1 compliant, AXI-PCIe® Bridge, and DMA modules. The AXI-PCIe® Bridge provides high-performance bridging between PCIe® and AXI.

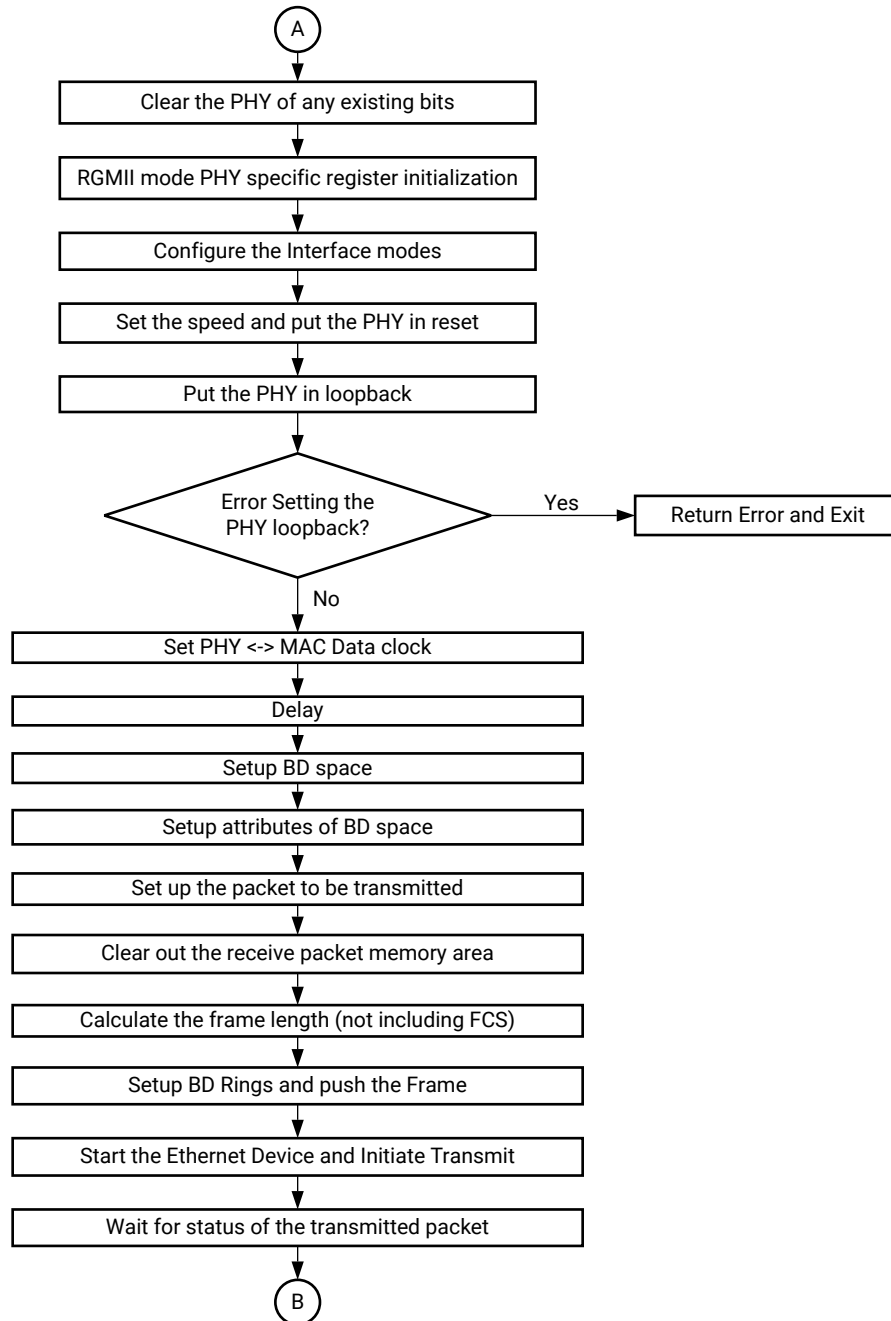
The following flow diagrams illustrate an example for configuring PCIe root complex for a data transfer.

Figure 75: Example PCIe Flow: Enable the Legacy Interrupts and Create PCIe Root Bus



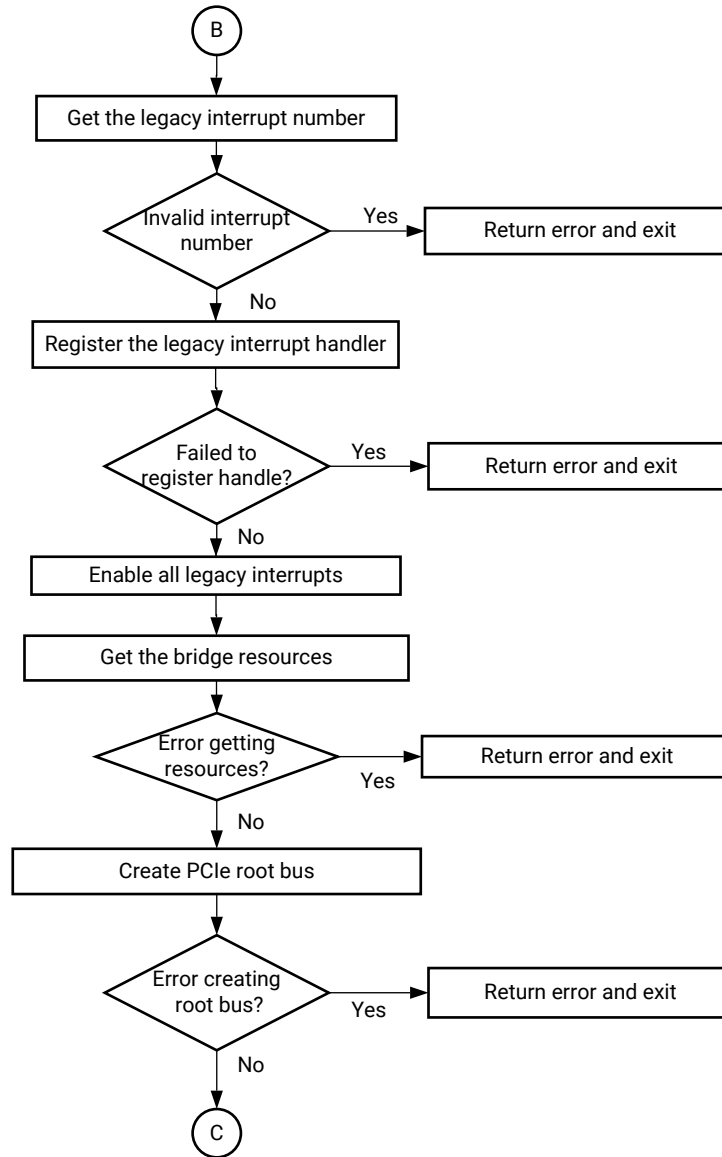
X15481-071217

Figure 76: Example PCIe Flow: Configure the PCIe Parameters and Initialize the Transmit



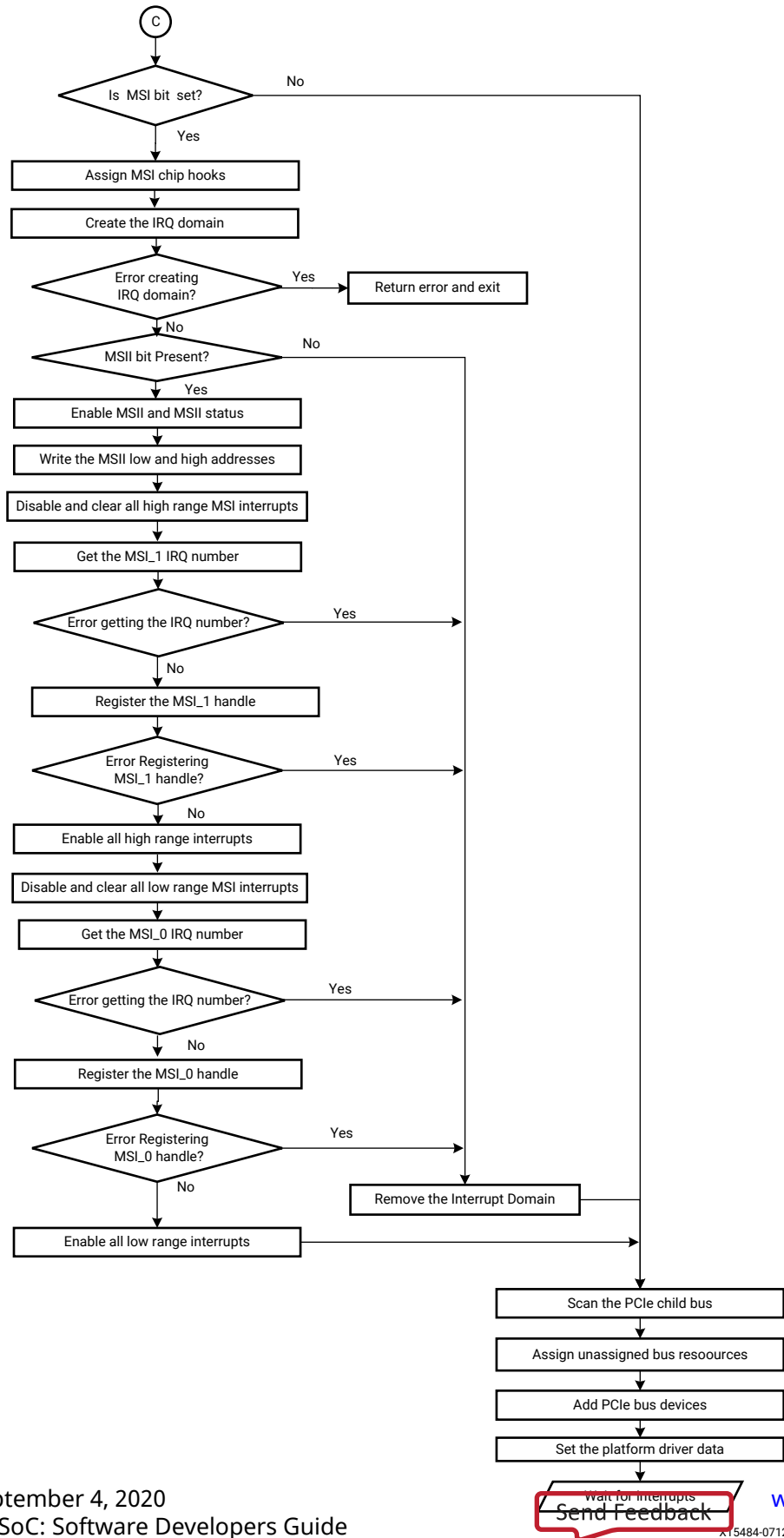
X15479-071017

Figure 77: Example PCIe Flow: Enable the Legacy Interrupts and Create PCIe Root Bus



X15483-071217

Figure 78: Example PCIe Flow: Enable MSI Interrupts and Wait for Interrupts



Note: For endpoint operation, refer to this [link](#) to “Controller for PCI Express” in the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*.

After the memory space for PCIe bridge and ECAM is mapped, ECAM is enabled for ECAM translations. You then acquire the bus range to set up the bus numbers, and write the primary, secondary, and subordinate bus numbers. The interrupt system must be set up by enabling all the miscellaneous and legacy interrupts. You can parse the ranges property of a PCI host bridge device node, and setup the resource mapping based on its content.

To create a root bus, allocate the PCIe root bus and add initial resources to the bus. If the MSI bit is set, you must enable the message signaling interrupt (MSI). After configuring the MSI interrupts, scan the PCIe slot and enumerate the entire PCIe bus and allocate bus resources to scanned buses. Now, you can add PCIe devices to the system.

For more information on PCI Express, see this [link](#) to the “DMA Controller” section and this link to “Controller for PCI Express” in the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*.

Clock and Frequency Management

The Zynq[®] UltraScale+[™] MPSoC architecture includes a programmable clock generator that takes a clock of a definite input frequency and generates multiple-derived clocks using the phase-locked loop (PLL) blocks in the PS. The output clock from each of the PLLs is used as a reference clock for the different PS peripherals.

Unlike the USB and Ethernet peripherals, some peripherals like the UART and SD allow you to dynamically change the device frequency setting.

This chapter provides information about changing the operating frequency of these peripherals dynamically. See [Chapter 11: Power Management Framework](#) for more information on reducing or adjusting the clock frequencies.

Changing the Peripheral Frequency

You can change the peripheral operation frequency by directly setting the frequency in the corresponding peripheral clock configuration register. The Zynq UltraScale+ MPSoC BSP provides APIs that aid in changing the peripheral clock frequency dynamically according to your requirements.

The following table shows the standalone APIs that can be used to change the frequency of the peripherals

Table 62: Standalone APIs

APIs	Description
XSDPS_change_clkfreq	Change the clock frequency of SD.
XSPIPS_setclkprescaler XSPIPS_getclkprescaler	Pre-scale the SPI frequency.
XRtcPSu_calculatecalibration	Change the oscillator frequency.
XQSPIPSU_setclkprescaler	Change the clock frequency of QSPI.

In case of a Linux application, the frequency of all the peripherals is set in the device tree file. The following code snippet shows the setting of peripheral clock.

```
ps7_qspi_0: ps7-qspi:dir0xFF0F0000 {
#address-cells = <0x1>;
#size-cells = <0x0>;
#bus-cells = <0x1>;
clock-names = "ref_clk", "pclk";
compatible = "xlnx,usmp-gqspi", "cdns,spi-r1p6"; stream-connected-dma =
<0x26>;
clocks = <0x1e 0x1e>; dma = <0xb>; interrupts = <0xf>;
num-chip-select = <0x2>;
reg = <0x0 0xff0f0000 0x1000 0x0 0xc0000000 0x8000000>;
speed-hz = <0xbebc200>; xlnx,fb-clk = <0x1>;
xlnx,qspi-clk-freq-hz = <0xbebc200>; xlnx,qspi-mode = <0x2>;
```

To avoid any error condition, the peripheral needs to be stopped before changing the corresponding clock frequency.

The steps to follow before changing the clock frequency for any peripheral are as follows:

1. Stop the transition pertaining to the peripheral (IP) and make it idle.
2. Stop the IP by appropriately configuring the registers.
3. Change the clock frequency of the peripheral.
4. Issue soft reset to the IP.
5. Restart the IP.

For more information on Zynq UltraScale+ MPSoC clock generator, see this [link](#) in the “Clocking” chapter in the in the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*.

Target Development Platforms

This chapter describes various development platforms available for the Zynq® UltraScale+™ MPSoC, such as Quick Emulators (QEMU) and the Zynq UltraScale+ MPSoC boards and kits.

QEMU

QEMU is a system emulation model that functions on an Intel-compatible Linux host system. Xilinx® QEMU implements a framework for generating custom machine models based on a device tree passed to it using the command line. See the *Xilinx Quick Emulator User Guide: QEMU (UG1169)* for more information about QEMU.

Boards and Kits

Xilinx provides the Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit for developers. To understand more about the ZCU102 evaluation kit, see the Preliminary ZCU102 Getting Started Guide Answer Record: [66249](#).

See the [Zynq UltraScale+ MPSoC Products Page](#) to know the different Zynq UltraScale+ MPSoCs.

Boot Image Creation

Zynq® UltraScale+™ MPSoC supports both secure and non-secure booting. While deploying the devices in field, it is important to prevent unauthorized or modified code from being run on these devices. Zynq UltraScale+ MPSoC provides the required confidentiality, integrity, and authentication to host applications securely. For more information on security features, see *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#)).

Zynq UltraScale+ MPSoCs typically have many hardware and software binaries that are used to boot them to function as designed and expected. These binaries includes FPGA bitstreams, Firmware, boot loaders, operating system, and applications that you select. For example: FPGA bitstream files, first stage boot loader (FSBL), PMU firmware, ATF, U-Boot, Linux kernel, Rootfs, device tree, standalone or RTOS applications and so on). Xilinx provides a standalone tool, Bootgen, to stitch all these binary images together and generate a device bootable image in a specific format that Xilinx loader programs can interpret while loading.

Bootgen has multiple attributes and commands that define its behavior while generating boot images. They are secure boot image generation, non-secure boot image generation, Secure key generation, HMI Mode and so on. For complete details of how to get the Bootgen tool, the installation procedure, and details of Zynq Ultrascale+ Boot Image format, Bootgen commands, attributes, and boot image generation procedure with examples, see *Bootgen User Guide* ([UG1283](#)).

Standalone Library v7.2

Xilinx Hardware Abstraction Layer API

This section describes the Xilinx Hardware Abstraction Layer API, These APIs are applicable for all processors supported by Xilinx.

Assert APIs and Macros

The `xil_assert.h` file contains assert related functions and macros. Assert APIs/Macros specifies that a application program satisfies certain conditions at particular points in its execution. These function can be used by application programs to ensure that, application code is satisfying certain conditions.

Table 63: Quick Function Reference

Type	Name	Arguments
void	Xil_Assert	file line
void	XNullHandler	void * NullParameter
void	Xil_AssertSetCallback	routine

Functions

Xil_Assert

Implement assert.

Currently, it calls a user-defined callback function if one has been set. Then, it potentially enters an infinite loop depending on the value of the `Xil_AssertWait` variable.

Note: None.

Prototype

```
void Xil_Assert(const char8 *File, s32 Line);
```

Parameters

The following table lists the `Xil_Assert` function arguments.

Table 64: Xil_Assert Arguments

Name	Description
file	filename of the source
line	linenumber within File

Returns

None.

XNullHandler

Null handler function.

This follows the `XInterruptHandler` signature for interrupt handlers. It can be used to assign a null handler (a stub) to an interrupt controller vector table.

Note: None.

Prototype

```
void XNullHandler(void *NullParameter);
```

Parameters

The following table lists the `XNullHandler` function arguments.

Table 65: XNullHandler Arguments

Name	Description
NullParameter	arbitrary void pointer and not used.

Returns

None.

Xil_AssertSetCallback

Set up a callback function to be invoked when an assert occurs.

If a callback is already installed, then it will be replaced.

Note: This function has no effect if NDEBUB is set

Prototype

```
void Xil_AssertSetCallback(Xil_AssertCallback Routine);
```

Parameters

The following table lists the `Xil_AssertSetCallback` function arguments.

Table 66: Xil_AssertSetCallback Arguments

Name	Description
routine	callback to be invoked when an assert is taken

Returns

None.

Register IO interfacing APIs

The `xil_io.h` file contains the interface for the general I/O component, which encapsulates the Input/Output functions for the processors that do not require any special I/O handling.

Table 67: Quick Function Reference

Type	Name	Arguments
u16	Xil_EndianSwap16	u16 Data
u32	Xil_EndianSwap32	u32 Data
INLINE u8	Xil_In8	UINTPTR Addr
INLINE u16	Xil_In16	UINTPTR Addr
INLINE u32	Xil_In32	UINTPTR Addr
INLINE u64	Xil_In64	UINTPTR Addr
INLINE void	Xil_Out8	UINTPTR Addr u8 Value

Table 67: Quick Function Reference (cont'd)

Type	Name	Arguments
INLINE void	Xil_Out16	UINTPTR Addr u16 Value
INLINE void	Xil_Out32	UINTPTR Addr u32 Value
INLINE void	Xil_Out64	UINTPTR Addr u64 Value
INLINE u32	Xil_SecureOut32	UINTPTR Addr u32 Value
INLINE u16	Xil_In16BE	void
INLINE u32	Xil_In32BE	void
INLINE void	Xil_Out16BE	void
INLINE void	Xil_Out32BE	void

Functions

Xil_EndianSwap16

Perform a 16-bit endian conversion.

Prototype

```
u16 Xil_EndianSwap16(u16 Data);
```

Parameters

The following table lists the `Xil_EndianSwap16` function arguments.

Table 68: Xil_EndianSwap16 Arguments

Name	Description
Data	16 bit value to be converted

Returns

16 bit Data with converted endianness

Xil_EndianSwap32

Perform a 32-bit endian conversion.

Prototype

```
u32 Xil_EndianSwap32(u32 Data);
```

Parameters

The following table lists the `Xil_EndianSwap32` function arguments.

Table 69: Xil_EndianSwap32 Arguments

Name	Description
Data	32 bit value to be converted

Returns

32 bit data with converted endianness

Xil_In8

Performs an input operation for a memory location by reading from the specified address and returning the 8 bit Value read from that address.

Prototype

```
INLINE u8 Xil_In8(UINTPTR Addr);
```

Parameters

The following table lists the `Xil_In8` function arguments.

Table 70: Xil_In8 Arguments

Name	Description
Addr	contains the address to perform the input operation

Returns

The 8 bit Value read from the specified input address.

Xil_In16

Performs an input operation for a memory location by reading from the specified address and returning the 16 bit Value read from that address.

Prototype

```
INLINE u16 Xil_In16(UINTPTR Addr);
```

Parameters

The following table lists the `Xil_In16` function arguments.

Table 71: Xil_In16 Arguments

Name	Description
Addr	contains the address to perform the input operation

Returns

The 16 bit Value read from the specified input address.

Xil_In32

Performs an input operation for a memory location by reading from the specified address and returning the 32 bit Value read from that address.

Prototype

```
INLINE u32 Xil_In32(UINTPTR Addr);
```

Parameters

The following table lists the `Xil_In32` function arguments.

Table 72: Xil_In32 Arguments

Name	Description
Addr	contains the address to perform the input operation

Returns

The 32 bit Value read from the specified input address.

Xil_In64

Performs an input operation for a memory location by reading the 64 bit Value read from that address.

Prototype

```
INLINE u64 Xil_In64(UINTPTR Addr);
```

Parameters

The following table lists the `Xil_In64` function arguments.

Table 73: Xil_In64 Arguments

Name	Description
Addr	contains the address to perform the input operation

Returns

The 64 bit Value read from the specified input address.

Xil_Out8

Performs an output operation for an memory location by writing the 8 bit Value to the the specified address.

Prototype

```
INLINE void Xil_Out8(UINTPTR Addr, u8 Value);
```

Parameters

The following table lists the `Xil_Out8` function arguments.

Table 74: Xil_Out8 Arguments

Name	Description
Addr	contains the address to perform the output operation
Value	contains the 8 bit Value to be written at the specified address.

Returns

None.

Xil_Out16

Performs an output operation for a memory location by writing the 16 bit Value to the the specified address.

Prototype

```
INLINE void Xil_Out16(UINTPTR Addr, u16 Value);
```

Parameters

The following table lists the `Xil_Out16` function arguments.

Table 75: Xil_Out16 Arguments

Name	Description
Addr	contains the address to perform the output operation
Value	contains the Value to be written at the specified address.

Returns

None.

Xil_Out32

Performs an output operation for a memory location by writing the 32 bit Value to the the specified address.

Prototype

```
INLINE void Xil_Out32(UINTPTR Addr, u32 Value);
```

Parameters

The following table lists the `Xil_Out32` function arguments.

Table 76: Xil_Out32 Arguments

Name	Description
Addr	contains the address to perform the output operation
Value	contains the 32 bit Value to be written at the specified address.

Returns

None.

Xil_Out64

Performs an output operation for a memory location by writing the 64 bit Value to the the specified address.

Prototype

```
INLINE void Xil_Out64(UINTPTR Addr, u64 Value);
```

Parameters

The following table lists the `Xil_Out64` function arguments.

Table 77: Xil_Out64 Arguments

Name	Description
Addr	contains the address to perform the output operation
Value	contains 64 bit Value to be written at the specified address.

Returns

None.

Xil_SecureOut32

Performs an output operation for a memory location by writing the 32 bit Value to the the specified address and then reading it back to verify the value written in the register.

Prototype

```
INLINE u32 Xil_SecureOut32(UINTPTR Addr, u32 Value);
```

Parameters

The following table lists the `Xil_SecureOut32` function arguments.

Table 78: Xil_SecureOut32 Arguments

Name	Description
Addr	contains the address to perform the output operation
Value	contains 32 bit Value to be written at the specified address

Returns

Returns Status

- XST_SUCCESS on success

- XST_FAILURE on failure

Definitions for available xilinx platforms

The `xplatform_info.h` file contains definitions for various available Xilinx platforms. Also, it contains prototype of APIs, which can be used to get the platform information.

Table 79: Quick Function Reference

Type	Name	Arguments
u32	XGetPlatform_Info	None.

Functions

XGetPlatform_Info

This API is used to provide information about platform.

Prototype

```
u32 XGetPlatform_Info(void);
```

Parameters

The following table lists the `XGetPlatform_Info` function arguments.

Table 80: XGetPlatform_Info Arguments

Name	Description
None.	

Returns

The information about platform defined in `xplatform_info.h`

Data types for Xilinx Software IP Cores

The `xil_types.h` file contains basic types for Xilinx software IP. These data types are applicable for all processors supported by Xilinx.

Customized APIs for Memory Operations

The `xil_mem.h` file contains prototype for functions related to memory operations. These APIs are applicable for all processors supported by Xilinx.

Table 81: Quick Function Reference

Type	Name	Arguments
void	Xil_MemCpy	void * dst const void * src u32 cnt

Functions

Xil_MemCpy

This function copies memory from once location to other.

Prototype

```
void Xil_MemCpy(void *dst, const void *src, u32 cnt);
```

Parameters

The following table lists the `Xil_MemCpy` function arguments.

Table 82: Xil_MemCpy Arguments

Name	Description
dst	pointer pointing to destination memory
src	pointer pointing to source memory
cnt	32 bit length of bytes to be copied

Xilinx software status codes

The `xstatus.h` file contains the Xilinx software status codes. These codes are used throughout the Xilinx device drivers.

Test Utilities for Memory and Caches

The `xil_testcache.h`, `xil_testio.h` and the `xil_testmem.h` files contain utility functions to test cache and memory. Details of supported tests and subtests are listed below.

The `xil_testcache.h` file contains utility functions to test cache.

The `xil_testio.h` file contains utility functions to test endian related memory IO functions.

A subset of the memory tests can be selected or all of the tests can be run in order. If there is an error detected by a subtest, the test stops and the failure code is returned. Further tests are not run even if all of the tests are selected.

The `xil_testmem.h` file contains utility functions to test memory. A subset of the memory tests can be selected or all of the tests can be run in order. If there is an error detected by a subtest, the test stops and the failure code is returned. Further tests are not run even if all of the tests are selected. Following list describes the supported memory tests:

- `XIL_TESTMEM_ALLMEMTESTS`: This test runs all of the subtests.
- `XIL_TESTMEM_INCREMENT`: This test starts at '`XIL_TESTMEM_INIT_VALUE`' and uses the incrementing value as the test value for memory.
- `XIL_TESTMEM_WALKONES`: Also known as the Walking ones test. This test uses a walking '1' as the test value for memory.

```
location 1 = 0x00000001
location 2 = 0x00000002
...
```

- `XIL_TESTMEM_WALKZEROS`: Also known as the Walking zero's test. This test uses the inverse value of the walking ones test as the test value for memory.

```
location 1 = 0xFFFFFFFF
location 2 = 0xFFFFFFF0
...
```

- `XIL_TESTMEM_INVERSEADDR`: Also known as the inverse address test. This test uses the inverse of the address of the location under test as the test value for memory.
- `XIL_TESTMEM_FIXEDPATTERN`: Also known as the fixed pattern test. This test uses the provided patters as the test value for memory. If zero is provided as the pattern the test uses '0xDEADBEEF'.



CAUTION! The tests are **DESTRUCTIVE**. Run before any initialized memory spaces have been set up. The address provided to the memory tests is not checked for validity except for the NULL case. It is possible to provide a code-space pointer for this test to start with and ultimately destroy executable code causing random failures.

Note: Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2^{**} width, the patterns used in `XIL_TESTMEM_WALKONES` and `XIL_TESTMEM_WALKZEROS` will repeat on a boundary of a power of two making it more difficult to detect addressing errors. The `XIL_TESTMEM_INCREMENT` and `XIL_TESTMEM_INVERSEADDR` tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

Table 83: Quick Function Reference

Type	Name	Arguments
s32	Xil_TestDCacheRange	void

Table 83: Quick Function Reference (cont'd)

Type	Name	Arguments
s32	Xil_TestDCacheAll	void
s32	Xil_TestICacheRange	void
s32	Xil_TestICacheAll	void
s32	Xil_TestIO8	u8 * Addr s32 Length u8 Value
s32	Xil_TestIO16	u16 * Addr s32 Length u16 Value s32 Kind s32 Swap
s32	Xil_TestIO32	u32 * Addr s32 Length u32 Value s32 Kind s32 Swap

Functions

Xil_TestIO8

Perform a destructive 8-bit wide register IO test where the register is accessed using Xil_Out8 and Xil_In8, and comparing the written values by reading them back.

Prototype

```
s32 Xil_TestIO8(u8 *Addr, s32 Length, u8 Value);
```

Parameters

The following table lists the Xil_TestIO8 function arguments.

Table 84: Xil_TestIO8 Arguments

Name	Description
Addr	a pointer to the region of memory to be tested.
Length	Length of the block.

Table 84: Xil_TestIO8 Arguments (cont'd)

Name	Description
Value	constant used for writing the memory.

Returns

- -1 is returned for a failure
- 0 is returned for a pass

Xil_TestIO16

Perform a destructive 16-bit wide register IO test.

Each location is tested by sequentially writing a 16-bit wide register, reading the register, and comparing value. This function tests three kinds of register IO functions, normal register IO, little-endian register IO, and big-endian register IO. When testing little/big-endian IO, the function performs the following sequence, Xil_Out16LE/Xil_Out16BE, Xil_In16, Compare In-Out values, Xil_Out16, Xil_In16LE/Xil_In16BE, Compare In-Out values. Whether to swap the read-in value before comparing is controlled by the 5th argument.

Prototype

```
s32 Xil_TestIO16(u16 *Addr, s32 Length, u16 Value, s32 Kind, s32 Swap);
```

Parameters

The following table lists the Xil_TestIO16 function arguments.

Table 85: Xil_TestIO16 Arguments

Name	Description
Addr	a pointer to the region of memory to be tested.
Length	Length of the block.
Value	constant used for writing the memory.
Kind	Type of test. Acceptable values are: XIL_TESTIO_DEFAULT, XIL_TESTIO_LE, XIL_TESTIO_BE.
Swap	indicates whether to byte swap the read-in value.

Returns

- -1 is returned for a failure
- 0 is returned for a pass

Xil_TestIO32

Perform a destructive 32-bit wide register IO test.

Each location is tested by sequentially writing a 32-bit wide register, reading the register, and comparing value. This function tests three kinds of register IO functions, normal register IO, little-endian register IO, and big-endian register IO. When testing little/big-endian IO, the function perform the following sequence, Xil_Out32LE/ Xil_Out32BE, Xil_In32, Compare, Xil_Out32, Xil_In32LE/Xil_In32BE, Compare. Whether to swap the read-in value *before comparing is controlled by the 5th argument.

Prototype

```
s32 Xil_TestIO32(u32 *Addr, s32 Length, u32 Value, s32 Kind, s32 Swap);
```

Parameters

The following table lists the `Xil_TestIO32` function arguments.

Table 86: Xil_TestIO32 Arguments

Name	Description
Addr	a pointer to the region of memory to be tested.
Length	Length of the block.
Value	constant used for writing the memory.
Kind	type of test. Acceptable values are: XIL_TESTIO_DEFAULT, XIL_TESTIO_LE, XIL_TESTIO_BE.
Swap	indicates whether to byte swap the read-in value.

Returns

- -1 is returned for a failure
- 0 is returned for a pass

MicroBlaze Processor API

This section provides a linked summary and detailed descriptions of the MicroBlaze Processor APIs.

MicroBlaze Pseudo-asm Macros and Interrupt Handling APIs

MicroBlaze BSP includes macros to provide convenient access to various registers in the MicroBlaze processor. Some of these macros are very useful within exception handlers for retrieving information about the exception. Also, the interrupt handling functions help manage interrupt handling on MicroBlaze processor devices. To use these functions, include the header file `mb_interface.h` in your source code

Table 87: Quick Function Reference

Type	Name	Arguments
void	microblaze_enable_interrupts	void
void	microblaze_disable_interrupts	void
void	microblaze_enable_icache	void
void	microblaze_disable_icache	void
void	microblaze_enable_dcache	void
void	microblaze_disable_dcache	void
void	microblaze_enable_exceptions	void
void	microblaze_disable_exceptions	void
void	microblaze_register_handler	XInterruptHandler Handler void * DataPtr
void	microblaze_register_exception_handler	u32 ExceptionId Top void * DataPtr
void	microblaze_invalidate_icache	void
void	microblaze_invalidate_dcache	void
void	microblaze_flush_dcache	void
void	microblaze_invalidate_icache_range	void

Table 87: Quick Function Reference (cont'd)

Type	Name	Arguments
void	microblaze_invalidate_dcache_range	void
void	microblaze_flush_dcache_range	void
void	microblaze_scrub	void
void	microblaze_invalidate_cache_ext	void
void	microblaze_flush_cache_ext	void
void	microblaze_flush_cache_ext_range	void
void	microblaze_invalidate_cache_ext_range	void
void	microblaze_update_icache	void
void	microblaze_init_icache_range	void
void	microblaze_update_dcache	void
void	microblaze_init_dcache_range	void

Functions

microblaze_register_handler

Registers a top-level interrupt handler for the MicroBlaze.

The argument provided in this call as the DataPtr is used as the argument for the handler when it is called.

Prototype

```
void microblaze_register_handler(XInterruptHandler Handler, void *DataPtr);
```

Parameters

The following table lists the `microblaze_register_handler` function arguments.

Table 88: microblaze_register_handler Arguments

Name	Description
Handler	Top level handler.
DataPtr	a reference to data that will be passed to the handler when it gets called.

Returns

None.

microblaze_register_exception_handler

Registers an exception handler for the MicroBlaze.

The argument provided in this call as the DataPtr is used as the argument for the handler when it is called.

None.

Note:

Prototype

```
void microblaze_register_exception_handler(u32 ExceptionId,
Xil_ExceptionHandler Handler, void *DataPtr);
```

Parameters

The following table lists the `microblaze_register_exception_handler` function arguments.

Table 89: microblaze_register_exception_handler Arguments

Name	Description
ExceptionId	is the id of the exception to register this handler for.
Top	level handler.
DataPtr	is a reference to data that will be passed to the handler when it gets called.

Returns

None.

MicroBlaze exception APIs

The `xil_exception.h` file contains MicroBlaze specific exception related APIs and macros. Application programs can use these APIs/Macros for various exception related operations (i.e. enable exception, disable exception, register exception handler etc.)

Note: To use exception related functions, the `xil_exception.h` file must be added in source code

Table 90: Quick Function Reference

Type	Name	Arguments
void	microblaze_enable_exceptions	void
void	microblaze_disable_exceptions	void
void	microblaze_enable_interrupts	void
void	microblaze_disable_interrupts	void
void	Xil_ExceptionNullHandler	void * Data
void	Xil_ExceptionInit	None.
void	Xil_ExceptionEnable	void
void	Xil_ExceptionDisable	None.
void	Xil_ExceptionRegisterHandler	u32 Id Xil_ExceptionHandler Handler void * Data
void	Xil_ExceptionRemoveHandler	u32 Id

Functions

Xil_ExceptionNullHandler

This function is a stub handler that is the default handler that gets called if the application has not setup a handler for a specific exception.

The function interface has to match the interface specified for a handler even though none of the arguments are used.

Prototype

```
void Xil_ExceptionNullHandler(void *Data);
```

Parameters

The following table lists the `Xil_ExceptionNullHandler` function arguments.

Table 91: Xil_ExceptionNullHandler Arguments

Name	Description
Data	unused by this function.

Xil_ExceptionInit

Initialize exception handling for the processor.

The exception vector table is setup with the stub handler for all exceptions.

Prototype

```
void Xil_ExceptionInit(void);
```

Parameters

The following table lists the `Xil_ExceptionInit` function arguments.

Table 92: Xil_ExceptionInit Arguments

Name	Description
None.	

Returns

None.

Xil_ExceptionEnable

Enable Exceptions.

Prototype

```
void Xil_ExceptionEnable(void);
```

Returns

None.

Xil_ExceptionDisable

Disable Exceptions.

Prototype

```
void Xil_ExceptionDisable(void);
```

Parameters

The following table lists the `Xil_ExceptionDisable` function arguments.

Table 93: Xil_ExceptionDisable Arguments

Name	Description
None.	

Returns

None.

Xil_ExceptionRegisterHandler

Makes the connection between the Id of the exception source and the associated handler that is to run when the exception is recognized.

The argument provided in this call as the `DataPtr` is used as the argument for the handler when it is called.

Prototype

```
void Xil_ExceptionRegisterHandler(u32 Id, Xil_ExceptionHandler Handler,
void *Data);
```

Parameters

The following table lists the `Xil_ExceptionRegisterHandler` function arguments.

Table 94: Xil_ExceptionRegisterHandler Arguments

Name	Description
Id	contains the 32 bit ID of the exception source and should be <code>XIL_EXCEPTION_INT</code> or be in the range of 0 to <code>XIL_EXCEPTION_LAST</code> . See <code>xil_mach_exception.h</code> for further information.
Handler	handler function to be registered for exception
Data	a reference to data that will be passed to the handler when it gets called.

Xil_ExceptionRemoveHandler

Removes the handler for a specific exception Id.

The stub handler is then registered for this exception Id.

Prototype

```
void Xil_ExceptionRemoveHandler(u32 Id);
```

Parameters

The following table lists the `Xil_ExceptionRemoveHandler` function arguments.

Table 95: Xil_ExceptionRemoveHandler Arguments

Name	Description
Id	contains the 32 bit ID of the exception source and should be XIL_EXCEPTION_INT or in the range of 0 to XIL_EXCEPTION_LAST. See <code>xexception_l.h</code> for further information.

MicroBlaze Processor FSL Macros

MicroBlaze BSP includes macros to provide convenient access to accelerators connected to the MicroBlaze Fast Simplex Link (FSL) Interfaces. To use these functions, include the header file `fsl.h` in your source code

MicroBlaze PVR access routines and macros

MicroBlaze processor v5.00.a and later versions have configurable Processor Version Registers (PVRs). The contents of the PVR are captured using the `pvr_t` data structure, which is defined as an array of 32-bit words, with each word corresponding to a PVR register on hardware. The number of PVR words is determined by the number of PVRs configured in the hardware. You should not attempt to access PVR registers that are not present in hardware, as the `pvr_t` data structure is resized to hold only as many PVRs as are present in hardware. To access information in the PVR:

1. Use the `microblaze_get_pvr()` function to populate the PVR data into a `pvr_t` data structure.
2. In subsequent steps, you can use any one of the PVR access macros list to get individual data stored in the PVR.
3. `pvr.h` header file must be included to source to use PVR macros.

Table 96: Quick Function Reference

Type	Name	Arguments
int	<code>microblaze_get_pvr</code>	<code>pvr-</code>

Functions

`microblaze_get_pvr`

Populate the PVR data structure to which `pvr` points, with the values of the hardware PVR registers.

Prototype

```
int microblaze_get_pvr(pvr_t *pvr);
```

Parameters

The following table lists the `microblaze_get_pvr` function arguments.

Table 97: microblaze_get_pvr Arguments

Name	Description
pvr-	address of PVR data structure to be populated

Returns

0 - SUCCESS -1 - FAILURE

Sleep Routines for MicroBlaze

The `microblaze_sleep.h` file contains microblaze sleep APIs. These APIs provides delay for requested duration.

Note: The `microblaze_sleep.h` file may contain architecture-dependent items.

Table 98: Quick Function Reference

Type	Name	Arguments
void	MB_Sleep	MilliSeconds-

Functions

MB_Sleep

Provides delay for requested duration.

Note: Instruction cache should be enabled for this to work.

Prototype

```
void MB_Sleep(u32 MilliSeconds) __attribute__((__deprecated__));
```

Parameters

The following table lists the `MB_Sleep` function arguments.

Table 99: MB_Sleep Arguments

Name	Description
MilliSeconds-	Delay time in milliseconds.

Returns

None.

Cortex R5 Processor API

Standalone BSP contains boot code, cache, exception handling, file and memory management, configuration, time and processor-specific include functions. It supports gcc compiler. This section provides a linked summary and detailed descriptions of the Cortex R5 processor APIs.

Cortex R5 Processor Boot Code

The boot code performs minimum configuration which is required for an application to run starting from processor's reset state. Below is a sequence illustrating what all configuration is performed before control reaches to main function.

1. Program vector table base for exception handling
2. Program stack pointer for various modes (IRQ, FIQ, supervisor, undefine, abort, system)
3. Disable instruction cache, data cache and MPU
4. Invalidate instruction and data cache
5. Configure MPU with short descriptor translation table format and program base address of translation table
6. Enable data cache, instruction cache and MPU
7. Enable Floating point unit
8. Transfer control to `_start` which clears BSS sections and jumping to main application

Cortex R5 Processor MPU specific APIs

MPU functions provides access to MPU operations such as enable MPU, disable MPU and set attribute for section of memory. Boot code invokes `Init_MPU` function to configure the MPU. A total of 10 MPU regions are allocated with another 6 being free for users. Overview of the memory attributes for different MPU regions is as given below,

	Memory Range	Attributes of MPURegion
DDR	0x00000000 - 0x7FFFFFFF	Normal write-back Cacheable
PL	0x80000000 - 0xBFFFFFFF	Strongly Ordered
QSPI	0xC0000000 - 0xDFFFFFFF	Device Memory
PCIe	0xE0000000 - 0xEFFFFFFF	Device Memory
STM_CORESIGHT	0xF8000000 - 0xF8FFFFFF	Device Memory
RPU_R5_GIC	0xF9000000 - 0xF90FFFFFFF	Device memory
FPS	0xFD000000 - 0xFDFFFFFFFF	Device Memory
LPS	0xFE000000 - 0xFFFFFFFF	Device Memory
OCM	0xFFFC0000 - 0xFFFFFFFF	Normal write-back Cacheable

Note: For a system where DDR is less than 2GB, region after DDR and before PL is marked as undefined in translation table. Memory range 0xFE000000-0xFFFFFFFF is allocated for upper LPS slaves, where as memory region 0xFF000000-0xFFFFFFFF is allocated for lower LPS slaves.

Table 100: Quick Function Reference

Type	Name	Arguments
void	Xil_SetTlbAttributes	INTPTR Addr u32 attrib
void	Xil_EnableMPU	None.
void	Xil_DisableMPU	None.
u32	Xil_SetMPURegion	Addr u64 size u32 attrib
u32	Xil_UpdateMPUConfig	u32 reg_num INTPTR address u32 size u32 attrib
void	Xil_GetMPUConfig	XMpu_Config mpuconfig
u32	Xil_GetNumOfFreeRegions	none
u32	Xil_GetNextMPURegion	none
u32	Xil_DisableMPURegionByRegNum	u32 reg_num
u16	Xil_GetMPUFreeRegMask	none

Table 100: Quick Function Reference (cont'd)

Type	Name	Arguments
u32	Xil_SetMPURegionByRegNum	u32 reg_num address u64 size u32 attrib
void *	Xil_MemMap	void

Functions

Xil_SetTlbAttributes

This function sets the memory attributes for a section covering 1MB, of memory in the translation table.

Prototype

```
void Xil_SetTlbAttributes(INTPTR Addr, u32 attrib);
```

Parameters

The following table lists the `Xil_SetTlbAttributes` function arguments.

Table 101: Xil_SetTlbAttributes Arguments

Name	Description
Addr	32-bit address for which memory attributes need to be set.
attrib	Attribute for the given memory region.

Returns

None.

Xil_EnableMPU

Enable MPU for Cortex R5 processor.

This function invalidates I cache and flush the D Caches, and then enables the MPU.

Prototype

```
void Xil_EnableMPU(void);
```

Parameters

The following table lists the `Xil_EnableMPU` function arguments.

Table 102: Xil_EnableMPU Arguments

Name	Description
None.	

Returns

None.

Xil_DisableMPU

Disable MPU for Cortex R5 processors.

This function invalidates I cache and flush the D Caches, and then disables the MPU.

Prototype

```
void Xil_DisableMPU(void);
```

Parameters

The following table lists the `Xil_DisableMPU` function arguments.

Table 103: Xil_DisableMPU Arguments

Name	Description
None.	

Returns

None.

Xil_SetMPURegion

Set the memory attributes for a section of memory in the translation table.

Prototype

```
u32 Xil_SetMPURegion(INTPTR addr, u64 size, u32 attrib);
```

Parameters

The following table lists the `Xil_SetMPURegion` function arguments.

Table 104: Xil_SetMPURegion Arguments

Name	Description
Addr	32-bit address for which memory attributes need to be set..
size	size is the size of the region.
attrib	Attribute for the given memory region.

Returns

None.

Xil_UpdateMPUConfig

Update the MPU configuration for the requested region number in the global MPU configuration table.

Prototype

```
u32 Xil_UpdateMPUConfig(u32 reg_num, INTPTR address, u32 size, u32 attrib);
```

Parameters

The following table lists the `Xil_UpdateMPUConfig` function arguments.

Table 105: Xil_UpdateMPUConfig Arguments

Name	Description
reg_num	The requested region number to be updated information for.
address	32 bit address for start of the region.
size	Requested size of the region.
attrib	Attribute for the corresponding region.

Returns

XST_FAILURE: When the requested region number is 16 or more. XST_SUCCESS: When the MPU configuration table is updated.

Xil_GetMPUConfig

The MPU configuration table is passed to the caller.

Prototype

```
void Xil_GetMPUConfig(XMpu_Config mpuconfig);
```

Parameters

The following table lists the `Xil_GetMPUConfig` function arguments.

Table 106: Xil_GetMPUConfig Arguments

Name	Description
mpuconfig	This is of type <code>XMpu_Config</code> which is an array of 16 entries of type structure representing the MPU config table

Returns

none

Xil_GetNumOfFreeRegions

Returns the total number of free MPU regions available.

Prototype

```
u32 Xil_GetNumOfFreeRegions(void);
```

Parameters

The following table lists the `Xil_GetNumOfFreeRegions` function arguments.

Table 107: Xil_GetNumOfFreeRegions Arguments

Name	Description
none	

Returns

Number of free regions available to users

Xil_GetNextMPURegion

Returns the next available free MPU region.

Prototype

```
u32 Xil_GetNextMPURegion(void);
```

Parameters

The following table lists the `Xil_GetNextMPURegion` function arguments.

Table 108: Xil_GetNextMPURegion Arguments

Name	Description
none	

Returns

The free MPU region available

Xil_DisableMPURegionByRegNum

Disables the corresponding region number as passed by the user.

Prototype

```
u32 Xil_DisableMPURegionByRegNum(u32 reg_num);
```

Parameters

The following table lists the `Xil_DisableMPURegionByRegNum` function arguments.

Table 109: Xil_DisableMPURegionByRegNum Arguments

Name	Description
reg_num	The region number to be disabled

Returns

XST_SUCCESS: If the region could be disabled successfully XST_FAILURE: If the requested region number is 16 or more.

Xil_GetMPUFreeRegMask

Returns the total number of free MPU regions available in the form of a mask.

A bit of 1 in the returned 16 bit value represents the corresponding region number to be available. For example, if this function returns 0xC0000, this would mean, the regions 14 and 15 are available to users.

Prototype

```
u16 Xil_GetMPUFreeRegMask(void);
```

Parameters

The following table lists the `Xil_GetMPUFreeRegMask` function arguments.

Table 110: Xil_GetMPUFreeRegMask Arguments

Name	Description
none	

Returns

The free region mask as a 16 bit value

Xil_SetMPURegionByRegNum

Enables the corresponding region number as passed by the user.

Prototype

```
u32 Xil_SetMPURegionByRegNum(u32 reg_num, INTPTR addr, u64 size, u32 attrib);
```

Parameters

The following table lists the Xil_SetMPURegionByRegNum function arguments.

Table 111: Xil_SetMPURegionByRegNum Arguments

Name	Description
reg_num	The region number to be enabled
address	32 bit address for start of the region.
size	Requested size of the region.
attrib	Attribute for the corresponding region.

Returns

XST_SUCCESS: If the region could be created successfully XST_FAILURE: If the requested region number is 16 or more.

Cortex R5 Processor Cache Functions

Cache functions provide access to cache related operations such as flush and invalidate for instruction and data caches. It gives option to perform the cache operations on a single cacheline, a range of memory and an entire cache.

Table 112: Quick Function Reference

Type	Name	Arguments
void	Xil_DCacheEnable	None.
void	Xil_DCacheDisable	None.
void	Xil_DCacheInvalidate	None.
void	Xil_DCacheInvalidateRange	INTPTR adr u32 len
void	Xil_DCacheFlush	None.
void	Xil_DCacheFlushRange	INTPTR adr u32 len
void	Xil_DCacheInvalidateLine	INTPTR adr
void	Xil_DCacheFlushLine	INTPTR adr
void	Xil_DCacheStoreLine	INTPTR adr
void	Xil_ICacheEnable	None.
void	Xil_ICacheDisable	None.
void	Xil_ICacheInvalidate	None.
void	Xil_ICacheInvalidateRange	INTPTR adr u32 len
void	Xil_ICacheInvalidateLine	INTPTR adr

Functions

Xil_DCacheEnable

Enable the Data cache.

Note: None.

Prototype

```
void Xil_DCacheEnable(void);
```

Parameters

The following table lists the `Xil_DCacheEnable` function arguments.

Table 113: Xil_DCacheEnable Arguments

Name	Description
None.	

Returns

None.

Xil_DCacheDisable

Disable the Data cache.

Note: None.

Prototype

```
void Xil_DCacheDisable(void);
```

Parameters

The following table lists the `Xil_DCacheDisable` function arguments.

Table 114: Xil_DCacheDisable Arguments

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidate

Invalidate the entire Data cache.

Prototype

```
void Xil_DCacheInvalidate(void);
```


Parameters

The following table lists the `Xil_DCacheInvalidate` function arguments.

Table 115: Xil_DCacheInvalidate Arguments

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidateRange

Invalidate the Data cache for the given address range.

If the bytes specified by the address (`adr`) are cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are NOT written to system memory before the line is invalidated.

Prototype

```
void Xil_DCacheInvalidateRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_DCacheInvalidateRange` function arguments.

Table 116: Xil_DCacheInvalidateRange Arguments

Name	Description
<code>adr</code>	32bit start address of the range to be invalidated.
<code>len</code>	Length of range to be invalidated in bytes.

Returns

None.

Xil_DCacheFlush

Flush the entire Data cache.

Prototype

```
void Xil_DCacheFlush(void);
```

Parameters

The following table lists the `Xil_DCacheFlush` function arguments.

Table 117: Xil_DCacheFlush Arguments

Name	Description
None.	

Returns

None.

Xil_DCacheFlushRange

Flush the Data cache for the given address range.

If the bytes specified by the address (`adr`) are cached by the Data cache, the cacheline containing those bytes is invalidated. If the cacheline is modified (dirty), the written to system memory before the lines are invalidated.

Prototype

```
void Xil_DCacheFlushRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_DCacheFlushRange` function arguments.

Table 118: Xil_DCacheFlushRange Arguments

Name	Description
<code>adr</code>	32bit start address of the range to be flushed.
<code>len</code>	Length of the range to be flushed in bytes

Returns

None.

Xil_DCacheInvalidateLine

Invalidate a Data cache line.

If the byte specified by the address (`adr`) is cached by the data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are NOT written to system memory before the line is invalidated.

Note: The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_DCacheInvalidateLine(INTPTR adr);
```

Parameters

The following table lists the `Xil_DCacheInvalidateLine` function arguments.

Table 119: Xil_DCacheInvalidateLine Arguments

Name	Description
adr	32bit address of the data to be flushed.

Returns

None.

Xil_DCacheFlushLine

Flush a Data cache line.

If the byte specified by the address (`adr`) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

Note: The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_DCacheFlushLine(INTPTR adr);
```

Parameters

The following table lists the `Xil_DCacheFlushLine` function arguments.

Table 120: Xil_DCacheFlushLine Arguments

Name	Description
adr	32bit address of the data to be flushed.

Returns

None.

Xil_DCacheStoreLine

Store a Data cache line.

If the byte specified by the address (*adr*) is cached by the Data cache and the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory. After the store completes, the cacheline is marked as unmodified (not dirty).

Note: The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_DCacheStoreLine(INTPTR adr);
```

Parameters

The following table lists the `Xil_DCacheStoreLine` function arguments.

Table 121: Xil_DCacheStoreLine Arguments

Name	Description
<i>adr</i>	32bit address of the data to be stored

Returns

None.

Xil_ICacheEnable

Enable the instruction cache.

Prototype

```
void Xil_ICacheEnable(void);
```

Parameters

The following table lists the `Xil_ICacheEnable` function arguments.

Table 122: Xil_ICacheEnable Arguments

Name	Description
None.	

Returns

None.

Xil_ICacheDisable

Disable the instruction cache.

Prototype

```
void Xil_ICacheDisable(void);
```

Parameters

The following table lists the `Xil_ICacheDisable` function arguments.

Table 123: Xil_ICacheDisable Arguments

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidate

Invalidate the entire instruction cache.

Prototype

```
void Xil_ICacheInvalidate(void);
```

Parameters

The following table lists the `Xil_ICacheInvalidate` function arguments.

Table 124: Xil_ICacheInvalidate Arguments

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidateRange

Invalidate the instruction cache for the given address range.

If the bytes specified by the address (`adr`) are cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are NOT written to system memory before the line is invalidated.

Prototype

```
void Xil_ICacheInvalidateRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_ICacheInvalidateRange` function arguments.

Table 125: Xil_ICacheInvalidateRange Arguments

Name	Description
adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Xil_ICacheInvalidateLine

Invalidate an instruction cache line. If the instruction specified by the address is cached by the instruction cache, the cacheline containing that instruction is invalidated.

Note: The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_ICacheInvalidateLine(INTPTR adr);
```

Parameters

The following table lists the `Xil_ICacheInvalidateLine` function arguments.

Table 126: Xil_ICacheInvalidateLine Arguments

Name	Description
adr	32bit address of the instruction to be invalidated.

Returns

None.

Cortex R5 Time Functions

The `xtime_l.h` provides access to 32-bit TTC timer counter. These functions can be used by applications to track the time.

Table 127: Quick Function Reference

Type	Name	Arguments
void	XTime_SetTime	XTime Xtime_Global
void	XTime_GetTime	XTime * Xtime_Global

Functions

XTime_SetTime

TTC Timer runs continuously and the time can not be set as desired.

This API doesn't contain anything. It is defined to have uniformity across platforms.

Note: In multiprocessor environment reference time will reset/lost for all processors, when this function called by any one processor.

Prototype

```
void XTime_SetTime(XTime Xtime_Global);
```

Parameters

The following table lists the `XTime_SetTime` function arguments.

Table 128: XTime_SetTime Arguments

Name	Description
Xtime_Global	32 bit value to be written to the timer counter register.

Returns

None.

XTime_GetTime

Get the time from the timer counter register.

Prototype

```
void XTime_GetTime(XTime *Xtime_Global);
```

Parameters

The following table lists the `XTime_GetTime` function arguments.

Table 129: XTime_GetTime Arguments

Name	Description
Xtime_Global	Pointer to the 32 bit location to be updated with the time current value of timer counter register.

Returns

None.

Cortex R5 Event Counters Functions

Cortex R5 event counter functions can be utilized to configure and control the Cortex-R5 performance monitor events. Cortex-R5 Performance Monitor has 3 event counters which can be used to count a variety of events described in Coretx-R5 TRM. The xpm_counter.h file defines configurations XPM_CNTRCFGx which can be used to program the event counters to count a set of events.

Table 130: Quick Function Reference

Type	Name	Arguments
void	Xpm_SetEvents	s32 PmcrCfg
void	Xpm_GetEventCounters	u32 * PmCtrValue
u32	Xpm_DisableEvent	Event
u32	Xpm_SetUpAnEvent	Event
u32	Xpm_GetEventCounter	Event Pointer
void	Xpm_DisableEventCounters	None.
void	Xpm_EnableEventCounters	None.
void	Xpm_ResetEventCounters	None.
void	Xpm_SleepPerfCounter	u32 delay u64 frequency

Functions

Xpm_SetEvents

This function configures the Cortex R5 event counters controller, with the event codes, in a configuration selected by the user and enables the counters.

Prototype

```
void Xpm_SetEvents(s32 PmcrCfg);
```

Parameters

The following table lists the `Xpm_SetEvents` function arguments.

Table 131: Xpm_SetEvents Arguments

Name	Description
PmcrCfg	Configuration value based on which the event counters are configured. XPM_CNTRCFG* values defined in xpm_counter.h can be utilized for setting configuration

Returns

None.

Xpm_GetEventCounters

This function disables the event counters and returns the counter values.

Prototype

```
void Xpm_GetEventCounters(u32 *PmCtrValue);
```

Parameters

The following table lists the `Xpm_GetEventCounters` function arguments.

Table 132: Xpm_GetEventCounters Arguments

Name	Description
PmCtrValue	Pointer to an array of type u32 PmCtrValue[6]. It is an output parameter which is used to return the PM counter values.

Returns

None.

Xpm_DisableEvent

Disables the requested event counter.

Note: None.

Prototype

```
u32 Xpm_DisableEvent(u32 EventHandlerId);
```

Parameters

The following table lists the `Xpm_DisableEvent` function arguments.

Table 133: Xpm_DisableEvent Arguments

Name	Description
Event	Counter ID. The counter ID is the same that was earlier returned through a call to <code>Xpm_SetUpAnEvent</code> . Cortex-R5 supports only 3 counters. The valid values are 0, 1, or 2.

Returns

- `XST_SUCCESS` if successful.
- `XST_FAILURE` if the passed Counter ID is invalid (i.e. greater than 2).

Xpm_SetUpAnEvent

Sets up one of the event counters to count events based on the Event ID passed.

For supported Event IDs please refer `xpm_counter.h`. Upon invoked, the API searches for an available counter. After finding one, it sets up the counter to count events for the requested event.

Note: None.

Prototype

```
u32 Xpm_SetUpAnEvent(u32 EventID);
```

Parameters

The following table lists the `Xpm_SetUpAnEvent` function arguments.

Table 134: Xpm_SetUpAnEvent Arguments

Name	Description
Event	ID. For valid values, please refer <code>xpm_counter.h</code> .

Returns

- Counter Number if successful. For Cortex-R5, valid return values are 0, 1, or 2.
- XPM_NO_COUNTERS_AVAILABLE (0xFF) if all counters are being used

Xpm_GetEventCounter

Reads the counter value for the requested counter ID.

This is used to read the number of events that has been counted for the requested event ID. This can only be called after a call to Xpm_SetUpAnEvent.

Note: None.

Prototype

```
u32 Xpm_GetEventCounter(u32 EventHandlerId, u32 *CntVal);
```

Parameters

The following table lists the Xpm_GetEventCounter function arguments.

Table 135: Xpm_GetEventCounter Arguments

Name	Description
Event	Counter ID. The counter ID is the same that was earlier returned through a call to Xpm_SetUpAnEvent. Cortex-R5 supports only 3 counters. The valid values are 0, 1, or 2.
Pointer	to a 32 bit unsigned int type. This is used to return the event counter value.

Returns

- XST_SUCCESS if successful.
- XST_FAILURE if the passed Counter ID is invalid (i.e. greater than 2).

Xpm_DisableEventCounters

This function disables the Cortex R5 event counters.

Prototype

```
void Xpm_DisableEventCounters(void);
```

Parameters

The following table lists the Xpm_DisableEventCounters function arguments.

Table 136: Xpm_DisableEventCounters Arguments

Name	Description
None.	

Returns

None.

Xpm_EnableEventCounters

This function enables the Cortex R5 event counters.

Prototype

```
void Xpm_EnableEventCounters(void);
```

Parameters

The following table lists the `Xpm_EnableEventCounters` function arguments.

Table 137: Xpm_EnableEventCounters Arguments

Name	Description
None.	

Returns

None.

Xpm_ResetEventCounters

This function resets the Cortex R5 event counters.

Prototype

```
void Xpm_ResetEventCounters(void);
```

Parameters

The following table lists the `Xpm_ResetEventCounters` function arguments.

Table 138: Xpm_ResetEventCounters Arguments

Name	Description
None.	

Returns

None.

Xpm_SleepPerfCounter

This is helper function used by sleep/usleep APIs to generate delay in sec/usec.

Prototype

```
void Xpm_SleepPerfCounter(u32 delay, u64 frequency);
```

Parameters

The following table lists the `Xpm_SleepPerfCounter` function arguments.

Table 139: Xpm_SleepPerfCounter Arguments

Name	Description
delay	- delay time in sec/usec
frequency	- Number of counts in second/micro second

Returns

None.

Cortex R5 Processor Specific Include Files

The `xpseudo_asm.h` includes `xreg_cortexr5.h` and `xpseudo_asm_gcc.h`.

The `xreg_cortexr5.h` file contains definitions for inline assembler code. It provides inline definitions for Cortex R5 GPRs, SPRs, co-processor registers and Debug register

The `xpseudo_asm_gcc.h` contains the definitions for the most often used inline assembler instructions, available as macros. These can be very useful for tasks such as setting or getting special purpose registers, synchronization, or cache manipulation. These inline assembler instructions can be used from drivers and user applications written in C.

Cortex R5 peripheral definitions

The `xparameters_ps.h` file contains the canonical definitions and constant declarations for peripherals within hardblock, attached to the ARM Cortex R5 core. These definitions can be used by drivers or applications to access the peripherals.

ARM Processor Common API

This section provides a linked summary and detailed descriptions of the ARM Processor Common APIs.

ARM Processor Exception Handling

ARM processors specific exception related APIs for cortex A53,A9 and R5 can utilized for enabling/disabling IRQ, registering/removing handler for exceptions or initializing exception vector table with null handler.

Table 140: Quick Function Reference

Type	Name	Arguments
void	Xil_ExceptionRegisterHandler	exception_id Xil_ExceptionHandler Handler void * Data
void	Xil_ExceptionRemoveHandler	exception_id
void	Xil_GetExceptionRegisterHandler	exception_id Xil_ExceptionHandler * Handler void ** Data
void	Xil_ExceptionInit	None.
void	Xil_DataAbortHandler	void
void	Xil_PrefetchAbortHandler	void
void	Xil_UndefinedExceptionHandler	void

Functions

Xil_ExceptionRegisterHandler

Register a handler for a specific exception.

This handler is being called when the processor encounters the specified exception.

Note: None.

Prototype

```
void Xil_ExceptionRegisterHandler(u32 Exception_id, Xil_ExceptionHandler
Handler, void *Data);
```

Parameters

The following table lists the `Xil_ExceptionRegisterHandler` function arguments.

Table 141: Xil_ExceptionRegisterHandler Arguments

Name	Description
exception_id	contains the ID of the exception source and should be in the range of 0 to XIL_EXCEPTION_ID_LAST. See xil_exception.h for further information.
Handler	to the Handler for that exception.
Data	is a reference to Data that will be passed to the Handler when it gets called.

Returns

None.

Xil_ExceptionRemoveHandler

Removes the Handler for a specific exception Id.

The stub Handler is then registered for this exception Id.

Note: None.

Prototype

```
void Xil_ExceptionRemoveHandler(u32 Exception_id);
```

Parameters

The following table lists the `Xil_ExceptionRemoveHandler` function arguments.

Table 142: Xil_ExceptionRemoveHandler Arguments

Name	Description
exception_id	contains the ID of the exception source and should be in the range of 0 to XIL_EXCEPTION_ID_LAST. See xil_exception.h for further information.

Returns

None.

Xil_GetExceptionRegisterHandler

Get a handler for a specific exception.

This handler is being called when the processor encounters the specified exception.

Note: None.

Prototype

```
void Xil_GetExceptionRegisterHandler(u32 Exception_id, Xil_ExceptionHandler *Handler, void **Data);
```

Parameters

The following table lists the `Xil_GetExceptionRegisterHandler` function arguments.

Table 143: Xil_GetExceptionRegisterHandler Arguments

Name	Description
exception_id	contains the ID of the exception source and should be in the range of 0 to XIL_EXCEPTION_ID_LAST. See <code>xil_exception.h</code> for further information.
Handler	to the Handler for that exception.
Data	is a reference to Data that will be passed to the Handler when it gets called.

Returns

None.

Xil_ExceptionInit

The function is a common API used to initialize exception handlers across all supported arm processors.

For ARM Cortex-A53, Cortex-R5, and Cortex-A9, the exception handlers are being initialized statically and this function does not do anything. However, it is still present to take care of backward compatibility issues (in earlier versions of BSPs, this API was being used to initialize exception handlers).

Note: None.

Prototype

```
void Xil_ExceptionInit(void);
```

Parameters

The following table lists the `Xil_ExceptionInit` function arguments.

Table 144: Xil_ExceptionInit Arguments

Name	Description
None.	

Returns

None.

Xil_DataAbortHandler

Default Data abort handler which prints data fault status register through which information about data fault can be acquired

Prototype

```
void Xil_DataAbortHandler(void *CallBackRef);
```

Xil_PrefetchAbortHandler

Default Prefetch abort handler which prints prefetch fault status register through which information about instruction prefetch fault can be acquired.

Prototype

```
void Xil_PrefetchAbortHandler(void *CallBackRef);
```

Xil_UndefinedExceptionHandler

Default undefined exception handler which prints address of the undefined instruction if debug prints are enabled.

Prototype

```
void Xil_UndefinedExceptionHandler(void *CallBackRef);
```

Cortex A9 Processor API

Standalone BSP contains boot code, cache, exception handling, file and memory management, configuration, time and processor-specific include functions. It supports gcc compilers.

Cortex A9 Processor Boot Code

The boot code performs minimum configuration which is required for an application to run starting from processor's reset state. Below is a sequence illustrating what all configuration is performed before control reaches to main function.

1. Program vector table base for exception handling
2. Invalidate instruction cache, data cache and TLBs
3. Program stack pointer for various modes (IRQ, FIQ, supervisor, undefine, abort, system)
4. Configure MMU with short descriptor translation table format and program base address of translation table
5. Enable data cache, instruction cache and MMU
6. Enable Floating point unit
7. Transfer control to `_start` which clears BSS sections, initializes global timer and runs global constructor before jumping to main application

None.

Note:

`translation_table.S` contains a static page table required by MMU for cortex-A9. This translation table is flat mapped (input address = output address) with default memory attributes defined for zynq architecture. It utilizes short descriptor translation table format with each section defining 1MB of memory.

The overview of translation table memory attributes is described below.

	Memory Range	Definition in Translation Table
DDR	0x00000000 - 0x3FFFFFFF	Normal write-back Cacheable
PL	0x40000000 - 0xBFFFFFFF	Strongly Ordered
Reserved	0xC0000000 - 0xDFFFFFFF	Unassigned
Memory mapped devices	0xE0000000 - 0xE02FFFFF	Device Memory
Reserved	0xE0300000 - 0xE0FFFFFF	Unassigned
NAND, NOR	0xE1000000 - 0xE3FFFFFF	Device memory
SRAM	0xE4000000 - 0xE5FFFFFF	Normal write-back Cacheable
Reserved	0xE6000000 - 0xF7FFFFFF	Unassigned
AMBA APB Peripherals	0xF8000000 - 0xF8FFFFFF	Device Memory
Reserved	0xF9000000 - 0xFBFFFFFF	Unassigned
Linear QSPI - XIP	0xFC000000 - 0xFDFFFFFF	Normal write-through cacheable
Reserved	0xFE000000 - 0xFFEFFFFFF	Unassigned
OCM	0xFFF00000 - 0xFFFFFFFF	Normal inner write-back cacheable

For region 0x00000000 - 0x3FFFFFFF, a system where DDR is less than 1GB, region after DDR and before PL is marked as undefined/reserved in translation table. In 0xF8000000 - 0xF8FFFFFF, 0xF800C00 - 0xF800FFF, 0xF8010000 - 0xF88FFFFFF and 0xF8F03000 to 0xF8FFFFFF are reserved but due to granual size of 1MB, it is not possible to define separate regions for them. For region 0xFFF00000 - 0xFFFFFFFF, 0xFFF00000 to 0xFFFB0000 is reserved but due to 1MB granual size, it is not possible to define separate region for it

Note:

Cortex A9 Processor Cache Functions

Cache functions provide access to cache related operations such as flush and invalidate for instruction and data caches. It gives option to perform the cache operations on a single cacheline, a range of memory and an entire cache.

Table 145: Quick Function Reference

Type	Name	Arguments
void	Xil_DCacheEnable	None.
void	Xil_DCacheDisable	None.
void	Xil_DCacheInvalidate	None.
void	Xil_DCacheInvalidateRange	INTPTR adr u32 len
void	Xil_DCacheFlush	None.
void	Xil_DCacheFlushRange	INTPTR adr u32 len
void	Xil_ICacheEnable	None.
void	Xil_ICacheDisable	None.
void	Xil_ICacheInvalidate	None.
void	Xil_ICacheInvalidateRange	INTPTR adr u32 len
void	Xil_DCacheInvalidateLine	u32 adr

Table 145: Quick Function Reference (cont'd)

Type	Name	Arguments
void	Xil_DCacheFlushLine	u32 adr
void	Xil_DCacheStoreLine	u32 adr
void	Xil_ICacheInvalidateLine	u32 adr
void	Xil_L1DCacheEnable	None.
void	Xil_L1DCacheDisable	None.
void	Xil_L1DCacheInvalidate	None.
void	Xil_L1DCacheInvalidateLine	u32 adr
void	Xil_L1DCacheInvalidateRange	u32 adr u32 len
void	Xil_L1DCacheFlush	None.
void	Xil_L1DCacheFlushLine	u32 adr
void	Xil_L1DCacheFlushRange	u32 adr u32 len
void	Xil_L1DCacheStoreLine	Address
void	Xil_L1ICacheEnable	None.
void	Xil_L1ICacheDisable	None.
void	Xil_L1ICacheInvalidate	None.
void	Xil_L1ICacheInvalidateLine	u32 adr
void	Xil_L1ICacheInvalidateRange	u32 adr u32 len
void	Xil_L2CacheEnable	None.

Table 145: Quick Function Reference (cont'd)

Type	Name	Arguments
void	Xil_L2CacheDisable	None.
void	Xil_L2CacheInvalidate	None.
void	Xil_L2CacheInvalidateLine	u32 adr
void	Xil_L2CacheInvalidateRange	u32 adr u32 len
void	Xil_L2CacheFlush	None.
void	Xil_L2CacheFlushLine	u32 adr
void	Xil_L2CacheFlushRange	u32 adr u32 len
void	Xil_L2CacheStoreLine	u32 adr

Functions

Xil_DCacheEnable

Enable the Data cache.

Note: None.

Prototype

```
void Xil_DCacheEnable(void);
```

Parameters

The following table lists the `Xil_DCacheEnable` function arguments.

Table 146: Xil_DCacheEnable Arguments

Name	Description
None.	

Returns

None.

Xil_DCacheDisable

Disable the Data cache.

Note: None.

Prototype

```
void Xil_DCacheDisable(void);
```

Parameters

The following table lists the `Xil_DCacheDisable` function arguments.

Table 147: Xil_DCacheDisable Arguments

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidate

Invalidate the entire Data cache.

Note: None.

Prototype

```
void Xil_DCacheInvalidate(void);
```

Parameters

The following table lists the `Xil_DCacheInvalidate` function arguments.

Table 148: Xil_DCacheInvalidate Arguments

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidateRange

Invalidate the Data cache for the given address range.

If the bytes specified by the address range are cached by the Data cache, the cachelines containing those bytes are invalidated. If the cachelines are modified (dirty), the modified contents are lost and NOT written to the system memory before the lines are invalidated. data. This issue raises few possibilities. work.

1. Avoid situations where invalidation has to be done after the data is updated by peripheral/DMA directly into the memory. It is not tough to achieve (may be a bit risky). The common use case to do invalidation is when a DMA happens. Generally for such use cases, buffers can be allocated first and then start the DMA. The practice that needs to be followed here is, immediately after buffer allocation and before starting the DMA, do the invalidation. With this approach, invalidation need not to be done after the DMA transfer is over. are brought into cache (between the time it is invalidated and DMA completes) because of some speculative prefetching or reading data for a variable present in the same cache line, then we will have to invalidate the cache after DMA is complete.

Note: None.

Prototype

```
void Xil_DCacheInvalidateRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_DCacheInvalidateRange` function arguments.

Table 149: Xil_DCacheInvalidateRange Arguments

Name	Description
adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Xil_DCacheFlush

Flush the entire Data cache.

Note: None.

Prototype

```
void Xil_DCacheFlush(void);
```

Parameters

The following table lists the `Xil_DCacheFlush` function arguments.

Table 150: Xil_DCacheFlush Arguments

Name	Description
None.	

Returns

None.

Xil_DCacheFlushRange

Flush the Data cache for the given address range.

If the bytes specified by the address range are cached by the data cache, the cachelines containing those bytes are invalidated. If the cachelines are modified (dirty), they are written to the system memory before the lines are invalidated.

Note: None.

Prototype

```
void Xil_DCacheFlushRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_DCacheFlushRange` function arguments.

Table 151: Xil_DCacheFlushRange Arguments

Name	Description
adr	32bit start address of the range to be flushed.
len	Length of the range to be flushed in bytes.

Returns

None.

Xil_ICacheEnable

Enable the instruction cache.

Note: None.

Prototype

```
void Xil_ICacheEnable(void);
```

Parameters

The following table lists the `Xil_ICacheEnable` function arguments.

Table 152: Xil_ICacheEnable Arguments

Name	Description
None.	

Returns

None.

Xil_ICacheDisable

Disable the instruction cache.

Note: None.

Prototype

```
void Xil_ICacheDisable(void);
```

Parameters

The following table lists the `Xil_ICacheDisable` function arguments.

Table 153: Xil_ICacheDisable Arguments

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidate

Invalidate the entire instruction cache.

Note: None.

Prototype

```
void Xil_ICacheInvalidate(void);
```

Parameters

The following table lists the `Xil_ICacheInvalidate` function arguments.

Table 154: Xil_ICacheInvalidate Arguments

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidateRange

Invalidate the instruction cache for the given address range.

If the instructions specified by the address range are cached by the instruction cache, the cachelines containing those instructions are invalidated.

Note: None.

Prototype

```
void Xil_ICacheInvalidateRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_ICacheInvalidateRange` function arguments.

Table 155: Xil_ICacheInvalidateRange Arguments

Name	Description
adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Xil_DCacheInvalidateLine

Invalidate a Data cache line.

If the byte specified by the address (*adr*) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are NOT written to the system memory before the line is invalidated.

Note: The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_DCacheInvalidateLine(u32 adr);
```

Parameters

The following table lists the `Xil_DCacheInvalidateLine` function arguments.

Table 156: Xil_DCacheInvalidateLine Arguments

Name	Description
<i>adr</i>	32bit address of the data to be flushed.

Returns

None.

Xil_DCacheFlushLine

Flush a Data cache line.

If the byte specified by the address (*adr*) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

Note: The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_DCacheFlushLine(u32 adr);
```

Parameters

The following table lists the `Xil_DCacheFlushLine` function arguments.

Table 157: Xil_DCacheFlushLine Arguments

Name	Description
<i>adr</i>	32bit address of the data to be flushed.

Returns

None.

Xil_DCacheStoreLine

Store a Data cache line.

If the byte specified by the address (*adr*) is cached by the Data cache and the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory. After the store completes, the cacheline is marked as unmodified (not dirty).

Note: The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_DCacheStoreLine(u32 adr);
```

Parameters

The following table lists the `Xil_DCacheStoreLine` function arguments.

Table 158: Xil_DCacheStoreLine Arguments

Name	Description
<code>adr</code>	32bit address of the data to be stored.

Returns

None.

Xil_ICacheInvalidateLine

Invalidate an instruction cache line.

If the instruction specified by the address is cached by the instruction cache, the cacheline containing that instruction is invalidated.

Note: The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_ICacheInvalidateLine(u32 adr);
```

Parameters

The following table lists the `Xil_ICacheInvalidateLine` function arguments.

Table 159: Xil_ICacheInvalidateLine Arguments

Name	Description
adr	32bit address of the instruction to be invalidated.

Returns

None.

Xil_L1DCacheEnable

Enable the level 1 Data cache.

Note: None.

Prototype

```
void Xil_L1DCacheEnable(void);
```

Parameters

The following table lists the `Xil_L1DCacheEnable` function arguments.

Table 160: Xil_L1DCacheEnable Arguments

Name	Description
None.	

Returns

None.

Xil_L1DCacheDisable

Disable the level 1 Data cache.

Note: None.

Prototype

```
void Xil_L1DCacheDisable(void);
```

Parameters

The following table lists the `Xil_L1DCacheDisable` function arguments.

Table 161: Xil_L1DCacheDisable Arguments

Name	Description
None.	

Returns

None.

Xil_L1DCacheInvalidate

Invalidate the level 1 Data cache.

Note: In Cortex A9, there is no cp instruction for invalidating the whole D-cache. This function invalidates each line by set/way.

Prototype

```
void Xil_L1DCacheInvalidate(void);
```

Parameters

The following table lists the `Xil_L1DCacheInvalidate` function arguments.

Table 162: Xil_L1DCacheInvalidate Arguments

Name	Description
None.	

Returns

None.

Xil_L1DCacheInvalidateLine

Invalidate a level 1 Data cache line.

If the byte specified by the address (`Addr`) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are NOT written to system memory before the line is invalidated.

Note: The bottom 5 bits are set to 0, forced by architecture.

Prototype

```
void Xil_L1DCacheInvalidateLine(u32 adr);
```

Parameters

The following table lists the `Xil_L1DCacheInvalidateLine` function arguments.

Table 163: Xil_L1DCacheInvalidateLine Arguments

Name	Description
adr	32bit address of the data to be invalidated.

Returns

None.

Xil_L1DCacheInvalidateRange

Invalidate the level 1 Data cache for the given address range.

If the bytes specified by the address range are cached by the Data cache, the cachelines containing those bytes are invalidated. If the cachelines are modified (dirty), the modified contents are lost and NOT written to the system memory before the lines are invalidated.

Note: None.

Prototype

```
void Xil_L1DCacheInvalidateRange(u32 adr, u32 len);
```

Parameters

The following table lists the `Xil_L1DCacheInvalidateRange` function arguments.

Table 164: Xil_L1DCacheInvalidateRange Arguments

Name	Description
adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Xil_L1DCacheFlush

Flush the level 1 Data cache.

Note: In Cortex A9, there is no cp instruction for flushing the whole D-cache. Need to flush each line.

Prototype

```
void Xil_L1DCacheFlush(void);
```

Parameters

The following table lists the `Xil_L1DCacheFlush` function arguments.

Table 165: Xil_L1DCacheFlush Arguments

Name	Description
None.	

Returns

None.

Xil_L1DCacheFlushLine

Flush a level 1 Data cache line.

If the byte specified by the address (`adr`) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

Note: The bottom 5 bits are set to 0, forced by architecture.

Prototype

```
void Xil_L1DCacheFlushLine(u32 adr);
```

Parameters

The following table lists the `Xil_L1DCacheFlushLine` function arguments.

Table 166: Xil_L1DCacheFlushLine Arguments

Name	Description
<code>adr</code>	32bit address of the data to be flushed.

Returns

None.

Xil_L1DCacheFlushRange

Flush the level 1 Data cache for the given address range.

If the bytes specified by the address range are cached by the Data cache, the cacheline containing those bytes are invalidated. If the cachelines are modified (dirty), they are written to system memory before the lines are invalidated.

Note: None.

Prototype

```
void Xil_L1DCacheFlushRange(u32 adr, u32 len);
```

Parameters

The following table lists the `Xil_L1DCacheFlushRange` function arguments.

Table 167: Xil_L1DCacheFlushRange Arguments

Name	Description
adr	32bit start address of the range to be flushed.
len	Length of the range to be flushed in bytes.

Returns

None.

Xil_L1DCacheStoreLine

Store a level 1 Data cache line.

If the byte specified by the address (adr) is cached by the Data cache and the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory. After the store completes, the cacheline is marked as unmodified (not dirty).

Note: The bottom 5 bits are set to 0, forced by architecture.

Prototype

```
void Xil_L1DCacheStoreLine(u32 adr);
```

Parameters

The following table lists the `Xil_L1DCacheStoreLine` function arguments.

Table 168: Xil_L1DCacheStoreLine Arguments

Name	Description
Address	to be stored.

Returns

None.

Xil_L1ICacheEnable

Enable the level 1 instruction cache.

Note: None.

Prototype

```
void Xil_L1ICacheEnable(void);
```

Parameters

The following table lists the `Xil_L1ICacheEnable` function arguments.

Table 169: Xil_L1ICacheEnable Arguments

Name	Description
None.	

Returns

None.

Xil_L1ICacheDisable

Disable level 1 the instruction cache.

Note: None.

Prototype

```
void Xil_L1ICacheDisable(void);
```

Parameters

The following table lists the `Xil_L1ICacheDisable` function arguments.

Table 170: Xil_L1ICacheDisable Arguments

Name	Description
None.	

Returns

None.

Xil_L1ICacheInvalidate

Invalidate the entire level 1 instruction cache.

Note: None.

Prototype

```
void Xil_L1ICacheInvalidate(void);
```

Parameters

The following table lists the `Xil_L1ICacheInvalidate` function arguments.

Table 171: Xil_L1ICacheInvalidate Arguments

Name	Description
None.	

Returns

None.

Xil_L1ICacheInvalidateLine

Invalidate a level 1 instruction cache line.

If the instruction specified by the address is cached by the instruction cache, the cacheline containing that instruction is invalidated.

Note: The bottom 5 bits are set to 0, forced by architecture.

Prototype

```
void Xil_L1ICacheInvalidateLine(u32 adr);
```

Parameters

The following table lists the `Xil_L1ICacheInvalidateLine` function arguments.

Table 172: Xil_L1ICacheInvalidateLine Arguments

Name	Description
adr	32bit address of the instruction to be invalidated.

Returns

None.

Xil_L1ICacheInvalidateRange

Invalidate the level 1 instruction cache for the given address range.

If the instructions specified by the address range are cached by the instruction cache, the cacheline containing those bytes are invalidated.

Note: None.

Prototype

```
void Xil_L1ICacheInvalidateRange(u32 adr, u32 len);
```

Parameters

The following table lists the `Xil_L1ICacheInvalidateRange` function arguments.

Table 173: Xil_L1ICacheInvalidateRange Arguments

Name	Description
adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Xil_L2CacheEnable

Enable the L2 cache.

Note: None.

Prototype

```
void Xil_L2CacheEnable(void);
```

Parameters

The following table lists the `Xil_L2CacheEnable` function arguments.

Table 174: Xil_L2CacheEnable Arguments

Name	Description
None.	

Returns

None.

Xil_L2CacheDisable

Disable the L2 cache.

Note: None.

Prototype

```
void Xil_L2CacheDisable(void);
```

Parameters

The following table lists the `Xil_L2CacheDisable` function arguments.

Table 175: Xil_L2CacheDisable Arguments

Name	Description
None.	

Returns

None.

Xil_L2CacheInvalidate

Invalidate the entire level 2 cache.

Note: None.

Prototype

```
void Xil_L2CacheInvalidate(void);
```

Parameters

The following table lists the `Xil_L2CacheInvalidate` function arguments.

Table 176: Xil_L2CacheInvalidate Arguments

Name	Description
None.	

Returns

None.

Xil_L2CacheInvalidateLine

Invalidate a level 2 cache line.

If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are NOT written to system memory before the line is invalidated.

Note: The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_L2CacheInvalidateLine(u32 adr);
```

Parameters

The following table lists the Xil_L2CacheInvalidateLine function arguments.

Table 177: Xil_L2CacheInvalidateLine Arguments

Name	Description
adr	32bit address of the data/instruction to be invalidated.

Returns

None.

Xil_L2CacheInvalidateRange

Invalidate the level 2 cache for the given address range.

If the bytes specified by the address range are cached by the L2 cache, the cacheline containing those bytes are invalidated. If the cachelines are modified (dirty), the modified contents are lost and are NOT written to system memory before the lines are invalidated.

Note: None.

Prototype

```
void Xil_L2CacheInvalidateRange(u32 adr, u32 len);
```

Parameters

The following table lists the `Xil_L2CacheInvalidateRange` function arguments.

Table 178: Xil_L2CacheInvalidateRange Arguments

Name	Description
adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Xil_L2CacheFlush

Flush the entire level 2 cache.

Note: None.

Prototype

```
void Xil_L2CacheFlush(void);
```

Parameters

The following table lists the `Xil_L2CacheFlush` function arguments.

Table 179: Xil_L2CacheFlush Arguments

Name	Description
None.	

Returns

None.

Xil_L2CacheFlushLine

Flush a level 2 cache line.

If the byte specified by the address (`adr`) is cached by the L2 cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

Note: The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_L2CacheFlushLine(u32 adr);
```

Parameters

The following table lists the `Xil_L2CacheFlushLine` function arguments.

Table 180: Xil_L2CacheFlushLine Arguments

Name	Description
adr	32bit address of the data/instruction to be flushed.

Returns

None.

Xil_L2CacheFlushRange

Flush the level 2 cache for the given address range.

If the bytes specified by the address range are cached by the L2 cache, the cacheline containing those bytes are invalidated. If the cachelines are modified (dirty), they are written to the system memory before the lines are invalidated.

Note: None.

Prototype

```
void Xil_L2CacheFlushRange(u32 adr, u32 len);
```

Parameters

The following table lists the `Xil_L2CacheFlushRange` function arguments.

Table 181: Xil_L2CacheFlushRange Arguments

Name	Description
adr	32bit start address of the range to be flushed.
len	Length of the range to be flushed in bytes.

Returns

None.

Xil_L2CacheStoreLine

Store a level 2 cache line.

If the byte specified by the address (adr) is cached by the L2 cache and the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory. After the store completes, the cacheline is marked as unmodified (not dirty).

Note: The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_L2CacheStoreLine(u32 adr);
```

Parameters

The following table lists the `Xil_L2CacheStoreLine` function arguments.

Table 182: Xil_L2CacheStoreLine Arguments

Name	Description
adr	32bit address of the data/instruction to be stored.

Returns

None.

Cortex A9 Processor MMU Functions

MMU functions equip users to enable MMU, disable MMU and modify default memory attributes of MMU table as per the need.

Table 183: Quick Function Reference

Type	Name	Arguments
void	Xil_SetTlbAttributes	INTPTR Addr u32 attrib
void	Xil_EnableMMU	None.
void	Xil_DisableMMU	None.
void *	Xil_MemMap	UINTPTR PhysAddr size_t size u32 flags

Functions

Xil_SetTlbAttributes

This function sets the memory attributes for a section covering 1MB of memory in the translation table.

Note: The MMU or D-cache does not need to be disabled before changing a translation table entry.

Prototype

```
void Xil_SetTlbAttributes(INTPTR Addr, u32 attrib);
```

Parameters

The following table lists the `Xil_SetTlbAttributes` function arguments.

Table 184: Xil_SetTlbAttributes Arguments

Name	Description
Addr	32-bit address for which memory attributes need to be set.
attrib	Attribute for the given memory region. <code>xil_mmu.h</code> contains definitions of commonly used memory attributes which can be utilized for this function.

Returns

None.

Xil_EnableMMU

Enable MMU for cortex A9 processor.

This function invalidates the instruction and data caches, and then enables MMU.

Prototype

```
void Xil_EnableMMU(void);
```

Parameters

The following table lists the `Xil_EnableMMU` function arguments.

Table 185: Xil_EnableMMU Arguments

Name	Description
None.	

Returns

None.

Xil_DisableMMU

Disable MMU for Cortex A9 processors.

This function invalidates the TLBs, Branch Predictor Array and flushed the D Caches before disabling the MMU.

Note: When the MMU is disabled, all the memory accesses are treated as strongly ordered.

Prototype

```
void Xil_DisableMMU(void);
```

Parameters

The following table lists the `Xil_DisableMMU` function arguments.

Table 186: Xil_DisableMMU Arguments

Name	Description
None.	

Returns

None.

Xil_MemMap

Memory mapping for Cortex A9 processor.

Note: : Previously this was implemented in libmetal. Move to embeddedsw as this functionality is specific to A9 processor.

Prototype

```
void * Xil_MemMap(UINTPTR PhysAddr, size_t size, u32 flags);
```

Parameters

The following table lists the `Xil_MemMap` function arguments.

Table 187: Xil_MemMap Arguments

Name	Description
PhysAddr	is physical address.
size	is size of region.
flags	is flags used to set translation table.

Returns

Pointer to virtual address.

Cortex A9 Time Functions

xtime_l.h provides access to the 64-bit Global Counter in the PMU. This counter increases by one at every two processor cycles. These functions can be used to get/set time in the global timer.

Table 188: Quick Function Reference

Type	Name	Arguments
void	XTime_SetTime	XTime Xtime_Global
void	XTime_GetTime	XTime * Xtime_Global

Functions

XTime_SetTime

Set the time in the Global Timer Counter Register.

Note: When this function is called by any one processor in a multi-processor environment, reference time will reset/lost for all processors.

Prototype

```
void XTime_SetTime(XTime Xtime_Global);
```

Parameters

The following table lists the XTime_SetTime function arguments.

Table 189: XTime_SetTime Arguments

Name	Description
Xtime_Global	64-bit Value to be written to the Global Timer Counter Register.

Returns

None.

XTime_GetTime

Get the time from the Global Timer Counter Register.

Note: None.

Prototype

```
void XTime_GetTime(XTime *Xtime_Global);
```

Parameters

The following table lists the `XTime_GetTime` function arguments.

Table 190: XTime_GetTime Arguments

Name	Description
Xtime_Global	Pointer to the 64-bit location which will be updated with the current timer value.

Returns

None.

Cortex A9 Event Counter Function

Cortex A9 event counter functions can be utilized to configure and control the Cortex-A9 performance monitor events.

Cortex-A9 performance monitor has six event counters which can be used to count a variety of events described in Cortex-A9 TRM. `xpm_counter.h` defines configurations `XPM_CNTRCFGx` which can be used to program the event counters to count a set of events.

Note: It doesn't handle the Cortex-A9 cycle counter, as the cycle counter is being used for time keeping.

Table 191: Quick Function Reference

Type	Name	Arguments
void	Xpm_SetEvents	s32 PmcrCfg
void	Xpm_GetEventCounters	u32 * PmCtrValue

Functions

Xpm_SetEvents

This function configures the Cortex A9 event counters controller, with the event codes, in a configuration selected by the user and enables the counters.

Note: None.

Prototype

```
void Xpm_SetEvents(s32 PmcrCfg);
```

Parameters

The following table lists the `Xpm_SetEvents` function arguments.

Table 192: Xpm_SetEvents Arguments

Name	Description
PmcrCfg	Configuration value based on which the event counters are configured. XPM_CNTRCFG* values defined in xpm_counter.h can be utilized for setting configuration.

Returns

None.

Xpm_GetEventCounters

This function disables the event counters and returns the counter values.

Note: None.

Prototype

```
void Xpm_GetEventCounters(u32 *PmCtrValue);
```

Parameters

The following table lists the `Xpm_GetEventCounters` function arguments.

Table 193: Xpm_GetEventCounters Arguments

Name	Description
PmCtrValue	Pointer to an array of type u32 PmCtrValue[6]. It is an output parameter which is used to return the PM counter values.

Returns

None.

PL310 L2 Event Counters Functions

xl2cc_counter.h contains APIs for configuring and controlling the event counters in PL310 L2 cache controller. PL310 has two event counters which can be used to count variety of events like DRHIT, DRREQ, DWHIT, DWREQ, etc. xl2cc_counter.h contains definitions for different configurations which can be used for the event counters to count a set of events.

Table 194: Quick Function Reference

Type	Name	Arguments
void	XL2cc_EventCtrInit	s32 Event0 s32 Event1
void	XL2cc_EventCtrStart	None.
void	XL2cc_EventCtrStop	u32 * EveCtr0

Functions

XL2cc_EventCtrInit

This function initializes the event counters in L2 Cache controller with a set of event codes specified by the user.

Note: The definitions for event codes XL2CC_* can be found in xl2cc_counter.h.

Prototype

```
void XL2cc_EventCtrInit(s32 Event0, s32 Event1);
```

Parameters

The following table lists the XL2cc_EventCtrInit function arguments.

Table 195: XL2cc_EventCtrInit Arguments

Name	Description
Event0	Event code for counter 0.
Event1	Event code for counter 1.

Returns

None.

XL2cc_EventCtrStart

This function starts the event counters in L2 Cache controller.

Note: None.

Prototype

```
void XL2cc_EventCtrStart(void);
```

Parameters

The following table lists the `XL2cc_EventCtrStart` function arguments.

Table 196: XL2cc_EventCtrStart Arguments

Name	Description
None.	

Returns

None.

XL2cc_EventCtrStop

This function disables the event counters in L2 Cache controller, saves the counter values and resets the counters.

Note: None.

Prototype

```
void XL2cc_EventCtrStop(u32 *EveCtr0, u32 *EveCtr1);
```

Parameters

The following table lists the `XL2cc_EventCtrStop` function arguments.

Table 197: XL2cc_EventCtrStop Arguments

Name	Description
EveCtr0	Output parameter which is used to return the value in event counter 0. EveCtr1: Output parameter which is used to return the value in event counter 1.

Returns

None.

Cortex A9 Processor and pl310 Errata Support

Various ARM errata are handled in the standalone BSP. The implementation for errata handling follows ARM guidelines and is based on the open source Linux support for these errata.

Note: The errata handling is enabled by default. To disable handling of all the errata globally, un-define the macro `ENABLE_ARM_ERRATA` in `xil_errata.h`. To disable errata on a per-erratum basis, un-define relevant macros in `xil_errata.h`.

Cortex A9 Processor Specific Include Files

The `xpseudo_asm.h` includes `xreg_cortexa9.h` and `xpseudo_asm_gcc.h`.

The `xreg_cortexa9.h` file contains definitions for inline assembler code. It provides inline definitions for Cortex A9 GPRs, SPRs, MPE registers, co-processor registers and Debug registers.

The `xpseudo_asm_gcc.h` contains the definitions for the most often used inline assembler instructions, available as macros. These can be very useful for tasks such as setting or getting special purpose registers, synchronization, or cache manipulation etc. These inline assembler instructions can be used from drivers and user applications written in C.

Cortex A53 32-bit Processor API

Cortex-A53 standalone BSP contains two separate BSPs for 32-bit mode and 64-bit mode. The 32-bit mode of cortex-A53 is compatible with ARMv7-A architecture.

Cortex A53 32-bit Processor Boot Code

The boot code performs minimum configuration which is required for an application to run starting from processor's reset state. Below is a sequence illustrating what all configuration is performed before control reaches to main function.

1. Program vector table base for exception handling
2. Invalidate instruction cache, data cache and TLBs
3. Program stack pointer for various modes (IRQ, FIQ, supervisor, undefine, abort, system)
4. Program counter frequency
5. Configure MMU with short descriptor translation table format and program base address of translation table

6. Enable data cache, instruction cache and MMU
7. Transfer control to `_start` which clears BSS sections and runs global constructor before jumping to main application

Cortex A53 32-bit Processor Cache Functions

Cache functions provide access to cache related operations such as flush and invalidate for instruction and data caches. It gives option to perform the cache operations on a single cacheline, a range of memory and an entire cache.

Table 198: Quick Function Reference

Type	Name	Arguments
void	Xil_DCacheEnable	None.
void	Xil_DCacheDisable	None.
void	Xil_DCacheInvalidate	None.
void	Xil_DCacheInvalidateRange	INTPTR adr u32 len
void	Xil_DCacheFlush	None.
void	Xil_DCacheFlushRange	INTPTR adr u32 len
void	Xil_DCacheInvalidateLine	u32 adr
void	Xil_DCacheFlushLine	u32 adr
void	Xil_ICacheInvalidateLine	u32 adr
void	Xil_ICacheEnable	None.
void	Xil_ICacheDisable	None.
void	Xil_ICacheInvalidate	None.
void	Xil_ICacheInvalidateRange	INTPTR adr u32 len

Functions

Xil_DCacheEnable

Enable the Data cache.

Note: None.

Prototype

```
void Xil_DCacheEnable(void);
```

Parameters

The following table lists the `Xil_DCacheEnable` function arguments.

Table 199: Xil_DCacheEnable Arguments

Name	Description
None.	

Returns

None.

Xil_DCacheDisable

Disable the Data cache.

Note: None.

Prototype

```
void Xil_DCacheDisable(void);
```

Parameters

The following table lists the `Xil_DCacheDisable` function arguments.

Table 200: Xil_DCacheDisable Arguments

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidate

Invalidate the Data cache.

The contents present in the data cache are cleaned and invalidated.

Note: In Cortex-A53, functionality to simply invalid the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate to avoid such corruption.

Prototype

```
void Xil_DCacheInvalidate(void);
```

Parameters

The following table lists the `Xil_DCacheInvalidate` function arguments.

Table 201: Xil_DCacheInvalidate Arguments

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidateRange

Invalidate the Data cache for the given address range.

The cachelines present in the address range are cleaned and invalidated

@notice In Cortex-A53, functionality to simply invalid the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate to avoid such corruption.

Prototype

```
void Xil_DCacheInvalidateRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_DCacheInvalidateRange` function arguments.

Table 202: Xil_DCacheInvalidateRange Arguments

Name	Description
adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Xil_DCacheFlush

Flush the Data cache.

@notice None.

Prototype

```
void Xil_DCacheFlush(void);
```

Parameters

The following table lists the `Xil_DCacheFlush` function arguments.

Table 203: Xil_DCacheFlush Arguments

Name	Description
None.	

Returns

None.

Xil_DCacheFlushRange

Flush the Data cache for the given address range.

If the bytes specified by the address range are cached by the Data cache, the cachelines containing those bytes are invalidated. If the cachelines are modified (dirty), they are written to system memory before the lines are invalidated.

@notice None.

Prototype

```
void Xil_DCacheFlushRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_DCacheFlushRange` function arguments.

Table 204: Xil_DCacheFlushRange Arguments

Name	Description
adr	32bit start address of the range to be flushed.
len	Length of range to be flushed in bytes.

Returns

None.

Xil_DCacheInvalidateLine

Invalidate a Data cache line.

The cacheline is cleaned and invalidated.

Note: In Cortex-A53, functionality to simply invalid the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate to avoid such corruption.

Prototype

```
void Xil_DCacheInvalidateLine(u32 adr);
```

Parameters

The following table lists the `Xil_DCacheInvalidateLine` function arguments.

Table 205: Xil_DCacheInvalidateLine Arguments

Name	Description
adr	32 bit address of the data to be invalidated.

Returns

None.

Xil_DCacheFlushLine

Flush a Data cache line.

If the byte specified by the address (*adr*) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

@notice The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_DCacheFlushLine(u32 adr);
```

Parameters

The following table lists the `Xil_DCacheFlushLine` function arguments.

Table 206: Xil_DCacheFlushLine Arguments

Name	Description
<i>adr</i>	32bit address of the data to be flushed.

Returns

None.

Xil_ICacheInvalidateLine

Invalidate an instruction cache line.

If the instruction specified by the address is cached by the instruction cache, the cacheline containing that instruction is invalidated.

@notice The bottom 4 bits are set to 0, forced by architecture.

Prototype

```
void Xil_ICacheInvalidateLine(u32 adr);
```

Parameters

The following table lists the `Xil_ICacheInvalidateLine` function arguments.

Table 207: Xil_ICacheInvalidateLine Arguments

Name	Description
<i>adr</i>	32bit address of the instruction to be invalidated..

Returns

None.

Xil_ICacheEnable

Enable the instruction cache.

@notice None.

Prototype

```
void Xil_ICacheEnable(void);
```

Parameters

The following table lists the `Xil_ICacheEnable` function arguments.

Table 208: Xil_ICacheEnable Arguments

Name	Description
None.	

Returns

None.

Xil_ICacheDisable

Disable the instruction cache.

Note: None.

Prototype

```
void Xil_ICacheDisable(void);
```

Parameters

The following table lists the `Xil_ICacheDisable` function arguments.

Table 209: Xil_ICacheDisable Arguments

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidate

Invalidate the entire instruction cache.

Note: None.

Prototype

```
void Xil_ICacheInvalidate(void);
```

Parameters

The following table lists the `Xil_ICacheInvalidate` function arguments.

Table 210: Xil_ICacheInvalidate Arguments

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidateRange

Invalidate the instruction cache for the given address range.

If the instructions specified by the address range are cached by the instruction cache, the cachelines containing those instructions are invalidated.

@notice None.

Prototype

```
void Xil_ICacheInvalidateRange(INTPTR adr, u32 len);
```

Parameters

The following table lists the `Xil_ICacheInvalidateRange` function arguments.

Table 211: Xil_ICacheInvalidateRange Arguments

Name	Description
adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Cortex A53 32-bit Processor MMU Handling

MMU functions equip users to enable MMU, disable MMU and modify default memory attributes of MMU table as per the need.

None.

Note:

Table 212: Quick Function Reference

Type	Name	Arguments
void	Xil_SetTlbAttributes	UINTPTR Addr u32 attrib
void	Xil_EnableMMU	None.
void	Xil_DisableMMU	None.

Functions

Xil_SetTlbAttributes

This function sets the memory attributes for a section covering 1MB of memory in the translation table.

Note: The MMU or D-cache does not need to be disabled before changing a translation table entry.

Prototype

```
void Xil_SetTlbAttributes(UINTPTR Addr, u32 attrib);
```

Parameters

The following table lists the `Xil_SetTlbAttributes` function arguments.

Table 213: Xil_SetTlbAttributes Arguments

Name	Description
Addr	32-bit address for which the attributes need to be set.
attrib	Attributes for the specified memory region. <code>xil_mmu.h</code> contains commonly used memory attributes definitions which can be utilized for this function.

Returns

None.

Xil_EnableMMU

Enable MMU for Cortex-A53 processor in 32bit mode.

This function invalidates the instruction and data caches before enabling MMU.

Prototype

```
void Xil_EnableMMU(void);
```

Parameters

The following table lists the `Xil_EnableMMU` function arguments.

Table 214: Xil_EnableMMU Arguments

Name	Description
None.	

Returns

None.

Xil_DisableMMU

Disable MMU for Cortex A53 processors in 32bit mode.

This function invalidates the TLBs, Branch Predictor Array and flushed the data cache before disabling the MMU.

Note: When the MMU is disabled, all the memory accesses are treated as strongly ordered.

Prototype

```
void Xil_DisableMMU(void);
```

Parameters

The following table lists the `Xil_DisableMMU` function arguments.

Table 215: Xil_DisableMMU Arguments

Name	Description
None.	

Returns

None.

Cortex A53 32-bit Mode Time Functions

xtime_l.h provides access to the 64-bit physical timer counter.

Table 216: Quick Function Reference

Type	Name	Arguments
void	XTime_StartTimer	None.
void	XTime_SetTime	XTime Xtime_Global
void	XTime_GetTime	XTime * Xtime_Global

Functions

XTime_StartTimer

Start the 64-bit physical timer counter.

Note: The timer is initialized only if it is disabled. If the timer is already running this function does not perform any operation.

Prototype

```
void XTime_StartTimer(void);
```

Parameters

The following table lists the XTime_StartTimer function arguments.

Table 217: XTime_StartTimer Arguments

Name	Description
None.	

Returns

None.

XTime_SetTime

Timer of A53 runs continuously and the time can not be set as desired.

This API doesn't contain anything. It is defined to have uniformity across platforms.

Note: None.

Prototype

```
void XTime_SetTime(XTime Xtime_Global);
```

Parameters

The following table lists the `XTime_SetTime` function arguments.

Table 218: XTime_SetTime Arguments

Name	Description
Xtime_Global	64bit Value to be written to the Global Timer Counter Register. But since the function does not contain anything, the value is not used for anything.

Returns

None.

XTime_GetTime

Get the time from the physical timer counter register.

Note: None.

Prototype

```
void XTime_GetTime(XTime *Xtime_Global);
```

Parameters

The following table lists the `XTime_GetTime` function arguments.

Table 219: XTime_GetTime Arguments

Name	Description
Xtime_Global	Pointer to the 64-bit location to be updated with the current value in physical timer counter.

Returns

None.

Cortex A53 32-bit Processor Specific Include Files

The `xpseudo_asm.h` includes `xreg_cortexa53.h` and `xpseudo_asm_gcc.h`. The `xreg_cortexa53.h` file contains definitions for inline assembler code. It provides inline definitions for Cortex A53 GPRs, SPRs, co-processor registers and floating point registers.

The `xpseudo_asm_gcc.h` contains the definitions for the most often used inline assembler instructions, available as macros. These can be very useful for tasks such as setting or getting special purpose registers, synchronization, or cache manipulation etc. These inline assembler instructions can be used from drivers and user applications written in C.

Cortex A53 64-bit Processor Boot Code

Cortex-A53 standalone BSP contains two separate BSPs for 32-bit mode and 64-bit mode. The 64-bit mode of cortex-A53 contains ARMv8-A architecture. This section provides a linked summary and detailed descriptions of the Cortex A53 64-bit Processor APIs.

Cortex A53 64-bit Processor Cache Functions

Cache functions provide access to cache related operations such as flush and invalidate for instruction and data caches. It gives option to perform the cache operations on a single cacheline, a range of memory and an entire cache.

Table 220: Quick Function Reference

Type	Name	Arguments
void	Xil_DCacheEnable	None.
void	Xil_DCacheDisable	None.
void	Xil_DCacheInvalidate	None.
void	Xil_DCacheInvalidateRange	INTPTR adr INTPTR len
void	Xil_DCacheInvalidateLine	INTPTR adr
void	Xil_DCacheFlush	None.
void	Xil_DCacheFlushLine	INTPTR adr

Table 220: Quick Function Reference (cont'd)

Type	Name	Arguments
void	Xil_ICacheEnable	None.
void	Xil_ICacheDisable	None.
void	Xil_ICacheInvalidate	None.
void	Xil_ICacheInvalidateRange	INTPTR adr INTPTR len
void	Xil_ICacheInvalidateLine	INTPTR adr
void	Xil_ConfigureL1Prefetch	u8 num

Functions

Xil_DCacheEnable

Enable the Data cache.

Note: None.

Prototype

```
void Xil_DCacheEnable(void);
```

Parameters

The following table lists the `Xil_DCacheEnable` function arguments.

Table 221: Xil_DCacheEnable Arguments

Name	Description
None.	

Returns

None.

Xil_DCacheDisable

Disable the Data cache.

Note: None.

Prototype

```
void Xil_DCacheDisable(void);
```

Parameters

The following table lists the `Xil_DCacheDisable` function arguments.

Table 222: Xil_DCacheDisable Arguments

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidate

Invalidate the Data cache.

The contents present in the cache are cleaned and invalidated.

Note: In Cortex-A53, functionality to simply invalidate the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate which avoids such corruption.

Prototype

```
void Xil_DCacheInvalidate(void);
```

Parameters

The following table lists the `Xil_DCacheInvalidate` function arguments.

Table 223: Xil_DCacheInvalidate Arguments

Name	Description
None.	

Returns

None.

Xil_DCacheInvalidateRange

Invalidate the Data cache for the given address range.

The cachelines present in the address range are cleaned and invalidated

Note: In Cortex-A53, functionality to simply invalidate the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate which avoids such corruption.

Prototype

```
void Xil_DCacheInvalidateRange(INTPTR adr, INTPTR len);
```

Parameters

The following table lists the `Xil_DCacheInvalidateRange` function arguments.

Table 224: Xil_DCacheInvalidateRange Arguments

Name	Description
adr	64bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Xil_DCacheInvalidateLine

Invalidate a Data cache line.

The cacheline is cleaned and invalidated.

Note: In Cortex-A53, functionality to simply invalidate the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate which avoids such corruption.

Prototype

```
void Xil_DCacheInvalidateLine(INTPTR adr);
```

Parameters

The following table lists the `Xil_DCacheInvalidateLine` function arguments.

Table 225: Xil_DCacheInvalidateLine Arguments

Name	Description
adr	64bit address of the data to be flushed.

Returns

None.

Xil_DCacheFlush

Flush the Data cache.

Note: None.

Prototype

```
void Xil_DCacheFlush(void);
```

Parameters

The following table lists the Xil_DCacheFlush function arguments.

Table 226: Xil_DCacheFlush Arguments

Name	Description
None.	

Returns

None.

Xil_DCacheFlushLine

Flush a Data cache line.

If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

Note: The bottom 6 bits are set to 0, forced by architecture.

Prototype

```
void Xil_DCacheFlushLine(INTPTR adr);
```

Parameters

The following table lists the `Xil_DCacheFlushLine` function arguments.

Table 227: Xil_DCacheFlushLine Arguments

Name	Description
adr	64bit address of the data to be flushed.

Returns

None.

Xil_ICacheEnable

Enable the instruction cache.

Note: None.

Prototype

```
void Xil_ICacheEnable(void);
```

Parameters

The following table lists the `Xil_ICacheEnable` function arguments.

Table 228: Xil_ICacheEnable Arguments

Name	Description
None.	

Returns

None.

Xil_ICacheDisable

Disable the instruction cache.

Note: None.

Prototype

```
void Xil_ICacheDisable(void);
```

Parameters

The following table lists the `Xil_ICacheDisable` function arguments.

Table 229: Xil_ICacheDisable Arguments

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidate

Invalidate the entire instruction cache.

Note: None.

Prototype

```
void Xil_ICacheInvalidate(void);
```

Parameters

The following table lists the `Xil_ICacheInvalidate` function arguments.

Table 230: Xil_ICacheInvalidate Arguments

Name	Description
None.	

Returns

None.

Xil_ICacheInvalidateRange

Invalidate the instruction cache for the given address range.

If the instructions specified by the address range are cached by the instruction cache, the cachelines containing those instructions are invalidated.

Note: None.

Prototype

```
void Xil_ICacheInvalidateRange(INTPTR adr, INTPTR len);
```

Parameters

The following table lists the `Xil_ICacheInvalidateRange` function arguments.

Table 231: Xil_ICacheInvalidateRange Arguments

Name	Description
adr	64bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Xil_ICacheInvalidateLine

Invalidate an instruction cache line.

If the instruction specified by the parameter `adr` is cached by the instruction cache, the cacheline containing that instruction is invalidated.

Note: The bottom 6 bits are set to 0, forced by architecture.

Prototype

```
void Xil_ICacheInvalidateLine(INTPTR adr);
```

Parameters

The following table lists the `Xil_ICacheInvalidateLine` function arguments.

Table 232: Xil_ICacheInvalidateLine Arguments

Name	Description
adr	64bit address of the instruction to be invalidated.

Returns

None.

Xil_ConfigureL1Prefetch

Configure the maximum number of outstanding data prefetches allowed in L1 cache.

Note: This function is implemented only for EL3 privilege level.

Prototype

```
void Xil_ConfigureL1Prefetch(u8 num);
```

Parameters

The following table lists the `Xil_ConfigureL1Prefetch` function arguments.

Table 233: Xil_ConfigureL1Prefetch Arguments

Name	Description
num	maximum number of outstanding data prefetches allowed, valid values are 0-7.

Returns

None.

Cortex A53 64-bit Processor MMU Handling

MMU function equip users to modify default memory attributes of MMU table as per the need.

None.

Note:

Table 234: Quick Function Reference

Type	Name	Arguments
void	Xil_SetTlbAttributes	UINTPTR Addr u64 attrib

Functions

Xil_SetTlbAttributes

brief It sets the memory attributes for a section, in the translation table.

If the address (defined by Addr) is less than 4GB, the memory attribute(attrib) is set for a section of 2MB memory. If the address (defined by Addr) is greater than 4GB, the memory attribute (attrib) is set for a section of 1GB memory.

Note: The MMU and D-cache need not be disabled before changing an translation table attribute.

Prototype

```
void Xil_SetTlbAttributes(UINTPTR Addr, u64 attrib);
```

Parameters

The following table lists the `Xil_SetTlbAttributes` function arguments.

Table 235: `Xil_SetTlbAttributes` Arguments

Name	Description
Addr	64-bit address for which attributes are to be set.
attrib	Attribute for the specified memory region. <code>xil_mmu.h</code> contains commonly used memory attributes definitions which can be utilized for this function.

Returns

None.

Cortex A53 64-bit Mode Time Functions

`xtime_l.h` provides access to the 64-bit physical timer counter.

Table 236: Quick Function Reference

Type	Name	Arguments
void	XTime_StartTimer	None.
void	XTime_SetTime	XTime Xtime_Global
void	XTime_GetTime	XTime * Xtime_Global

Functions

XTime_StartTimer

Start the 64-bit physical timer counter.

Note: The timer is initialized only if it is disabled. If the timer is already running this function does not perform any operation. This API is effective only if BSP is built for EL3. For EL1 Non-secure, it simply exits.

Prototype

```
void XTime_StartTimer(void);
```

Parameters

The following table lists the `XTime_StartTimer` function arguments.

Table 237: XTime_StartTimer Arguments

Name	Description
None.	

Returns

None.

XTime_SetTime

Timer of A53 runs continuously and the time can not be set as desired.

This API doesn't contain anything. It is defined to have uniformity across platforms.

Note: None.

Prototype

```
void XTime_SetTime(XTime Xtime_Global);
```

Parameters

The following table lists the XTime_SetTime function arguments.

Table 238: XTime_SetTime Arguments

Name	Description
Xtime_Global	64bit value to be written to the physical timer counter register. Since API does not do anything, the value is not utilized.

Returns

None.

XTime_GetTime

Get the time from the physical timer counter register.

Note: None.

Prototype

```
void XTime_GetTime(XTime *Xtime_Global);
```

Parameters

The following table lists the XTime_GetTime function arguments.

Table 239: XTime_GetTime Arguments

Name	Description
Xtime_Global	Pointer to the 64-bit location to be updated with the current value of physical timer counter register.

Returns

None.

Cortex A53 64-bit Processor Specific Include Files

The `xpseudo_asm.h` includes `xreg_cortexa53.h` and `xpseudo_asm_gcc.h`. The `xreg_cortexa53.h` file contains definitions for inline assembler code. It provides inline definitions for Cortex A53 GPRs, SPRs and floating point registers.

The `xpseudo_asm_gcc.h` contains the definitions for the most often used inline assembler instructions, available as macros. These can be very useful for tasks such as setting or getting special purpose registers, synchronization, or cache manipulation etc. These inline assembler instructions can be used from drivers and user applications written in C.

LwIP 2.1.1 Library

Introduction

The lwIP is an open source TCP/IP protocol suite available under the BSD license. The lwIP is a standalone stack; there are no operating systems dependencies, although it can be used along with operating systems. The lwIP provides two APIs for use by applications:

- RAW API: Provides access to the core lwIP stack.
- Socket API: Provides a BSD sockets style interface to the stack.

The lwip211_v1.2 is built on the open source lwIP library version 2.1.1. The lwip211 library provides adapters for the Ethernetlite (axi_ethernetlite), the TEMAC (axi_ethernet), and the Gigabit Ethernet controller and MAC (GigE) cores. The library can run on MicroBlaze™, ARM Cortex-A9, ARM Cortex-A53, and ARM Cortex-R5 processors. The Ethernetlite and TEMAC cores apply for MicroBlaze systems. The Gigabit Ethernet controller and MAC (GigE) core is applicable only for ARM Cortex-A9 system (Zynq-7000 processor devices) and ARM Cortex-A53 & ARM Cortex-R5 system (Zynq UltraScale+ MPSoC).

Features

The lwIP provides support for the following protocols:

- Internet Protocol (IP)
- Internet Control Message Protocol (ICMP)
- User Datagram Protocol (UDP)
- TCP (Transmission Control Protocol (TCP))
- Address Resolution Protocol (ARP)
- Dynamic Host Configuration Protocol (DHCP)
- Internet Group Message Protocol (IGMP)

References

- FreeRTOS: http://www.freertos.org/Interactive_Frames/Open_Frames.html?http://interactive.freertos.org/forums
- lwIP wiki: <http://lwip.scribblewiki.com>
- Xilinx® lwIP designs and application examples: http://www.xilinx.com/support/documentation/application_notes/xapp1026.pdf
- lwIP examples using RAW and Socket APIs: <http://savannah.nongnu.org/projects/lwip/>
- FreeRTOS Port for Zynq is available for download from the [FreeRTOS][freertos] website

Using lwIP

Overview

The following are the key steps to use lwIP for networking:

1. Creating a hardware system containing the processor, ethernet core, and a timer. The timer and ethernet interrupts must be connected to the processor using an interrupt controller.
2. Configuring lwip211_v1.2 to be a part of the software platform. For operating with lwIP socket API, the Xilkernel library or FreeRTOS BSP is a prerequisite. See the Note below.

Note: The Xilkernel library is available only for MicroBlaze systems. For Cortex-A9 based systems (Zynq) and Cortex-A53 or Cortex-R5 based systems (Zynq® UltraScale™+ MPSoC), there is no support for Xilkernel. Instead, use FreeRTOS. A FreeRTOS BSP is available for Zynq and Zynq UltraScale+ MPSoC systems and must be included for using lwIP socket API. The FreeRTOS BSP for Zynq and Zynq UltraScale+ MPSoC is available for download from the the [FreeRTOS][http://www.freertos.org/Interactive_Frames/Open_Frames.html?http://interactive.freertos.org/forums] website.

Setting up the Hardware System

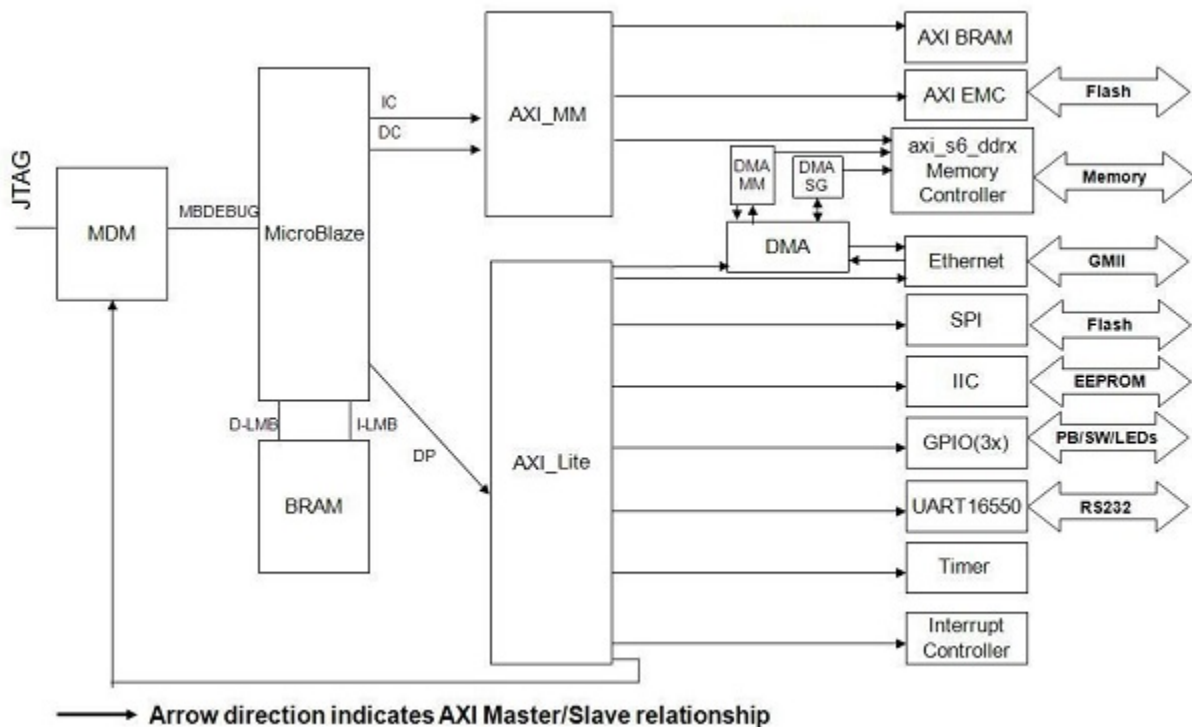
This section describes the hardware configurations supported by lwIP. The key components of the hardware system include:

- Processor: Either a MicroBlaze or a Cortex-A9 or a Cortex-A53 or a Cortex-R5 processor. The Cortex-A9 processor applies to Zynq systems. The Cortex-A53 and Cortex-R5 processors apply to Zynq UltraScale+ MPSoC systems.
- MAC: LwIP supports axi_ethernetlite, axi_ethernet, and Gigabit Ethernet controller and MAC (GigE) cores.
- Timer: to maintain TCP timers, lwIP raw API based applications require that certain functions are called at periodic intervals by the application. An application can do this by registering an interrupt handler with a timer.

- DMA: For axi_ethernet based systems, the axi_ethernet cores can be configured with a soft DMA engine (AXI DMA and MCDMA) or a FIFO interface. For GigE-based Zynq and Zynq UltraScale+ MPSoC systems, there is a built-in DMA and so no extra configuration is needed. Same applies to axi_ethernetlite based systems, which have their built-in buffer management provisions.

The following figure shows a sample system architecture with a Kintex-6 device utilizing the axi_ethernet core with DMA.

Figure 79: System Architecture using axi_ethernet core with DMA



Setting up the Software System

To use lwIP in a software application, you must first compile the lwIP library as a part of the software application.

1. Click File > New > Platform Project.
2. Click Specify to create a new Hardware Platform Specification.
3. Provide a new name for the domain in the Project name field if you wish to override the default value.
4. Select the location for the board support project files. To use the default location, as displayed in the Location field, leave the Use default location check box selected. Otherwise, deselect the checkbox and then type or browse to the directory location.

5. From the Hardware Platform drop-down choose the appropriate platform for your application or click the New button to browse to an existing Hardware Platform.
6. Select the target CPU from the drop-down list.
7. From the Board Support Package OS list box, select the type of board support package to create. A description of the platform types displays in the box below the drop-down list.
8. Click Finish. The wizard creates a new software platform and displays it in the Vitis Navigator pane.
9. Select Project > Build Automatically to automatically build the board support package. The Board Support Package Settings dialog box opens. Here you can customize the settings for the domain.
10. Click OK to accept the settings, build the platform, and close the dialog box.
11. From the Explorer, double-click platform.spr file and select the appropriate domain/board support package. The overview page opens.
12. In the overview page, click Modify BSP Settings.
13. Using the Board Support Package Settings page, you can select the OS Version and which of the Supported Libraries are to be enabled in this domain/BSP.
14. Select the lwip211 library from the list of Supported Libraries.
15. Expand the Overview tree and select lwip211. The configuration options for the lwip211 library are listed.
16. Configure the lwip211 library and click OK.

Configuring lwIP Options

The lwIP library provides configurable parameters. There are two major categories of configurable options:

- Xilinx Adapter to lwIP options: These control the settings used by Xilinx adapters for the ethernet cores.
- Base lwIP options: These options are part of lwIP library itself, and include parameters for TCP, UDP, IP and other protocols supported by lwIP. The following sections describe the available lwIP configurable options.

Customizing lwIP API Mode

The lwip211_v1.3 supports both raw API and socket API:

- The raw API is customized for high performance and lower memory overhead. The limitation of raw API is that it is callback-based, and consequently does not provide portability to other TCP stacks.

- The socket API provides a BSD socket-style interface and is very portable; however, this mode is not as efficient as raw API mode in performance and memory requirements. The lwip211_v1.3 also provides the ability to set the priority on TCP/IP and other lwIP application threads.

The following table describes the lwIP library API mode options.

Attribute	Description	Type	Default
api_mode {RAW_API SOCKET_API}	The lwIP library mode of operation	enum	RAW_API
socket_mode_thread_prio	Priority of lwIP TCP/IP thread and all lwIP application threads. This setting applies only when Xilkernel is used in priority mode. It is recommended that all threads using lwIP run at the same priority level. For GigE based Zynq-7000 and Zynq UltraScale+ MPSoC systems using FreeRTOS, appropriate priority should be set. The default priority of 1 will not give the expected behaviour. For FreeRTOS (Zynq-7000 and Zynq UltraScale+ MPSoC systems), all internal lwIP tasks (except the main TCP/IP task) are created with the priority level set for this attribute. The TCP/IP task is given a higher priority than other tasks for improved performance. The typical TCP/IP task priority is 1 more than the priority set for this attribute for FreeRTOS.	integer	1
use_axieth_on_zynq	In the event that the AxiEthernet soft IP is used on a Zynq-7000 device or a Zynq UltraScale+ MPSoC device. This option ensures that the GigE on the Zynq-7000 PS (EmacPs) is not enabled and the device uses the AxiEthernet soft IP for Ethernet traffic. The existing Xilinx-provided lwIP adapters are not tested for multiple MACs. Multiple Axi Ethernet's are not supported on Zynq UltraScale+ MPSoC devices.	integer	0 = Use Zynq-7000 PS-based or ZynMP PS-based GigE controller 1= User AxiEthernet

Configuring Xilinx Adapter Options

The Xilinx adapters for EMAC/GigE cores are configurable.

Ethernetlite Adapter Options

The following table describes the configuration parameters for the axi_ethernetlite adapter.

Attribute	Description	Type	Default
sw_rx_fifo_size	Software Buffer Size in bytes of the receive data between EMAC and processor	integer	8192
sw_tx_fifo_size	Software Buffer Size in bytes of the transmit data between processor and EMAC	integer	8192

TEMAC Adapter Options

The following table describes the configuration parameters for the axi_ethernet and GigE adapters.

Attribute	Type	Description
n_tx_descriptors	integer	Number of Tx descriptors to be used. For high performance systems there might be a need to use a higher value. Default is 64.
n_rx_descriptors	integer	Number of Rx descriptors to be used. For high performance systems there might be a need to use a higher value. Typical values are 128 and 256. Default is 64.
n_tx_coalesce	integer	Setting for Tx interrupt coalescing. Default is 1.
n_rx_coalesce	integer	Setting for Rx interrupt coalescing. Default is 1.
tcp_rx_checksum_offload	boolean	Offload TCP Receive checksum calculation (hardware support required). For GigE in Zynq and Zynq UltraScale+ MPSoC, the TCP receive checksum offloading is always present, so this attribute does not apply. Default is false.
tcp_tx_checksum_offload	boolean	Offload TCP Transmit checksum calculation (hardware support required). For GigE cores (Zynq and Zynq UltraScale+ MPSoC), the TCP transmit checksum offloading is always present, so this attribute does not apply. Default is false.
tcp_ip_rx_checksum_ofload	boolean	Offload TCP and IP Receive checksum calculation (hardware support required). Applicable only for AXI systems. For GigE in Zynq and Zynq UltraScale+ MPSoC devices, the TCP and IP receive checksum offloading is always present, so this attribute does not apply. Default is false.

Attribute	Type	Description
tcp_ip_tx_checksum_offload	boolean	Offload TCP and IP Transmit checksum calculation (hardware support required). Applicable only for AXI systems. For GigE in Zynq and Zynq UltraScale+ MPSoC devices, the TCP and IP transmit checksum offloading is always present, so this attribute does not apply. Default is false.
phy_link_speed	CONFIG_LINKSPEED_AUTODETECT	Link speed as auto-negotiated by the PHY. LwIP configures the TEMAC/GigE for this speed setting. This setting must be correct for the TEMAC/GigE to transmit or receive packets. The CONFIG_LINKSPEED_AUTODETECT setting attempts to detect the correct link speed by reading the PHY registers; however, this is PHY dependent, and has been tested with the Marvell and TI PHYs present on Xilinx development boards. For other PHYs, select the correct speed. Default is enum.
temac_use_jumbo_frames_experimental	boolean	Use TEMAC jumbo frames (with a size up to 9k bytes). If this option is selected, jumbo frames are allowed to be transmitted and received by the TEMAC. For GigE in Zynq there is no support for jumbo frames, so this attribute does not apply. Default is false.

Configuring Memory Options

The LwIP stack provides different kinds of memories. Similarly, when the application uses socket mode, different memory options are used. All the configurable memory options are provided as a separate category. Default values work well unless application tuning is required. The following table describes the memory parameter options.

Attribute	Default	Type	Description
mem_size	131072	Integer	Total size of the heap memory available, measured in bytes. For applications which use a lot of memory from heap (using C library malloc or LwIP routine mem_malloc or pbuf_alloc with PBUF_RAM option), this number should be made higher as per the requirements.
memp_n_pbuf	16	Integer	The number of memp struct pbufs. If the application sends a lot of data out of ROM (or other static memory), this should be set high.
memp_n_udp_pcb	4	Integer	The number of UDP protocol control blocks. One per active UDP connection.

Attribute	Default	Type	Description
memp_n_tcp_pcb	32	Integer	The number of simultaneously active TCP connections.
memp_n_tcp_pcb_listen	8	Integer	The number of listening TC connections.
memp_n_tcp_seg	256	Integer	The number of simultaneously queued TCP segments.
memp_n_sys_timeout	8	Integer	Number of simultaneously active timeouts.
memp_num_netbuf	8	Integer	Number of allowed structure instances of type netbufs. Applicable only in socket mode.
memp_num_netconn	16	Integer	Number of allowed structure instances of type netconns. Applicable only in socket mode.
memp_num_api_msg	16	Integer	Number of allowed structure instances of type api_msg. Applicable only in socket mode.
memp_num_tcpip_msg	64	Integer	Number of TCPIP msg structures (socket mode only).

Note: Because Sockets Mode support uses Xilkernel services, the number of semaphores chosen in the Xilkernel configuration must take the value set for the `memp_num_netbuf` parameter into account. For FreeRTOS BSP there is no setting for the maximum number of semaphores. For FreeRTOS, you can create semaphores as long as memory is available.

Configuring Packet Buffer (Pbuf) Memory Options

Packet buffers (Pbufs) carry packets across various layers of the TCP/IP stack. The following are the pbuf memory options provided by the lwIP stack. Default values work well unless application tuning is required. The following table describes the parameters for the Pbuf memory options.

Attribute	Default	Type	Description
pbuf_pool_size	256	Integer	Number of buffers in pbuf pool. For high performance systems, you might consider increasing the pbuf pool size to a higher value, such as 512.
pbuf_pool_bufsize	1700	Integer	Size of each pbuf in pbuf pool. For systems that support jumbo frames, you might consider using a pbuf pool buffer size that is more than the maximum jumbo frame size.

Attribute	Default	Type	Description
pbuf_link_hlen	16	Integer	Number of bytes that should be allocated for a link level header.

Configuring ARP Options

The following table describes the parameters for the ARP options. Default values work well unless application tuning is required.

Attribute	Default	Type	Description
arp_table_size	10	Integer	Number of active hardware address IP address pairs cached.
arp_queueing	1	Integer	If enabled outgoing packets are queued during hardware address resolution. This attribute can have two values: 0 or 1.

Configuring IP Options

The following table describes the IP parameter options. Default values work well unless application tuning is required.

Attribute	Default	Type	Description
ip_forward	0	Integer	Set to 1 for enabling ability to forward IP packets across network interfaces. If running lwIP on a single network interface, set to 0. This attribute can have two values: 0 or 1.
ip_options	0	Integer	When set to 1, IP options are allowed (but not parsed). When set to 0, all packets with IP options are dropped. This attribute can have two values: 0 or 1.
ip_reassembly	1	Integer	Reassemble incoming fragmented IP packets.
ip_frag	1	Integer	Fragment outgoing IP packets if their size exceeds MTU.
ip_reass_max_pbufs	128	Integer	Reassembly pbuf queue length.
ip_frag_max_mtu	1500	Integer	Assumed max MTU on any interface for IP fragmented buffer.
ip_default_ttl	255	Integer	Global default TTL used by transport layers.

Configuring ICMP Options

The following table describes the parameter for ICMP protocol option. Default values work well unless application tuning is required.

For GigE cores (for Zynq and Zynq MPSoC) there is no support for ICMP in the hardware.

Attribute	Default	Type	Description
icmp_ttl	255	Integer	ICMP TTL value.

Configuring IGMP Options

The IGMP protocol is supported by lwIP stack. When set true, the following option enables the IGMP protocol.

Attribute	Default	Type	Description
imgp_options	false	Boolean	Specify whether IGMP is required.

Configuring UDP Options

The following table describes UDP protocol options. Default values work well unless application tuning is required.

Attribute	Default	Type	Description
lwip_udp	true	Boolean	Specify whether UDP is required.
udp_ttl	255	Integer	UDP TTL value.

Configuring TCP Options

The following table describes the TCP protocol options. Default values work well unless application tuning is required.

Attribute	Default	Type	Description
lwip_tcp	true	Boolean	Require TCP.
tcp_ttl	255	Integer	TCP TTL value.
tcp_wnd	2048	Integer	TCP Window size in bytes.
tcp_maxrtx	12	Integer	TCP Maximum retransmission value.
tcp_synmaxrtx	4	Integer	TCP Maximum SYN retransmission value.
tcp_queue_ooseq	1	Integer	Accept TCP queue segments out of order. Set to 0 if your device is low on memory.

Attribute	Default	Type	Description
tcp_mss	1460	Integer	TCP Maximum segment size.
tcp_snd_buf	8192	Integer	TCP sender buffer space in bytes.

Configuring DHCP Options

The DHCP protocol is supported by lwIP stack. The following table describes DHCP protocol options. Default values work well unless application tuning is required.

Attribute	Default	Type	Description
lwip_dhcp	false	Boolean	Specify whether DHCP is required.
dhcp_does_arp_check	false	Boolean	Specify whether ARP checks on offered addresses.

Configuring the Stats Option

lwIP stack has been written to collect statistics, such as the number of connections used; amount of memory used; and number of semaphores used, for the application. The library provides the stats_display() API to dump out the statistics relevant to the context in which the call is used. The stats option can be turned on to enable the statistics information to be collected and displayed when the stats_display API is called from user code. Use the following option to enable collecting the stats information for the application.

Attribute	Description	Type	Default
lwip_stats	Turn on lwIP Statistics	int	0

Configuring the Debug Option

lwIP provides debug information. The following table lists all the available options.

Attribute	Default	Type	Description
lwip_debug	false	Boolean	Turn on/off lwIP debugging.
ip_debug	false	Boolean	Turn on/off IP layer debugging.
tcp_debug	false	Boolean	Turn on/off TCP layer debugging.
udp_debug	false	Boolean	Turn on/off UDP layer debugging.
icmp_debug	false	Boolean	Turn on/off ICMP protocol debugging.
igmp_debug	false	Boolean	Turn on/off IGMP protocol debugging.
netif_debug	false	Boolean	Turn on/off network interface layer debugging.

Attribute	Default	Type	Description
sys_debug	false	Boolean	Turn on/off sys arch layer debugging.
pbuf_debug	false	Boolean	Turn on/off pbuf layer debugging

LwIP Library APIs

The lwIP library provides two different APIs: RAW API and Socket API.

Raw API

The Raw API is callback based. Applications obtain access directly into the TCP stack and vice-versa. As a result, there is no extra socket layer, and using the Raw API provides excellent performance at the price of compatibility with other TCP stacks.

Xilinx Adapter Requirements when using the RAW API

In addition to the lwIP RAW API, the Xilinx adapters provide the `xemacif_input` utility function for receiving packets. This function must be called at frequent intervals to move the received packets from the interrupt handlers to the lwIP stack. Depending on the type of packet received, lwIP then calls registered application callbacks. The `<Vitis_install_path>/sw/ThirdParty/sw_services/lwip211/src/lwip-2.1.1/doc/rawapi.txt` file describes the lwIP Raw API.

LwIP Performance

The following table provides the maximum TCP throughput achievable by FPGA, CPU, EMAC, and system frequency in RAW modes. Applications requiring high performance should use the RAW API.

FPGA	CPU	EMAC	System Frequency	Max TCP Throughput in RAW Mode (Mbps)
Virtex	MicroBlaze	axi-ethernet	100 MHz	RX Side: 182 TX Side: 100
Virtex	MicroBlaze	xps-ll-temac	100 MHz	RX Side: 178 TX Side: 100
Virtex	MicroBlaze	xps-ethernetlite	100 MHz	RX Side: 50 TX Side: 38

RAW API Example

Applications using the RAW API are single threaded. The following pseudo-code illustrates a typical RAW mode program structure.

```
int main()
{
    struct netif *netif, server_netif;
    ip_addr_t ipaddr, netmask, gw;

    unsigned char mac_ethernet_address[] =
        {0x00, 0x0a, 0x35, 0x00, 0x01, 0x02};

    lwip_init();

    if (!xemac_add(netif, &ipaddr, &netmask,
        &gw, mac_ethernet_address,
        EMAC_BASEADDR)) {
        printf("Error adding N/W interface\n\r");
        return -1;
    }
    netif_set_default(netif);

    platform_enable_interrupts();

    netif_set_up(netif);

    start_application();

    while (1) {
        xemacif_input(netif);

        transfer_data();
    }
}
```

Socket API

The lwIP socket API provides a BSD socket-style API to programs. This API provides an execution model that is a blocking, open-read-write-close paradigm.

Xilinx Adapter Requirements when using the Socket API

Applications using the Socket API with Xilinx adapters need to spawn a separate thread called `xemacif_input_thread`. This thread takes care of moving received packets from the interrupt handlers to the `tcpip_thread` of the lwIP. Application threads that use lwIP must be created using the lwIP `sys_thread_new` API. Internally, this function makes use of the appropriate thread or task creation routines provided by XilKernel or FreeRTOS.

Xilkernel/FreeRTOS scheduling policy when using the Socket API

LwIP in socket mode requires the use of the Xilkernel or FreeRTOS, which provides two policies for thread scheduling: round-robin and priority based. There are no special requirements when round-robin scheduling policy is used because all threads or tasks with same priority receive the same time quanta. This quanta is fixed by the RTOS (Xilkernel or FreeRTOS) being used. With priority scheduling, care must be taken to ensure that lwIP threads or tasks are not starved. For Xilkernel, lwIP internally launches all threads at the priority level specified in `socket_mode_thread_prio`. For FreeRTOS, lwIP internally launches all tasks except the main TCP/IP task at the priority specified in `socket_mode_thread_prio`. The TCP/IP task in FreeRTOS is launched with a higher priority (one more than priority set in `socket_mode_thread_prio`). In addition, application threads must launch `xemacif_input_thread`. The priorities of both `xemacif_input_thread`, and the lwIP internal threads (`socket_mode_thread_prio`) must be high enough in relation to the other application threads so that they are not starved.

Socket API Example

XilKernel-based applications in socket mode can specify a static list of threads that Xilkernel spawns on startup in the Xilkernel Software Platform Settings dialog box. Assuming that `main_thread()` is a thread specified to be launched by Xilkernel, control reaches this first thread from application `main` after the Xilkernel schedule is started. In `main_thread`, one more thread (`network_thread`) is created to initialize the MAC layer. For FreeRTOS (Zynq and Zynq UltraScale+ MPSoC processor systems) based applications, once the control reaches application `main` routine, a task (can be termed as `main_thread`) with an entry point function as `main_thread()` is created before starting the scheduler. After the FreeRTOS scheduler starts, the control reaches `main_thread()`, where the lwIP internal initialization happens. The application then creates one more thread (`network_thread`) to initialize the MAC layer. The following pseudo-code illustrates a typical socket mode program structure.

```
void network_thread(void *p)
{
    struct netif *netif;
    ip_addr_t ipaddr, netmask, gw;

    unsigned char mac_ethernet_address[] =
        {0x00, 0x0a, 0x35, 0x00, 0x01, 0x02};

    netif = &server_netif;

    IP4_ADDR(&ipaddr, 192, 168, 1, 10);
    IP4_ADDR(&netmask, 255, 255, 255, 0);
    IP4_ADDR(&gw, 192, 168, 1, 1);

    if (!xemac_add(netif, &ipaddr, &netmask,
                  &gw, mac_ethernet_address,
                  EMAC_BASEADDR)) {
        printf("Error adding N/W interface\n\r");
    }
}
```

```

        return;
    }
    netif_set_default(netif);

    netif_set_up(netif);

    sys_thread_new("xemacif_input_thread", xemacif_input_thread,
                  netif,
                  THREAD_STACKSIZE, DEFAULT_THREAD_PRIO);

    sys_thread_new("httpd" web_application_thread, 0,
                  THREAD_STACKSIZE DEFAULT_THREAD_PRIO);
}

int main_thread()
{
    lwip_init();

    sys_thread_new("network_thread" network_thread, NULL,
                  THREAD_STACKSIZE DEFAULT_THREAD_PRIO);

    return 0;
}

```

Using the Xilinx Adapter Helper Functions

The Xilinx adapters provide the following helper functions to simplify the use of the lwIP APIs.

Table 240: Quick Function Reference

Type	Name	Arguments
void	xemacif_input_thread	void
struct netif *	xemac_add	void
void	lwip_init	void
int	xemacif_input	void
void	xemacpsif_resetrx_on_no_rxdata	void

Functions

xemacif_input_thread

In the socket mode, the application thread must launch a separate thread to receive the input packets. This performs the same work as the RAW mode function, `xemacif_input()`, except that it resides in its own separate thread; consequently, any lwIP socket mode application is required to have code similar to the following in its main thread:

Note: For Socket mode only.

```
sys_thread_new("xemacif_input_thread",
               xemacif_input_thread
               , netif, THREAD_STACK_SIZE, DEFAULT_THREAD_PRIO);
```

The application can then continue launching separate threads for doing application specific tasks. The `xemacif_input_thread()` receives data processed by the interrupt handlers, and passes them to the lwIP `tcpip_thread`.

Prototype

```
void xemacif_input_thread(struct netif *netif);
```

Returns

xemac_add

The `xemac_add()` function provides a unified interface to add any Xilinx EMAC IP as well as GigE core. This function is a wrapper around the lwIP `netif_add` function that initializes the network interface 'netif' given its IP address `ipaddr`, netmask, the IP address of the gateway, `gw`, the 6 byte ethernet address `mac_ethernet_address`, and the base address, `mac_baseaddr`, of the `axi_ethernetlite` or `axi_ethernet` MAC core.

Prototype

```
struct netif * xemac_add(struct netif *netif, ip_addr_t *ipaddr, ip_addr_t
                        *netmask, ip_addr_t *gw, unsigned char *mac_ethernet_address, unsigned
                        mac_baseaddr);
```

lwip_init

Initialize all modules. Use this in NO_SYS mode. Use `tcpip_init()` otherwise.

This function provides a single initialization function for the lwIP data structures. This replaces specific calls to initialize stats, system, memory, pbufs, ARP, IP, UDP, and TCP layers.

Prototype

```
void lwip_init(void);
```

xemacif_input

The Xilinx lwIP adapters work in interrupt mode. The receive interrupt handlers move the packet data from the EMAC/GigE and store them in a queue. The `xemacif_input()` function takes those packets from the queue, and passes them to lwIP; consequently, this function is required for lwIP operation in RAW mode. The following is a sample lwIP application in RAW mode.

Note: For RAW mode only.

```
while (1) {

    xemacif_input
    (netif);

}
```

Note: The program is notified of the received data through callbacks.

Prototype

```
int xemacif_input(struct netif *netif);
```

Returns

xemacpsif_resetrx_on_no_rxddata

There is an errata on the GigE controller that is related to the Rx path. The errata describes conditions whereby the Rx path of GigE becomes completely unresponsive with heavy Rx traffic of small sized packets. The condition occurrence is rare; however a software reset of the Rx logic in the controller is required when such a condition occurs. This API must be called periodically (approximately every 100 milliseconds using a timer or thread) from user applications to ensure that the Rx path never becomes unresponsive for more than 100 milliseconds.

Note: Used in both Raw and Socket mode and applicable only for the Zynq-7000 and Zynq MPSoC processors and the GigE controller

Prototype

```
void xemacpsif_resetrx_on_no_rxddata(struct netif *netif);
```

Returns

XiIsf Library v5.15

Overview

The LibXil Isf library:

- Allows you to Write, Read, and Erase the Serial Flash.
- Allows protection of the data stored in the Serial Flash from unwarranted modification by enabling the Sector Protection feature.
- Supports multiple instances of Serial Flash at a time, provided they are of the same device family (Atmel, Intel, STM, Winbond, SST, or Spansion) as the device family is selected at compile time.
- Allows your application to perform Control operations on Intel, STM, Winbond, SST, and Spansion Serial Flash.
- Requires the underlying hardware platform to contain the `axi_quad_spi`, `ps7_spi`, `ps7_qspi`, `psu_qspi`, `psv_osp`, or `psu_spi` device for accessing the Serial Flash.
- Uses the Xilinx SPI interface drivers in interrupt-driven mode or polled mode for communicating with the Serial Flash. In interrupt mode, the user application must acknowledge any associated interrupts from the Interrupt Controller.

Additional information

- In interrupt mode, the application is required to register a callback to the library and the library registers an internal status handler to the selected interface driver.
- When your application requests a library operation, it is initiated and control is given back to the application. The library tracks the status of the interface transfers, and notifies the user application upon completion of the selected library operation.
- Added support in the library for SPI PS and QSPI PS. You must select one of the interfaces at compile time.
- Added support for QSPIPSU and SPIPS flash interface on Zynq UltraScale+ MPSoC.
- Added support for OSPIPSV flash interface
- When your application requests selection of QSPIPS interface during compilation, the QSPI PS or QSPI PSU interface, based on the hardware platform, are selected.

- When the SPIPS interface is selected during compilation, the SPI PS or the SPI PSU interface is selected.
- When the OSPI interface is selected during compilation, the OSPIPSV interface is selected.

Supported Devices

The table below lists the supported Xilinx in-system and external serial flash memories.

Device Series	Manufacturer
AT45DB011D AT45DB021D AT45DB041D AT45DB081D AT45DB161D AT45DB321D AT45DB642D	Atmel
W25Q16 W25Q32 W25Q64 W25Q80 W25Q128 W25X10 W25X20 W25X40 W25X80 W25X16 W25X32 W25X64	Winbond
S25FL004 S25FL008 S25FL016 S25FL032 S25FL064 S25FL128 S25FL129 S25FL256 S25FL512 S70FL01G	Spansion
SST25WF080	SST
N25Q032 N25Q064 N25Q128 N25Q256 N25Q512 N25Q00AA MT25Q01 MT25Q02 MT25Q512 MT25QL02G MT25QU02G MT35XU512ABA	Micron
MX66L1G45G MX66U1G45G	Macronix
IS25WP256D IS25LP256D IS25LWP512M IS25LP512M IS25WP064A IS25LP064A IS25WP032D IS25LP032D IS25WP016D IS25LP016D IS25WP080D IS25LP080D IS25LP128F IS25WP128F	ISSI

Note: Intel, STM, and Numonyx serial flash devices are now a part of Serial Flash devices provided by Micron.

References

- Spartan-3AN FPGA In-System Flash User Guide (UG333):http://www.xilinx.com/support/documentation/user_guides/ug333.pdf
- Winbond Serial Flash Page:http://www.winbond.com/hq/product/code-storage-flash-memory/serial-nor-flash/?__locale=en
- Intel (Numonyx) S33 Serial Flash Memory, SST SST25WF080, Micron N25Q flash family : <https://www.micron.com/products/nor-flash/serial-nor-flash>

XilIsf Library API

This section provides a linked summary and detailed descriptions of the XilIsf library APIs.

Table 241: Quick Function Reference

Type	Name	Arguments
int	XIsf_Initialize	XIsf * InstancePtr XIsf_Iface * SpiInstPtr u8 SlaveSelect u8 * WritePtr
int	XIsf_GetStatus	XIsf * InstancePtr u8 * ReadPtr
int	XIsf_GetStatusReg2	XIsf * InstancePtr u8 * ReadPtr
int	XIsf_GetDeviceInfo	XIsf * InstancePtr u8 * ReadPtr
int	XIsf_Transfer	void
u32	GetRealAddr	XIsf_Iface * QspiPtr u32 Address
int	XIsf_Write	XIsf * InstancePtr XIsf_WriteOperation Operation void * OpParamPtr
int	XIsf_Read	XIsf * InstancePtr XIsf_ReadOperation Operation void * OpParamPtr
int	XIsf_Erase	XIsf * InstancePtr XIsf_EraseOperation Operation u32 Address
int	XIsf_SectorProtect	XIsf * InstancePtr XIsf_SpOperation Operation u8 * BufferPtr
int	XIsf_Ioctl	XIsf * InstancePtr XIsf_IoctlOperation Operation
int	XIsf_WriteEnable	XIsf * InstancePtr u8 WriteEnable

Table 241: Quick Function Reference (cont'd)

Type	Name	Arguments
void	XIsf_RegisterInterface	XIsf * InstancePtr
int	XIsf_SetSpiConfiguration	XIsf * InstancePtr XIsf_Iface * SpiInstPtr u32 Options u8 PreScaler
void	XIsf_SetStatusHandler	XIsf * InstancePtr XIsf_Iface * XIfaceInstancePtr XIsf_StatusHandler XilIsf_Handler
void	XIsf_IfaceHandler	void * CallBackRef u32 StatusEvent unsigned int ByteCount

Functions

XIsf_Initialize

This API when called initializes the SPI interface with default settings.

With custom settings, user should call [XIsf_SetSpiConfiguration\(\)](#) and then call this API. The geometry of the underlying Serial Flash is determined by reading the Joint Electron Device Engineering Council (JEDEC) Device Information and the Status Register of the Serial Flash.

Note:

- The [XIsf_Initialize\(\)](#) API is a blocking call (for both polled and interrupt modes of the Spi driver). It reads the JEDEC information of the device and waits till the transfer is complete before checking if the information is valid.
- This library can support multiple instances of Serial Flash at a time, provided they are of the same device family (either Atmel, Intel or STM, Winbond or Spansion) as the device family is selected at compile time.

Prototype

```
int XIsf_Initialize(XIsf *InstancePtr, XIsf_Iface *SpiInstPtr, u8
SlaveSelect, u8 *WritePtr);
```

Parameters

The following table lists the [XIsf_Initialize](#) function arguments.

Table 242: XIsf_Initialize Arguments

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.
XIsf_Iface *	SpiInstPtr	Pointer to XIsf_Iface instance to be worked on.
u8	SlaveSelect	It is a 32-bit mask with a 1 in the bit position of slave being selected. Only one slave can be selected at a time.
u8 *	WritePtr	Pointer to the buffer allocated by the user to be used by the In-system and Serial Flash Library to perform any read/write operations on the Serial Flash device. User applications must pass the address of this buffer for the Library to work. <ul style="list-style-type: none"> • Write operations : <ul style="list-style-type: none"> ◦ The size of this buffer should be equal to the Number of bytes to be written to the Serial Flash + XISF_CMD_MAX_EXTRA_BYTES. ◦ The size of this buffer should be large enough for usage across all the applications that use a common instance of the Serial Flash. ◦ A minimum of one byte and a maximum of ISF_PAGE_SIZE bytes can be written to the Serial Flash, through a single Write operation. • Read operations : <ul style="list-style-type: none"> ◦ The size of this buffer should be equal to XISF_CMD_MAX_EXTRA_BYTES, if the application only reads from the Serial Flash (no write operations).

Returns

- XST_SUCCESS if successful.
- XST_DEVICE_IS_STOPPED if the device must be started before transferring data.
- XST_FAILURE, otherwise.

XIsf_GetStatus

This API reads the Serial Flash Status Register.

Note: The contents of the Status Register is stored at second byte pointed by the ReadPtr.

Prototype

```
int XIsf_GetStatus(XIsf *InstancePtr, u8 *ReadPtr);
```

Parameters

The following table lists the XIsf_GetStatus function arguments.

Table 243: XIsf_GetStatus Arguments

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.
u8 *	ReadPtr	Pointer to the memory where the Status Register content is copied.

Returns

XST_SUCCESS if successful else XST_FAILURE.

XIsf_GetStatusReg2

This API reads the Serial Flash Status Register 2.

Note: The contents of the Status Register 2 is stored at the second byte pointed by the ReadPtr. This operation is available only in Winbond Serial Flash.

Prototype

```
int XIsf_GetStatusReg2(XIsf *InstancePtr, u8 *ReadPtr);
```

Parameters

The following table lists the XIsf_GetStatusReg2 function arguments.

Table 244: XIsf_GetStatusReg2 Arguments

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.
u8 *	ReadPtr	Pointer to the memory where the Status Register content is copied.

Returns

XST_SUCCESS if successful else XST_FAILURE.

XIsf_GetDeviceInfo

This API reads the Joint Electron Device Engineering Council (JEDEC) information of the Serial Flash.

Note: The Device information is stored at the second byte pointed by the ReadPtr.

Prototype

```
int XIsf_GetDeviceInfo(XIsf *InstancePtr, u8 *ReadPtr);
```


Parameters

The following table lists the `XIsf_GetDeviceInfo` function arguments.

Table 245: XIsf_GetDeviceInfo Arguments

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.
u8 *	ReadPtr	Pointer to the buffer where the Device information is copied.

Returns

XST_SUCCESS if successful else XST_FAILURE.

XIsf_Transfer

Prototype

```
int XIsf_Transfer(XIsf *InstancePtr, u8 *WritePtr, u8 *ReadPtr, u32
ByteCount);
```

GetRealAddr

Function to get the real address of flash in case dual parallel and stacked configuration.

Function to get the real address of flash in case dual parallel and stacked configuration.

This functions translates the address based on the type of interconnection. In case of stacked, this function asserts the corresponding slave select.

Note: None.

Prototype

```
u32 GetRealAddr(XIsf_Iface *QspiPtr, u32 Address);
```

Parameters

The following table lists the `GetRealAddr` function arguments.

Table 246: GetRealAddr Arguments

Type	Name	Description
XIsf_Iface *	QspiPtr	is a pointer to XIsf_Iface instance to be worked on.
u32	Address	which is to be accessed (for erase, write or read)

Returns

RealAddr is the translated address - for single it is unchanged for stacked, the lower flash size is subtracted for parallel the address is divided by 2.

XIsf_Write

This API writes the data to the Serial Flash.

Operations

- Normal Write(XISF_WRITE), Dual Input Fast Program (XISF_DUAL_IP_PAGE_WRITE), Dual Input Extended Fast Program(XISF_DUAL_IP_EXT_PAGE_WRITE), Quad Input Fast Program(XISF_QUAD_IP_PAGE_WRITE), Quad Input Extended Fast Program (XISF_QUAD_IP_EXT_PAGE_WRITE):
 - The OpParamPtr must be of type struct XIsf_WriteParam.
 - OpParamPtr->Address is the start address in the Serial Flash.
 - OpParamPtr->WritePtr is a pointer to the data to be written to the Serial Flash.
 - OpParamPtr->NumBytes is the number of bytes to be written to Serial Flash.
 - This operation is supported for Atmel, Intel, STM, Winbond and Spansion Serial Flash.
- Auto Page Write (XISF_AUTO_PAGE_WRITE):
 - The OpParamPtr must be of 32 bit unsigned integer variable.
 - This is the address of page number in the Serial Flash which is to be refreshed.
 - This operation is only supported for Atmel Serial Flash.
- Buffer Write (XISF_BUFFER_WRITE):
 - The OpParamPtr must be of type struct XIsf_BufferToFlashWriteParam.
 - OpParamPtr->BufferNum specifies the internal SRAM Buffer of the Serial Flash. The valid values are XISF_PAGE_BUFFER1 or XISF_PAGE_BUFFER2. XISF_PAGE_BUFFER2 is not valid in case of AT45DB011D Flash as it contains a single buffer.
 - OpParamPtr->WritePtr is a pointer to the data to be written to the Serial Flash SRAM Buffer.
 - OpParamPtr->ByteOffset is byte offset in the buffer from where the data is to be written.
 - OpParamPtr->NumBytes is number of bytes to be written to the Buffer. This operation is supported only for Atmel Serial Flash.
- Buffer To Memory Write With Erase (XISF_BUF_TO_PAGE_WRITE_WITH_ERASE)/ Buffer To Memory Write Without Erase (XISF_BUF_TO_PAGE_WRITE_WITHOUT_ERASE):
 - The OpParamPtr must be of type struct XIsf_BufferToFlashWriteParam.

- OpParamPtr->BufferNum specifies the internal SRAM Buffer of the Serial Flash. The valid values are XISF_PAGE_BUFFER1 or XISF_PAGE_BUFFER2. XISF_PAGE_BUFFER2 is not valid in case of AT45DB011D Flash as it contains a single buffer.
- OpParamPtr->Address is starting address in the Serial Flash memory from where the data is to be written. These operations are only supported for Atmel Serial Flash.
- Write Status Register (XISF_WRITE_STATUS_REG):
 - The OpParamPtr must be of type of 8 bit unsigned integer variable. This is the value to be written to the Status Register.
 - This operation is only supported for Intel, STM Winbond and Spansion Serial Flash.
- Write Status Register2 (XISF_WRITE_STATUS_REG2):
 - The OpParamPtr must be of type (u8 *) and should point to two 8 bit unsigned integer values. This is the value to be written to the 16 bit Status Register. This operation is only supported in Winbond (W25Q) Serial Flash.
- One Time Programmable Area Write(XISF_OTP_WRITE):
 - The OpParamPtr must be of type struct XIsf_WriteParam.
 - OpParamPtr->Address is the address in the SRAM Buffer of the Serial Flash to which the data is to be written.
 - OpParamPtr->WritePtr is a pointer to the data to be written to the Serial Flash.
 - OpParamPtr->NumBytes should be set to 1 when performing OTPWrite operation. This operation is only supported for Intel Serial Flash.

Note:

- Application must fill the structure elements of the third argument and pass its pointer by type casting it with void pointer.
- For Intel, STM, Winbond and Spansion Serial Flash, the user application must call the [XIsf_WriteEnable\(\)](#) API by passing XISF_WRITE_ENABLE as an argument, before calling the [XIsf_Write\(\)](#) API.

Prototype

```
int XIsf_Write(XIsf *InstancePtr, XIsf_WriteOperation Operation, void *OpParamPtr);
```

Parameters

The following table lists the XIsf_Write function arguments.

Table 247: XIsf_Write Arguments

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.

Table 247: XIsf_Write Arguments (cont'd)

Type	Name	Description
XIsf_WriteOperation	Operation	<p>Type of write operation to be performed on the Serial Flash. The different operations are</p> <ul style="list-style-type: none"> • XISF_WRITE: Normal Write • XISF_DUAL_IP_PAGE_WRITE: Dual Input Fast Program • XISF_DUAL_IP_EXT_PAGE_WRITE: Dual Input Extended Fast Program • XISF_QUAD_IP_PAGE_WRITE: Quad Input Fast Program • XISF_QUAD_IP_EXT_PAGE_WRITE: Quad Input Extended Fast Program • XISF_AUTO_PAGE_WRITE: Auto Page Write • XISF_BUFFER_WRITE: Buffer Write • XISF_BUF_TO_PAGE_WRITE_WITH_ERASE: Buffer to Page Transfer with Erase • XISF_BUF_TO_PAGE_WRITE_WITHOUT_ERASE: Buffer to Page Transfer without Erase • XISF_WRITE_STATUS_REG: Status Register Write • XISF_WRITE_STATUS_REG2: 2 byte Status Register Write • XISF_OTP_WRITE: OTP Write.
void *	OpParamPtr	<p>Pointer to a structure variable which contains operational parameters of the specified operation. This parameter type is dependant on value of first argument(Operation). For more details, refer <i>Operations</i> .</p>

Returns

XST_SUCCESS if successful else XST_FAILURE.

XIsf_Read

This API reads the data from the Serial Flash.

Operations

- Normal Read (XISF_READ), Fast Read (XISF_FAST_READ), One Time Programmable Area Read(XISF_OTP_READ), Dual Output Fast Read (XISF_CMD_DUAL_OP_FAST_READ), Dual Input/Output Fast Read (XISF_CMD_DUAL_IO_FAST_READ), Quad Output Fast Read (XISF_CMD_QUAD_OP_FAST_READ) and Quad Input/Output Fast Read (XISF_CMD_QUAD_IO_FAST_READ):
 - The OpParamPtr must be of type struct XIsf_ReadParam.
 - OpParamPtr->Address is start address in the Serial Flash.

- OpParamPtr->ReadPtr is a pointer to the memory where the data read from the Serial Flash is stored.
- OpParamPtr->NumBytes is number of bytes to read.
- OpParamPtr->NumDummyBytes is the number of dummy bytes to be transmitted for the Read command. This parameter is only used in case of Dual and Quad reads.
- Normal Read and Fast Read operations are supported for Atmel, Intel, STM, Winbond and Spansion Serial Flash.
- Dual and quad reads are supported for Winbond (W25QXX), Numonyx(N25QXX) and Spansion (S25FL129) quad flash.
- OTP Read operation is only supported in Intel Serial Flash.
- Page To Buffer Transfer (XISF_PAGE_TO_BUF_TRANS):
 - The OpParamPtr must be of type struct Xisf_FlashToBufTransferParam .
 - OpParamPtr->BufferNum specifies the internal SRAM Buffer of the Serial Flash. The valid values are XISF_PAGE_BUFFER1 or XISF_PAGE_BUFFER2. XISF_PAGE_BUFFER2 is not valid in case of AT45DB011D Flash as it contains a single buffer.
 - OpParamPtr->Address is start address in the Serial Flash. This operation is only supported in Atmel Serial Flash.
- Buffer Read (XISF_BUFFER_READ) and Fast Buffer Read(XISF_FAST_BUFFER_READ):
 - The OpParamPtr must be of type struct Xisf_BufferReadParam.
 - OpParamPtr->BufferNum specifies the internal SRAM Buffer of the Serial Flash. The valid values are XISF_PAGE_BUFFER1 or XISF_PAGE_BUFFER2. XISF_PAGE_BUFFER2 is not valid in case of AT45DB011D Flash as it contains a single buffer.
 - OpParamPtr->ReadPtr is pointer to the memory where data read from the SRAM buffer is to be stored.
 - OpParamPtr->ByteOffset is byte offset in the SRAM buffer from where the first byte is read.
 - OpParamPtr->NumBytes is the number of bytes to be read from the Buffer. These operations are supported only in Atmel Serial Flash.

Note:

- Application must fill the structure elements of the third argument and pass its pointer by type casting it with void pointer.
- The valid data is available from the fourth location pointed to by the ReadPtr for Normal Read and Buffer Read operations.
- The valid data is available from fifth location pointed to by the ReadPtr for Fast Read, Fast Buffer Read and OTP Read operations.
- The valid data is available from the (4 + NumDummyBytes)th location pointed to by ReadPtr for Dual/Quad Read operations.

Prototype

```
int XIsf_Read(XIsf *InstancePtr, XIsf_ReadOperation Operation, void *OpParamPtr);
```

Parameters

The following table lists the `XIsf_Read` function arguments.

Table 248: XIsf_Read Arguments

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.
XIsf_ReadOperation	Operation	Type of the read operation to be performed on the Serial Flash. The different operations are <ul style="list-style-type: none"> XISF_READ: Normal Read XISF_FAST_READ: Fast Read XISF_PAGE_TO_BUF_TRANS: Page to Buffer Transfer XISF_BUFFER_READ: Buffer Read XISF_FAST_BUFFER_READ: Fast Buffer Read XISF_OTP_READ: One Time Programmable Area (OTP) Read XISF_DUAL_OP_FAST_READ: Dual Output Fast Read XISF_DUAL_IO_FAST_READ: Dual Input/Output Fast Read XISF_QUAD_OP_FAST_READ: Quad Output Fast Read XISF_QUAD_IO_FAST_READ: Quad Input/Output Fast Read
void *	OpParamPtr	Pointer to structure variable which contains operational parameter of specified Operation. This parameter type is dependant on the type of Operation to be performed. For more details, refer Operations .

Returns

XST_SUCCESS if successful else XST_FAILURE.

XIsf_Erase

This API erases the contents of the specified memory in the Serial Flash.

Note:

- The erased bytes will read as 0xFF.
- For Intel, STM, Winbond or Spansion Serial Flash the user application must call [XIsf_WriteEnable\(\)](#) API by passing XISF_WRITE_ENABLE as an argument before calling [XIsf_Erase\(\)](#) API.

- Atmel Serial Flash support Page/Block/Sector Erase operations.
- Intel, Winbond, Numonyx (N25QXX) and Spansion Serial Flash support Sector/Block/Bulk Erase operations.
- STM (M25PXX) Serial Flash support Sector/Bulk Erase operations.

Prototype

```
int XIsf_Erase(XIsf *InstancePtr, XIsf_EraseOperation Operation, u32 Address);
```

Parameters

The following table lists the `XIsf_Erase` function arguments.

Table 249: XIsf_Erase Arguments

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.
XIsf_EraseOperation	Operation	Type of Erase operation to be performed on the Serial Flash. The different operations are <ul style="list-style-type: none"> • XISF_PAGE_ERASE: Page Erase • XISF_BLOCK_ERASE: Block Erase • XISF_SECTOR_ERASE: Sector Erase • XISF_BULK_ERASE: Bulk Erase
u32	Address	Address of the Page/Block/Sector to be erased. The address can be either Page address, Block address or Sector address based on the Erase operation to be performed.

Returns

XST_SUCCESS if successful else XST_FAILURE.

XIsf_SectorProtect

This API is used for performing Sector Protect related operations.

Note:

- The SPR content is stored at the fourth location pointed by the BufferPtr when performing XISF_SPR_READ operation.
- For Intel, STM, Winbond and Spansion Serial Flash, the user application must call the `XIsf_WriteEnable()` API by passing XISF_WRITE_ENABLE as an argument, before calling the `XIsf_SectorProtect()` API, for Sector Protect Register Write (XISF_SPR_WRITE) operation.
- Atmel Flash supports all these Sector Protect operations.

- Intel, STM, Winbond and Spansion Flash support only Sector Protect Read and Sector Protect Write operations.

Prototype

```
int XIsf_SectorProtect(XIsf *InstancePtr, XIsf_SpOperation Operation, u8 *BufferPtr);
```

Parameters

The following table lists the `XIsf_SectorProtect` function arguments.

Table 250: XIsf_SectorProtect Arguments

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.
XIsf_SpOperation	Operation	Type of Sector Protect operation to be performed on the Serial Flash. The different operations are <ul style="list-style-type: none"> • XISF_SPR_READ: Read Sector Protection Register • XISF_SPR_WRITE: Write Sector Protection Register • XISF_SPR_ERASE: Erase Sector Protection Register • XISF_SP_ENABLE: Enable Sector Protection • XISF_SP_DISABLE: Disable Sector Protection
u8 *	BufferPtr	Pointer to the memory where the SPR content is read to/written from. This argument can be NULL if the Operation is SprErase, SpEnable and SpDisable.

Returns

- XST_SUCCESS if successful.
- XST_FAILURE if it fails.

XIsf_Ioctl

This API configures and controls the Intel, STM, Winbond and Spansion Serial Flash.

Note:

- Atmel Serial Flash does not support any of these operations.
- Intel Serial Flash support Enter/Release from DPD Mode and Clear Status Register Fail Flags.
- STM, Winbond and Spansion Serial Flash support Enter/Release from DPD Mode.
- Winbond (W25QXX) Serial Flash support Enable High Performance mode.

Prototype

```
int XIsf_Ioctl(XIsf *InstancePtr, XIsf_IoctlOperation Operation);
```

Parameters

The following table lists the `XIsf_Ioctl` function arguments.

Table 251: XIsf_Ioctl Arguments

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.
XIsf_IoctlOperation	Operation	Type of Control operation to be performed on the Serial Flash. The different control operations are <ul style="list-style-type: none"> XISF_RELEASE_DPD: Release from Deep Power Down (DPD) Mode XISF_ENTER_DPD: Enter DPD Mode XISF_CLEAR_SR_FAIL_FLAGS: Clear Status Register Fail Flags

Returns

XST_SUCCESS if successful else XST_FAILURE.

XIsf_WriteEnable

This API Enables/Disables writes to the Intel, STM, Winbond and Spansion Serial Flash.

Note: This API works only for Intel, STM, Winbond and Spansion Serial Flash. If this API is called for Atmel Flash, XST_FAILURE is returned.

Prototype

```
int XIsf_WriteEnable(XIsf *InstancePtr, u8 WriteEnable);
```

Parameters

The following table lists the `XIsf_WriteEnable` function arguments.

Table 252: XIsf_WriteEnable Arguments

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.
u8	WriteEnable	Specifies whether to Enable (XISF_CMD_ENABLE_WRITE) or Disable (XISF_CMD_DISABLE_WRITE) the writes to the Serial Flash.

Returns

XST_SUCCESS if successful else XST_FAILURE.

XIsf_RegisterInterface

This API registers the interface SPI/SPI PS/QSPI PS.

Prototype

```
void XIsf_RegisterInterface(XIsf *InstancePtr);
```

Parameters

The following table lists the `XIsf_RegisterInterface` function arguments.

Table 253: XIsf_RegisterInterface Arguments

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.

Returns

None

XIsf_SetSpiConfiguration

This API sets the configuration of SPI.

This will set the options and clock prescaler (if applicable).

Note: This API can be called before calling `XIsf_Initialize()` to initialize the SPI interface in other than default options mode. PreScaler is only applicable to PS SPI/QSPI.

Prototype

```
int XIsf_SetSpiConfiguration(XIsf *InstancePtr, XIsf_Iface *SpiInstPtr, u32 Options, u8 PreScaler);
```

Parameters

The following table lists the `XIsf_SetSpiConfiguration` function arguments.

Table 254: XIsf_SetSpiConfiguration Arguments

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf instance.

Table 254: XIsf_SetSpiConfiguration Arguments (cont'd)

Type	Name	Description
XIsf_Iface *	SpiInstPtr	Pointer to XIsf_Iface instance to be worked on.
u32	Options	Specified options to be set.
u8	PreScaler	Value of the clock prescaler to set.

Returns

XST_SUCCESS if successful else XST_FAILURE.

XIsf_SetStatusHandler

This API is to set the Status Handler when an interrupt is registered.

Note: None.

Prototype

```
void XIsf_SetStatusHandler(XIsf *InstancePtr, XIsf_Iface
*XIfaceInstancePtr, XIsf_StatusHandler XilIsf_Handler);
```

Parameters

The following table lists the XIsf_SetStatusHandler function arguments.

Table 255: XIsf_SetStatusHandler Arguments

Type	Name	Description
XIsf *	InstancePtr	Pointer to the XIsf Instance.
XIsf_Iface *	XIfaceInstancePtr	Pointer to the XIsf_Iface instance to be worked on.
XIsf_StatusHandler	XilIsf_Handler	Status handler for the application.

Returns

None

XIsf_IfaceHandler

This API is the handler which performs processing for the QSPI driver.

It is called from an interrupt context such that the amount of processing performed should be minimized. It is called when a transfer of QSPI data completes or an error occurs.

This handler provides an example of how to handle QSPI interrupts but is application specific.

Note: None.

Prototype

```
void XIsf_IfaceHandler(void *CallBackRef, u32 StatusEvent, unsigned int
ByteCount);
```

Parameters

The following table lists the `XIsf_IfaceHandler` function arguments.

Table 256: XIsf_IfaceHandler Arguments

Type	Name	Description
void *	CallBackRef	Reference passed to the handler.
u32	StatusEvent	Status of the QSPI .
unsigned int	ByteCount	Number of bytes transferred.

Returns

None

Library Parameters in MSS File

XilIsf Library can be integrated with a system using the following snippet in the Microprocessor Software Specification (MSS) file:

```
BEGIN LIBRARY`
PARAMETER LIBRARY_NAME = xilIsf
PARAMETER LIBRARY_VER = 5.15
PARAMETER serial_flash_family = 1
PARAMETER serial_flash_interface = 1
END
```

The table below describes the `libgen` customization parameters.

Parameter	Default Value	Description
LIBRARY_NAME	xilIsf	Specifies the library name.
LIBRARY_VER	5.15	Specifies the library version.
serial_flash_family	1	Specifies the serial flash family. Supported numerical values are: 1 = Xilinx In-system Flash or Atmel Serial Flash 2 = Intel (Numonyx) S33 Serial Flash 3 = STM (Numonyx) M25PXX/N25QXX Serial Flash 4 = Winbond Serial Flash 5 = Spansion Serial Flash/Micron Serial Flash/Cypress Serial Flash 6 = SST Serial Flash

Parameter	Default Value	Description
Serial_flash_interface	1	Specifies the serial flash interface. Supported numerical values are: 1 = AXI QSPI Interface 2 = SPI PS Interface 3 = QSPI PS Interface or QSPI PSU Interface 4 = OSPIPSV Interface for OSPI

Note: Intel, STM, and Numonyx serial flash devices are now a part of Serial Flash devices provided by Micron.

XilFFS Library v4.3

XilFFS Library API Reference

The Xilinx fat file system (FFS) library consists of a file system and a glue layer. This FAT file system can be used with an interface supported in the glue layer. The file system code is open source and is used as it is. Currently, the Glue layer implementation supports the SD/eMMC interface and a RAM based file system. Application should make use of APIs provided in ff.h. These file system APIs access the driver functions through the glue layer.

The file system supports FAT16, FAT32, and exFAT (optional). The APIs are standard file system APIs. For more information, see the http://elm-chan.org/fsw/ff/00index_e.html.

Note: The XilFFS library uses Revision R0.13b of the generic FAT filesystem module.

Library Files

The table below lists the file system files.

File	Description
ff.c	Implements all the file system APIs
ff.h	File system header
ffconf.h	File system configuration header – File system configurations such as READ_ONLY, MINIMAL, can be set here. This library uses FF_FS_MINIMIZE and FF_FS_TINY and Read/Write (NOT read only)

The table below lists the glue layer files.

File	Description
diskio.c	Glue layer – implements the function used by file system to call the driver APIs
ff.h	File system header
diskio.h	Glue layer header

Selecting a File System with an SD Interface

To select a file system with an SD interface:

1. Click **File** → **New** → **Platform Project**.
2. Click **Specify** to create a new hardware platform specification.
3. Provide a new name for the domain in the **Project name** field if you wish to override the default value.
4. Select the location for the board support project files. To use the default location, as displayed in the **Location** field, leave the **Use default location** check box selected. Otherwise, deselect the checkbox and then type or browse to the directory location.
5. From the **Hardware Platform** drop-down, choose the appropriate platform for your application or click the **New** button to browse to an existing hardware platform.
6. Select the target CPU from the drop-down list.
7. From the **Board Support Package OS** list box, select the type of board support package to create. A description of the platform types displays in the box below the drop-down list.
8. Click **Finish**. The wizard creates a new software platform and displays it in the Vitis Navigator pane.
9. Select **Project** → **Build Automatically** to automatically build the board support package. The **Board Support Package Settings** dialog box opens. Here you can customize the settings for the domain.
10. Click **OK** to accept the settings, build the platform, and close the dialog box.
11. From the Explorer, double-click `platform.spr` file and select the appropriate domain/board support package. The **Overview** page opens.
12. In the overview page, click **Modify BSP Settings**.
13. Using the Board Support Package Settings page, you can select the OS version and which of the supported libraries are to be enabled in this domain/BSP.
14. Select the **xilffs** library from the list of **Supported Libraries**.
15. Expand the **Overview** tree and select **xilffs**. The configuration options for xilffs are listed.
16. Configure the xilffs by setting the `fs_interface = 1` to select the SD/eMMC. This is the default value. Ensure that the SD/eMMC interface is available, prior to selecting the `fs_interface = 1` option.
17. Build the bsp and the application to use the file system with SD/eMMC. SD or eMMC will be recognized by the low level driver.

Selecting a RAM Based File System

To select a RAM based file system:

1. Click **File** → **New** → **Platform Project**.
2. Click **Specify** to create a new hardware platform specification.
3. Provide a new name for the domain in the **Project name** field if you wish to override the default value.
4. Select the location for the board support project files. To use the default location, as displayed in the **Location** field, leave the **Use default location** check box selected. Otherwise, deselect the checkbox and then type or browse to the directory location.
5. From the **Hardware Platform** drop-down, choose the appropriate platform for your application or click the **New** button to browse to an existing hardware platform.
6. Select the target CPU from the drop-down list.
7. From the **Board Support Package OS** list box, select the type of board support package to create. A description of the platform types displays in the box below the drop-down list.
8. Click **Finish**. The wizard creates a new software platform and displays it in the Vitis Navigator pane.
9. Select **Project** → **Build Automatically** to automatically build the board support package. The **Board Support Package Settings** dialog box opens. Here you can customize the settings for the domain.
10. Click **OK** to accept the settings, build the platform, and close the dialog box.
11. From the Explorer, double-click `platform.spr` file and select the appropriate domain/board support package. The **Overview** page opens.
12. In the **Overview** page, click **Modify BSP Settings**.
13. Using the Board Support Package Settings page, you can select the OS version and which of the supported libraries are to be enabled in this domain/BSP.
14. Select the **xilffs** library from the list of **Supported Libraries**.
15. Expand the **Overview** tree and select **xilffs**. The configuration options for xilffs are listed.
16. Configure the xilffs by setting the `fs_interface = 2` to select the RAM.
17. As this project is used by LWIP based application, select **lwip library** and configure according to your requirements. For more information, see the LwIP Library API Reference documentation.
18. Use any lwip application that requires a RAM based file system - TCP/UDP performance test apps or tftp or webserver examples.
19. Build the bsp and the application to use the RAM based file system.

Library Parameters in MSS File

XilFFS Library can be integrated with a system using the following code snippet in the Microprocessor Software Specification (MSS) file:

```

BEGIN LIBRARY
  PARAMETER LIBRARY_NAME = xilffs
  PARAMETER LIBRARY_VER = 4.3
  PARAMETER fs_interface = 1
  PARAMETER read_only = false
  PARAMETER use_lfn = 0
  PARAMETER enable_multi_partition = false
  PARAMETER num_logical_vol = 2
  PARAMETER use_mkfs = true
  PARAMETER use_strfunc = 0
  PARAMETER set_fs_rpath = 0
  PARAMETER enable_exfat = false
  PARAMETER word_access = true
  PARAMETER use_chmod = false
END
    
```

The table below describes the libgen customization parameters.

Parameter	Default Value	Description
LIBRARY_NAME	xilffs	Specifies the library name.
LIBRARY_VER	4.3	Specifies the library version.
fs_interface	1 for SD/eMMC 2 for RAM	File system interface. SD/eMMC and RAM based file system are supported.
read_only	False	Enables the file system in Read Only mode, if true. Default is false. For Zynq UltraScale+ MPSoC devices, sets this option as true.
use_lfn	0	Enables the Long File Name(LFN) support if non-zero. 0: Disabled (Default) 1: LFN with static working buffer 2 (on stack) or 3 (on heap): Dynamic working buffer
enable_multi_partitio	False	Enables the multi partition support, if true.
num_logical_vol	2	Number of volumes (logical drives, from 1 to 10) to be used.
use_mkfs	True	Enables the mkfs support, if true. For Zynq UltraScale+ MPSoC devices, set this option as false.
use_strfunc	0	Enables the string functions (valid values 0 to 2). Default is 0.
set_fs_rpath	0	Configures relative path feature (valid values 0 to 2). Default is 0.
ramfs_size	3145728	Ram FS size is applicable only when RAM based file system is selected.
ramfs_start_addr	0x10000000	RAM FS start address is applicable only when RAM based file system is selected.

Parameter	Default Value	Description
enable_exfat	false	Enables support for exFAT file system. 0: Disable exFAT 1: Enable exFAT(Also Enables LFN)
word_access	True	Enables word access for misaligned memory access platform.
use_chmod	false	Enables use of CHMOD functionality for changing attributes (valid only with read_only set to false).

XilSecure Library v4.2

Overview

The XilSecure library provides APIs to access cryptographic accelerators on the Zynq UltraScale+ MPSoC devices. The library is designed to run on top of Xilinx standalone BSPs. It is tested for A53, R5 and MicroBlaze. XilSecure is used during the secure boot process. The primary post-boot use case is to run this library on the PMU MicroBlaze with PMUFW to service requests from Uboot or Linux for cryptographic acceleration.

Note: The XilSecure library does not check for memory bounds while performing cryptographic operations. You must check the bounds before using the functions provided in this library. If needed, you can take advantage of the XMPU, the XPPU, and/or TrustZone to limit memory access.

The XilSecure library includes:

- SHA-3/384 engine for 384 bit hash calculation.
- AES-GCM engine for symmetric key encryption and decryption using a 256-bit key.
- RSA engine for signature generation, signature verification, encryption and decryption. Key sizes supported include 2048, 3072, and 4096.



CAUTION! SDK defaults to using a software stack in DDR and any variables used by XilSecure will be placed in DDR. For better security, change the linker settings to make sure the stack used by XilSecure is either in the OCM or the TCM.

Board Support Package Settings

XilSecure provides an user configuration under BSP settings to enable or disable secure environment, this bsp parameter is valid only when BSP is build for the PMU MicroBlaze for post boot use cases and XilSecure is been accessed using the IPI response calls to PMUFW from Linux or U-boot or baremetal applications. When the application environment is secure and trusted this variable should be set to TRUE.

Parameter	Description
secure_environment	Default = FALSE. Set the value to TRUE to allow usage of device key through the IPI response calls.

By default, PMUFW will not allow device key for any decryption operation requested through IPI response unless authentication is enabled. If the user space is secure and trusted PMUFW can be build by setting the `secure_environment` variable. Only then the PMUFW allows usage of the device key for encrypting or decrypting the data blobs, decryption of bitstream or image.

Source Files

The source files for the library can be found at:

- https://github.com/Xilinx/embeddedsw/blob/master/lib/sw_services/xilsecure/
- https://github.com/Xilinx/embeddedsw/tree/master/lib/sw_services/xilsecure/src/common

AES-GCM

This software uses AES-GCM hardened cryptographic accelerator to encrypt or decrypt the provided data and requires a key of size 256 bits and initialization vector(IV) of size 96 bits.

XilSecure library supports the following features:

- Encryption of data with provided key and IV
- Decryption of data with provided key and IV
- Authentication using a GCM tag.
- Key loading based on key selection, the key can be either the user provided key loaded into the KUP key or the device key used during boot.

For either encryption or decryption the AES-GCM engine should be initialized first using the `XSecure_AesInitiaze` function.

AES Encryption Function Usage

When all the data to be encrypted is available, the `XSecure_AesEncryptData()` can be used. When all the data is not available, use the following functions in the suggested order:

1. `XSecure_AesEncryptInit()`
2. `XSecure_AesEncryptUpdate()` - This function can be called multiple times till input data is completed.

AES Decryption Function Usage

When all the data to be decrypted is available, the `XSecure_AesDecryptData()` can be used. When all the data is not available, use the following functions in the suggested order:

1. `XSecure_AesDecryptInit()`

2. `XSecure_AesDecryptUpdate()` - This function can be called multiple times till input data is completed.

During decryption, the passed in GCM tag will be compared to the GCM tag calculated by the engine. The two tags are then compared in the software and returned to the user as to whether or not the tags matched.



CAUTION! when using the KUP key for encryption/decryption of the data, where the key is stored should be carefully considered. Key should be placed in an internal memory region that has access controls. Not doing so may result in security vulnerability.

Table 257: Quick Function Reference

Type	Name	Arguments
s32	<code>XSecure_AesInitialize</code>	XSecure_Aes * InstancePtr XCsuDma * CsuDmaPtr u32 KeySel Iv Key
u32	<code>XSecure_AesDecryptInit</code>	XSecure_Aes * InstancePtr u8 * DecData u32 Size u8 * GcmTagAddr
s32	<code>XSecure_AesDecryptUpdate</code>	XSecure_Aes * InstancePtr u8 * EncData u32 Size
s32	<code>XSecure_AesDecryptData</code>	XSecure_Aes * InstancePtr u8 * DecData u8 * EncData u32 Size
s32	<code>XSecure_AesDecrypt</code>	XSecure_Aes * InstancePtr const u8 * Src u8 * Dst u32 Length
u32	<code>XSecure_AesEncryptInit</code>	XSecure_Aes * InstancePtr u8 * EncData u32 Size
u32	<code>XSecure_AesEncryptUpdate</code>	XSecure_Aes * InstancePtr const u8 * Data u32 Size

Table 257: Quick Function Reference (cont'd)

Type	Name	Arguments
u32	XSecure_AesEncryptData	XSecure_Aes * InstancePtr u8 * Dst const u8 * Src u32 Len
void	XSecure_AesReset	XSecure_Aes * InstancePtr

Functions

XSecure_AesInitialize

This function initializes the instance pointer.

Note: All the inputs are accepted in little endian format but the AES engine accepts the data in big endian format, The decryption and encryption functions in `xsecure_aes` handle the little endian to big endian conversion using few API's, `Xil_Htonl` (provided by Xilinx `xil_io` library) and `XSecure_AesCsuDmaConfigureEndiannes` for handling data endianness conversions. If higher performance is needed, users can strictly use data in big endian format and modify the `xsecure_aes` functions to remove the use of the `Xil_Htonl` and `XSecure_AesCsuDmaConfigureEndiannes` functions as required.

Prototype

```
s32 XSecure_AesInitialize(XSecure_Aes *InstancePtr, XCsuDma *CsuDmaPtr, u32 KeySel, u32 *IvPtr, u32 *KeyPtr);
```

Parameters

The following table lists the `XSecure_AesInitialize` function arguments.

Table 258: XSecure_AesInitialize Arguments

Name	Description
InstancePtr	Pointer to the XSecure_Aes instance.
CsuDmaPtr	Pointer to the XCsuDma instance.
KeySel	Key source for decryption, can be KUP/device key <ul style="list-style-type: none"> XSECURE_CSU_AES_KEY_SRC_KUP :For KUP key XSECURE_CSU_AES_KEY_SRC_DEV :For Device Key
Iv	Pointer to the Initialization Vector for decryption
Key	Pointer to Aes key in case KUP key is used. Pass Null if the device key is to be used.

Returns

XST_SUCCESS if initialization was successful.

XSecure_AesDecryptInit

This function initializes the AES engine for decryption and is required to be called before calling XSecure_AesDecryptUpdate.

Note: If all of the data to be decrypted is available, the XSecure_AesDecryptData function can be used instead.

Prototype

```
u32 XSecure_AesDecryptInit(XSecure_Aes *InstancePtr, u8 *DecData, u32 Size,
u8 *GcmTagAddr);
```

Parameters

The following table lists the XSecure_AesDecryptInit function arguments.

Table 259: XSecure_AesDecryptInit Arguments

Name	Description
InstancePtr	Pointer to the XSecure_Aes instance.
DecData	Pointer in which decrypted data will be stored.
Size	Expected size of the data in bytes whereas the number of bytes provided should be multiples of 4.
GcmTagAddr	Pointer to the GCM tag which needs to be verified during decryption of the data.

Returns

None

XSecure_AesDecryptUpdate

This function decrypts the encrypted data passed in and updates the GCM tag from any previous calls. The size from XSecure_AesDecryptInit is decremented from the size passed into this function to determine when the GCM tag passed to XSecure_AesDecryptInit needs to be compared to the GCM tag calculated in the AES engine.

Note: When Size of the data equals to size of the remaining data that data will be treated as final data. This API can be called multiple times but sum of all Sizes should be equal to Size mention in init. Return of the final call of this API tells whether GCM tag is matching or not.

Prototype

```
s32 XSecure_AesDecryptUpdate(XSecure_Aes *InstancePtr, u8 *EncData, u32 Size);
```

Parameters

The following table lists the `XSecure_AesDecryptUpdate` function arguments.

Table 260: XSecure_AesDecryptUpdate Arguments

Name	Description
InstancePtr	Pointer to the XSecure_Aes instance.
EncData	Pointer to the encrypted data which needs to be decrypted.
Size	Expected size of data to be decrypted in bytes, whereas the number of bytes should be multiples of 4.

Returns

Final call of this API returns the status of GCM tag matching.

- `XSECURE_CSU_AES_GCM_TAG_MISMATCH`: If GCM tag is mismatched
- `XSECURE_CSU_AES_ZEROIZATION_ERROR`: If GCM tag is mismatched, zeroize the decrypted data and send the status of zeroization.
- `XST_SUCCESS`: If GCM tag is matching.

XSecure_AesDecryptData

This function decrypts the encrypted data provided and updates the DecData buffer with decrypted data.

Note: When using this function to decrypt data that was encrypted with `XSecure_AesEncryptData`, the GCM tag will be stored as the last sixteen (16) bytes of data in `XSecure_AesEncryptData`'s Dst (destination) buffer and should be used as the `GcmTagAddr`'s pointer.

Prototype

```
s32 XSecure_AesDecryptData(XSecure_Aes *InstancePtr, u8 *DecData, u8 *EncData, u32 Size, u8 *GcmTagAddr);
```

Parameters

The following table lists the `XSecure_AesDecryptData` function arguments.

Table 261: XSecure_AesDecryptData Arguments

Name	Description
InstancePtr	Pointer to the XSecure_Aes instance.
DecData	Pointer to a buffer in which decrypted data will be stored.
EncData	Pointer to the encrypted data which needs to be decrypted.
Size	Size of data to be decrypted in bytes, whereas the number of bytes should be multiples of 4.

Returns

This API returns the status of GCM tag matching.

- XSECURE_CSU_AES_GCM_TAG_MISMATCH: If GCM tag was mismatched
- XST_SUCCESS: If GCM tag was matched.

XSecure_AesDecrypt

This function will handle the AES-GCM Decryption.

Note: This function is used for decrypting the Image's partition encrypted by Bootgen

Prototype

```
s32 XSecure_AesDecrypt(XSecure_Aes *InstancePtr, u8 *Dst, const u8 *Src,
u32 Length);
```

Parameters

The following table lists the XSecure_AesDecrypt function arguments.

Table 262: XSecure_AesDecrypt Arguments

Name	Description
InstancePtr	Pointer to the XSecure_Aes instance.
Src	Pointer to encrypted data source location
Dst	Pointer to location where decrypted data will be written.
Length	Expected total length of decrypted image expected.

Returns

returns XST_SUCCESS if successful, or the relevant errorcode.

XSecure_AesEncryptInit

This function is used to initialize the AES engine for encryption.

Note: If all of the data to be encrypted is available, the `XSecure_AesEncryptData` function can be used instead.

Prototype

```
u32 XSecure_AesEncryptInit(XSecure_Aes *InstancePtr, u8 *EncData, u32 Size);
```

Parameters

The following table lists the `XSecure_AesEncryptInit` function arguments.

Table 263: XSecure_AesEncryptInit Arguments

Name	Description
InstancePtr	Pointer to the XSecure_Aes instance.
EncData	Pointer of a buffer in which encrypted data along with GCM TAG will be stored. Buffer size should be Size of data plus 16 bytes.
Size	A 32 bit variable, which holds the size of the input data to be encrypted in bytes, whereas number of bytes provided should be multiples of 4.

Returns

None

XSecure_AesEncryptUpdate

This function encrypts the clear-text data passed in and updates the GCM tag from any previous calls. The size from `XSecure_AesEncryptInit` is decremented from the size passed into this function to determine when the final CSU DMA transfer of data to the AES-GCM cryptographic core.

Note: If all of the data to be encrypted is available, the `XSecure_AesEncryptData` function can be used instead.

Prototype

```
u32 XSecure_AesEncryptUpdate(XSecure_Aes *InstancePtr, const u8 *Data, u32 Size);
```

Parameters

The following table lists the `XSecure_AesEncryptUpdate` function arguments.

Table 264: XSecure_AesEncryptUpdate Arguments

Name	Description
InstancePtr	Pointer to the XSecure_Aes instance.

Table 264: XSecure_AesEncryptUpdate Arguments (cont'd)

Name	Description
Data	Pointer to the data for which encryption should be performed.
Size	A 32 bit variable, which holds the size of the input data in bytes, whereas the number of bytes provided should be multiples of 4.

Returns

None

XSecure_AesEncryptData

This function encrypts Len (length) number of bytes of the passed in Src (source) buffer and stores the encrypted data along with its associated 16 byte tag in the Dst (destination) buffer.

Note: If data to be encrypted is not available in one buffer one can call `XSecure_AesEncryptInit()` and update the AES engine with data to be encrypted by calling `XSecure_AesEncryptUpdate()` API multiple times as required.

Prototype

```
u32 XSecure_AesEncryptData(XSecure_Aes *InstancePtr, u8 *Dst, const u8 *Src, u32 Len);
```

Parameters

The following table lists the `XSecure_AesEncryptData` function arguments.

Table 265: XSecure_AesEncryptData Arguments

Name	Description
InstancePtr	A pointer to the XSecure_Aes instance.
Dst	A pointer to a buffer where encrypted data along with GCM tag will be stored. The Size of buffer provided should be Size of the data plus 16 bytes
Src	A pointer to input data for encryption.
Len	Size of input data in bytes, whereas the number of bytes provided should be multiples of 4.

Returns

None

XSecure_AesReset

This function sets and then clears the AES-GCM's reset line.

Prototype

```
void XSecure_AesReset(XSecure_Aes *InstancePtr);
```

Parameters

The following table lists the `XSecure_AesReset` function arguments.

Table 266: XSecure_AesReset Arguments

Name	Description
InstancePtr	is a pointer to the XSecure_Aes instance.

Returns

None

Definitions

XSecure_AesWaitForDone

This macro waits for AES engine completes configured operation.

Definition

```
#define XSecure_AesWaitForDoneXil_WaitForEvent((InstancePtr)->BaseAddress +
XSECURE_CSU_AES_STS_OFFSET, \
        XSECURE_CSU_AES_STS_AES_BUSY, \
        0U, \
        XSECURE_AES_TIMEOUT_MAX)
```

Parameters

The following table lists the `XSecure_AesWaitForDone` definition values.

Table 267: XSecure_AesWaitForDone Values

Name	Description
InstancePtr	Pointer to the XSecure_Aes instance.

Returns

XST_SUCCESS if the AES engine completes configured operation. XST_FAILURE if a timeout has occurred.

AES-GCM Error Codes

The table below lists the AES-GCM error codes.

Error Code	Error Value	Description
XSECURE_CSU_AES_GCM_TAG_MISMATCH	0x1	User provided GCM tag does not match with GCM calculated on data
XSECURE_CSU_AES_IMAGE_LENGTH_MISMATCH	0x2	When there is a Image length mismatch
XSECURE_CSU_AES_DEVICE_COPY_ERROR	0x3	When there is device copy error.
XSECURE_CSU_AES_ZEROIZATION_ERROR	0x4	When there is an error with Zeroization. Note: In case of any error during Aes decryption, we perform zeroization of the decrypted data.
XSECURE_CSU_AES_KEY_CLEAR_ERROR	0x20	Error when clearing key storage registers after Aes operation.

AES-GCM API Example Usage

The following example illustrates the usage of AES encryption and decryption APIs.

```
static s32 SecureAesExample(void)
{
    XCsuDma_Config *Config;
    s32 Status;
    u32 Index;
    XCsuDma CsuDmaInstance;
    XSecure_Aes Secure_Aes;

    /* Initialize CSU DMA driver */
    Config = XCsuDma_LookupConfig(XSECURE_CSUDMA_DEVICEID);
    if (NULL == Config) {
        return XST_FAILURE;
    }

    Status = XCsuDma_CfgInitialize(&CsuDmaInstance, Config,
                                   Config->BaseAddress);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    /* Initialize the Aes driver so that it's ready to use */
    XSecure_AesInitialize(&Secure_Aes, &CsuDmaInstance,
                          XSECURE_CSU_AES_KEY_SRC_KUP,
                          (u32 *)Iv, (u32 *)Key);

    xil_printf("Data to be encrypted: \n\r");
    for (Index = 0; Index < XSECURE_DATA_SIZE; Index++) {
        xil_printf("%02x", Data[Index]);
    }
    xil_printf( "\r\n\n");

    /* Encryption of Data */
}
```

```

/*
 * If all the data to be encrypted is contiguous one can call
 * XSecure_AesEncryptData API directly.
 */
XSecure_AesEncryptInit(&Secure_Aes, EncData, XSECURE_DATA_SIZE);
XSecure_AesEncryptUpdate(&Secure_Aes, Data, XSECURE_DATA_SIZE);

xil_printf("Encrypted data: \n\r");
for (Index = 0; Index < XSECURE_DATA_SIZE; Index++) {
    xil_printf("%02x", EncData[Index]);
}
xil_printf( "\r\n");

xil_printf("GCM tag: \n\r");
for (Index = 0; Index < XSECURE_SECURE_GCM_TAG_SIZE; Index++) {
    xil_printf("%02x", EncData[XSECURE_DATA_SIZE + Index]);
}
xil_printf( "\r\n\r\n");

/* Decrypt's the encrypted data */
/*
 * If data to be decrypted is contiguous one can also call
 * single API XSecure_AesDecryptData
 */
XSecure_AesDecryptInit(&Secure_Aes, DecData, XSECURE_DATA_SIZE,
                      EncData + XSECURE_DATA_SIZE);
/* Only the last update will return the GCM TAG matching status */
Status = XSecure_AesDecryptUpdate(&Secure_Aes, EncData,
                                  XSECURE_DATA_SIZE);
if (Status != XST_SUCCESS) {
    xil_printf("Decryption failure- GCM tag was not matched\n\r");
    return Status;
}

xil_printf("Decrypted data\n\r");
for (Index = 0; Index < XSECURE_DATA_SIZE; Index++) {
    xil_printf("%02x", DecData[Index]);
}
xil_printf( "\r\n");

/* Comparison of Decrypted Data with original data */
for(Index = 0; Index < XSECURE_DATA_SIZE; Index++) {
    if (Data[Index] != DecData[Index]) {
        xil_printf("Failure during comparison of the data\n\r");
        return XST_FAILURE;
    }
}

return XST_SUCCESS;
}
    
```

Note: Relevant examples are available in the <library-install-path>\examples folder. Where <library-install-path> is the XilSecure library installation path.

AES-GCM Usage to decrypt Boot Image

The Multiple key(Key Rolling) or Single key encrypted images will have the same format. The images include:

- Secure header - This includes the dummy AES key of 32byte + Block 0 IV of 12byte + DLC for Block 0 of 4byte + GCM tag of 16byte(Un-Enc).
- Block N - This includes the boot image data for the block N of n size + Block N+1 AES key of 32byte + Block N+1 IV of 12byte + GCM tag for Block N of 16byte(Un-Enc).

The Secure header and Block 0 will be decrypted using the device key or user provided key. If more than one block is found then the key and the IV obtained from previous block will be used for decryption.

Following are the instructions to decrypt an image:

1. Read the first 64 bytes and decrypt 48 bytes using the selected Device key.
2. Decrypt Block 0 using the IV + Size and the selected Device key.
3. After decryption, you will get the decrypted data+KEY+IV+Block Size. Store the KEY/IV into KUP/IV registers.
4. Using Block size, IV and the next Block key information, start decrypting the next block.
5. If the current image size is greater than the total image length, perform the next step. Else, go back to the previous step.
6. If there are failures, an error code is returned. Else, the decryption is successful.

RSA

The `xsecure_rsa.h` file contains hardware interface related information for the RSA hardware accelerator. This hardened cryptographic accelerator, within the CSU, performs the modulus math based on the Rivest-Shamir-Adelman (RSA) algorithm. It is an asymmetric algorithm.

Initialization & Configuration

The RSA driver instance can be initialized by using the `XSecure_RsaInitialize()` function. The method used for RSA implementation can take a pre-calculated value of $R^2 \bmod N$. If you do not have the pre-calculated exponential value pass NULL, the controller will take care of the exponential value.

Note:

- From the RSA key modulus, the exponent should be extracted.
- For verification, PKCS v1.5 padding scheme has to be applied for comparing the data hash with decrypted hash.

Table 268: Quick Function Reference

Type	Name	Arguments
s32	XSecure_RsaInitialize	XSecure_Rsa * InstancePtr u8 * Mod u8 * ModExt u8 * ModExpo
u32	XSecure_RsaSignVerification	u8 * Signature u8 * Hash u32 HashLen
s32	XSecure_RsaPublicEncrypt	XSecure_Rsa * InstancePtr u8 * Input u32 Size u8 * Result
s32	XSecure_RsaPrivateDecrypt	XSecure_Rsa * InstancePtr u8 * Input u32 Size u8 * Result

Functions

XSecure_RsaInitialize

This function initializes a XSecure_Rsa structure with the default values required for operating the RSA cryptographic engine.

Note: Modulus, ModExt and ModExpo are part of prtion signature when authenticated boot image is generated by bootgen, else the all of them should be extracted from the key.

Prototype

```
s32 XSecure_RsaInitialize(XSecure_Rsa *InstancePtr, u8 *Mod, u8 *ModExt, u8 *ModExpo);
```

Parameters

The following table lists the XSecure_RsaInitialize function arguments.

Table 269: XSecure_RsaInitialize Arguments

Name	Description
InstancePtr	Pointer to the XSecure_Rsa instance.

Table 269: XSecure_RsaInitialize Arguments (cont'd)

Name	Description
Mod	A character Pointer which contains the key Modulus of key size.
ModExt	A Pointer to the pre-calculated exponential ($R^2 \text{ Mod } N$) value. <ul style="list-style-type: none"> • NULL - if user doesn't have pre-calculated $R^2 \text{ Mod } N$ value, control will take care of this calculation internally.
ModExpo	Pointer to the buffer which contains key exponent.

Returns

XST_SUCCESS if initialization was successful.

XSecure_RsaSignVerification

This function verifies the RSA decrypted data provided is either matching with the provided expected hash by taking care of PKCS padding.

Prototype

```
u32 XSecure_RsaSignVerification(u8 *Signature, u8 *Hash, u32 HashLen);
```

Parameters

The following table lists the XSecure_RsaSignVerification function arguments.

Table 270: XSecure_RsaSignVerification Arguments

Name	Description
Signature	Pointer to the buffer which holds the decrypted RSA signature
Hash	Pointer to the buffer which has the hash calculated on the data to be authenticated.
HashLen	Length of Hash used. <ul style="list-style-type: none"> • For SHA3 it should be 48 bytes • For SHA2 it should be 32 bytes

Returns

- XST_SUCCESS if decryption was successful.
- XST_FAILURE in case of mismatch.

XSecure_RsaPublicEncrypt

This function handles the RSA encryption with the public key components provided when initializing the RSA cryptographic core with the XSecure_RsaInitialize function.

Note: The Size passed here needs to match the key size used in the XSecure_RsaInitialize function.

Prototype

```
s32 XSecure_RsaPublicEncrypt(XSecure_Rsa *InstancePtr, u8 *Input, u32 Size,
u8 *Result);
```

Parameters

The following table lists the XSecure_RsaPublicEncrypt function arguments.

Table 271: XSecure_RsaPublicEncrypt Arguments

Name	Description
InstancePtr	Pointer to the XSecure_Rsa instance.
Input	Pointer to the buffer which contains the input data to be encrypted.
Size	Key size in bytes, Input size also should be same as Key size mentioned. Inputs supported are <ul style="list-style-type: none"> • XSECURE_RSA_4096_KEY_SIZE • XSECURE_RSA_2048_KEY_SIZE • XSECURE_RSA_3072_KEY_SIZE
Result	Pointer to the buffer where resultant decrypted data to be stored .

Returns

- XST_SUCCESS if encryption was successful.
- Error code on failure

XSecure_RsaPrivateDecrypt

This function handles the RSA decryption with the private key components provided when initializing the RSA cryptographic core with the XSecure_RsaInitialize function.

Note: The Size passed in needs to match the key size used in the XSecure_RsaInitialize function..

Prototype

```
s32 XSecure_RsaPrivateDecrypt(XSecure_Rsa *InstancePtr, u8 *Input, u32
Size, u8 *Result);
```

Parameters

The following table lists the `XSecure_RsaPrivateDecrypt` function arguments.

Table 272: **XSecure_RsaPrivateDecrypt Arguments**

Name	Description
InstancePtr	Pointer to the XSecure_Rsa instance.
Input	Pointer to the buffer which contains the input data to be decrypted.
Size	Key size in bytes, Input size also should be same as Key size mentioned. Inputs supported are <ul style="list-style-type: none"> • XSECURE_RSA_4096_KEY_SIZE, • XSECURE_RSA_2048_KEY_SIZE • XSECURE_RSA_3072_KEY_SIZE
Result	Pointer to the buffer where resultant decrypted data to be stored .

Returns

- XST_SUCCESS if decryption was successful.
- XSECURE_RSA_DATA_VALUE_ERROR - if input data is greater than modulus.
- XST_FAILURE - on RSA operation failure.

RSA API Example Usage

The following example illustrates the usage of the RSA library to encrypt data using the public key and to decrypt the data using private key.

Note: Application should take care of the padding.

```

u32 SecureRsaExample(void)
{
    u32 Index;

    /* RSA signature decrypt with private key */
    /*
     * Initialize the Rsa driver with private key components
     * so that it's ready to use
     */
    XSecure_RsaInitialize(&Secure_Rsa, Modulus, NULL, PrivateExp);

    if(XST_SUCCESS != XSecure_RsaPrivateDecrypt(&Secure_Rsa, Data,
                                                Size, Signature)) {
        xil_printf("Failed at RSA signature decryption\n\r");
        return XST_FAILURE;
    }

    xil_printf("\r\n Decrypted Signature with private key\r\n ");
    for(Index = 0; Index < Size; Index++) {

```

```

        xil_printf(" %02x ", Signature[Index]);
    }
    xil_printf(" \r\n ");

    /* Verification if Data is expected */
    for(Index = 0; Index < Size; Index++) {
        if (Signature[Index] != ExpectedSign[Index]) {
signature"
            xil_printf("\r\nError at verification of RSA
                " Decryption\n\r");
            return XST_FAILURE;
        }
    }

    /* RSA signature encrypt with Public key components */

    /*
    * Initialize the Rsa driver with public key components
    * so that it's ready to use
    */

    XSecure_RsaInitialize(&Secure_Rsa, Modulus, NULL, (u8 *)&PublicExp);

    if(XST_SUCCESS != XSecure_RsaPublicEncrypt(&Secure_Rsa, Signature,
EncryptSignatureOut)
        Size,
        {
            xil_printf("\r\nFailed at RSA signature encryption\n\r");
            return XST_FAILURE;
        }
        xil_printf("\r\n Encrypted Signature with public key\r\n ");

        for(Index = 0; Index < Size; Index++) {
            xil_printf(" %02x ", EncryptSignatureOut[Index]);
        }

        /* Verification if Data is expected */
        for(Index = 0; Index < Size; Index++) {
signature"
            if (EncryptSignatureOut[Index] != Data[Index]) {
                xil_printf("\r\nError at verification of RSA
                    " encryption\n\r");
                return XST_FAILURE;
            }
        }

        return XST_SUCCESS;
    }
}

```

Note: Relevant examples are available in the <library-install-path>\examples folder. Where <library-install-path> is the XilSecure library installation path.

SHA-3

This block uses the NIST-approved SHA-3 algorithm to generate a 384-bit hash on the input data. Because the SHA-3 hardware only accepts 104 byte blocks as the minimum input size, the input data will be padded with user selectable Keccak or NIST SHA-3 padding and is handled internally in the SHA-3 library.

Initialization & Configuration

The SHA-3 driver instance can be initialized using the `XSecure_Sha3Initialize()` function. A pointer to `CsuDma` instance has to be passed during initialization as the CSU DMA will be used for data transfers to the SHA module.

SHA-3 Function Usage

When all the data is available on which the SHA3 hash must be calculated, the `XSecure_Sha3Digest()` can be used with the appropriate parameters as described. When all the data is not available, use the SHA3 functions in the following order:

1. `XSecure_Sha3Start()`
2. `XSecure_Sha3Update()` - This function can be called multiple times until all input data has been passed to the SHA-3 cryptographic core.
3. `XSecure_Sha3Finish()` - Provides the final hash of the data. To get intermediate hash values after each `XSecure_Sha3Update()`, you can call `XSecure_Sha3_ReadHash()` after the `XSecure_Sha3Update()` call.

Table 273: Quick Function Reference

Type	Name	Arguments
s32	<code>XSecure_Sha3Initialize</code>	<code>XSecure_Sha3 * InstancePtr</code> <code>XCsuDma * CsuDmaPtr</code>
void	<code>XSecure_Sha3Start</code>	<code>XSecure_Sha3 * InstancePtr</code>
u32	<code>XSecure_Sha3Update</code>	<code>XSecure_Sha3 * InstancePtr</code> <code>const u8 * Data</code> <code>const u32 Size</code>
u32	<code>XSecure_Sha3Finish</code>	<code>XSecure_Sha3 * InstancePtr</code> <code>u8 * Hash</code>

Table 273: Quick Function Reference (cont'd)

Type	Name	Arguments
u32	XSecure_Sha3Digest	XSecure_Sha3 * InstancePtr const u8 * In const u32 Size u8 * Out
void	XSecure_Sha3_ReadHash	XSecure_Sha3 * InstancePtr u8 * Hash
s32	XSecure_Sha3PadSelection	XSecure_Sha3 * InstancePtr Sha3Type
s32	XSecure_Sha3LastUpdate	XSecure_Sha3 * InstancePtr
u32	XSecure_Sha3WaitForDone	XSecure_Sha3 * InstancePtr

Functions

XSecure_Sha3Initialize

This function initializes a XSecure_Sha3 structure with the default values required for operating the SHA3 cryptographic engine.

Note: The base address is initialized directly with value from xsecure_hw.h The default is NIST SHA3 padding, to change to KECCAK padding call [XSecure_Sha3PadSelection\(\)](#) after [XSecure_Sha3Initialize\(\)](#).

Prototype

```
s32 XSecure_Sha3Initialize(XSecure_Sha3 *InstancePtr, XCsuDma *CsuDmaPtr);
```

Parameters

The following table lists the XSecure_Sha3Initialize function arguments.

Table 274: XSecure_Sha3Initialize Arguments

Name	Description
InstancePtr	Pointer to the XSecure_Sha3 instance.
CsuDmaPtr	Pointer to the XCsuDma instance.

Returns

XST_SUCCESS if initialization was successful

XSecure_Sha3Start

This function configures Secure Stream Switch and starts the SHA-3 engine.

Prototype

```
void XSecure_Sha3Start(XSecure_Sha3 *InstancePtr);
```

Parameters

The following table lists the XSecure_Sha3Start function arguments.

Table 275: XSecure_Sha3Start Arguments

Name	Description
InstancePtr	Pointer to the XSecure_Sha3 instance.

Returns

None

XSecure_Sha3Update

This function updates the SHA3 engine with the input data.

Prototype

```
u32 XSecure_Sha3Update(XSecure_Sha3 *InstancePtr, const u8 *Data, const u32 Size);
```

Parameters

The following table lists the XSecure_Sha3Update function arguments.

Table 276: XSecure_Sha3Update Arguments

Name	Description
InstancePtr	Pointer to the XSecure_Sha3 instance.
Data	Pointer to the input data for hashing.
Size	Size of the input data in bytes.

Returns

XST_SUCCESS if the update is successful XST_FAILURE if there is a failure in SSS config

XSecure_Sha3Finish

This function updates SHA3 engine with final data which includes SHA3 padding and reads final hash on complete data.

Prototype

```
u32 XSecure_Sha3Finish(XSecure_Sha3 *InstancePtr, u8 *Hash);
```

Parameters

The following table lists the `XSecure_Sha3Finish` function arguments.

Table 277: XSecure_Sha3Finish Arguments

Name	Description
InstancePtr	Pointer to the XSecure_Sha3 instance.
Hash	Pointer to location where resulting hash will be written

Returns

XST_SUCCESS if finished without any errors XST_FAILURE if Sha3PadType is other than KECCAK or NIST

XSecure_Sha3Digest

This function calculates the SHA-3 digest on the given input data.

Prototype

```
u32 XSecure_Sha3Digest(XSecure_Sha3 *InstancePtr, const u8 *In, const u32 Size, u8 *Out);
```

Parameters

The following table lists the `XSecure_Sha3Digest` function arguments.

Table 278: XSecure_Sha3Digest Arguments

Name	Description
InstancePtr	Pointer to the XSecure_Sha3 instance.
In	Pointer to the input data for hashing
Size	Size of the input data

Table 278: XSecure_Sha3Digest Arguments (cont'd)

Name	Description
Out	Pointer to location where resulting hash will be written.

Returns

XST_SUCCESS if digest calculation done successfully XST_FAILURE if any error from Sha3Update or Sha3Finish.

XSecure_Sha3_ReadHash

This function reads the SHA3 hash of the data and it can be called between calls to XSecure_Sha3Update.

Prototype

```
void XSecure_Sha3_ReadHash(XSecure_Sha3 *InstancePtr, u8 *Hash);
```

Parameters

The following table lists the XSecure_Sha3_ReadHash function arguments.

Table 279: XSecure_Sha3_ReadHash Arguments

Name	Description
InstancePtr	Pointer to the XSecure_Sha3 instance.
Hash	Pointer to a buffer in which read hash will be stored.

Returns

None

XSecure_Sha3PadSelection

This function provides an option to select the SHA-3 padding type to be used while calculating the hash.

Note: The default provides support for NIST SHA-3. If a user wants to change the padding to Keccak SHA-3, this function should be called after `XSecure_Sha3Initialize()`

Prototype

```
s32 XSecure_Sha3PadSelection(XSecure_Sha3 *InstancePtr, XSecure_Sha3PadType Sha3PadType);
```

Parameters

The following table lists the `XSecure_Sha3PadSelection` function arguments.

Table 280: XSecure_Sha3PadSelection Arguments

Name	Description
InstancePtr	Pointer to the XSecure_Sha3 instance.
Sha3Type	Type of SHA3 padding to be used. <ul style="list-style-type: none"> For NIST SHA-3 padding - XSECURE_CSU_NIST_SHA3 For KECCAK SHA-3 padding - XSECURE_CSU_KECCAK_SHA3

Returns

XST_SUCCESS if pad selection is successful. XST_FAILURE if pad selection is failed.

XSecure_Sha3LastUpdate

This function is to notify this is the last update of data where sha padding is also been included along with the data in the next update call.

Prototype

```
s32 XSecure_Sha3LastUpdate(XSecure_Sha3 *InstancePtr);
```

Parameters

The following table lists the `XSecure_Sha3LastUpdate` function arguments.

Table 281: XSecure_Sha3LastUpdate Arguments

Name	Description
InstancePtr	Pointer to the XSecure_Sha3 instance.

Returns

XST_SUCCESS if last update can be accepted

XSecure_Sha3WaitForDone

This inline function waits till SHA3 completes its operation.

Prototype

```
u32 XSecure_Sha3WaitForDone(XSecure_Sha3 *InstancePtr);
```

Parameters

The following table lists the `XSecure_Sha3WaitForDone` function arguments.

Table 282: XSecure_Sha3WaitForDone Arguments

Name	Description
InstancePtr	Pointer to the XSecure_Sha3 instance.

Returns

`XST_SUCCESS` if the SHA3 completes its operation. `XST_FAILURE` if a timeout has occurred.

SHA-3 API Example Usage

The `xilsecure_sha_example.c` file is a simple example application that demonstrates the usage of SHA-3 accelerator to calculate a 384-bit hash on the Hello World string. A typical use case for the SHA3 accelerator is for calculation of the boot image hash as part of the authentication operation. This is illustrated in the `xilsecure_rsa_example.c`.

The contents of the `xilsecure_sha_example.c` file are shown below:

```
int SecureHelloWorldExample()
{
    u8 HelloWorld[4] = {'h','e','l','l'};
    u32 Size = sizeof(HelloWorld);
    u8 Out[384/8];
    XCsuDma_Config *Config;

    int Status;

    Config = XCsuDma_LookupConfig(0);
    if (NULL == Config) {
        xil_printf("config failed\n\r");
        return XST_FAILURE;
    }

    Status = XCsuDma_CfgInitialize(&CsuDma, Config, Config->BaseAddress);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    /*
     * Initialize the SHA-3 driver so that it's ready to use
     */
    XSecure_Sha3Initialize(&Secure_Sha3, &CsuDma);

    XSecure_Sha3Digest(&Secure_Sha3, HelloWorld, Size, Out);

    xil_printf(" Calculated Digest \r\n ");
    int i= 0;
    for(i=0; i< (384/8); i++)
    {
        xil_printf(" %0x ", Out[i]);
    }
}
```

```

    }
    xil_printf(" \r\n ");

    return XST_SUCCESS;
}
    
```

Note: The `xilsecure_sha_example.c` and `xilsecure_rsa_example.c` example files are available in the `<library-install-path>\examples` folder. Where `<library-install-path>` is the XilSecure library installation path.

XilSecure Utilities

The `xsecure_utils.h` file contains common functions used among the XilSecure library like holding hardware crypto engines in Reset or bringing them out of reset, and secure stream switch configuration for AES and SHA3.

Table 283: Quick Function Reference

Type	Name	Arguments
u32	XSecure_ReadReg	u32 BaseAddress u16 RegOffset
void	XSecure_WriteReg	u32 BaseAddress u32 RegOffset u32 RegisterValue
void	XSecure_SetReset	u32 BaseAddress u32 BaseAddress
void	XSecure_ReleaseReset	u32 BaseAddress u32 BaseAddress

Functions

XSecure_ReadReg

Read from the register.

Note: C-Style signature: `u32 XSecure_ReadReg(u32 BaseAddress, u16 RegOffset)`

Prototype

```
u32 XSecure_ReadReg(u32 BaseAddress, u16 RegOffset);
```

Parameters

The following table lists the `XSecure_ReadReg` function arguments.

Table 284: XSecure_ReadReg Arguments

Name	Description
BaseAddress	contains the base address of the device.
RegOffset	contains the offset from the base address of the device.

Returns

The value read from the register.

XSecure_WriteReg

Write to the register.

Note: C-Style signature: `void XSecure_WriteReg(u32 BaseAddress, u16 RegOffset, u16 RegisterValue)`

Prototype

```
void XSecure_WriteReg(u32 BaseAddress, u32 RegOffset, u32 RegisterValue);
```

Parameters

The following table lists the `XSecure_WriteReg` function arguments.

Table 285: XSecure_WriteReg Arguments

Name	Description
BaseAddress	contains the base address of the device.
RegOffset	contains the offset from the base address of the device.
RegisterValue	is the value to be written to the register

Returns

None.

XSecure_SetReset

This function places the hardware core into the reset.

Prototype

```
void XSecure_SetReset(u32 BaseAddress, u32 Offset);
```

Parameters

The following table lists the `XSecure_SetReset` function arguments.

Table 286: XSecure_SetReset Arguments

Name	Description
BaseAddress	Base address of the core.
BaseAddress	Offset of the reset register.

Returns

None

XSecure_ReleaseReset

This function takes the hardware core out of reset.

Prototype

```
void XSecure_ReleaseReset(u32 BaseAddress, u32 Offset);
```

Parameters

The following table lists the `XSecure_ReleaseReset` function arguments.

Table 287: XSecure_ReleaseReset Arguments

Name	Description
BaseAddress	Base address of the core.
BaseAddress	Offset of the reset register.

Returns

None

Additional References

For more information on Linux interface to Zynq® UltraScale+™ MPSoC secure modules, see:

- <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/152305667/Zynq+Ultrascale+MPSoC+Secure+Driver+for+Linux>
- <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841936/RSA+Driver>

- <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841654/Linux+SHA+Driver+for+Zynq+UltraScale+MPSoc>
- <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/64749783/ZynqMP+AES+Driver>

For more information on the U-boot interface, see <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842432>Loading+authenticated+and+or+encrypted+image+partitions+from+u-boot>.

XiISkey Library v4.9

Overview

The XiISKey library provides APIs for programming and reading eFUSE bits and for programming the battery-backed RAM (BBRAM) of Zynq-7000 SoC, UltraScale, UltraScale+ and the Zynq UltraScale+ MPSoC devices.

- In Zynq-7000 devices:
 - PS eFUSE holds the RSA primary key hash bits and user feature bits, which can enable or disable some Zynq-7000 processor features.
 - PL eFUSE holds the AES key, the user key and some of the feature bits.
 - PL BBRAM holds the AES key.
- In Kintex/Virtex UltraScale or UltraScale+:
 - PL eFUSE holds the AES key, 32 bit and 128 bit user key, RSA hash and some of the feature bits.
 - PL BBRAM holds AES key with or without DPA protection enable or obfuscated key programming.
- In Zynq UltraScale+ MPSoC:
 - PUF registration and Regeneration.
 - PS eFUSE holds:

Programming AES key and can perform CRC verification of AES key

- Programming/Reading User fuses
- Programming/Reading PPK0/PPK1 sha3 hash
- Programming/Reading SPKID
- Programming/Reading secure control bits
 - PS BBRAM holds the AES key.
 - PL eFUSE holds the AES key, 32 bit and 128 bit user key, RSA hash and some of the feature bits.

- PL BBRAM holds AES key with or without DPA protection enable or obfuscated key programming.

Board Support Package Settings

There are few configurable parameters available under bsp settings, which can be configured during compilation of board support package.

Configurations For Adding New device

The below configurations helps in adding new device information not supported by default. Currently, MicroBlaze, Zynq UltraScale and Zynq UltraScale+ MPSoC devices are supported.

Parameter Name	Description
device_id	Mention the device ID
device_irlen	Mention IR length of the device. Default is 0
device_numslr	Mention number of SLRs available. Range of values can be 1 to 4. Default is 1. If no slaves are present and only one master SLR is available then only 1 number of SLR is available.
device_series	Select the device series. Default is FPGA SERIES ZYNQ. The following device series are supported: XSK_FPGA_SERIES_ZYNQ - Select if the device belongs to the Zynq-7000 family. XSK_FPGA_SERIES_ULTRA - Select if the device belongs to the Zynq UltraScale family. XSK_FPGA_SERIES_ULTRA_PLUS - Select if the device belongs to Zynq UltraScale MPSoC family.
device_masterslr	Mention the master SLR number. Default is 0.

Configurations For Zynq UltraScale+ MPSoC devices

Parameter Name	Description
override_sysmon_cfg	Default = TRUE, library configures sysmon before accessing efuse memory. If you are using the Sysmon library and XilSkey library together, XilSkey overwrites the user defined sysmon configuration by default. When override_sysmon_cfg is set to false, XilSkey expects you to configure the sysmon to read the 3 ADC channels - Supply 1 (VPINT), Supply 3 (VPAUX) and LPD Temperature. XilSkey validates the user defined sysmon configuration is correct before performing the eFuse operations.

Note: On Ultrascale and Ultrascale plus devices there can be multiple or single SLRs and among which one can be master and the others are slaves, where SLR 0 is not always the master SLR. Based on master and slave SLR order SLRs in this library are referred with config order index. Master SLR is mentioned with CONFIG ORDER 0, then follows the slaves config order, CONFIG ORDER 1,2 and 3 are for slaves in order. Due to the added support for the SSIT devices, it is recommended to use the updated library with updated examples only for the UltraScale and the UltraScale+ devices.

Hardware Setup

This section describes the hardware setup required for programming PL BBRAM or PL eFUSE.

Hardware setup for Zynq PL

This section describes the hardware setup required for programming BBRAM or eFUSE of Zynq PL devices. PL eFUSE or PL BBRAM is accessed through PS via MIO pins which are used for communication PL eFUSE or PL BBRAM through JTAG signals, these can be changed depending on the hardware setup. A hardware setup which dedicates four MIO pins for JTAG signals should be used and the MIO pins should be mentioned in application header file (xilskey_input.h). There should be a method to download this example and have the MIO pins connected to JTAG before running this application. You can change the listed pins at your discretion.

MUX Usage Requirements

To write the PL eFUSE or PL BBRAM using a driver you must:

- Use four MIO lines (TCK,TMS,TDO,TDI)
- Connect the MIO lines to a JTAG port

If you want to switch between the external JTAG and JTAG operation driven by the MIOs, you must:

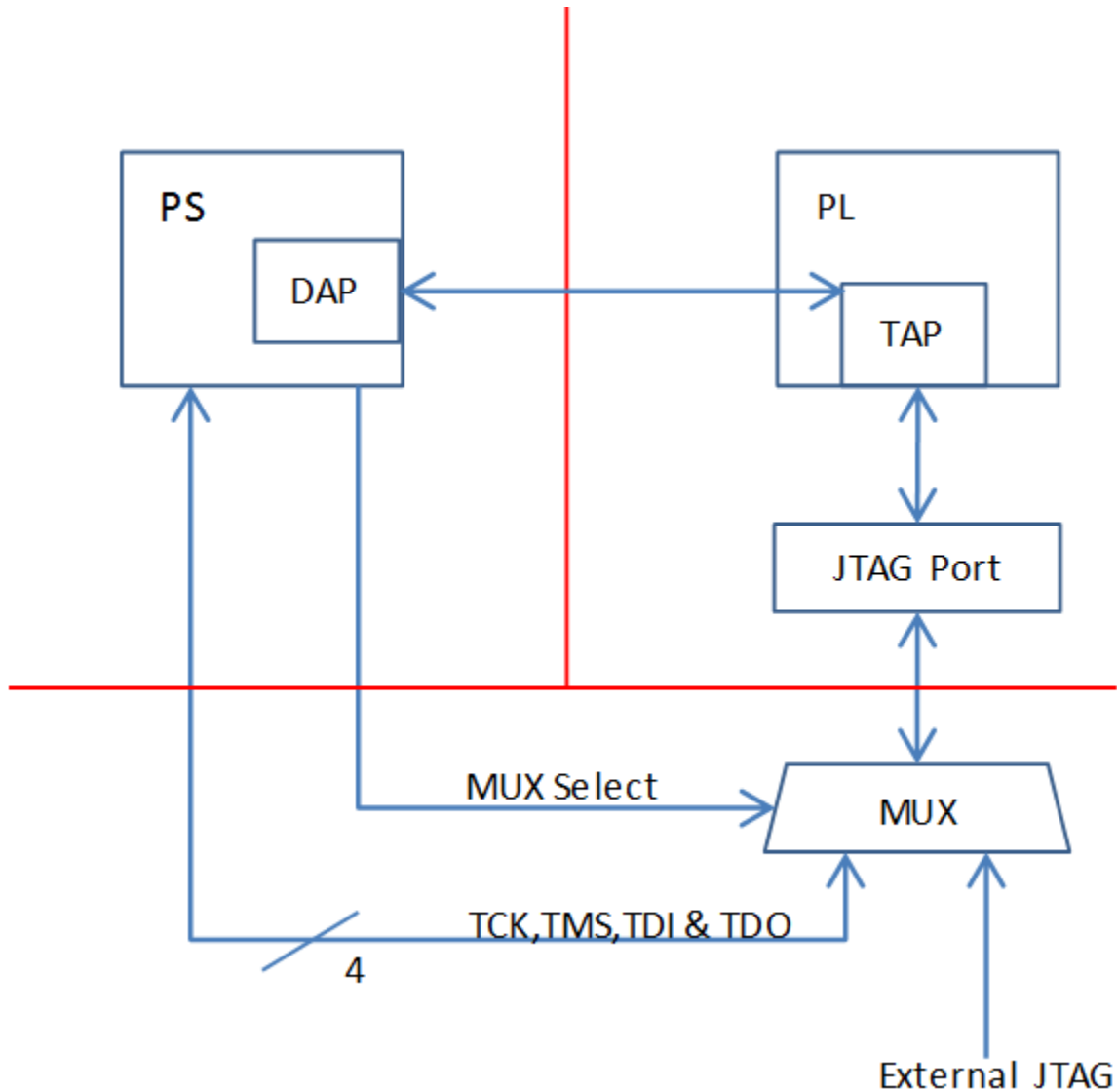
- Include a MUX between the external JTAG and the JTAG operation driven by the MIOs
- Assign a MUX selection PIN

To rephrase, to select JTAG for PL EFUSE or PL BBRAM writing, you must define the following:

- The MIOs used for JTAG operations (TCK,TMS,TDI,TDO).
- The MIO used for the MUX Select Line.
- The Value on the MUX Select line, to select JTAG for PL eFUSE or PL BBRAM writing.

The following graphic illustrates the correct MUX usage:

Figure 80: MUX Usage



Note: If you use the Vivado Device Programmer tool to burn PL eFUSEs, there is no need for MUX circuitry or MIO pins.

Hardware setup for UltraScale or UltraScale+

This section describes the hardware setup required for programming BBRAM or eFUSE of UltraScale devices. Accessing UltraScale MicroBlaze eFuse is done by using block RAM initialization. UltraScale eFUSE programming is done through MASTER JTAG. Crucial Programming sequence will be taken care by Hardware module. It is mandatory to add Hardware module in the design. Use hardware module's vhd code and instructions provided to add Hardware module in the design.

- You need to add the Master JTAG primitive to design, that is, the MASTER_JTAG_inst instantiation has to be performed and AXI GPIO pins have to be connected to TDO, TDI, TMS and TCK signals of the MASTER_JTAG primitive.
- For programming eFUSE, along with master JTAG, hardware module(HWM) has to be added in design and it's signals XSK_EFUSEPL_AXI_GPIO_HWM_READY , XSK_EFUSEPL_AXI_GPIO_HWM_END and XSK_EFUSEPL_AXI_GPIO_HWM_START, needs to be connected to AXI GPIO pins to communicate with HWM. Hardware module is not mandatory for programming BBRAM. If your design has a HWM, it is not harmful for accessing BBRAM.
- All inputs (Master JTAG's TDO and HWM's HWM_READY, HWM_END) and all outputs (Master JTAG TDI, TMS, TCK and HWM's HWM_START) can be connected in one channel (or) inputs in one channel and outputs in other channel.
- Some of the outputs of GPIO in one channel and some others in different channels are not supported.
- The design should contain AXI BRAM control memory mapped (1MB).

Note: MASTER_JTAG will disable all other JTAGs.

For providing inputs of MASTER JTAG signals and HWM signals connected to the GPIO pins and GPIO channels, refer GPIO Pins Used for PL Master JTAG Signal and GPIO Channels sections of the UltraScale User-Configurable PL eFUSE Parameters and UltraScale User-Configurable PL BBRAM Parameters. The procedure for programming BBRAM of eFUSE of UltraScale or UltraScale+ can be referred at UltraScale BBRAM Access Procedure and UltraScale eFUSE Access Procedure.

Source Files

The following is a list of eFUSE and BBRAM application project files, folders and macros.

- xilskey_efuse_example.c: This file contains the main application code. The file helps in the PS/PL structure initialization and writes/reads the PS/PL eFUSE based on the user settings provided in the xilskey_input.h file.
- xilskey_input.h: This file contains all the actions that are supported by the eFUSE library. Using the preprocessor directives given in the file, you can read/write the bits in the PS/PL eFUSE. More explanation of each directive is provided in the following sections. Burning or reading the PS/PL eFUSE bits is based on the values set in the xilskey_input.h file. Also contains GPIO pins and channels connected to MASTER JTAG primitive and hardware module to access Ultrascale eFUSE.

In this file:

- specify the 256 bit key to be programmed into BBRAM.
- specify the AES(256 bit) key, User (32 bit and 128 bit) keys and RSA key hash(384 bit) key to be programmed into UltraScale eFUSE.

- XSK_EFUSEPS_DRIVER: Define to enable the writing and reading of PS eFUSE.
 - XSK_EFUSEPL_DRIVER: Define to enable the writing of PL eFUSE.
 - xilskey_bbram_example.c: This file contains the example to program a key into BBRAM and verify the key.
- Note:** This algorithm only works when programming and verifying key are both executed in the recommended order.
- xilskey_efuseps_zynqmp_example.c: This file contains the example code to program the PS eFUSE and read back of eFUSE bits from the cache.
 - xilskey_efuseps_zynqmp_input.h: This file contains all the inputs supported for eFUSE PS of Zynq UltraScale+ MPSoC. eFUSE bits are programmed based on the inputs from the xilskey_efuseps_zynqmp_input.h file.
 - xilskey_bbramps_zynqmp_example.c: This file contains the example code to program and verify BBRAM key of Zynq UltraScale+ MPSoC. Default is zero. You can modify this key on top of the file.
 - xilskey_bbram_ultrascale_example.c: This file contains example code to program and verify BBRAM key of UltraScale.

Note: Programming and verification of BBRAM key cannot be done separately.

- xilskey_bbram_ultrascale_input.h: This file contains all the preprocessor directives you need to provide. In this file, specify BBRAM AES key or Obfuscated AES key to be programmed, DPA protection enable and, GPIO pins and channels connected to MASTER JTAG primitive.
- xilskey_puf_registration.c: This file contains all the PUF related code. This example illustrates PUF registration and generating black key and programming eFUSE with PUF helper data, CHash and Auxiliary data along with the Black key.
- xilskey_puf_registration.h: This file contains all the preprocessor directives based on which read/write the eFUSE bits and Syndrome data generation. More explanation of each directive is provided in the following sections.



CAUTION! *Ensure that you enter the correct information before writing or 'burning' eFUSE bits. Once burned, they cannot be changed. The BBRAM key can be programmed any number of times.*

Note: POR reset is required for the eFUSE values to be recognized.

BBRAM PL API

This section provides a linked summary and detailed descriptions of the battery-backed RAM (BBRAM) APIs of Zynq PL and UltraScale devices.

Example Usage

- Zynq BBRAM PL example usage:
 - The Zynq BBRAM PL example application should contain the `xilskey_bbram_example.c` and `xilskey_input.h` files.
 - You should provide user configurable parameters in the `xilskey_input.h` file. For more information, refer [Zynq User-Configurable PL BBRAM Parameters](#).
- UltraScale BBRAM example usage:
 - The UltraScale BBRAM example application should contain the `xilskey_bbram_ultrascale_input.h` and `xilskey_bbram_ultrascale_example.c` files.
 - You should provide user configurable parameters in the `xilskey_bbram_ultrascale_input.h` file. For more information, refer [UltraScale or UltraScale+ User-Configurable BBRAM PL Parameters](#).

Note: It is assumed that you have set up your hardware prior to working on the example application. For more information, refer [Hardware Setup](#).

Table 288: Quick Function Reference

Type	Name	Arguments
int	XilSKey_Bbram_Program	XilSKey_Bbram * InstancePtr
int	XilSKey_Bbram_JTAGServerInit	void

Functions

XilSKey_Bbram_Program

This function implements the BBRAM algorithm for programming and verifying key.

The program and verify will only work together in and in that order.

Note: This function will program BBRAM of Ultrascale and Zynq as well.

Prototype

```
int XilSKey_Bbram_Program(XilSKey_Bbram *InstancePtr);
```

Parameters

The following table lists the `XilSKey_Bbram_Program` function arguments.

Table 289: XilSKey_Bbram_Program Arguments

Type	Name	Description
XilSKey_Bbram *	InstancePtr	Pointer to XilSKey_Bbram

Returns

XilSKey_Bbram_JTAGServerInit

Prototype

```
int XilSKey_Bbram_JTAGServerInit(XilSKey_Bbram *InstancePtr);
```

Zynq UltraScale+ MPSoC BBRAM PS API

This section provides a linked summary and detailed descriptions of the battery-backed RAM (BBRAM) APIs for Zynq UltraScale+ MPSoC devices.

Example Usage

- The Zynq UltraScale+ MPSoC example application should contain the `xilskey_bbramps_zynqmp_example.c` file.
- User configurable key can be modified in the same file (`xilskey_bbramps_zynqmp_example.c`), at the `XSK_ZYNQMP_BBRAMPS_AES_KEY` macro.

Table 290: Quick Function Reference

Type	Name	Arguments
u32	XilSKey_ZynqMp_Bbram_Program	u32 * AesKey
u32	XilSKey_ZynqMp_Bbram_Zeroise	None.

Functions

XilSKey_ZynqMp_Bbram_Program

This function implements the BBRAM programming and verifying the key written.

Program and verification of AES will work only together. CRC of the provided key will be calculated internally and verified after programming.

Prototype

```
u32 XilSKey_ZynqMp_Bbram_Program(u32 *AesKey);
```

Parameters

The following table lists the `XilSKey_ZynqMp_Bbram_Program` function arguments.

Table 291: XilSKey_ZynqMp_Bbram_Program Arguments

Type	Name	Description
u32 *	AesKey	Pointer to the key which has to be programmed.

Returns

- Error code from `XskZynqMp_Ps_Bbram_ErrorCodes` enum if it fails
- `XST_SUCCESS` if programming is done.

XilSKey_ZynqMp_Bbram_Zeroise

This function zeroize's Bbram Key.

Note: BBRAM key will be zeroized.

Prototype

```
u32 XilSKey_ZynqMp_Bbram_Zeroise(void);
```

Parameters

The following table lists the `XilSKey_ZynqMp_Bbram_Zeroise` function arguments.

Table 292: XilSKey_ZynqMp_Bbram_Zeroise Arguments

Type	Name	Description
Commented parameter None. does not exist in function <code>XilSKey_ZynqMp_Bbram_Zeroise</code> .	None.	

Returns

None.

Zynq eFUSE PS API

This chapter provides a linked summary and detailed descriptions of the Zynq eFUSE PS APIs.

Example Usage

- The Zynq eFUSE PS example application should contain the `xilsky_efuse_example.c` and the `xilsky_input.h` files.
- There is no need of any hardware setup. By default, both the eFUSE PS and PL are enabled in the application. You can comment 'XSK_EFUSEPL_DRIVER' to execute only the PS. For more details, refer [Zynq User-Configurable PS eFUSE Parameters](#).

Table 293: Quick Function Reference

Type	Name	Arguments
u32	XilSKey_EfusePs_Write	InstancePtr
u32	XilSKey_EfusePs_Read	InstancePtr
u32	XilSKey_EfusePs_ReadStatus	XilSKey_EPs * InstancePtr u32 * StatusBits

Functions

XilSKey_EfusePs_Write

PS eFUSE interface functions.

PS eFUSE interface functions.

Note: When called, this Initializes the timer, XADC subsystems. Unlocks the PS eFUSE controller. Configures the PS eFUSE controller. Writes the hash and control bits if requested. Programs the PS eFUSE to enable the RSA authentication if requested. Locks the PS eFUSE controller. Returns an error, if the reference clock frequency is not in between 20 and 60 MHz or if the system not in a position to write the requested PS eFUSE bits (because the bits are already written or not allowed to write) or if the temperature and voltage are not within range

Prototype

```
u32 XilSKey_EfusePs_Write(XilSKey_EPs *PsInstancePtr);
```

Parameters

The following table lists the `XilSKey_EfusePs_Write` function arguments.

Table 294: XilSKey_EfusePs_Write Arguments

Type	Name	Description
Commented parameter InstancePtr does not exist in function XilSKey_EfusePs_Write.	InstancePtr	Pointer to the PsEfuseHandle which describes which PS eFUSE bit should be burned.

Returns

- XST_SUCCESS.
- In case of error, value is as defined in `xilskey_utils.h` Error value is a combination of Upper 8 bit value and Lower 8 bit value. For example, 0x8A03 should be checked in `error.h` as 0x8A00 and 0x03. Upper 8 bit value signifies the major error and lower 8 bit values tells more precisely.

XilSKey_EfusePs_Read

This function is used to read the PS eFUSE.

Note: When called: This API initializes the timer, XADC subsystems. Unlocks the PS eFUSE Controller. Configures the PS eFUSE Controller and enables read-only mode. Reads the PS eFUSE (Hash Value), and enables read-only mode. Locks the PS eFUSE Controller. Returns an error, if the reference clock frequency is not in between 20 and 60MHz. or if unable to unlock PS eFUSE controller or requested address corresponds to restricted bits. or if the temperature and voltage are not within range

Prototype

```
u32 XilSKey_EfusePs_Read(XilSKey_EPs *PsInstancePtr);
```

Parameters

The following table lists the `XilSKey_EfusePs_Read` function arguments.

Table 295: XilSKey_EfusePs_Read Arguments

Type	Name	Description
Commented parameter InstancePtr does not exist in function XilSKey_EfusePs_Read.	InstancePtr	Pointer to the PsEfuseHandle which describes which PS eFUSE should be burned.

Returns

- XST_SUCCESS no errors occurred.

- In case of error, value is as defined in `xilskey_utils.h`. Error value is a combination of Upper 8 bit value and Lower 8 bit value. For example, `0x8A03` should be checked in `error.h` as `0x8A00` and `0x03`. Upper 8 bit value signifies the major error and lower 8 bit values tells more precisely.

XilSkey_EfusePs_ReadStatus

This function is used to read the PS efuse status register.

Note: This API unlocks the controller and reads the Zynq PS eFUSE status register.

Prototype

```
u32 XilSKey_EfusePs_ReadStatus(XilSKey_EPs *InstancePtr, u32 *StatusBits);
```

Parameters

The following table lists the `XilSKey_EfusePs_ReadStatus` function arguments.

Table 296: XilSKey_EfusePs_ReadStatus Arguments

Type	Name	Description
XilSKey_EPs *	InstancePtr	Pointer to the PS eFUSE instance.
u32 *	StatusBits	Buffer to store the status register read.

Returns

- XST_SUCCESS.
- XST_FAILURE

Zynq UltraScale+ MPSoC eFUSE PS API

This chapter provides a linked summary and detailed descriptions of the Zynq MPSoC UltraScale+ eFUSE PS APIs.

Example Usage

- For programming eFUSES other than the PUF, the Zynq UltraScale+ MPSoC example application should contain the `xilskey_efuseps_zynqmp_example.c` and the `xilskey_efuseps_zynqmp_input.h` files.
- For PUF registration, programming PUF helper data, AUX, chash, and black key, the Zynq UltraScale+ MPSoC example application should contain the `xilskey_puf_registration.c` and the `xilskey_puf_registration.h` files.

- For more details on the user configurable parameters, refer [Zynq UltraScale+ MPSoC User-Configurable PS eFUSE Parameters](#) and [Zynq UltraScale+ MPSoC User-Configurable PS PUF Parameters](#).

Table 297: Quick Function Reference

Type	Name	Arguments
u32	XiISKey_ZynqMp_EfusePs_CheckAesKeyCrc	u32 CrcValue
u32	XiISKey_ZynqMp_EfusePs_ReadUserFuse	u32 * UseFusePtr u8 UserFuse_Num u8 ReadOption
u32	XiISKey_ZynqMp_EfusePs_ReadPpk0Hash	u32 * Ppk0Hash u8 ReadOption
u32	XiISKey_ZynqMp_EfusePs_ReadPpk1Hash	u32 * Ppk1Hash u8 ReadOption
u32	XiISKey_ZynqMp_EfusePs_ReadSpkId	u32 * SpkId u8 ReadOption
void	XiISKey_ZynqMp_EfusePs_ReadDna	u32 * DnaRead
u32	XiISKey_ZynqMp_EfusePs_ReadSecCtrlBits	XiISKey_SecCtrlBits * ReadBackSecCtrlBits u8 ReadOption
u32	XiISKey_ZynqMp_EfusePs_CacheLoad	void
u32	XiISKey_ZynqMp_EfusePs_Write	XiISKey_ZynqMpEPs * InstancePtr
u32	XiISKey_ZynqMpEfuseAccess	void
u32	XiISKey_ZynqMp_EfusePs_WritePufHelpData	XiISKey_Puf * InstancePtr
u32	XiISKey_ZynqMp_EfusePs_ReadPufHelpData	u32 * Address
u32	XiISKey_ZynqMp_EfusePs_WritePufHash	XiISKey_Puf * InstancePtr
u32	XiISKey_ZynqMp_EfusePs_ReadPufHash	u32 * Address u8 ReadOption
u32	XiISKey_ZynqMp_EfusePs_WritePufAux	XiISKey_Puf * InstancePtr

Table 297: Quick Function Reference (cont'd)

Type	Name	Arguments
u32	XiISKey_ZynqMp_EfusePs_ReadPufAux	u32 * Address u8 ReadOption
u32	XiISKey_Write_Puf_EfusePs_SecureBits	XiISKey_Puf_Secure * WriteSecureBits
u32	XiISKey_Read_Puf_EfusePs_SecureBits	SecureBits u8 ReadOption
u32	XiISKey_Puf_Registration	XiISKey_Puf * InstancePtr
u32	XiISKey_Puf_Regeneration	XiISKey_Puf * InstancePtr

Functions

XiISKey_ZynqMp_EfusePs_CheckAesKeyCrc

This function performs the CRC check of AES key.

Note: For Calculating the CRC of the AES key use the [XiISKey_CrcCalculation\(\)](#) function or [XiISKey_CrcCalculation_AesKey\(\)](#) function

Prototype

```
u32 XiISKey_ZynqMp_EfusePs_CheckAesKeyCrc(u32 CrcValue);
```

Parameters

The following table lists the `XiISKey_ZynqMp_EfusePs_CheckAesKeyCrc` function arguments.

Table 298: XiISKey_ZynqMp_EfusePs_CheckAesKeyCrc Arguments

Type	Name	Description
u32	CrcValue	A 32 bit CRC value of an expected AES key.

Returns

- XST_SUCCESS on successful CRC check.
- ErrorCode on failure

XilSKey_ZynqMp_EfusePs_ReadUserFuse

This function is used to read a user fuse from the eFUSE or cache.

Note: It is highly recommended to read from eFuse cache. Because reading from efuse may reduce the life of the efuse. And Cache reload is required for obtaining updated values for ReadOption 0.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_ReadUserFuse(u32 *UseFusePtr, u8 UserFuse_Num,
u8 ReadOption);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_ReadUserFuse` function arguments.

Table 299: XilSKey_ZynqMp_EfusePs_ReadUserFuse Arguments

Type	Name	Description
u32 *	UseFusePtr	Pointer to an array which holds the readback user fuse.
u8	UserFuse_Num	A variable which holds the user fuse number. Range is (User fuses: 0 to 7)
u8	ReadOption	Indicates whether or not to read from the actual eFUSE array or from the eFUSE cache. <ul style="list-style-type: none"> • 0(XSK_EFUSEPS_READ_FROM_CACHE) Reads from eFUSE cache • 1(XSK_EFUSEPS_READ_FROM_EFUSE) Reads from eFUSE array

Returns

- XST_SUCCESS on successful read
- ErrorCode on failure

XilSKey_ZynqMp_EfusePs_ReadPpk0Hash

This function is used to read the PPK0 hash from an eFUSE or eFUSE cache.

Note: It is highly recommended to read from eFuse cache. Because reading from efuse may reduce the life of the efuse. And Cache reload is required for obtaining updated values for ReadOption 0.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_ReadPpk0Hash(u32 *Ppk0Hash, u8 ReadOption);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_ReadPpk0Hash` function arguments.

Table 300: XilSKey_ZynqMp_EfusePs_ReadPpk0Hash Arguments

Type	Name	Description
u32 *	Ppk0Hash	A pointer to an array which holds the readback PPK0 hash.
u8	ReadOption	Indicates whether or not to read from the actual eFUSE array or from the eFUSE cache. <ul style="list-style-type: none"> 0(XSK_EFUSEPS_READ_FROM_CACHE) Reads from eFUSE cache 1(XSK_EFUSEPS_READ_FROM_EFUSE) Reads from eFUSE array

Returns

- XST_SUCCESS on successful read
- ErrorCode on failure

XilSKey_ZynqMp_EfusePs_ReadPpk1Hash

This function is used to read the PPK1 hash from eFUSE or cache.

Note: It is highly recommended to read from eFuse cache. Because reading from efuse may reduce the life of the efuse. And Cache reload is required for obtaining updated values for ReadOption 0.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_ReadPpk1Hash(u32 *Ppk1Hash, u8 ReadOption);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_ReadPpk1Hash` function arguments.

Table 301: XilSKey_ZynqMp_EfusePs_ReadPpk1Hash Arguments

Type	Name	Description
u32 *	Ppk1Hash	Pointer to an array which holds the readback PPK1 hash.
u8	ReadOption	Indicates whether or not to read from the actual eFUSE array or from the eFUSE cache. <ul style="list-style-type: none"> 0(XSK_EFUSEPS_READ_FROM_CACHE) Reads from eFUSE cache 1(XSK_EFUSEPS_READ_FROM_EFUSE) Reads from eFUSE array

Returns

- XST_SUCCESS on successful read
- ErrorCode on failure

XilSKey_ZynqMp_EfusePs_ReadSpkId

This function is used to read SPKID from eFUSE or cache based on user's read option.

Note: It is highly recommended to read from eFuse cache. Because reading from efuse may reduce the life of the efuse. And Cache reload is required for obtaining updated values for ReadOption 0.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_ReadSpkId(u32 *SpkId, u8 ReadOption);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_ReadSpkId` function arguments.

Table 302: XilSKey_ZynqMp_EfusePs_ReadSpkId Arguments

Type	Name	Description
u32 *	SpkId	Pointer to a 32 bit variable which holds SPK ID.
u8	ReadOption	Indicates whether or not to read from the actual eFUSE array or from the eFUSE cache. <ul style="list-style-type: none"> • 0(XSK_EFUSEPS_READ_FROM_CACHE) Reads from eFUSE cache • 1(XSK_EFUSEPS_READ_FROM_EFUSE) Reads from eFUSE array

Returns

- XST_SUCCESS on successful read
- ErrorCode on failure

XilSKey_ZynqMp_EfusePs_ReadDna

This function is used to read DNA from eFUSE.

Prototype

```
void XilSKey_ZynqMp_EfusePs_ReadDna(u32 *DnaRead);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_ReadDna` function arguments.

Table 303: XilSKey_ZynqMp_EfusePs_ReadDna Arguments

Type	Name	Description
u32 *	DnaRead	Pointer to an array of 3 x u32 words which holds the readback DNA.

Returns

None.

XilSKey_ZynqMp_EfusePs_ReadSecCtrlBits

This function is used to read the PS eFUSE secure control bits from cache or eFUSE based on user input provided.

Note: It is highly recommended to read from eFuse cache. Because reading from efuse may reduce the life of the efuse. And Cache reload is required for obtaining updated values for ReadOption 0.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_ReadSecCtrlBits(XilSKey_SecCtrlBits
*ReadBackSecCtrlBits, u8 ReadOption);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_ReadSecCtrlBits` function arguments.

Table 304: XilSKey_ZynqMp_EfusePs_ReadSecCtrlBits Arguments

Type	Name	Description
XilSKey_SecCtrlBits *	ReadBackSecCtrlBits	Pointer to the XilSKey_SecCtrlBits which holds the read secure control bits.
u8	ReadOption	Indicates whether or not to read from the actual eFUSE array or from the eFUSE cache. <ul style="list-style-type: none"> 0(XSK_EFUSEPS_READ_FROM_CACHE) Reads from eFUSE cache 1(XSK_EFUSEPS_READ_FROM_EFUSE) Reads from eFUSE array

Returns

- XST_SUCCESS if reads successfully
- XST_FAILURE if reading is failed

XilSKey_ZynqMp_EfusePs_CacheLoad

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_CacheLoad(void);
```

XilSKey_ZynqMp_EfusePs_Write

This function is used to program the PS eFUSE of ZynqMP, based on user inputs.

Note: After eFUSE programming is complete, the cache is automatically reloaded so all programmed eFUSE bits can be directly read from cache.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_Write(XilSKey_ZynqMpEPs *InstancePtr);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_Write` function arguments.

Table 305: XilSKey_ZynqMp_EfusePs_Write Arguments

Type	Name	Description
XilSKey_ZynqMpEPs *	InstancePtr	Pointer to the XilSKey_ZynqMpEPs.

Returns

- XST_SUCCESS if programs successfully.
- Errorcode on failure

XilSkey_ZynqMpEfuseAccess

Prototype

```
u32 XilSkey_ZynqMpEfuseAccess(const u32 AddrHigh, const u32 AddrLow);
```

XilSKey_ZynqMp_EfusePs_WritePufHelprData

This function programs the PS eFUSEs with the PUF helper data.

Note: To generate PufSyndromeData please use XilSKey_Puf_Registration API

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_WritePufHelprData(XilSKey_Puf *InstancePtr);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_WritePufHelprData` function arguments.

Table 306: XilSKey_ZynqMp_EfusePs_WritePufHelprData Arguments

Type	Name	Description
XilSKey_Puf *	InstancePtr	Pointer to the XilSKey_Puf instance.

Returns

- XST_SUCCESS if programs successfully.
- Errorcode on failure

XilSKey_ZynqMp_EfusePs_ReadPufHelprData

This function reads the PUF helper data from eFUSE.

Note: This function only reads from eFUSE non-volatile memory. There is no option to read from Cache.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_ReadPufHelprData(u32 *Address);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_ReadPufHelprData` function arguments.

Table 307: XilSKey_ZynqMp_EfusePs_ReadPufHelprData Arguments

Type	Name	Description
u32 *	Address	Pointer to data array which holds the PUF helper data read from eFUSES.

Returns

- XST_SUCCESS if reads successfully.
- Errorcode on failure.

XilSKey_ZynqMp_EfusePs_WritePufChash

This function programs eFUSE with CHash value.

Note: To generate the CHash value, please use `XilSKey_Puf_Registration` function.

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_WritePufChash(XilSKey_Puf *InstancePtr);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_WritePufChash` function arguments.

Table 308: XilSKey_ZynqMp_EfusePs_WritePufChash Arguments

Type	Name	Description
XilSKey_Puf *	InstancePtr	Pointer to the XilSKey_Puf instance.

Returns

- XST_SUCCESS if chash is programmed successfully.
- An Error code on failure

XilSKey_ZynqMp_EfusePs_ReadPufChash

This function reads eFUSE PUF CHash data from the eFUSE array or cache based on the user read option.

Note: Cache reload is required for obtaining updated values for reading from cache..

Prototype

```
u32 XilSKey_ZynqMp_EfusePs_ReadPufChash(u32 *Address, u8 ReadOption);
```

Parameters

The following table lists the `XilSKey_ZynqMp_EfusePs_ReadPufChash` function arguments.

Table 309: XilSKey_ZynqMp_EfusePs_ReadPufChash Arguments

Type	Name	Description
u32 *	Address	Pointer which holds the read back value of the chash.
u8	ReadOption	Indicates whether or not to read from the actual eFUSE array or from the eFUSE cache. <ul style="list-style-type: none"> • 0(XSK_EFUSEPS_READ_FROM_CACHE) Reads from cache • 1(XSK_EFUSEPS_READ_FROM_EFUSE) Reads from eFUSE array

Returns

- XST_SUCCESS if programs successfully.
- Errorcode on failure

XilSkey_ZynqMp_EfusePs_WritePufAux

This function programs eFUSE PUF auxiliary data.

Note: To generate auxiliary data, please use XilSkey_Puf_Registration function.

Prototype

```
u32 XilSkey_ZynqMp_EfusePs_WritePufAux(XilSkey_Puf *InstancePtr);
```

Parameters

The following table lists the `XilSkey_ZynqMp_EfusePs_WritePufAux` function arguments.

Table 310: XilSkey_ZynqMp_EfusePs_WritePufAux Arguments

Type	Name	Description
XilSkey_Puf *	InstancePtr	Pointer to the XilSkey_Puf instance.

Returns

- XST_SUCCESS if the eFUSE is programmed successfully.
- Errorcode on failure

XilSkey_ZynqMp_EfusePs_ReadPufAux

This function reads eFUSE PUF auxiliary data from eFUSE array or cache based on user read option.

Note: Cache reload is required for obtaining updated values for reading from cache.

Prototype

```
u32 XilSkey_ZynqMp_EfusePs_ReadPufAux(u32 *Address, u8 ReadOption);
```

Parameters

The following table lists the `XilSkey_ZynqMp_EfusePs_ReadPufAux` function arguments.

Table 311: XilSKey_ZynqMp_EfusePs_ReadPufAux Arguments

Type	Name	Description
u32 *	Address	Pointer which holds the read back value of PUF's auxiliary data.
u8	ReadOption	Indicates whether or not to read from the actual eFUSE array or from the eFUSE cache. <ul style="list-style-type: none"> 0(XSK_EFUSEPS_READ_FROM_CACHE) Reads from cache 1(XSK_EFUSEPS_READ_FROM_EFUSE) Reads from eFUSE array

Returns

- XST_SUCCESS if PUF auxiliary data is read successfully.
- Errorcode on failure

XilSKey_Write_Puf_EfusePs_SecureBits

This function programs the eFUSE PUF secure bits.

Prototype

```
u32 XilSKey_Write_Puf_EfusePs_SecureBits(XilSKey_Puf_Secure
*WriteSecureBits);
```

Parameters

The following table lists the `XilSKey_Write_Puf_EfusePs_SecureBits` function arguments.

Table 312: XilSKey_Write_Puf_EfusePs_SecureBits Arguments

Type	Name	Description
XilSKey_Puf_Secure *	WriteSecureBits	Pointer to the XilSKey_Puf_Secure structure

Returns

- XST_SUCCESS if eFUSE PUF secure bits are programmed successfully.
- Errorcode on failure.

XilSKey_Read_Puf_EfusePs_SecureBits

This function is used to read the PS eFUSE PUF secure bits from cache or from eFUSE array.

Prototype

```
u32 XilSKey_Read_Puf_EfusePs_SecureBits(XilSKey_Puf_Secure *SecureBitsRead,
u8 ReadOption);
```

Parameters

The following table lists the `XilSKey_Read_Puf_EfusePs_SecureBits` function arguments.

Table 313: XilSKey_Read_Puf_EfusePs_SecureBits Arguments

Type	Name	Description
Commented parameter SecureBits does not exist in function <code>XilSKey_Read_Puf_EfusePs_SecureBits</code> .	SecureBits	Pointer to the <code>XilSKey_Puf_Secure</code> structure which holds the read eFUSE secure bits from the PUF.
u8	ReadOption	Indicates whether or not to read from the actual eFUSE array or from the eFUSE cache. <ul style="list-style-type: none"> 0(<code>XSK_EFUSEPS_READ_FROM_CACHE</code>) Reads from cache 1(<code>XSK_EFUSEPS_READ_FROM_EFUSE</code>) Reads from eFUSE array

Returns

- `XST_SUCCESS` if reads successfully.
- Errorcode on failure.

XilSKey_Puf_Registration

This function performs registration of PUF which generates a new KEK and associated CHash, Auxiliary and PUF-syndrome data which are unique for each silicon.

Note: With the help of generated PUF syndrome data, it will be possible to re-generate same PUF KEK.

Prototype

```
u32 XilSKey_Puf_Registration(XilSKey_Puf *InstancePtr);
```

Parameters

The following table lists the `XilSKey_Puf_Registration` function arguments.

Table 314: XilSKey_Puf_Registration Arguments

Type	Name	Description
<code>XilSKey_Puf *</code>	InstancePtr	Pointer to the <code>XilSKey_Puf</code> instance.

Returns

- XST_SUCCESS if registration/re-registration was successful.
- ERROR if registration was unsuccessful

XilSkey_Puf_Regeneration

This function regenerates the PUF data so that the PUF's output can be used as the key source to the AES-GCM hardware cryptographic engine.

Prototype

```
u32 XilSkey_Puf_Regeneration(XilSkey_Puf *InstancePtr);
```

Parameters

The following table lists the `XilSkey_Puf_Regeneration` function arguments.

Table 315: XilSkey_Puf_Regeneration Arguments

Type	Name	Description
XilSkey_Puf *	InstancePtr	is a pointer to the XilSkey_Puf instance.

Returns

- XST_SUCCESS if regeneration was successful.
- ERROR if regeneration was unsuccessful

eFUSE PL API

This chapter provides a linked summary and detailed descriptions of the eFUSE APIs of Zynq eFUSE PL and UltraScale eFUSE.

Example Usage

- The Zynq eFUSE PL and UltraScale example application should contain the `xilskey_efuse_example.c` and the `xilskey_input.h` files.
- By default, both the eFUSE PS and PL are enabled in the application. You can comment 'XSK_EFUSEPL_DRIVER' to execute only the PS.
- For UltraScale, it is mandatory to comment ``XSK_EFUSEPS_DRIVER` else the example will generate an error.

- For more details on the user configurable parameters, refer [Zynq User-Configurable PL eFUSE Parameters](#) and [UltraScale or UltraScale+ User-Configurable PL eFUSE Parameters](#).
- Requires hardware setup to program PL eFUSE of Zynq or UltraScale.

Table 316: Quick Function Reference

Type	Name	Arguments
u32	XilSKey_EfusePl_SystemInit	XilSKey_EP1 * InstancePtr
u32	XilSKey_EfusePl_Program	InstancePtr
u32	XilSKey_EfusePl_ReadStatus	XilSKey_EP1 * InstancePtr u32 * StatusBits
u32	XilSKey_EfusePl_ReadKey	XilSKey_EP1 * InstancePtr

Functions

XilSKey_EfusePl_SystemInit

Note: Updates the global variable ErrorCode with error code(if any).

Prototype

```
u32 XilSKey_EfusePl_SystemInit(XilSKey_EP1 *InstancePtr);
```

Parameters

The following table lists the `XilSKey_EfusePl_SystemInit` function arguments.

Table 317: XilSKey_EfusePl_SystemInit Arguments

Type	Name	Description
XilSKey_EP1 *	InstancePtr	- Input data to be written to PL eFUSE

Returns

XilSKey_EfusePl_Program

Programs PL eFUSE with input data given through InstancePtr.

Note: When this API is called: Initializes the timer, XADC/xsysmon and JTAG server subsystems. Returns an error in the following cases, if the reference clock frequency is not in the range or if the PL DAP ID is not identified, if the system is not in a position to write the requested PL eFUSE bits (because the bits are already written or not allowed to write) if the temperature and voltage are not within range.

Prototype

```
u32 XilSKey_EfusePl_Program(XilSKey_EPl *PlInstancePtr);
```

Parameters

The following table lists the `XilSKey_EfusePl_Program` function arguments.

Table 318: XilSKey_EfusePl_Program Arguments

Type	Name	Description
Commented parameter InstancePtr does not exist in function XilSKey_EfusePl_Program.	InstancePtr	Pointer to PL eFUSE instance which holds the input data to be written to PL eFUSE.

Returns

- XST_FAILURE - In case of failure
- XST_SUCCESS - In case of Success

XilSKey_EfusePl_ReadStatus

Reads the PL efuse status bits and gets all secure and control bits.

Prototype

```
u32 XilSKey_EfusePl_ReadStatus(XilSKey_EPl *InstancePtr, u32 *StatusBits);
```

Parameters

The following table lists the `XilSKey_EfusePl_ReadStatus` function arguments.

Table 319: XilSKey_EfusePl_ReadStatus Arguments

Type	Name	Description
<code>XilSKey_EPl *</code>	InstancePtr	Pointer to PL eFUSE instance.
<code>u32 *</code>	StatusBits	Buffer to store the status bits read.

Returns

XilSKey_EfusePl_ReadKey

Reads the PL efuse keys and stores them in the corresponding arrays in instance structure.

Note: This function initializes the timer, XADC and JTAG server subsystems, if not already done so. In Zynq - Reads AES key and User keys. In Ultrascale - Reads 32 bit and 128 bit User keys and RSA hash But AES key cannot be read directly it can be verified with CRC check (for that we need to update the instance with 32 bit CRC value, API updates whether provided CRC value is matched with actuals or not). To calculate the CRC of expected AES key one can use any of the following APIs [XilSKey_CrcCalculation\(\)](#) or [XilSKey_CrcCalculation_AesKey\(\)](#)

Prototype

```
u32 XilSKey_EfusePl_ReadKey(XilSKey_EPl *InstancePtr);
```

Parameters

The following table lists the `XilSKey_EfusePl_ReadKey` function arguments.

Table 320: XilSKey_EfusePl_ReadKey Arguments

Type	Name	Description
XilSKey_EPl *	InstancePtr	Pointer to PL eFUSE instance.

Returns

CRC Calculation API

This chapter provides a linked summary and detailed descriptions of the CRC calculation APIs. For UltraScale and Zynq UltraScale+ MPSoC devices, the programmed AES cannot be read back. The programmed AES key can only be verified by reading the CRC value of AES key.

Table 321: Quick Function Reference

Type	Name	Arguments
u32	XilSKey_CrcCalculation	u8 * Key
u32	XilSKey_CrcCalculation_AesKey	u8 * Key

Functions

XilSkey_CrcCalculation

This function Calculates CRC value based on hexadecimal string passed.

Note: If the length of the string provided is less than 64, this function appends the string with zeros. For calculation of AES key's CRC one can use u32 `XilSkey_CrcCalculation(u8 *Key)` API or reverse polynomial 0x82F63B78.

Prototype

```
u32 XilSkey_CrcCalculation(u8 *Key);
```

Parameters

The following table lists the `XilSkey_CrcCalculation` function arguments.

Table 322: XilSkey_CrcCalculation Arguments

Type	Name	Description
u8 *	Key	Pointer to the string contains AES key in hexadecimal of length less than or equal to 64.

Returns

- On Success returns the Crc of AES key value.
- On failure returns the error code when string length is greater than 64

XilSkey_CrcCalculation_AesKey

Calculates CRC value of the provided key.

Key should be provided in hexa buffer.

Prototype

```
u32 XilSkey_CrcCalculation_AesKey(u8 *Key);
```

Parameters

The following table lists the `XilSkey_CrcCalculation_AesKey` function arguments.

Table 323: XilSkey_CrcCalculation_AesKey Arguments

Type	Name	Description
u8 *	Key	Pointer to an array of 32 bytes, which holds an AES key.

Returns

Crc of provided AES key value. To calculate CRC on the AES key in string format please use XilSkey_CrcCalculation.

User-Configurable Parameters

This section provides detailed descriptions of the various user configurable parameters.

Zynq User-Configurable PS eFUSE Parameters

Define the XSK_EFUSEPS_DRIVER macro to use the PS eFUSE. After defining the macro, provide the inputs defined with XSK_EFUSEPS_DRIVER to burn the bits in PS eFUSE. If the bit is to be burned, define the macro as TRUE; otherwise define the macro as FALSE. For details, refer the following table.

Macro Name	Description
XSK_EFUSEPS_ENABLE_WRITE_PROTECT	<p>Default = FALSE.</p> <p>TRUE to burn the write-protect bits in eFUSE array. Write protect has two bits. When either of the bits is burned, it is considered write-protected. So, while burning the write-protected bits, even if one bit is blown, write API returns success. As previously mentioned, POR reset is required after burning for write protection of the eFUSE bits to go into effect. It is recommended to do the POR reset after write protection. Also note that, after write-protect bits are burned, no more eFUSE writes are possible.</p> <p>If the write-protect macro is TRUE with other macros, write protect is burned in the last iteration, after burning all the defined values, so that for any error while burning other macros will not effect the total eFUSE array.</p> <p>FALSE does not modify the write-protect bits.</p>
XSK_EFUSEPS_ENABLE_RSA_AUTH	<p>Default = FALSE.</p> <p>Use TRUE to burn the RSA enable bit in the PS eFUSE array. After enabling the bit, every successive boot must be RSA-enabled apart from JTAG. Before burning (blowing) this bit, make sure that eFUSE array has the valid PPK hash. If the PPK hash burning is enabled, only after writing the hash successfully, RSA enable bit will be blown. For the RSA enable bit to take effect, POR reset is required. FALSE does not modify the RSA enable bit.</p>

Macro Name	Description
XSK_EFUSEPS_ENABLE_ROM_128K_CRC	Default = FALSE. TRUE burns the ROM 128K CRC bit. In every successive boot, BootROM calculates 128k CRC. FALSE does not modify the ROM CRC 128K bit.
XSK_EFUSEPS_ENABLE_RSA_KEY_HASH	Default = FALSE. TRUE burns (blows) the eFUSE hash, that is given in XSK_EFUSEPS_RSA_KEY_HASH_VALUE when write API is used. TRUE reads the eFUSE hash when the read API is used and is read into structure. FALSE ignores the provided value.
XSK_EFUSEPS_RSA_KEY_HASH_VALUE	Default = The specified value is converted to a hexadecimal buffer and written into the PS eFUSE array when the write API is used. This value should be the Primary Public Key (PPK) hash provided in string format. The buffer must be 64 characters long: valid characters are 0-9, a-f, and A-F. Any other character is considered an invalid string and will not burn RSA hash. When the XilSkey_EfusePs_Write() API is used, the RSA hash is written, and the XSK_EFUSEPS_ENABLE_RSA_KEY_HASH must have a value of TRUE.
XSK_EFUSEPS_DISABLE_DFT_JTAG	Default = FALSE TRUE disables DFT JTAG permanently. FALSE will not modify the eFuse PS DFT JTAG disable bit.
XSK_EFUSEPS_DISABLE_DFT_MODE	Default = FALSE TRUE disables DFT mode permanently. FALSE will not modify the eFuse PS DFT mode disable bit.

Zynq User-Configurable PL eFUSE Parameters

Define the XSK_EFUSEPL_DRIVER macro to use the PL eFUSE. After defining the macro, provide the inputs defined with XSK_EFUSEPL_DRIVER to burn the bits in PL eFUSE bits. If the bit is to be burned, define the macro as TRUE; otherwise define the macro as FALSE. The table below lists the user-configurable PL eFUSE parameters for Zynq devices.

Macro Name	Description
XSK_EFUSEPL_FORCE_PCYCLE_RECONFIG	Default = FALSE If the value is set to TRUE, then the part has to be power-cycled to be reconfigured. FALSE does not set the eFUSE control bit.
XSK_EFUSEPL_DISABLE_KEY_WRITE	Default = FALSE TRUE disables the eFUSE write to FUSE_AES and FUSE_USER blocks. FALSE does not affect the eFUSE bit.
XSK_EFUSEPL_DISABLE_AES_KEY_READ	Default = FALSE TRUE disables the write to FUSE_AES and FUSE_USER key and disables the read of FUSE_AES. FALSE does not affect the eFUSE bit.

Macro Name	Description
XSK_EFUSEPL_DISABLE_USER_KEY_READ	Default = FALSE. TRUE disables the write to FUSE_AES and FUSE_USER key and disables the read of FUSE_USER. FALSE does not affect the eFUSE bit.
XSK_EFUSEPL_DISABLE_FUSE_CNTRL_WRITE	Default = FALSE. TRUE disables the eFUSE write to FUSE_CTRL block. FALSE does not affect the eFUSE bit.
XSK_EFUSEPL_FORCE_USE_AES_ONLY	Default = FALSE. TRUE forces the use of secure boot with eFUSE AES key only. FALSE does not affect the eFUSE bit.
XSK_EFUSEPL_DISABLE_JTAG_CHAIN	Default = FALSE. TRUE permanently disables the Zynq ARM DAP and PL TAP. FALSE does not affect the eFUSE bit.
XSK_EFUSEPL_BBRAM_KEY_DISABLE	Default = FALSE. TRUE forces the eFUSE key to be used if booting Secure Image. FALSE does not affect the eFUSE bit.

MIO Pins for Zynq PL eFUSE JTAG Operations

The table below lists the MIO pins for Zynq PL eFUSE JTAG operations. You can change the listed pins at your discretion.

Note: The pin numbers listed in the table below are examples. You must assign appropriate pin numbers as per your hardware design.

Pin Name	Pin Number
XSK_EFUSEPL_MIO_JTAG_TDI	(17)
XSK_EFUSEPL_MIO_JTAG_TDO	(21)
XSK_EFUSEPL_MIO_JTAG_TCK	(19)
XSK_EFUSEPL_MIO_JTAG_TMS	(20)

MUX Selection Pin for Zynq PL eFUSE JTAG Operations

The table below lists the MUX selection pin.

Pin Name	Pin Number	Description
XSK_EFUSEPL_MIO_JTAG_MUX_SELECT	(11)	This pin toggles between the external JTAG or MIO driving JTAG operations.

MUX Parameter for Zynq PL eFUSE JTAG Operations

The table below lists the MUX parameter.

Parameter Name	Description
XSK_EFUSEPL_USER_HIGH_KEY	Default = 000000 The default value is converted to a hexadecimal buffer and written into the PL eFUSE array when the write API is used. This value is the User High Key given in string format. The buffer must be six characters long: valid characters are 0-9, a-f, A-F. Any other character is considered to be an invalid string and does not burn User High Key. To write the User High Key, the XSK_EFUSEPL_PROGRAM_USER_HIGH_KEY must have a value of TRUE.

Zynq User-Configurable PL BBRAM Parameters

The table below lists the MIO pins for Zynq PL BBRAM JTAG operations.

The table below lists the MUX selection pin for Zynq BBRAM PL JTAG operations.

Note: The pin numbers listed in the table below are examples. You must assign appropriate pin numbers as per your hardware design.

Pin Name	Pin Number
XSK_BBRAM_MIO_JTAG_TDI	(17)
XSK_BBRAM_MIO_JTAG_TDO	(21)
XSK_BBRAM_MIO_JTAG_TCK	(19)
XSK_BBRAM_MIO_JTAG_TMS	(20)

Pin Name	Pin Number
XSK_BBRAM_MIO_JTAG_MUX_SELECT	(11)

MUX Parameter for Zynq BBRAM PL JTAG Operations

The table below lists the MUX parameter for Zynq BBRAM PL JTAG operations.

Parameter Name	Description
XSK_BBRAM_MIO_MUX_SEL_DEFAULT_VAL	Default = LOW. LOW writes zero on the MUX select line before PL_eFUSE writing. HIGH writes one on the MUX select line before PL_eFUSE writing.

AES and User Key Parameters

The table below lists the AES and user key parameters.

Parameter Name	Description
XSK_BBRAM_AES_KEY	Default = XX. AES key (in HEX) that must be programmed into BBRAM.
XSK_BBRAM_AES_KEY_SIZE_IN_BITS	Default = 256. Size of AES key. Must be 256 bits.

UltraScale or UltraScale+ User-Configurable BBRAM PL Parameters

Following parameters need to be configured. Based on your inputs, BBRAM is programmed with the provided AES key.

AES Keys and Related Parameters

The following table shows AES key related parameters.

Parameter Name	Description
XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_0	Default = FALSE By default, XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_0 is FALSE. BBRAM is programmed with a non-obfuscated key provided in XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_0 and DPA protection can be either in enabled/disabled state. TRUE programs the BBRAM with key provided in XSK_BBRAM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_0 and DPA protection cannot be enabled.
XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_1	Default = FALSE By default, XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_1 is FALSE. BBRAM is programmed with a non-obfuscated key provided in XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_1 and DPA protection can be either in enabled/disabled state. TRUE programs the BBRAM with key provided in XSK_BBRAM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_1 and DPA protection cannot be enabled.
XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_2	Default = FALSE By default, XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_2 is FALSE. BBRAM is programmed with a non-obfuscated key provided in XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_2 and DPA protection can be either in enabled/disabled state. TRUE programs the BBRAM with key provided in XSK_BBRAM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_2 and DPA protection cannot be enabled.

Parameter Name	Description
XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_3	Default = FALSE By default, XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_3 is FALSE. BBRAM is programmed with a non-obfuscated key provided in XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_3 and DPA protection can be either in enabled/disabled state. TRUE programs the BBRAM with key provided in XSK_BBRAM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_3 and DPA protection cannot be enabled.
XSK_BBRAM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_0	Default = b1c276899d71fb4cdd4a0a7905ea46c2e11f9574d09c7ea23b70b67de713ccd1 The value mentioned in this will be converted to hex buffer and the key is programmed into BBRAM, when program API is called. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not program BBRAM. Note: For writing the OBFUSCATED Key, XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_0 should have TRUE value.
XSK_BBRAM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_1	Default = b1c276899d71fb4cdd4a0a7905ea46c2e11f9574d09c7ea23b70b67de713ccd1 The value mentioned in this will be converted to hex buffer and the key is programmed into BBRAM, when program API is called. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not program BBRAM. Note: For writing the OBFUSCATED Key, XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_1 should have TRUE value.
XSK_BBRAM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_2	Default = b1c276899d71fb4cdd4a0a7905ea46c2e11f9574d09c7ea23b70b67de713ccd1 The value mentioned in this will be converted to hex buffer and the key is programmed into BBRAM, when program API is called. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not program BBRAM. Note: For writing the OBFUSCATED Key, XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_2 should have TRUE value.

Parameter Name	Description
XSK_BBRAM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_3	<p>Default = b1c276899d71fb4cdd4a0a7905ea46c2e11f9574d09c7ea23b70b67de713ccd1</p> <p>The value mentioned in this will be converted to hex buffer and the key is programmed into BBRAM, when program API is called. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not program BBRAM.</p> <p>Note: For writing the OBFUSCATED Key, XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_3 should have TRUE value.</p>
XSK_BBRAM_PGM_AES_KEY_SLR_CONFIG_ORDER_0	<p>Default = FALSE</p> <p>TRUE will program BBRAM with AES key provided in XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_0</p>
XSK_BBRAM_PGM_AES_KEY_SLR_CONFIG_ORDER_1	<p>Default = FALSE</p> <p>TRUE will program BBRAM with AES key provided in XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_1</p>
XSK_BBRAM_PGM_AES_KEY_SLR_CONFIG_ORDER_2	<p>Default = FALSE</p> <p>TRUE will program BBRAM with AES key provided in XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_2</p>
XSK_BBRAM_PGM_AES_KEY_SLR_CONFIG_ORDER_3	<p>Default = FALSE</p> <p>TRUE will program BBRAM with AES key provided in XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_3</p>
XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_0	<p>Default = 000000000000000524156a63950bcedafeadcdeabaadee34216615aaaaabbaaa</p> <p>The value mentioned in this will be converted to hex buffer and the key is programmed into BBRAM, when program API is called. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not program BBRAM.</p> <p>Note: For writing AES key, XSK_BBRAM_PGM_AES_KEY_SLR_CONFIG_ORDER_0 should have TRUE value , and XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_0 should have FALSE value.</p>

Parameter Name	Description
XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_1	<p>Default = 000000000000000524156a63950bcdafeadcdeabaadee34216615aaaabbbaaa</p> <p>The value mentioned in this will be converted to hex buffer and the key is programmed into BBRAM, when program API is called. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not program BBRAM.</p> <p>Note: For writing AES key, XSK_BBRAM_PGM_AES_KEY_SLR_CONFIG_ORDER_1 should have TRUE value , and XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_1 should have FALSE value</p>
XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_2	<p>Default = 000000000000000524156a63950bcdafeadcdeabaadee34216615aaaabbbaaa</p> <p>The value mentioned in this will be converted to hex buffer and the key is programmed into BBRAM, when program API is called. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not program BBRAM.</p> <p>Note: For writing AES key, XSK_BBRAM_PGM_AES_KEY_SLR_CONFIG_ORDER_2 should have TRUE value , and XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_2 should have FALSE value</p>
XSK_BBRAM_AES_KEY_SLR_CONFIG_ORDER_3	<p>Default = 000000000000000524156a63950bcdafeadcdeabaadee34216615aaaabbbaaa</p> <p>The value mentioned in this will be converted to hex buffer and the key is programmed into BBRAM, when program API is called. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not program BBRAM.</p> <p>Note: For writing AES key, XSK_BBRAM_PGM_AES_KEY_SLR_CONFIG_ORDER_3 should have TRUE value , and XSK_BBRAM_PGM_OBFUSCATED_KEY_SLR_CONFIG_ORDER_3 should have FALSE value</p>
XSK_BBRAM_AES_KEY_SIZE_IN_BITS	Default= 256 Size of AES key must be 256 bits.

DPA Protection for BBRAM key

The following table shows DPA protection configurable parameter

Parameter Name	Description
XSK_BBRAM_DPA_PROTECT_ENABLE	Default = FALSE By default, the DPA protection will be in disabled state. TRUE will enable DPA protection with provided DPA count and configuration in XSK_BBRAM_DPA_COUNT and XSK_BBRAM_DPA_MODE respectively. DPA protection cannot be enabled if BBRAM is been programmed with an obfuscated key.
XSK_BBRAM_DPA_COUNT	Default = 0 This input is valid only when DPA protection is enabled. Valid range of values are 1 - 255 when DPA protection is enabled else 0.
XSK_BBRAM_DPA_MODE	Default = XSK_BBRAM_INVALID_CONFIGURATIONS When DPA protection is enabled it can be XSK_BBRAM_INVALID_CONFIGURATIONS or XSK_BBRAM_ALL_CONFIGURATIONS If DPA protection is disabled this input provided over here is ignored.

GPIO Device Used for Connecting PL Master JTAG Signals

In hardware design MASTER JTAG can be connected to any one of the available GPIO devices, based on the design the following parameter should be provided with corresponding device ID of selected GPIO device.

Master JTAG Signal	Description
XSK_BBRAM_AXI_GPIO_DEVICE_ID	Default = XPAR_AXI_GPIO_0_DEVICE_ID This is for providing exact GPIO device ID, based on the design configuration this parameter can be modified to provide GPIO device ID which is used for connecting master jtag pins.

GPIO Pins Used for PL Master JTAG Signals

In Ultrascale the following GPIO pins are used for connecting MASTER_JTAG pins to access BBRAM. These can be changed depending on your hardware. The table below shows the GPIO pins used for PL MASTER JTAG signals.

Master JTAG Signal	Default PIN Number
XSK_BBRAM_AXI_GPIO_JTAG_TDO	0
XSK_BBRAM_AXI_GPIO_JTAG_TDI	0
XSK_BBRAM_AXI_GPIO_JTAG_TMS	1
XSK_BBRAM_AXI_GPIO_JTAG_TCK	2

GPIO Channels

The following table shows GPIO channel number.

Parameter	Default Channel Number	Master JTAG Signal Connected
XSK_BBRAM_GPIO_INPUT_CH	2	TDO
XSK_BBRAM_GPIO_OUTPUT_CH	1	TDI, TMS, TCK

Note: All inputs and outputs of GPIO should be configured in single channel. For example, XSK_BBRAM_GPIO_INPUT_CH = XSK_BBRAM_GPIO_OUTPUT_CH = 1 or 2. Among (TDI, TCK, TMS) Outputs of GPIO cannot be connected to different GPIO channels all the 3 signals should be in same channel. TDO can be a other channel of (TDI, TCK, TMS) or the same. DPA protection can be enabled only when programming non-obfuscated key.

UltraScale or UltraScale+ User-Configurable PL eFUSE Parameters

The table below lists the user-configurable PL eFUSE parameters for UltraScale devices.

Macro Name	Description
XSK_EFUSEPL_DISABLE_AES_KEY_READ	Default = FALSE TRUE will permanently disable the write to FUSE_AES and check CRC for AES key by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_USER_KEY_READ	Default = FALSE TRUE will permanently disable the write to 32 bit FUSE_USER and read of FUSE_USER key by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_SECURE_READ	Default = FALSE TRUE will permanently disable the write to FUSE_Secure block and reading of secure block by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_FUSE_CNTRL_WRITE	Default = FALSE. TRUE will permanently disable the write to FUSE_CNTRL block by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_RSA_KEY_READ	Default = FALSE. TRUE will permanently disable the write to FUSE_RSA block and reading of FUSE_RSA Hash by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_KEY_WRITE	Default = FALSE. TRUE will permanently disable the write to FUSE_AES block by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.

Macro Name	Description
XSK_EFUSEPL_DISABLE_USER_KEY_WRITE	Default = FALSE. TRUE will permanently disable the write to FUSE_USER block by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_SECURE_WRITE	Default = FALSE. TRUE will permanently disable the write to FUSE_SECURE block by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_RSA_HASH_WRITE	Default = FALSE. TRUE will permanently disable the write to FUSE_RSA authentication key by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_128BIT_USER_KEY_WRITE	Default = FALSE. TRUE will permanently disable the write to 128 bit FUSE_USER by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_ALLOW_ENCRYPTED_ONLY	Default = FALSE. TRUE will permanently allow encrypted bitstream only. FALSE will not modify this Secure bit of eFuse.
XSK_EFUSEPL_FORCE_USE_FUSE_AES_ONLY	Default = FALSE. TRUE then allows only FUSE's AES key as source of encryption FALSE then allows FPGA to configure an unencrypted bitstream or bitstream encrypted using key stored BBRAM or eFuse.
XSK_EFUSEPL_ENABLE_RSA_AUTH	Default = FALSE. TRUE will enable RSA authentication of bitstream FALSE will not modify this secure bit of eFuse.
XSK_EFUSEPL_DISABLE_JTAG_CHAIN	Default = FALSE. TRUE will disable JTAG permanently. FALSE will not modify this secure bit of eFuse.
XSK_EFUSEPL_DISABLE_TEST_ACCESS	Default = FALSE. TRUE will disables Xilinx test access. FALSE will not modify this secure bit of eFuse.
XSK_EFUSEPL_DISABLE_AES_DECRYPTOR	Default = FALSE. TRUE will disables decoder completely. FALSE will not modify this secure bit of eFuse.
XSK_EFUSEPL_ENABLE_OBFUSCATION_EFUSEAES	Default = FALSE. TRUE will enable obfuscation feature for eFUSE AES key.

GPIO Device Used for Connecting PL Master JTAG Signals

In hardware design MASTER JTAG can be connected to any one of the available GPIO devices, based on the design the following parameter should be provided with corresponding device ID of selected GPIO device.

Master JTAG Signal	Description
XSK_EFUSEPL_AXI_GPIO_DEVICE_ID	Default = XPAR_AXI_GPIO_0_DEVICE_ID This is for providing exact GPIO device ID, based on the design configuration this parameter can be modified to provide GPIO device ID which is used for connecting master jtag pins.

GPIO Pins Used for PL Master JTAG and HWM Signals

In Ultrascale the following GPIO pins are used for connecting MASTER_JTAG pins to access eFUSE. These can be changed depending on your hardware. The table below shows the GPIO pins used for PL MASTER JTAG signals.

Master JTAG Signal	Default PIN Number
XSK_EFUSEPL_AXI_GPIO_JTAG_TDO	0
XSK_EFUSEPL_AXI_GPIO_HWM_READY	0
XSK_EFUSEPL_AXI_GPIO_HWM_END	1
XSK_EFUSEPL_AXI_GPIO_JTAG_TDI	2
XSK_EFUSEPL_AXI_GPIO_JTAG_TMS	1
XSK_EFUSEPL_AXI_GPIO_JTAG_TCK	2
XSK_EFUSEPL_AXI_GPIO_HWM_START	3

GPIO Channels

The following table shows GPIO channel number.

Parameter	Default Channel Number	Master JTAG Signal Connected
XSK_EFUSEPL_GPIO_INPUT_CH	2	TDO
XSK_EFUSEPL_GPIO_OUTPUT_CH	1	TDI, TMS, TCK

Note: All inputs and outputs of GPIO should be configured in single channel. For example, XSK_EFUSEPL_GPIO_INPUT_CH = XSK_EFUSEPL_GPIO_OUTPUT_CH = 1 or 2. Among (TDI, TCK, TMS) Outputs of GPIO cannot be connected to different GPIO channels all the 3 signals should be in same channel. TDO can be a other channel of (TDI, TCK, TMS) or the same.

SLR Selection to Program eFUSE on MONO/SSIT Devices

The following table shows parameters for programming different SLRs.

Parameter Name	Description
XSK_EFUSEPL_PGM_SLR_CONFIG_ORDER_0	Default = FALSE TRUE will enable programming SLR config order 0's eFUSE. FALSE will disable programming.

Parameter Name	Description
XSK_EFUSEPL_PGM_SLR_CONFIG_ORDER_1	Default = FALSE TRUE will enable programming SLR config order 1's eFUSE. FALSE will disable programming.
XSK_EFUSEPL_PGM_SLR_CONFIG_ORDER_2	Default = FALSE TRUE will enable programming SLR config order 2's eFUSE. FALSE will disable programming.
XSK_EFUSEPL_PGM_SLR_CONFIG_ORDER_3	Default = FALSE TRUE will enable programming SLR config order 3's eFUSE. FALSE will disable programming.

eFUSE PL Read Parameters

The following table shows parameters related to read USER 32/128bit keys and RSA hash.

By enabling any of the below parameters, by default will read corresponding hash/key associated with all the available SLRs. For example, if XSK_EFUSEPL_READ_USER_KEY is TRUE, USER key for all the available SLRs will be read.

Note: For only reading keys it is not required to enable XSK_EFUSEPL_PGM_SLR1, XSK_EFUSEPL_PGM_SLR2, XSK_EFUSEPL_PGM_SLR3, XSK_EFUSEPL_PGM_SLR4 macros, they can be in FALSE state.

Parameter Name	Description
XSK_EFUSEPL_READ_USER_KEY	Default = FALSE TRUE will read 32 bit FUSE_USER from eFUSE of all available SLRs and each time updates in XilSKey_EP1 instance parameter UserKeyReadback, which will be displayed on UART by example before reading next SLR. FALSE 32-bit FUSE_USER key read will not be performed.
XSK_EFUSEPL_READ_RSA_KEY_HASH	Default = FALSE TRUE will read FUSE_USER from eFUSE of all available SLRs and each time updates in XilSKey_EP1 instance parameter RSAHashReadback, which will be displayed on UART by example before reading next SLR. FALSE FUSE_RSA_HASH read will not be performed.
XSK_EFUSEPL_READ_USER_KEY128_BIT	Default = FALSE TRUE will read 128 bit USER key eFUSE of all available SLRs and each time updates in XilSKey_EP1 instance parameter User128BitReadBack, which will be displayed on UART by example before reading next SLR. FALSE 128 bit USER key read will not be performed.

AES Keys and Related Parameters

Note: For programming AES key for MONO/SSIT device, the corresponding SLR should be selected and AES key programming should be enabled.

USER Keys (32-bit) and Related Parameters

Note: For programming USER key for MONO/SSIT device, the corresponding SLR should be selected and USER key programming should be enabled.

RSA Hash and Related Parameters

Note: For programming RSA hash for MONO/SSIT device, the corresponding SLR should be selected and RSA hash programming should be enabled.

USER Keys (128-bit) and Related Parameters

Note: For programming USER key 128 bit for MONO/SSIT device, the corresponding SLR and programming for USER key 128 bit should be enabled.

AES key CRC verification

You cannot read the AES key. You can verify only by providing the CRC of the expected AES key. The following lists the parameters that may help you in verifying the AES key:

Parameter Name	Description
XSK_EFUSEPL_CHECK_AES_KEY_CRC	Default = FALSE TRUE will perform CRC check of FUSE_AES with provided CRC value in macro XSK_EFUSEPL_CRC_OF_EXPECTED_AES_KEY. And result of CRC check will be updated in XiISKey_EP1 instance parameter AESKeyMatched with either TRUE or FALSE. FALSE CRC check of FUSE_AES will not be performed.
XSK_EFUSEPL_CRC_OF_EXPECTED_AES_KEY_CONFIG_ORDER_0	Default = XSK_EFUSEPL_AES_CRC_OF_ALL_ZEROS CRC value of FUSE_AES with all Zeros. Expected FUSE_AES key's CRC value of SLR config order 0 has to be updated in place of XSK_EFUSEPL_AES_CRC_OF_ALL_ZEROS. For Checking CRC of FUSE_AES XSK_EFUSEPL_CHECK_AES_KEY_ULTRA macro should be TRUE otherwise CRC check will not be performed. For calculation of AES key's CRC one can use u32 XiISKey_CrcCalculation(u8_Key) API. For UltraScale, the value of XSK_EFUSEPL_AES_CRC_OF_ALL_ZEROS is 0x621C42AA(XSK_EFUSEPL_CRC_FOR_AES_ZEROS). For UltraScale+, the value of XSK_EFUSEPL_AES_CRC_OF_ALL_ZEROS is 0x3117503A(XSK_EFUSEPL_CRC_FOR_AES_ZEROS_ULTRA_PLUS)

Parameter Name	Description
XSK_EFUSEPL_CRC_OF_EXPECTED_AES_KEY_CONFIG_ORDER_1	Default = XSK_EFUSEPL_AES_CRC_OF_ALL_ZEROS CRC value of FUSE_AES with all Zeros. Expected FUSE_AES key's CRC value of SLR config order 1 has to be updated in place of XSK_EFUSEPL_AES_CRC_OF_ALL_ZEROS. ZEROS. For Checking CRC of FUSE_AES XSK_EFUSEPL_CHECK_AES_KEY_ULTRA macro should be TRUE otherwise CRC check will not be performed. For calculation of AES key's CRC one can use u32 XilSkey_CrcCalculation(u8_Key) API. For UltraScale, the value of XSK_EFUSEPL_AES_CRC_OF_ALL_ZEROS is 0x621C42AA(XSK_EFUSEPL_CRC_FOR_AES_ZEROS). For UltraScale+, the value of XSK_EFUSEPL_AES_CRC_OF_ALL_ZEROS is 0x3117503A(XSK_EFUSEPL_CRC_FOR_AES_ZEROS_ULTRA_PLUS)
XSK_EFUSEPL_CRC_OF_EXPECTED_AES_KEY_CONFIG_ORDER_2	Default = XSK_EFUSEPL_AES_CRC_OF_ALL_ZEROS CRC value of FUSE_AES with all Zeros. Expected FUSE_AES key's CRC value of SLR config order 2 has to be updated in place of XSK_EFUSEPL_AES_CRC_OF_ALL_ZEROS. ZEROS. For Checking CRC of FUSE_AES XSK_EFUSEPL_CHECK_AES_KEY_ULTRA macro should be TRUE otherwise CRC check will not be performed. For calculation of AES key's CRC one can use u32 XilSkey_CrcCalculation(u8_Key) API. For UltraScale, the value of XSK_EFUSEPL_AES_CRC_OF_ALL_ZEROS is 0x621C42AA(XSK_EFUSEPL_CRC_FOR_AES_ZEROS). For UltraScale+, the value of XSK_EFUSEPL_AES_CRC_OF_ALL_ZEROS is 0x3117503A(XSK_EFUSEPL_CRC_FOR_AES_ZEROS_ULTRA_PLUS)
XSK_EFUSEPL_CRC_OF_EXPECTED_AES_KEY_CONFIG_ORDER_3	Default = XSK_EFUSEPL_AES_CRC_OF_ALL_ZEROS CRC value of FUSE_AES with all Zeros. Expected FUSE_AES key's CRC value of SLR config order 3 has to be updated in place of XSK_EFUSEPL_AES_CRC_OF_ALL_ZEROS. ZEROS. For Checking CRC of FUSE_AES XSK_EFUSEPL_CHECK_AES_KEY_ULTRA macro should be TRUE otherwise CRC check will not be performed. For calculation of AES key's CRC one can use u32 XilSkey_CrcCalculation(u8_Key) API. For UltraScale, the value of XSK_EFUSEPL_AES_CRC_OF_ALL_ZEROS is 0x621C42AA(XSK_EFUSEPL_CRC_FOR_AES_ZEROS). For UltraScale+, the value of XSK_EFUSEPL_AES_CRC_OF_ALL_ZEROS is 0x3117503A(XSK_EFUSEPL_CRC_FOR_AES_ZEROS_ULTRA_PLUS)

Zynq UltraScale+ MPSoC User-Configurable PS eFUSE Parameters

The table below lists the user-configurable PS eFUSE parameters for Zynq UltraScale+ MPSoC devices.

Macro Name	Description
XSK_EFUSEPS_AES_RD_LOCK	Default = FALSE TRUE will permanently disable the CRC check of FUSE_AES. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_AES_WR_LOCK	Default = FALSE TRUE will permanently disable the writing to FUSE_AES block. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_ENC_ONLY	Default = FALSE TRUE will permanently enable encrypted booting only using the Fuse key. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_BBRAM_DISABLE	Default = FALSE TRUE will permanently disable the BBRAM key. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_ERR_DISABLE	Default = FALSE TRUE will permanently disables the error messages in JTAG status register. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_JTAG_DISABLE	Default = FALSE TRUE will permanently disable JTAG controller. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_DFT_DISABLE	Default = FALSE TRUE will permanently disable DFT boot mode. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_PROG_GATE_DISABLE	Default = FALSE TRUE will permanently disable PROG_GATE feature in PPD. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_SECURE_LOCK	Default = FALSE TRUE will permanently disable reboot into JTAG mode when doing a secure lockdown. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_RSA_ENABLE	Default = FALSE TRUE will permanently enable RSA authentication during boot. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_PPK0_WR_LOCK	Default = FALSE TRUE will permanently disable writing to PPK0 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_PPK0_INVLD	Default = FALSE TRUE will permanently revoke PPK0. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_PPK1_WR_LOCK	Default = FALSE TRUE will permanently disable writing PPK1 efuses. FALSE will not modify this control bit of eFuse.

Macro Name	Description
XSK_EFUSEPS_PPK1_INVLD	Default = FALSE TRUE will permanently revoke PPK1. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_0	Default = FALSE TRUE will permanently disable writing to USER_0 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_1	Default = FALSE TRUE will permanently disable writing to USER_1 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_2	Default = FALSE TRUE will permanently disable writing to USER_2 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_3	Default = FALSE TRUE will permanently disable writing to USER_3 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_4	Default = FALSE TRUE will permanently disable writing to USER_4 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_5	Default = FALSE TRUE will permanently disable writing to USER_5 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_6	Default = FALSE TRUE will permanently disable writing to USER_6 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_7	Default = FALSE TRUE will permanently disable writing to USER_7 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_LBIST_EN	Default = FALSE TRUE will permanently enables logic BIST to be run during boot. FALSE will not modify this control bit of eFUSE.
XSK_EFUSEPS_LPD_SC_EN	Default = FALSE TRUE will permanently enables zeroization of registers in Low Power Domain(LPD) during boot. FALSE will not modify this control bit of eFUSE.
XSK_EFUSEPS_FPD_SC_EN	Default = FALSE TRUE will permanently enables zeroization of registers in Full Power Domain(FPD) during boot. FALSE will not modify this control bit of eFUSE.
XSK_EFUSEPS_PBR_BOOT_ERR	Default = FALSE TRUE will permanently enables the boot halt when there is any PMU error. FALSE will not modify this control bit of eFUSE.

AES Keys and Related Parameters

The following table shows AES key related parameters.

Parameter Name	Description
XSK_EFUSEPS_WRITE_USER3_FUSE	Default = FALSE TRUE will burn User3 Fuse provided in XSK_EFUSEPS_USER3_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER3_FUSES
XSK_EFUSEPS_WRITE_USER4_FUSE	Default = FALSE TRUE will burn User4 Fuse provided in XSK_EFUSEPS_USER4_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER4_FUSES
XSK_EFUSEPS_WRITE_USER5_FUSE	Default = FALSE TRUE will burn User5 Fuse provided in XSK_EFUSEPS_USERS5_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USERS5_FUSES
XSK_EFUSEPS_WRITE_USER6_FUSE	Default = FALSE TRUE will burn User6 Fuse provided in XSK_EFUSEPS_USER6_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER6_FUSES
XSK_EFUSEPS_WRITE_USER7_FUSE	Default = FALSE TRUE will burn User7 Fuse provided in XSK_EFUSEPS_USER7_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER7_FUSES
XSK_EFUSEPS_USER0_FUSES	Default = 00000000 The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID. Note: For writing the User0 Fuse, XSK_EFUSEPS_WRITE_USER0_FUSE should have TRUE value
XSK_EFUSEPS_USER1_FUSES	Default = 00000000 The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID. Note: For writing the User1 Fuse, XSK_EFUSEPS_WRITE_USER1_FUSE should have TRUE value
XSK_EFUSEPS_USER2_FUSES	Default = 00000000 The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID. Note: For writing the User2 Fuse, XSK_EFUSEPS_WRITE_USER2_FUSE should have TRUE value

Parameter Name	Description
XSK_EFUSEPS_USER3_FUSES	<p>Default = 00000000</p> <p>The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID.</p> <p>Note: For writing the User3 Fuse, XSK_EFUSEPS_WRITE_USER3_FUSE should have TRUE value</p>
XSK_EFUSEPS_USER4_FUSES	<p>Default = 00000000</p> <p>The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID.</p> <p>Note: For writing the User4 Fuse, XSK_EFUSEPS_WRITE_USER4_FUSE should have TRUE value</p>
XSK_EFUSEPS_USER5_FUSES	<p>Default = 00000000</p> <p>The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID.</p> <p>Note: For writing the User5 Fuse, XSK_EFUSEPS_WRITE_USER5_FUSE should have TRUE value</p>
XSK_EFUSEPS_USER6_FUSES	<p>Default = 00000000</p> <p>The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID.</p> <p>Note: For writing the User6 Fuse, XSK_EFUSEPS_WRITE_USER6_FUSE should have TRUE value</p>
XSK_EFUSEPS_USER7_FUSES	<p>Default = 00000000</p> <p>The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID.</p> <p>Note: For writing the User7 Fuse, XSK_EFUSEPS_WRITE_USER7_FUSE should have TRUE value</p>

PPK0 Keys and Related Parameters

The following table shows the PPK0 keys and related parameters.

Parameter Name	Description
XSK_ZYNQMP_BBRAMPS_AES_KEY_LEN_IN_BYTES	Default = 32. Length of AES key in bytes.
XSK_ZYNQMP_BBRAMPS_AES_KEY_LEN_IN_BITS	Default = 256. Length of AES key in bits.
XSK_ZYNQMP_BBRAMPS_AES_KEY_STR_LEN	Default = 64. String length of the AES key.

Zynq UltraScale+ MPSoC User-Configurable PS PUF Parameters

The table below lists the user-configurable PS PUF parameters for Zynq UltraScale+ MPSoC devices.

Macro Name	Description
XSK_PUF_INFO_ON_UART	Default = FALSE TRUE will display syndrome data on UART com port FALSE will display any data on UART com port.
XSK_PUF_PROGRAM_EFUSE	Default = FALSE TRUE will program the generated syndrome data, CHash and Auxilary values, Black key. FALSE will not program data into eFUSE.
XSK_PUF_IF_CONTRACT_MANUFACTURER	Default = FALSE This should be enabled when application is hand over to contract manufacturer. TRUE will allow only authenticated application. FALSE authentication is not mandatory.
XSK_PUF_REG_MODE	Default = XSK_PUF_MODE4K PUF registration is performed in 4K mode. For only understanding it is provided in this file, but user is not supposed to modify this.
XSK_PUF_READ_SECUREBITS	Default = FALSE TRUE will read status of the puf secure bits from eFUSE and will be displayed on UART. FALSE will not read secure bits.
XSK_PUF_PROGRAM_SECUREBITS	Default = FALSE TRUE will program PUF secure bits based on the user input provided at XSK_PUF_SYN_INVALID, XSK_PUF_SYN_WRLK and XSK_PUF_REGISTER_DISABLE. FALSE will not program any PUF secure bits.
XSK_PUF_SYN_INVALID	Default = FALSE TRUE will permanently invalidate the already programmed syndrome data. FALSE will not modify anything

- Write the error code in the reboot status register

PL eFUSE Error Codes

Enumerations

Enumeration XSKEfusePI_ErrorCodes

Table 324: Enumeration XSKEfusePI_ErrorCodes Values

Value	Description
XSK_EFUSEPL_ERROR_NONE	0 No error.
XSK_EFUSEPL_ERROR_ROW_NOT_ZERO	0x10 Row is not zero.
XSK_EFUSEPL_ERROR_READ_ROW_OUT_OF_RANGE	0x11 Read Row is out of range.
XSK_EFUSEPL_ERROR_READ_MARGIN_OUT_OF_RANGE	0x12 Read Margin is out of range.
XSK_EFUSEPL_ERROR_READ_BUFFER_NULL	0x13 No buffer for read.
XSK_EFUSEPL_ERROR_READ_BIT_VALUE_NOT_SET	0x14 Read bit not set.
XSK_EFUSEPL_ERROR_READ_BIT_OUT_OF_RANGE	0x15 Read bit is out of range.
XSK_EFUSEPL_ERROR_READ_TEMPERATURE_OUT_OF_RANGE	0x16 Temperature obtained from XADC is out of range to read.
XSK_EFUSEPL_ERROR_READ_VCCAUX_VOLTAGE_OUT_OF_RANGE	0x17 VCCAUX obtained from XADC is out of range to read.
XSK_EFUSEPL_ERROR_READ_VCCINT_VOLTAGE_OUT_OF_RANGE	0x18 VCCINT obtained from XADC is out of range to read.
XSK_EFUSEPL_ERROR_WRITE_ROW_OUT_OF_RANGE	0x19 To write row is out of range.
XSK_EFUSEPL_ERROR_WRITE_BIT_OUT_OF_RANGE	0x1A To read bit is out of range.
XSK_EFUSEPL_ERROR_WRITE_TEMPERATURE_OUT_OF_RANGE	0x1B To eFUSE write Temperature obtained from XADC is out of range.
XSK_EFUSEPL_ERROR_WRITE_VCCAUX_VOLTAGE_OUT_OF_RANGE	0x1C To write eFUSE VCCAUX obtained from XADC is out of range.
XSK_EFUSEPL_ERROR_WRITE_VCCINT_VOLTAGE_OUT_OF_RANGE	0x1D To write into eFUSE VCCINT obtained from XADC is out of range.
XSK_EFUSEPL_ERROR_FUSE_CNTRL_WRITE_DISABLED	0x1E Fuse control write is disabled.

Table 324: Enumeration XSKefusePl_ErrorCodes Values (cont'd)

Value	Description
XSK_EFUSEPL_ERROR_CNTRL_WRITE_BUFFER_NULL	0x1F Buffer pointer that is supposed to contain control data is null.
XSK_EFUSEPL_ERROR_NOT_VALID_KEY_LENGTH	0x20 Key length invalid.
XSK_EFUSEPL_ERROR_ZERO_KEY_LENGTH	0x21 Key length zero.
XSK_EFUSEPL_ERROR_NOT_VALID_KEY_CHAR	0x22 Invalid key characters.
XSK_EFUSEPL_ERROR_NULL_KEY	0x23 Null key.
XSK_EFUSEPL_ERROR_FUSE_SEC_WRITE_DISABLED	0x24 Secure bits write is disabled.
XSK_EFUSEPL_ERROR_FUSE_SEC_READ_DISABLED	0x25 Secure bits reading is disabled.
XSK_EFUSEPL_ERROR_SEC_WRITE_BUFFER_NULL	0x26 Buffer to write into secure block is NULL.
XSK_EFUSEPL_ERROR_READ_PAGE_OUT_OF_RANGE	0x27 Page is out of range.
XSK_EFUSEPL_ERROR_FUSE_ROW_RANGE	0x28 Row is out of range.
XSK_EFUSEPL_ERROR_IN_PROGRAMMING_ROW	0x29 Error programming fuse row.
XSK_EFUSEPL_ERROR_PRGRMG_ROWS_NOT_EMPTY	0x2A Error when tried to program non Zero rows of eFUSE.
XSK_EFUSEPL_ERROR_HWM_TIMEOUT	0x80 Error when hardware module is exceeded the time for programming eFUSE.
XSK_EFUSEPL_ERROR_USER_FUSE_REVERT	0x90 Error occurs when user requests to revert already programmed user eFUSE bit.
XSK_EFUSEPL_ERROR_KEY_VALIDATION	0xF00 Invalid key.
XSK_EFUSEPL_ERROR_PL_STRUCT_NULL	0x1000 Null PL structure.
XSK_EFUSEPL_ERROR_JTAG_SERVER_INIT	0x1100 JTAG server initialization error.
XSK_EFUSEPL_ERROR_READING_FUSE_CNTRL	0x1200 Error reading fuse control.
XSK_EFUSEPL_ERROR_DATA_PROGRAMMING_NOT_ALLOWED	0x1300 Data programming not allowed.
XSK_EFUSEPL_ERROR_FUSE_CTRL_WRITE_NOT_ALLOWED	0x1400 Fuse control write is disabled.

Table 324: Enumeration XSKefusePl_ErrorCodes Values (cont'd)

Value	Description
XSK_EFUSEPL_ERROR_READING_FUSE_AES_ROW	0x1500 Error reading fuse AES row.
XSK_EFUSEPL_ERROR_AES_ROW_NOT_EMPTY	0x1600 AES row is not empty.
XSK_EFUSEPL_ERROR_PROGRAMMING_FUSE_AES_ROW	0x1700 Error programming fuse AES row.
XSK_EFUSEPL_ERROR_READING_FUSE_USER_DATA_ROW	0x1800 Error reading fuse user row.
XSK_EFUSEPL_ERROR_USER_DATA_ROW_NOT_EMPTY	0x1900 User row is not empty.
XSK_EFUSEPL_ERROR_PROGRAMMING_FUSE_DATA_ROW	0x1A00 Error programming fuse user row.
XSK_EFUSEPL_ERROR_PROGRAMMING_FUSE_CTRL_ROW	0x1B00 Error programming fuse control row.
XSK_EFUSEPL_ERROR_XADC	0x1C00 XADC error.
XSK_EFUSEPL_ERROR_INVALID_REF_CLK	0x3000 Invalid reference clock.
XSK_EFUSEPL_ERROR_FUSE_SEC_WRITE_NOT_ALLOWED	0x1D00 Error in programming secure block.
XSK_EFUSEPL_ERROR_READING_FUSE_STATUS	0x1E00 Error in reading FUSE status.
XSK_EFUSEPL_ERROR_FUSE_BUSY	0x1F00 Fuse busy.
XSK_EFUSEPL_ERROR_READING_FUSE_RSA_ROW	0x2000 Error in reading FUSE RSA block.
XSK_EFUSEPL_ERROR_TIMER_INITIALISE_ULTRA	0x2200 Error in initiating Timer.
XSK_EFUSEPL_ERROR_READING_FUSE_SEC	0x2300 Error in reading FUSE secure bits.
XSK_EFUSEPL_ERROR_PRGRMG_FUSE_SEC_ROW	0x2500 Error in programming Secure bits of efuse.
XSK_EFUSEPL_ERROR_PRGRMG_USER_KEY	0x4000 Error in programming 32 bit user key.
XSK_EFUSEPL_ERROR_PRGRMG_128BIT_USER_KEY	0x5000 Error in programming 128 bit User key.
XSK_EFUSEPL_ERROR_PRGRMG_RSA_HASH	0x8000 Error in programming RSA hash.

PS eFUSE Error Codes

Enumerations

Enumeration XSKEfusePs_ErrorCodes

Table 325: Enumeration XSKEfusePs_ErrorCodes Values

Value	Description
XSK_EFUSEPS_ERROR_NONE	0 No error.
XSK_EFUSEPS_ERROR_ADDRESS_XIL_RESTRICTED	0x01 Address is restricted.
XSK_EFUSEPS_ERROR_READ_TEMPERATURE_OUT_OF_RANGE	0x02 Temperature obtained from XADC is out of range.
XSK_EFUSEPS_ERROR_READ_VCCPAUX_VOLTAGE_OUT_OF_RANGE	0x03 VCCAUX obtained from XADC is out of range.
XSK_EFUSEPS_ERROR_READ_VCCPINT_VOLTAGE_OUT_OF_RANGE	0x04 VCCINT obtained from XADC is out of range.
XSK_EFUSEPS_ERROR_WRITE_TEMPERATURE_OUT_OF_RANGE	0x05 Temperature obtained from XADC is out of range.
XSK_EFUSEPS_ERROR_WRITE_VCCPAUX_VOLTAGE_OUT_OF_RANGE	0x06 VCCAUX obtained from XADC is out of range.
XSK_EFUSEPS_ERROR_WRITE_VCCPINT_VOLTAGE_OUT_OF_RANGE	0x07 VCCINT obtained from XADC is out of range.
XSK_EFUSEPS_ERROR_VERIFICATION	0x08 Verification error.
XSK_EFUSEPS_ERROR_RSA_HASH_ALREADY_PROGRAMMED	0x09 RSA hash was already programmed.
XSK_EFUSEPS_ERROR_CONTROLLER_MODE	0x0A Controller mode error
XSK_EFUSEPS_ERROR_REF_CLOCK	0x0B Reference clock not between 20 to 60MHz
XSK_EFUSEPS_ERROR_READ_MODE	0x0C Not supported read mode
XSK_EFUSEPS_ERROR_XADC_CONFIG	0x0D XADC configuration error.
XSK_EFUSEPS_ERROR_XADC_INITIALIZE	0x0E XADC initialization error.
XSK_EFUSEPS_ERROR_XADC_SELF_TEST	0x0F XADC self-test failed.
XSK_EFUSEPS_ERROR_PARAMETER_NULL	0x10 Passed parameter null.

Table 325: Enumeration XSKefusePs_ErrorCodes Values (cont'd)

Value	Description
XSK_EFUSEPS_ERROR_STRING_INVALID	0x20 Passed string is invalid.
XSK_EFUSEPS_ERROR_AES_ALREADY_PROGRAMMED	0x12 AES key is already programmed.
XSK_EFUSEPS_ERROR_SPKID_ALREADY_PROGRAMMED	0x13 SPK ID is already programmed.
XSK_EFUSEPS_ERROR_PPK0_HASH_ALREADY_PROGRAMMED	0x14 PPK0 hash is already programmed.
XSK_EFUSEPS_ERROR_PPK1_HASH_ALREADY_PROGRAMMED	0x15 PPK1 hash is already programmed.
XSK_EFUSEPS_ERROR_IN_TBIT_PATTERN	0x16 Error in TBITS pattern .
XSK_EFUSEPS_ERROR_PROGRAMMING	0x00A0 Error in programming eFUSE.
XSK_EFUSEPS_ERROR_PGM_NOT_DONE	0x00A1 Program not done
XSK_EFUSEPS_ERROR_READ	0x00B0 Error in reading.
XSK_EFUSEPS_ERROR_BYTES_REQUEST	0x00C0 Error in requested byte count.
XSK_EFUSEPS_ERROR_RESRVD_BITS_PRGRMG	0x00D0 Error in programming reserved bits.
XSK_EFUSEPS_ERROR_ADDR_ACCESS	0x00E0 Error in accessing requested address.
XSK_EFUSEPS_ERROR_READ_NOT_DONE	0x00F0 Read not done
XSK_EFUSEPS_ERROR_PS_STRUCT_NULL	0x8100 PS structure pointer is null.
XSK_EFUSEPS_ERROR_XADC_INIT	0x8200 XADC initialization error.
XSK_EFUSEPS_ERROR_CONTROLLER_LOCK	0x8300 PS eFUSE controller is locked.
XSK_EFUSEPS_ERROR_EFUSE_WRITE_PROTECTED	0x8400 PS eFUSE is write protected.
XSK_EFUSEPS_ERROR_CONTROLLER_CONFIG	0x8500 Controller configuration error.
XSK_EFUSEPS_ERROR_PS_PARAMETER_WRONG	0x8600 PS eFUSE parameter is not TRUE/FALSE.
XSK_EFUSEPS_ERROR_WRITE_128K_CRC_BIT	0x9100 Error in enabling 128K CRC.

Table 325: Enumeration XSKefusePs_ErrorCodes Values (cont'd)

Value	Description
XSK_EFUSEPS_ERROR_WRITE_NONSECURE_INIT_B_BIT	0x9200 Error in programming NON secure bit.
XSK_EFUSEPS_ERROR_WRITE_UART_STATUS_BIT	0x9300 Error in writing UART status bit.
XSK_EFUSEPS_ERROR_WRITE_RSA_HASH	0x9400 Error in writing RSA key.
XSK_EFUSEPS_ERROR_WRITE_RSA_AUTH_BIT	0x9500 Error in enabling RSA authentication bit.
XSK_EFUSEPS_ERROR_WRITE_WRITE_PROTECT_BIT	0x9600 Error in writing write-protect bit.
XSK_EFUSEPS_ERROR_READ_HASH_BEFORE_PROGRAMMING	0x9700 Check RSA key before trying to program.
XSK_EFUSEPS_ERROR_WRTIE_DFT_JTAG_DIS_BIT	0x9800 Error in programming DFT JTAG disable bit.
XSK_EFUSEPS_ERROR_WRTIE_DFT_MODE_DIS_BIT	0x9900 Error in programming DFT MODE disable bit.
XSK_EFUSEPS_ERROR_WRTIE_AES_CRC_LK_BIT	0x9A00 Error in enabling AES's CRC check lock.
XSK_EFUSEPS_ERROR_WRTIE_AES_WR_LK_BIT	0x9B00 Error in programming AES write lock bit.
XSK_EFUSEPS_ERROR_WRTIE_USE_AESONLY_EN_BIT	0x9C00 Error in programming use AES only bit.
XSK_EFUSEPS_ERROR_WRTIE_BBRAM_DIS_BIT	0x9D00 Error in programming BBRAM disable bit.
XSK_EFUSEPS_ERROR_WRTIE_PMU_ERR_DIS_BIT	0x9E00 Error in programming PMU error disable bit.
XSK_EFUSEPS_ERROR_WRTIE_JTAG_DIS_BIT	0x9F00 Error in programming JTAG disable bit.
XSK_EFUSEPS_ERROR_READ_RSA_HASH	0xA100 Error in reading RSA key.
XSK_EFUSEPS_ERROR_WRONG_TBIT_PATTERN	0xA200 Error in programming TBIT pattern.
XSK_EFUSEPS_ERROR_WRITE_AES_KEY	0xA300 Error in programming AES key.
XSK_EFUSEPS_ERROR_WRITE_SPK_ID	0xA400 Error in programming SPK ID.
XSK_EFUSEPS_ERROR_WRITE_USER_KEY	0xA500 Error in programming USER key.
XSK_EFUSEPS_ERROR_WRITE_PPK0_HASH	0xA600 Error in programming PPK0 hash.

Table 325: Enumeration XSKefusePs_ErrorCodes Values (cont'd)

Value	Description
XSK_EFUSEPS_ERROR_WRITE_PPK1_HASH	0xA700 Error in programming PPK1 hash.
XSK_EFUSEPS_ERROR_WRITE_USER0_FUSE	0xC000 Error in programming USER 0 Fuses.
XSK_EFUSEPS_ERROR_WRITE_USER1_FUSE	0xC100 Error in programming USER 1 Fuses.
XSK_EFUSEPS_ERROR_WRITE_USER2_FUSE	0xC200 Error in programming USER 2 Fuses.
XSK_EFUSEPS_ERROR_WRITE_USER3_FUSE	0xC300 Error in programming USER 3 Fuses.
XSK_EFUSEPS_ERROR_WRITE_USER4_FUSE	0xC400 Error in programming USER 4 Fuses.
XSK_EFUSEPS_ERROR_WRITE_USER5_FUSE	0xC500 Error in programming USER 5 Fuses.
XSK_EFUSEPS_ERROR_WRITE_USER6_FUSE	0xC600 Error in programming USER 6 Fuses.
XSK_EFUSEPS_ERROR_WRITE_USER7_FUSE	0xC700 Error in programming USER 7 Fuses.
XSK_EFUSEPS_ERROR_WRTIE_USER0_LK_BIT	0xC800 Error in programming USER 0 fuses lock bit.
XSK_EFUSEPS_ERROR_WRTIE_USER1_LK_BIT	0xC900 Error in programming USER 1 fuses lock bit.
XSK_EFUSEPS_ERROR_WRTIE_USER2_LK_BIT	0xCA00 Error in programming USER 2 fuses lock bit.
XSK_EFUSEPS_ERROR_WRTIE_USER3_LK_BIT	0xCB00 Error in programming USER 3 fuses lock bit.
XSK_EFUSEPS_ERROR_WRTIE_USER4_LK_BIT	0xCC00 Error in programming USER 4 fuses lock bit.
XSK_EFUSEPS_ERROR_WRTIE_USER5_LK_BIT	0xCD00 Error in programming USER 5 fuses lock bit.
XSK_EFUSEPS_ERROR_WRTIE_USER6_LK_BIT	0xCE00 Error in programming USER 6 fuses lock bit.
XSK_EFUSEPS_ERROR_WRTIE_USER7_LK_BIT	0xCF00 Error in programming USER 7 fuses lock bit.
XSK_EFUSEPS_ERROR_WRTIE_PROG_GATE0_DIS_BIT	0xD000 Error in programming PROG_GATE0 disabling bit.
XSK_EFUSEPS_ERROR_WRTIE_PROG_GATE1_DIS_BIT	0xD100 Error in programming PROG_GATE1 disabling bit.
XSK_EFUSEPS_ERROR_WRTIE_PROG_GATE2_DIS_BIT	0xD200 Error in programming PROG_GATE2 disabling bit.

Table 325: Enumeration XSKefusePs_ErrorCodes Values (cont'd)

Value	Description
XSK_EFUSEPS_ERROR_WRTIE_SEC_LOCK_BIT	0xD300 Error in programming SEC_LOCK bit.
XSK_EFUSEPS_ERROR_WRTIE_PPK0_WR_LK_BIT	0xD400 Error in programming PPK0 write lock bit.
XSK_EFUSEPS_ERROR_WRTIE_PPK0_RVK_BIT	0xD500 Error in programming PPK0 revoke bit.
XSK_EFUSEPS_ERROR_WRTIE_PPK1_WR_LK_BIT	0xD600 Error in programming PPK1 write lock bit.
XSK_EFUSEPS_ERROR_WRTIE_PPK1_RVK_BIT	0xD700 Error in programming PPK0 revoke bit.
XSK_EFUSEPS_ERROR_WRITE_PUF_SYN_INVLD	0xD800 Error while programming the PUF syndrome invalidate bit.
XSK_EFUSEPS_ERROR_WRITE_PUF_SYN_WRLK	0xD900 Error while programming Syndrome write lock bit.
XSK_EFUSEPS_ERROR_WRITE_PUF_SYN_REG_DIS	0xDA00 Error while programming PUF syndrome register disable bit.
XSK_EFUSEPS_ERROR_WRITE_PUF_RESERVED_BIT	0xDB00 Error while programming PUF reserved bit.
XSK_EFUSEPS_ERROR_WRITE_LBIST_EN_BIT	0xDC00 Error while programming LBIST enable bit.
XSK_EFUSEPS_ERROR_WRITE_LPD_SC_EN_BIT	0xDD00 Error while programming LPD SC enable bit.
XSK_EFUSEPS_ERROR_WRITE_FPD_SC_EN_BIT	0xDE00 Error while programming FPD SC enable bit.
XSK_EFUSEPS_ERROR_WRITE_PBR_BOOT_ERR_BIT	0xDF00 Error while programming PBR boot error bit.
XSK_EFUSEPS_ERROR_PUF_INVALID_REG_MODE	0xE000 Error when PUF registration is requested with invalid registration mode.
XSK_EFUSEPS_ERROR_PUF_REG_WO_AUTH	0xE100 Error when write not allowed without authentication enabled.
XSK_EFUSEPS_ERROR_PUF_REG_DISABLED	0xE200 Error when trying to do PUF registration and when PUF registration is disabled.
XSK_EFUSEPS_ERROR_PUF_INVALID_REQUEST	0xE300 Error when an invalid mode is requested.
XSK_EFUSEPS_ERROR_PUF_DATA_ALREADY_PROGRAMMED	0xE400 Error when PUF is already programmed in eFUSE.
XSK_EFUSEPS_ERROR_PUF_DATA_OVERFLOW	0xE500 Error when an over flow occurs.
XSK_EFUSEPS_ERROR_SPKID_BIT_CANT_REVERT	0xE600 Already programmed SPKID bit cannot be reverted

Table 325: Enumeration XSKefusePs_ErrorCodes Values (cont'd)

Value	Description
XSK_EFUSEPS_ERROR_PUF_DATA_UNDERFLOW	0xE700 Error when an under flow occurs.
XSK_EFUSEPS_ERROR_PUF_TIMEOUT	0xE800 Error when an PUF generation timedout.
XSK_EFUSEPS_ERROR_PUF_ACCESS	0xE900 Error when an PUF Access violation.
XSK_EFUSEPS_ERROR_PUF_CHASH_ALREADY_PROGRAMMED	
XSK_EFUSEPS_ERROR_PUF_AUX_ALREADY_PROGRAMMED	0xEA00 Error When PUF Chash already programmed in eFuse.
XSK_EFUSEPS_ERROR_CMPLTD_EFUSE_PRGRM_WITH_ERR	0xEB00 Error When PUF AUX already programmed in eFuse. 0x10000 eFUSE programming is completed with temp and vol read errors.
XSK_EFUSEPS_ERROR_CACHE_LOAD	0x20000U Error in re-loading CACHE.
XSK_EFUSEPS_RD_FROM_EFUSE_NOT_ALLOWED	0x30000U Read from eFuse is not allowed.
XSK_EFUSEPS_ERROR_FUSE_PROTECTED	0x00080000 Requested eFUSE is write protected.
XSK_EFUSEPS_ERROR_USER_BIT_CANT_REVERT	0x00800000 Already programmed user FUSE bit cannot be reverted.
XSK_EFUSEPS_ERROR_BEFORE_PROGRAMMING	0x08000000U Error occurred before programming.

Zynq UltraScale+ MPSoC BBRAM PS Error Codes

Enumerations

Enumeration XskZynqMp_Ps_Bbram_ErrorCodes

Table 326: Enumeration XskZynqMp_Ps_Bbram_ErrorCodes Values

Value	Description
XSK_ZYNQMP_BBRAMPS_ERROR_NONE	0 No error.
XSK_ZYNQMP_BBRAMPS_ERROR_IN_PRGRMG_ENABLED	0x010 If this error is occurred programming is not possible.
XSK_ZYNQMP_BBRAMPS_ERROR_IN_ZEROISE	0x20 zeroize bbram is failed.

Table 326: Enumeration XskZynqMp_Ps_Bbram_ErrorCodes Values (cont'd)

Value	Description
XSK_ZYNQMP_BBRAMPS_ERROR_IN_CRC_CHECK	0xB000 If this error is occurred programming is done but CRC check is failed.
XSK_ZYNQMP_BBRAMPS_ERROR_IN_PRGRMG	0xC000 programming of key is failed.
XSK_ZYNQMP_BBRAMPS_ERROR_IN_WRITE_CRC	0xE800 error write CRC value.

Status Codes

For Zynq and UltraScale, the status in the xilskey_efuse_example.c file is conveyed through a UART or reboot status register in the following format: 0xYYYYZZZZ, where:

- YYYY represents the PS eFUSE Status.
- ZZZZ represents the PL eFUSE Status.

The table below lists the status codes.

Status Code Values	Description
0x0000ZZZZ	Represents PS eFUSE is successful and PL eFUSE process returned with error.
0xYYYY0000	Represents PL eFUSE is successful and PS eFUSE process returned with error.
0xFFFF0000	Represents PS eFUSE is not initiated and PL eFUSE is successful.
0x0000FFFF	Represents PL eFUSE is not initiated and PS eFUSE is successful.
0xFFFFZZZZ	Represents PS eFUSE is not initiated and PL eFUSE is process returned with error.
0xYYYYFFFF	Represents PL eFUSE is not initiated and PS eFUSE is process returned with error.

For Zynq UltraScale+ MPSoC, the status in the xilskey_bbramps_zynqmp_example.c, xilskey_puf_registration.c and xilskey_efuseps_zynqmp_example.c files is conveyed as 32 bit error code. Where Zero represents that no error has occurred and if the value is other than Zero, a 32 bit error code is returned.

Procedures

This section provides detailed descriptions of the various procedures.

Zynq eFUSE Writing Procedure Running from DDR as an Application

This sequence is same as the existing flow described below.

1. Provide the required inputs in `xilskey_input.h`, then compile the platform project.
2. Take the latest FSBL (ELF), stitch the `<output>.elf` generated to it (using the bootgen utility), and generate a bootable image.
3. Write the generated binary image into the flash device (for example: QSPI, NAND).
4. To burn the eFUSE key bits, execute the image.

Zynq eFUSE Driver Compilation Procedure for OCM

The procedure is as follows:

1. Open the linker script (`lscript.ld`) in the platform project.
2. Map all the sections to point to `ps7_ram_0_S_AXI_BASEADDR` instead of `ps7_ddr_0_S_AXI_BASEADDR`. For example, Click the Memory Region tab for the `.text` section and select `ps7_ram_0_S_AXI_BASEADDR` from the drop-down list.
3. Copy the `ps7_init.c` and `ps7_init.h` files from the `hw_platform` folder into the example folder.
4. In `xilskey_efuse_example.c`, un-comment the code that calls the `ps7_init()` routine.
5. Compile the project.

The `<Project name>.elf` file is generated and is executed out of OCM.

When executed, this example displays the success/failure of the eFUSE application in a display message via UART (if UART is present and initialized) or the reboot status register.

UltraScale eFUSE Access Procedure

The procedure is as follows:

1. After providing the required inputs in `xilskey_input.h`, compile the project.
2. Generate a memory mapped interface file using TCL command `write_mem_info`
3. Update memory has to be done using the tcl command `updatemem`.
4. Program the board using `$Final.bit` bitstream.
5. Output can be seen in UART terminal.

UltraScale BBRAM Access Procedure

The procedure is as follows:

1. After providing the required inputs in the `xilskey_bbram_ultrascale_input.h`` file, compile the project.
2. Generate a memory mapped interface file using TCL command
3. Update memory has to be done using the tcl command `updatemem`:
4. Program the board using `$Final.bit` bitstream.
5. Output can be seen in UART terminal.

Data Structure Index

The following is a list of data structures:

- [XilSKey_EPI](#)

XilSKey_EPI

XEfusePI is the PL eFUSE driver instance.

Using this structure, user can define the eFUSE bits to be blown.

Declaration

```
typedef struct
{
    u32 ForcePowerCycle,
    u32 KeyWrite,
    u32 AESKeyRead,
    u32 UserKeyRead,
    u32 CtrlWrite,
    u32 RSARead,
    u32 UserKeyWrite,
    u32 SecureWrite,
    u32 RSAWrite,
    u32 User128BitWrite,
    u32 SecureRead,
    u32 AESKeyExclusive,
    u32 JtagDisable,
    u32 UseAESOnly,
    u32 EncryptOnly,
    u32 IntTestAccessDisable,
    u32 DecoderDisable,
    u32 RSAEnable,
    u32 FuseObfusEn,
    u32 ProgAESandUserLowKey,
    u32 ProgUserHighKey,
    u32 ProgAESKeyUltra,
    u32 ProgUserKeyUltra,
    u32 ProgRSAKeyUltra,
    u32 ProgUser128BitUltra,
    u32 CheckAESKeyUltra,
    u32 ReadUserKeyUltra,
```

```

u32 ReadRSAKeyUltra,
u32 ReadUser128BitUltra,
u8 AESKey[XSK_EFUSEPL_AES_KEY_SIZE_IN_BYTES],
u8 UserKey[XSK_EFUSEPL_USER_KEY_SIZE_IN_BYTES],
u8 RSAKeyHash[XSK_EFUSEPL_RSA_KEY_HASH_SIZE_IN_BYTES],
u8 User128Bit[XSK_EFUSEPL_128BIT_USERKEY_SIZE_IN_BYTES],
u32 JtagMioTDI,
u32 JtagMioTDO,
u32 JtagMioTCK,
u32 JtagMioTMS,
u32 JtagMioMuxSel,
u32 JtagMuxSelLineDefVal,
u32 JtagGpioID,
u32 HwmGpioStart,
u32 HwmGpioReady,
u32 HwmGpioEnd,
u32 JtagGpioTDI,
u32 JtagGpioTDO,
u32 JtagGpioTMS,
u32 JtagGpioTCK,
u32 GpioInputCh,
u32 GpioOutPutCh,
u8 AESKeyReadback[XSK_EFUSEPL_AES_KEY_SIZE_IN_BYTES],
u8 UserKeyReadback[XSK_EFUSEPL_USER_KEY_SIZE_IN_BYTES],
u32 CrcOfAESKey,
u8 AESKeyMatched,
u8 RSAHashReadback[XSK_EFUSEPL_RSA_KEY_HASH_SIZE_IN_BYTES],
u8 User128BitReadBack[XSK_EFUSEPL_128BIT_USERKEY_SIZE_IN_BYTES],
u32 SystemInitDone,
XSKefusePl_Fpga FpgaFlag,
u32 CrcToVerify,
u32 NumSlr,
u32 MasterSlr,
u32 SlrConfigOrderIndex
} XilSKey_EPl;
    
```

Table 327: Structure XilSKey_EPl member description

Member	Description
ForcePowerCycle	Following are the FUSE CNTRL bits[1:5, 8-10]. If XTRUE then part has to be power cycled to be able to be reconfigured only for zynq
KeyWrite	If XTRUE will disable eFUSE write to FUSE_AES and FUSE_USER blocks valid only for zynq but in ultrascale If XTRUE will disable eFUSE write to FUSE_AESKEY block in Ultrascale.
AESKeyRead	If XTRUE will disable eFUSE read to FUSE_AES block and also disables eFUSE write to FUSE_AES and FUSE_USER blocks in Zynq Pl.but in Ultrascale if XTRUE will disable eFUSE read to FUSE_KEY block and also disables eFUSE write to FUSE_KEY blocks.
UserKeyRead	If XTRUE will disable eFUSE read to FUSE_USER block and also disables eFUSE write to FUSE_AES and FUSE_USER blocks in zynq but in ultrascale if XTRUE will disable eFUSE read to FUSE_USER block and also disables eFUSE write to FUSE_USER blocks.
CtrlWrite	If XTRUE will disable eFUSE write to FUSE_CNTRL block in both Zynq and Ultrascale.
RSARead	If XTRUE will disable eFuse read to FUSE_RSA block and also disables eFuse write to FUSE_RSA block in Ultrascale.
UserKeyWrite	

Table 327: Structure XilSkey_EPI member description (cont'd)

Member	Description
SecureWrite	
RSASecureWrite	
User128BitWrite	If TRUE will disable eFUSE write to 128BIT FUSE_USER block in Ultrascale.
SecureRead	If XTRUE will disable eFuse read to FUSE_SEC block and also disables eFuse write to FUSE_SEC block in Ultrascale.
AESKeyExclusive	If XTRUE will force eFUSE key to be used if booting Secure Image In Zynq.
JtagDisable	If XTRUE then permanently sets the Zynq ARM DAP controller in bypass mode in both zynq and ultrascale.
UseAESOnly	If XTRUE will force to use Secure boot with eFUSE key only for both Zynq and Ultrascale.
EncryptOnly	If XTRUE will only allow encrypted bitstreams only.
IntTestAccessDisable	If XTRUE then sets the disable's Xilinx internal test access in Ultrascale.
DecoderDisable	If XTRUE then permanently disables the decryptor in Ultrascale.
RSASecureEnable	Enable RSA authentication in ultrascale.
FuseObfusEn	
ProgAESandUserLowKey	Following is the define to select if the user wants to select AES key and User Low Key for Zynq.
ProgUserHighKey	Following is the define to select if the user wants to select User Low Key for Zynq.
ProgAESKeyUltra	Following is the define to select if the user wants to select User key for Ultrascale.
ProgUserKeyUltra	Following is the define to select if the user wants to select User key for Ultrascale.
ProgRSAKeyUltra	Following is the define to select if the user wants to select RSA key for Ultrascale.
ProgUser128BitUltra	Following is the define to select if the user wants to program 128 bit User key for Ultrascale.
CheckAESKeyUltra	Following is the define to select if the user wants to read AES key for Ultrascale.
ReadUserKeyUltra	Following is the define to select if the user wants to read User key for Ultrascale.
ReadRSAKeyUltra	Following is the define to select if the user wants to read RSA key for Ultrascale.
ReadUser128BitUltra	Following is the define to select if the user wants to read 128 bit User key for Ultrascale.
AESKey	This is the REF_CLK value in Hz. This is for the aes_key value
UserKey	This is for the user_key value.
RSASecureHash	This is for the rsa_key value for Ultrascale.
User128Bit	This is for the User 128 bit key value for Ultrascale.
JtagMioTDI	TDI MIO Pin Number for ZYNQ.
JtagMioTDO	TDO MIO Pin Number for ZYNQ.
JtagMioTCK	TCK MIO Pin Number for ZYNQ.
JtagMioTMS	TMS MIO Pin Number for ZYNQ.

Table 327: Structure XilSkey_EPI member description (cont'd)

Member	Description
JtagMioMuxSel	MUX Selection MIO Pin Number for ZYNQ.
JtagMuxSelLineDefVal	Value on the MUX Selection line for ZYNQ.
JtagGpioID	GPIO device ID.
HwmGpioStart	
HwmGpioReady	
HwmGpioEnd	
JtagGpioTDI	
JtagGpioTDO	
JtagGpioTMS	
JtagGpioTCK	
GpioInputCh	
GpioOutPutCh	
AESKeyReadback	AES key read only for Zynq.
UserKeyReadback	User key read in Ultrascale and Zynq.
CrcOfAESKey	Expected AES key's CRC for Ultrascale here we can't read AES key directly.
AESKeyMatched	
RSHashReadback	
User128BitReadBack	User 128 bit key read back for Ultrascale.
SystemInitDone	Internal variable to check if timer, XADC and JTAG are initialized.
FpgaFlag	
CrcToVerify	
NumSlr	
MasterSlr	
SlrConfigOrderIndex	

XilPM Library v3.1

XilPM Zynq UltraScale+ MPSoC APIs

Xilinx Power Management (XilPM) provides Embedded Energy Management Interface (EEMI) APIs for power management on Zynq UltraScale+ MPSoC. For more details about EEMI, see the Embedded Energy Management Interface (EEMI) API User Guide (UG1200).

Table 328: Quick Function Reference

Type	Name	Arguments
XStatus	XPm_InitXilpm	XIpiPsu * IpiInst
enum XPmBootStatus	XPm_GetBootStatus	void
void	XPm_SuspendFinalize	void
XStatus	pm_ipi_send	struct XPm_Master *const master u32 payload
XStatus	pm_ipi_buff_read32	struct XPm_Master *const master u32 * value1 u32 * value2 u32 * value3
XStatus	XPm_SelfSuspend	const enum XPmNodeId nid const u32 latency const u8 state const u64 address
XStatus	XPm_SetConfiguration	const u32 address
XStatus	XPm_InitFinalize	void

Table 328: Quick Function Reference (cont'd)

Type	Name	Arguments
XStatus	XPm_RequestSuspend	const enum XPmNodeId target const enum XPmRequestAck ack const u32 latency const u8 state
XStatus	XPm_RequestWakeUp	const enum XPmNodeId target const bool setAddress const u64 address const enum XPmRequestAck ack
XStatus	XPm_ForcePowerDown	const enum XPmNodeId target const enum XPmRequestAck ack
XStatus	XPm_AbortSuspend	const enum XPmAbortReason reason
XStatus	XPm_SetWakeUpSource	const enum XPmNodeId target const enum XPmNodeId wkup_node const u8 enable
XStatus	XPm_SystemShutdown	restart
XStatus	XPm_RequestNode	const enum XPmNodeId node const u32 capabilities const u32 qos const enum XPmRequestAck ack
XStatus	XPm_SetRequirement	const enum XPmNodeId nid const u32 capabilities const u32 qos const enum XPmRequestAck ack
XStatus	XPm_ReleaseNode	const enum XPmNodeId node
XStatus	XPm_SetMaxLatency	const enum XPmNodeId node const u32 latency
void	XPm_InitSuspendCb	const enum XPmSuspendReason reason const u32 latency const u32 state const u32 timeout

Table 328: Quick Function Reference (cont'd)

Type	Name	Arguments
void	XPm_AcknowledgeCb	const enum XPmNodeId node const XStatus status const u32 oppoint
void	XPm_NotifyCb	const enum XPmNodeId node const enum XPmNotifyEvent event const u32 oppoint
XStatus	XPm_GetApiVersion	u32 * version
XStatus	XPm_GetNodeStatus	const enum XPmNodeId node XPm_NodeStatus *const nodestatus
XStatus	XPm_GetOpCharacteristic	const enum XPmNodeId node const enum XPmOpCharType type u32 *const result
XStatus	XPm_ResetAssert	const enum XPmReset reset assert
XStatus	XPm_ResetGetStatus	const enum XPmReset reset u32 * status
XStatus	XPm_RegisterNotifier	XPm_Notifier *const notifier
XStatus	XPm_UnregisterNotifier	XPm_Notifier *const notifier
XStatus	XPm_MmioWrite	const u32 address const u32 mask const u32 value
XStatus	XPm_MmioRead	const u32 address u32 *const value
XStatus	XPm_ClockEnable	const enum XPmClock clock
XStatus	XPm_ClockDisable	const enum XPmClock clock
XStatus	XPm_ClockGetStatus	const enum XPmClock clock u32 *const status

Table 328: Quick Function Reference (cont'd)

Type	Name	Arguments
XStatus	XPm_ClockSetOneDivider	const enum XPmClock clock const u32 divider const u32 divId
XStatus	XPm_ClockSetDivider	const enum XPmClock clock const u32 divider
XStatus	XPm_ClockGetOneDivider	const enum XPmClock clock u32 *const divider
XStatus	XPm_ClockGetDivider	const enum XPmClock clock u32 *const divider
XStatus	XPm_ClockSetParent	const enum XPmClock clock const enum XPmClock parent
XStatus	XPm_ClockGetParent	const enum XPmClock clock enum XPmClock *const parent
XStatus	XPm_ClockSetRate	const enum XPmClock clock const u32 rate
XStatus	XPm_ClockGetRate	const enum XPmClock clock u32 *const rate
XStatus	XPm_PllSetParameter	const enum XPmNodeId node const enum XPmPllParam parameter const u32 value
XStatus	XPm_PllGetParameter	const enum XPmNodeId node const enum XPmPllParam parameter u32 *const value
XStatus	XPm_PllSetMode	const enum XPmNodeId node const enum XPmPllMode mode
XStatus	XPm_PllGetMode	const enum XPmNodeId node enum XPmPllMode *const mode
XStatus	XPm_PinCtrlAction	const u32 pin

Table 328: Quick Function Reference (cont'd)

Type	Name	Arguments
XStatus	XPm_PinCtrlRequest	const u32 pin
XStatus	XPm_PinCtrlRelease	const u32 pin
XStatus	XPm_PinCtrlSetFunction	const u32 pin const enum XPmPinFn fn
XStatus	XPm_PinCtrlGetFunction	const u32 pin enum XPmPinFn *const fn
XStatus	XPm_PinCtrlSetParameter	const u32 pin const enum XPmPinParam param const u32 value
XStatus	XPm_PinCtrlGetParameter	const u32 pin const enum XPmPinParam param u32 *const value

Functions

XPm_InitXilpm

Initialize xilpm library.

Note: None

Prototype

```
XStatus XPm_InitXilpm(XIpiPsu *IpiInst);
```

Parameters

The following table lists the `XPm_InitXilpm` function arguments.

Table 329: XPm_InitXilpm Arguments

Type	Name	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_GetBootStatus

This Function returns information about the boot reason. If the boot is not a system startup but a resume, power down request bitfield for this processor will be cleared.

Note: None

Prototype

```
enum
    XPmBootStatus
XPm_GetBootStatus(void);
```

Returns

Returns processor boot status

- PM_RESUME : If the boot reason is because of system resume.
- PM_INITIAL_BOOT : If this boot is the initial system startup.

XPm_SuspendFinalize

This Function waits for PMU to finish all previous API requests sent by the PU and performs client specific actions to finish suspend procedure (e.g. execution of wfi instruction on A53 and R5 processors).

Note: This function should not return if the suspend procedure is successful.

Prototype

```
void XPm_SuspendFinalize(void);
```

Returns

pm_ipi_send

Sends IPI request to the PMU.

Note: None

Prototype

```
XStatus pm_ipi_send(struct XPm_Master *const master, u32
payload[PAYLOAD_ARG_CNT]);
```

Parameters

The following table lists the `pm_ipi_send` function arguments.

Table 330: pm_ipi_send Arguments

Type	Name	Description
struct <code>XPm_Master</code> *const	master	Pointer to the master who is initiating request
u32	payload	API id and call arguments to be written in IPI buffer

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

pm_ipi_buff_read32

Reads IPI response after PMU has handled interrupt.

Note: None

Prototype

```
XStatus pm_ipi_buff_read32(struct XPm_Master *const master, u32 *value1,
u32 *value2, u32 *value3);
```

Parameters

The following table lists the `pm_ipi_buff_read32` function arguments.

Table 331: pm_ipi_buff_read32 Arguments

Type	Name	Description
struct <code>XPm_Master</code> *const	master	Pointer to the master who is waiting and reading response
u32 *	value1	Used to return value from 2nd IPI buffer element (optional)
u32 *	value2	Used to return value from 3rd IPI buffer element (optional)
u32 *	value3	Used to return value from 4th IPI buffer element (optional)

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_SelfSuspend

This function is used by a CPU to declare that it is about to suspend itself. After the PMU processes this call it will wait for the requesting CPU to complete the suspend procedure and become ready to be put into a sleep state.

Note: This is a blocking call, it will return only once PMU has responded

Prototype

```
XStatus XPm_SelfSuspend(const enum XPmNodeId nid, const u32 latency, const u8 state, const u64 address);
```

Parameters

The following table lists the `XPm_SelfSuspend` function arguments.

Table 332: XPm_SelfSuspend Arguments

Type	Name	Description
const enum XPmNodeId	nid	Node ID of the CPU node to be suspended.
const u32	latency	Maximum wake-up latency requirement in us(microsecs)
const u8	state	Instead of specifying a maximum latency, a CPU can also explicitly request a certain power state.
const u64	address	Address from which to resume when woken up.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_SetConfiguration

This function is called to configure the power management framework. The call triggers power management controller to load the configuration object and configure itself according to the content of the object.

Note: The provided address must be in 32-bit address space which is accessible by the PMU.

Prototype

```
XStatus XPm_SetConfiguration(const u32 address);
```

Parameters

The following table lists the `XPm_SetConfiguration` function arguments.

Table 333: XPm_SetConfiguration Arguments

Type	Name	Description
const u32	address	Start address of the configuration object

Returns

XST_SUCCESS if successful, otherwise an error code

XPm_InitFinalize

This function is called to notify the power management controller about the completed power management initialization.

Note: It is assumed that all used nodes are requested when this call is made. The power management controller may power down the nodes which are not requested after this call is processed.

Prototype

```
XStatus XPm_InitFinalize(void);
```

Returns

XST_SUCCESS if successful, otherwise an error code

XPm_RequestSuspend

This function is used by a PU to request suspend of another PU. This call triggers the power management controller to notify the PU identified by 'nodeID' that a suspend has been requested. This will allow said PU to gracefully suspend itself by calling XPm_SelfSuspend for each of its CPU nodes, or else call XPm_AbortSuspend with its PU node as argument and specify the reason.

Note: If 'ack' is set to PM_ACK_NON_BLOCKING, the requesting PU will be notified upon completion of suspend or if an error occurred, such as an abort. REQUEST_ACK_BLOCKING is not supported for this command.

Prototype

```
XStatus XPm_RequestSuspend(const enum XPmNodeId target, const enum XPmRequestAck ack, const u32 latency, const u8 state);
```

Parameters

The following table lists the XPm_RequestSuspend function arguments.

Table 334: XPm_RequestSuspend Arguments

Type	Name	Description
const enum XPmNodeId	target	Node ID of the PU node to be suspended
const enum XPmRequestAck	ack	Requested acknowledge type
const u32	latency	Maximum wake-up latency requirement in us(micro sec)
const u8	state	Instead of specifying a maximum latency, a PU can also explicitly request a certain power state.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_RequestWakeUp

This function can be used to request power up of a CPU node within the same PU, or to power up another PU.

Note: If acknowledge is requested, the calling PU will be notified by the power management controller once the wake-up is completed.

Prototype

```
XStatus XPm_RequestWakeUp(const enum XPmNodeId target, const bool
setAddress, const u64 address, const enum XPmRequestAck ack);
```

Parameters

The following table lists the XPm_RequestWakeUp function arguments.

Table 335: XPm_RequestWakeUp Arguments

Type	Name	Description
const enum XPmNodeId	target	Node ID of the CPU or PU to be powered/woken up.
const bool	setAddress	Specifies whether the start address argument is being passed. <ul style="list-style-type: none"> 0 : do not set start address 1 : set start address
const u64	address	Address from which to resume when woken up. Will only be used if setAddress is 1.
const enum XPmRequestAck	ack	Requested acknowledge type

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_ForcePowerDown

One PU can request a forced poweroff of another PU or its power island or power domain. This can be used for killing an unresponsive PU, in which case all resources of that PU will be automatically released.

Note: Force power down may not be requested by a PU for itself.

Prototype

```
XStatus XPm_ForcePowerDown(const enum XPmNodeId target, const enum
XPmRequestAck ack);
```

Parameters

The following table lists the `XPm_ForcePowerDown` function arguments.

Table 336: XPm_ForcePowerDown Arguments

Type	Name	Description
const enum XPmNodeId	target	Node ID of the PU node or power island/domain to be powered down.
const enum XPmRequestAck	ack	Requested acknowledge type

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_AbortSuspend

This function is called by a CPU after a `XPm_SelfSuspend` call to notify the power management controller that CPU has aborted suspend or in response to an init suspend request when the PU refuses to suspend.

Note: Calling PU expects the PMU to abort the initiated suspend procedure. This is a non-blocking call without any acknowledge.

Prototype

```
XStatus XPm_AbortSuspend(const enum XPmAbortReason reason);
```

Parameters

The following table lists the `XPm_AbortSuspend` function arguments.

Table 337: XPm_AbortSuspend Arguments

Type	Name	Description
const enum XPmAbortReason	reason	Reason code why the suspend can not be performed or completed <ul style="list-style-type: none"> ABORT_REASON_WKUP_EVENT : local wakeup-event received ABORT_REASON_PU_BUSY : PU is busy ABORT_REASON_NO_PWRDN : no external powerdown supported ABORT_REASON_UNKNOWN : unknown error during suspend procedure

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_SetWakeUpSource

This function is called by a PU to add or remove a wake-up source prior to going to suspend. The list of wake sources for a PU is automatically cleared whenever the PU is woken up or when one of its CPUs aborts the suspend procedure.

Note: Declaring a node as a wakeup source will ensure that the node will not be powered off. It also will cause the PMU to configure the GIC Proxy accordingly if the FPD is powered off.

Prototype

```
XStatus XPm_SetWakeUpSource(const enum XPmNodeId target, const enum XPmNodeId wkup_node, const u8 enable);
```

Parameters

The following table lists the `XPm_SetWakeUpSource` function arguments.

Table 338: XPm_SetWakeUpSource Arguments

Type	Name	Description
const enum XPmNodeId	target	Node ID of the target to be woken up.
const enum XPmNodeId	wkup_node	Node ID of the wakeup device.
const u8	enable	Enable flag: <ul style="list-style-type: none"> 1 : the wakeup source is added to the list 0 : the wakeup source is removed from the list

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_SystemShutdown

This function can be used by a privileged PU to shut down or restart the complete device.

Note: In either case the PMU will call XPm_InitSuspendCb for each of the other PUs, allowing them to gracefully shut down. If a PU is asleep it will be woken up by the PMU. The PU making the XPm_SystemShutdown should perform its own suspend procedure after calling this API. It will not receive an init suspend callback.

Prototype

```
XStatus XPm_SystemShutdown(u32 type, u32 subtype);
```

Parameters

The following table lists the XPm_SystemShutdown function arguments.

Table 339: XPm_SystemShutdown Arguments

Type	Name	Description
Commented parameter restart does not exist in function XPm_SystemShutdown.	restart	Should the system be restarted automatically? <ul style="list-style-type: none"> PM_SHUTDOWN : no restart requested, system will be powered off permanently PM_RESTART : restart is requested, system will go through a full reset

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_RequestNode

Used to request the usage of a PM-slave. Using this API call a PU requests access to a slave device and asserts its requirements on that device. Provided the PU is sufficiently privileged, the PMU will enable access to the memory mapped region containing the control registers of that device. For devices that can only be serving a single PU, any other privileged PU will now be blocked from accessing this device until the node is released.

Note: None

Prototype

```
XStatus XPm_RequestNode(const enum XPmNodeId node, const u32 capabilities,
const u32 qos, const enum XPmRequestAck ack);
```

Parameters

The following table lists the `XPm_RequestNode` function arguments.

Table 340: `XPm_RequestNode` Arguments

Type	Name	Description
const enum XPmNodeId	node	Node ID of the PM slave requested
const u32	capabilities	Slave-specific capabilities required, can be combined <ul style="list-style-type: none"> • <code>PM_CAP_ACCESS</code> : full access / functionality • <code>PM_CAP_CONTEXT</code> : preserve context • <code>PM_CAP_WAKEUP</code> : emit wake interrupts
const u32	qos	Quality of Service (0-100) required
const enum XPmRequestAck	ack	Requested acknowledge type

Returns

`XST_SUCCESS` if successful else `XST_FAILURE` or an error code or a reason code

XPm_SetRequirement

This function is used by a PU to announce a change in requirements for a specific slave node which is currently in use.

Note: If this function is called after the last awake CPU within the PU calls `SelfSuspend`, the requirement change shall be performed after the CPU signals the end of suspend to the power management controller, (e.g. WFI interrupt).

Prototype

```
XStatus XPm_SetRequirement(const enum XPmNodeId nid, const u32
capabilities, const u32 qos, const enum XPmRequestAck ack);
```

Parameters

The following table lists the `XPm_SetRequirement` function arguments.

Table 341: XPm_SetRequirement Arguments

Type	Name	Description
const enum XPmNodeId	nid	Node ID of the PM slave.
const u32	capabilities	Slave-specific capabilities required.
const u32	qos	Quality of Service (0-100) required.
const enum XPmRequestAck	ack	Requested acknowledge type

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_ReleaseNode

This function is used by a PU to release the usage of a PM slave. This will tell the power management controller that the node is no longer needed by that PU, potentially allowing the node to be placed into an inactive state.

Note: None

Prototype

```
XStatus XPm_ReleaseNode(const enum XPmNodeId node);
```

Parameters

The following table lists the `XPm_ReleaseNode` function arguments.

Table 342: XPm_ReleaseNode Arguments

Type	Name	Description
const enum XPmNodeId	node	Node ID of the PM slave.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_SetMaxLatency

This function is used by a PU to announce a change in the maximum wake-up latency requirements for a specific slave node currently used by that PU.

Note: Setting maximum wake-up latency can constrain the set of possible power states a resource can be put into.

Prototype

```
XStatus XPm_SetMaxLatency(const enum XPmNodeId node, const u32 latency);
```

Parameters

The following table lists the `XPm_SetMaxLatency` function arguments.

Table 343: XPm_SetMaxLatency Arguments

Type	Name	Description
const enum XPmNodeId	node	Node ID of the PM slave.
const u32	latency	Maximum wake-up latency required.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_InitSuspendCb

Callback function to be implemented in each PU, allowing the power management controller to request that the PU suspend itself.

Note: If the PU fails to act on this request the power management controller or the requesting PU may choose to employ the forceful power down option.

Prototype

```
void XPm_InitSuspendCb(const enum XPmSuspendReason reason, const u32 latency, const u32 state, const u32 timeout);
```

Parameters

The following table lists the `XPm_InitSuspendCb` function arguments.

Table 344: XPm_InitSuspendCb Arguments

Type	Name	Description
const enum XPmSuspendReason	reason	Suspend reason: <ul style="list-style-type: none"> SUSPEND_REASON_PU_REQ : Request by another PU SUSPEND_REASON_ALERT : Unrecoverable SysMon alert SUSPEND_REASON_SHUTDOWN : System shutdown SUSPEND_REASON_RESTART : System restart
const u32	latency	Maximum wake-up latency in us(micro secs). This information can be used by the PU to decide what level of context saving may be required.

Table 344: **XPm_InitSuspendCb Arguments** (cont'd)

Type	Name	Description
const u32	state	Targeted sleep/suspend state.
const u32	timeout	Timeout in ms, specifying how much time a PU has to initiate its suspend procedure before it's being considered unresponsive.

Returns

None

XPm_AcknowledgeCb

This function is called by the power management controller in response to any request where an acknowledge callback was requested, i.e. where the 'ack' argument passed by the PU was REQUEST_ACK_NON_BLOCKING.

Note: None

Prototype

```
void XPm_AcknowledgeCb(const enum XPmNodeId node, const XStatus status,
const u32 oppoint);
```

Parameters

The following table lists the `XPm_AcknowledgeCb` function arguments.

 Table 345: **XPm_AcknowledgeCb Arguments**

Type	Name	Description
const enum XPmNodeId	node	ID of the component or sub-system in question.
const XStatus	status	Status of the operation: <ul style="list-style-type: none"> OK: the operation completed successfully ERR: the requested operation failed
const u32	oppoint	Operating point of the node in question

Returns

None

XPm_NotifyCb

This function is called by the power management controller if an event the PU was registered for has occurred. It will populate the notifier data structure passed when calling `XPm_RegisterNotifier`.

Note: None

Prototype

```
void XPm_NotifyCb(const enum XPmNodeId node, const enum XPmNotifyEvent
event, const u32 oppoint);
```

Parameters

The following table lists the `XPm_NotifyCb` function arguments.

Table 346: XPm_NotifyCb Arguments

Type	Name	Description
const enum XPmNodeId	node	ID of the node the event notification is related to.
const enum XPmNotifyEvent	event	ID of the event
const u32	oppoint	Current operating state of the node.

Returns

None

XPm_GetApiVersion

This function is used to request the version number of the API running on the power management controller.

Note: None

Prototype

```
XStatus XPm_GetApiVersion(u32 *version);
```

Parameters

The following table lists the `XPm_GetApiVersion` function arguments.

Table 347: XPm_GetApiVersion Arguments

Type	Name	Description
u32 *	version	Returns the API 32-bit version number. Returns 0 if no PM firmware present.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_GetNodeStatus

This function is used to obtain information about the current state of a component. The caller must pass a pointer to an `XPm_NodeStatus` structure, which must be pre-allocated by the caller.

- status - The current power state of the requested node.
 - For CPU nodes:
 - 0 : if CPU is powered down,
 - 1 : if CPU is active (powered up),
 - 2 : if CPU is suspending (powered up)
 - For power islands and power domains:
 - 0 : if island is powered down,
 - 1 : if island is powered up
 - For PM slaves:
 - 0 : if slave is powered down,
 - 1 : if slave is powered up,
 - 2 : if slave is in retention
- requirement - Slave nodes only: Returns current requirements the requesting PU has requested of the node.
- usage - Slave nodes only: Returns current usage status of the node:
 - 0 : node is not used by any PU,
 - 1 : node is used by caller exclusively,
 - 2 : node is used by other PU(s) only,
 - 3 : node is used by caller and by other PU(s)

Note: None

Prototype

```
XStatus XPm_GetNodeStatus(const enum XPmNodeId node, XPm_NodeStatus *const
nodestatus);
```

Parameters

The following table lists the `XPm_GetNodeStatus` function arguments.

Table 348: XPm_GetNodeStatus Arguments

Type	Name	Description
const enum <code>XPmNodeId</code>	node	ID of the component or sub-system in question.
<code>XPm_NodeStatus</code> *const	nodestatus	Used to return the complete status of the node.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_GetOpCharacteristic

Call this function to request the power management controller to return information about an operating characteristic of a component.

Note: None

Prototype

```
XStatus XPm_GetOpCharacteristic(const enum XPmNodeId node, const enum
XPmOpCharType type, u32 *const result);
```

Parameters

The following table lists the `XPm_GetOpCharacteristic` function arguments.

Table 349: XPm_GetOpCharacteristic Arguments

Type	Name	Description
const enum <code>XPmNodeId</code>	node	ID of the component or sub-system in question.
const enum <code>XPmOpCharType</code>	type	Type of operating characteristic requested: <ul style="list-style-type: none"> power (current power consumption), latency (current latency in us to return to active state), temperature (current temperature),
u32 *const	result	Used to return the requested operating characteristic.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_ResetAssert

This function is used to assert or release reset for a particular reset line. Alternatively a reset pulse can be requested as well.

Note: None

Prototype

```
XStatus XPm_ResetAssert(const enum XPmReset reset, const enum
XPmResetAction resetaction);
```

Parameters

The following table lists the `XPm_ResetAssert` function arguments.

Table 350: XPm_ResetAssert Arguments

Type	Name	Description
const enum XPmReset	reset	ID of the reset line
Commented parameter assert does not exist in function <code>XPm_ResetAssert</code> .	assert	Identifies action: <ul style="list-style-type: none"> PM_RESET_ACTION_RELEASE : release reset, PM_RESET_ACTION_ASSERT : assert reset, PM_RESET_ACTION_PULSE : pulse reset,

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_ResetGetStatus

Call this function to get the current status of the selected reset line.

Note: None

Prototype

```
XStatus XPm_ResetGetStatus(const enum XPmReset reset, u32 *status);
```

Parameters

The following table lists the `XPm_ResetGetStatus` function arguments.

Table 351: XPm_ResetGetStatus Arguments

Type	Name	Description
const enum <code>XPmReset</code>	reset	Reset line
u32 *	status	Status of specified reset (true - asserted, false - released)

Returns

Returns 1/XST_FAILURE for 'asserted' or 0/XST_SUCCESS for 'released'.

XPm_RegisterNotifier

A PU can call this function to request that the power management controller call its notify callback whenever a qualifying event occurs. One can request to be notified for a specific or any event related to a specific node.

- `nodeID` : ID of the node to be notified about,
- `eventID` : ID of the event in question, '-1' denotes all events (- EVENT_STATE_CHANGE, EVENT_ZERO_USERS),
- `wake` : true: wake up on event, false: do not wake up (only notify if awake), no buffering/queueing
- `callback` : Pointer to the custom callback function to be called when the notification is available. The callback executes from interrupt context, so the user must take special care when implementing the callback. Callback is optional, may be set to NULL.
- `received` : Variable indicating how many times the notification has been received since the notifier is registered.

Note: The caller shall initialize the notifier object before invoking the `XPm_RegisteredNotifier` function. While notifier is registered, the notifier object shall not be modified by the caller.

Prototype

```
XStatus XPm_RegisterNotifier(XPm_Notifier *const notifier);
```

Parameters

The following table lists the `XPm_RegisterNotifier` function arguments.

Table 352: XPm_RegisterNotifier Arguments

Type	Name	Description
<code>XPm_Notifier</code> *const	notifier	Pointer to the notifier object to be associated with the requested notification. The notifier object contains the following data related to the notification:

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_UnregisterNotifier

A PU calls this function to unregister for the previously requested notifications.

Note: None

Prototype

```
XStatus XPm_UnregisterNotifier(XPm_Notifier *const notifier);
```

Parameters

The following table lists the `XPm_UnregisterNotifier` function arguments.

Table 353: XPm_UnregisterNotifier Arguments

Type	Name	Description
<code>XPm_Notifier *const</code>	notifier	Pointer to the notifier object associated with the previously requested notification

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_MmioWrite

Call this function to write a value directly into a register that isn't accessible directly, such as registers in the clock control unit. This call is bypassing the power management logic. The permitted addresses are subject to restrictions as defined in the PCW configuration.

Note: If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_MmioWrite(const u32 address, const u32 mask, const u32 value);
```

Parameters

The following table lists the `XPm_MmioWrite` function arguments.

Table 354: XPm_MmioWrite Arguments

Type	Name	Description
const u32	address	Physical 32-bit address of memory mapped register to write to.
const u32	mask	32-bit value used to limit write to specific bits in the register.
const u32	value	Value to write to the register bits specified by the mask.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_MmioRead

Call this function to read a value from a register that isn't accessible directly. The permitted addresses are subject to restrictions as defined in the PCW configuration.

Note: If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_MmioRead(const u32 address, u32 *const value);
```

Parameters

The following table lists the XPm_MmioRead function arguments.

Table 355: XPm_MmioRead Arguments

Type	Name	Description
const u32	address	Physical 32-bit address of memory mapped register to read from.
u32 *const	value	Returns the 32-bit value read from the register

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_ClockEnable

Call this function to enable (activate) a clock.

Note: If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_ClockEnable(const enum XPmClock clock);
```

Parameters

The following table lists the `XPm_ClockEnable` function arguments.

Table 356: XPm_ClockEnable Arguments

Type	Name	Description
const enum <code>XPmClock</code>	clock	Identifier of the target clock to be enabled

Returns

Status of performing the operation as returned by the PMU-FW

XPm_ClockDisable

Call this function to disable (gate) a clock.

Note: If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_ClockDisable(const enum XPmClock clock);
```

Parameters

The following table lists the `XPm_ClockDisable` function arguments.

Table 357: XPm_ClockDisable Arguments

Type	Name	Description
const enum <code>XPmClock</code>	clock	Identifier of the target clock to be disabled

Returns

Status of performing the operation as returned by the PMU-FW

XPm_ClockGetStatus

Call this function to get status of a clock gate state.

Prototype

```
XStatus XPm_ClockGetStatus(const enum XPmClock clock, u32 *const status);
```

Parameters

The following table lists the `XPm_ClockGetStatus` function arguments.

Table 358: XPm_ClockGetStatus Arguments

Type	Name	Description
const enum XPmClock	clock	Identifier of the target clock
u32 *const	status	Location to store clock gate state (1=enabled, 0=disabled)

Returns

Status of performing the operation as returned by the PMU-FW

XPm_ClockSetOneDivider

Call this function to set divider for a clock.

Note: If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_ClockSetOneDivider(const enum XPmClock clock, const u32
divider, const u32 divId);
```

Parameters

The following table lists the XPm_ClockSetOneDivider function arguments.

Table 359: XPm_ClockSetOneDivider Arguments

Type	Name	Description
const enum XPmClock	clock	Identifier of the target clock
const u32	divider	Divider value to be set
const u32	divId	ID of the divider to be set

Returns

Status of performing the operation as returned by the PMU-FW

XPm_ClockSetDivider

Call this function to set divider for a clock.

Note: If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_ClockSetDivider(const enum XPmClock clock, const u32 divider);
```

Parameters

The following table lists the `XPm_ClockSetDivider` function arguments.

Table 360: XPm_ClockSetDivider Arguments

Type	Name	Description
const enum XPmClock	clock	Identifier of the target clock
const u32	divider	Divider value to be set

Returns

XST_INVALID_PARAM or status of performing the operation as returned by the PMU-FW

XPm_ClockGetOneDivider

Local function to get one divider (DIV0 or DIV1) of a clock.

Prototype

```
XStatus XPm_ClockGetOneDivider(const enum XPmClock clock, u32 *const
divider, const u32 divId);
```

Parameters

The following table lists the `XPm_ClockGetOneDivider` function arguments.

Table 361: XPm_ClockGetOneDivider Arguments

Type	Name	Description
const enum XPmClock	clock	Identifier of the target clock
u32 *const	divider	Location to store the divider value

Returns

Status of performing the operation as returned by the PMU-FW

XPm_ClockGetDivider

Call this function to get divider of a clock.

Prototype

```
XStatus XPm_ClockGetDivider(const enum XPmClock clock, u32 *const divider);
```

Parameters

The following table lists the `XPm_ClockGetDivider` function arguments.

Table 362: `XPm_ClockGetDivider` Arguments

Type	Name	Description
const enum <code>XPmClock</code>	clock	Identifier of the target clock
u32 *const	divider	Location to store the divider value

Returns

`XST_INVALID_PARAM` or status of performing the operation as returned by the PMU-FW

`XPm_ClockSetParent`

Call this function to set parent for a clock.

Note: If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_ClockSetParent(const enum XPmClock clock, const enum XPmClock parent);
```

Parameters

The following table lists the `XPm_ClockSetParent` function arguments.

Table 363: `XPm_ClockSetParent` Arguments

Type	Name	Description
const enum <code>XPmClock</code>	clock	Identifier of the target clock
const enum <code>XPmClock</code>	parent	Identifier of the target parent clock

Returns

`XST_INVALID_PARAM` or status of performing the operation as returned by the PMU-FW.

`XPm_ClockGetParent`

Call this function to get parent of a clock.

Prototype

```
XStatus XPm_ClockGetParent(const enum XPmClock clock, enum XPmClock *const parent);
```


Parameters

The following table lists the `XPm_ClockGetParent` function arguments.

Table 364: `XPm_ClockGetParent` Arguments

Type	Name	Description
const enum <code>XPmClock</code>	clock	Identifier of the target clock
enum <code>XPmClock</code> *const	parent	Location to store clock parent ID

Returns

`XST_INVALID_PARAM` or status of performing the operation as returned by the PMU-FW.

`XPm_ClockSetRate`

Call this function to set rate of a clock.

Note: If the action isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_ClockSetRate(const enum XPmClock clock, const u32 rate);
```

Parameters

The following table lists the `XPm_ClockSetRate` function arguments.

Table 365: `XPm_ClockSetRate` Arguments

Type	Name	Description
const enum <code>XPmClock</code>	clock	Identifier of the target clock
const u32	rate	Clock frequency (rate) to be set

Returns

Status of performing the operation as returned by the PMU-FW

`XPm_ClockGetRate`

Call this function to get rate of a clock.

Prototype

```
XStatus XPm_ClockGetRate(const enum XPmClock clock, u32 *const rate);
```

Parameters

The following table lists the `XPm_ClockGetRate` function arguments.

Table 366: XPm_ClockGetRate Arguments

Type	Name	Description
const enum <code>XPmClock</code>	clock	Identifier of the target clock
u32 *const	rate	Location where the rate should be stored

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PllSetParameter

Call this function to set a PLL parameter.

Note: If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_PllSetParameter(const enum XPmNodeId node, const enum
XPmPllParam parameter, const u32 value);
```

Parameters

The following table lists the `XPm_PllSetParameter` function arguments.

Table 367: XPm_PllSetParameter Arguments

Type	Name	Description
const enum <code>XPmNodeId</code>	node	PLL node identifier
const enum <code>XPmPllParam</code>	parameter	PLL parameter identifier
const u32	value	Value of the PLL parameter

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PllGetParameter

Call this function to get a PLL parameter.

Prototype

```
XStatus XPm_PllGetParameter(const enum XPmNodeId node, const enum
XPmPllParam parameter, u32 *const value);
```

Parameters

The following table lists the `XPm_PllGetParameter` function arguments.

Table 368: XPm_PllGetParameter Arguments

Type	Name	Description
const enum XPmNodeId	node	PLL node identifier
const enum XPmPllParam	parameter	PLL parameter identifier
u32 *const	value	Location to store value of the PLL parameter

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PllSetMode

Call this function to set a PLL mode.

Note: If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_PllSetMode(const enum XPmNodeId node, const enum XPmPllMode
mode);
```

Parameters

The following table lists the `XPm_PllSetMode` function arguments.

Table 369: XPm_PllSetMode Arguments

Type	Name	Description
const enum XPmNodeId	node	PLL node identifier
const enum XPmPllMode	mode	PLL mode to be set

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PllGetMode

Call this function to get a PLL mode.

Prototype

```
XStatus XPm_PllGetMode(const enum XPmNodeId node, enum XPmPllMode *const mode);
```

Parameters

The following table lists the `XPm_PllGetMode` function arguments.

Table 370: XPm_PllGetMode Arguments

Type	Name	Description
const enum <code>XPmNodeId</code>	node	PLL node identifier
enum <code>XPmPllMode</code> *const	mode	Location to store the PLL mode

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PinCtrlAction

Locally used function to request or release a pin control.

Prototype

```
XStatus XPm_PinCtrlAction(const u32 pin, const enum XPmApiId api);
```

Parameters

The following table lists the `XPm_PinCtrlAction` function arguments.

Table 371: XPm_PinCtrlAction Arguments

Type	Name	Description
const u32	pin	PIN identifier (index from range 0-77) @api API identifier (request or release pin control)

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PinCtrlRequest

Call this function to request a pin control.

Prototype

```
XStatus XPm_PinCtrlRequest(const u32 pin);
```

Parameters

The following table lists the `XPm_PinCtrlRequest` function arguments.

Table 372: XPm_PinCtrlRequest Arguments

Type	Name	Description
const u32	pin	PIN identifier (index from range 0-77)

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PinCtrlRelease

Call this function to release a pin control.

Prototype

```
XStatus XPm_PinCtrlRelease(const u32 pin);
```

Parameters

The following table lists the `XPm_PinCtrlRelease` function arguments.

Table 373: XPm_PinCtrlRelease Arguments

Type	Name	Description
const u32	pin	PIN identifier (index from range 0-77)

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PinCtrlSetFunction

Call this function to set a pin function.

Note: If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_PinCtrlSetFunction(const u32 pin, const enum XPmPinFn fn);
```

Parameters

The following table lists the `XPm_PinCtrlSetFunction` function arguments.

Table 374: XPm_PinCtrlSetFunction Arguments

Type	Name	Description
const u32	pin	Pin identifier
const enum XPmPinFn	fn	Pin function to be set

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PinCtrlGetFunction

Call this function to get currently configured pin function.

Prototype

```
XStatus XPm_PinCtrlGetFunction(const u32 pin, enum XPmPinFn *const fn);
```

Parameters

The following table lists the `XPm_PinCtrlGetFunction` function arguments.

Table 375: XPm_PinCtrlGetFunction Arguments

Type	Name	Description
const u32	pin	PLL node identifier
enum XPmPinFn *const	fn	Location to store the pin function

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PinCtrlSetParameter

Call this function to set a pin parameter.

Note: If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_PinCtrlSetParameter(const u32 pin, const enum XPmPinParam
param, const u32 value);
```

Parameters

The following table lists the `XPm_PinCtrlSetParameter` function arguments.

Table 376: XPm_PinCtrlSetParameter Arguments

Type	Name	Description
const u32	pin	Pin identifier
const enum XPmPinParam	param	Pin parameter identifier
const u32	value	Value of the pin parameter to set

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PinCtrlGetParameter

Call this function to get currently configured value of pin parameter.

Prototype

```
XStatus XPm_PinCtrlGetParameter(const u32 pin, const enum XPmPinParam
param, u32 *const value);
```

Parameters

The following table lists the `XPm_PinCtrlGetParameter` function arguments.

Table 377: XPm_PinCtrlGetParameter Arguments

Type	Name	Description
const u32	pin	Pin identifier
const enum XPmPinParam	param	Pin parameter identifier
u32 *const	value	Location to store value of the pin parameter

Returns

Status of performing the operation as returned by the PMU-FW

Error Status

This section lists the Power management specific return error statuses.

PLM error codes format is '0xXXXXYYYY'. Where:

- XXXX - PLM/LOADER/XPLMI error codes as defined in the xplmi_status.h file.
- YYYY - Libraries / Drivers error code as defined in respective modules.

Definitions

Define XST_PM_INTERNAL

Definition

```
#define XST_PM_INTERNAL2000L
```

Description

An internal error occurred while performing the requested operation

Define XST_PM_CONFLICT

Definition

```
#define XST_PM_CONFLICT2001L
```

Description

Conflicting requirements have been asserted when more than one processing cluster is using the same PM slave

Define XST_PM_NO_ACCESS

Definition

```
#define XST_PM_NO_ACCESS2002L
```

Description

The processing cluster does not have access to the requested node or operation

Define XST_PM_INVALID_NODE

Definition

```
#define XST_PM_INVALID_NODE2003L
```

Description

The API function does not apply to the node passed as argument

Define XST_PM_DOUBLE_REQ

Definition

```
#define XST_PM_DOUBLE_REQ2004L
```

Description

A processing cluster has already been assigned access to a PM slave and has issued a duplicate request for that PM slave

Define XST_PM_ABORT_SUSPEND

Definition

```
#define XST_PM_ABORT_SUSPEND2005L
```

Description

The target processing cluster has aborted suspend

Define XST_PM_TIMEOUT

Definition

```
#define XST_PM_TIMEOUT2006L
```

Description

A timeout occurred while performing the requested operation

Define XST_PM_NODE_USED

Definition

```
#define XST_PM_NODE_USED2007L
```

Description

Slave request cannot be granted since node is non-shareable and used

Data Structure Index

The following is a list of data structures:

- [XPm_Master](#)
- [XPm_NodeStatus](#)
- [XPm_Notifier](#)
- [pm_acknowledge](#)
- [pm_init_suspend](#)

pm_acknowledge

Declaration

```
typedef struct
{
    u8 received,
    u32 node,
    XStatus status,
    u32 opp
} pm_acknowledge;
```

Table 378: Structure pm_acknowledge member description

Member	Description
received	Has acknowledge argument been received?
node	Node argument about which the acknowledge is
status	Acknowledged status
opp	Operating point of node in question

pm_init_suspend

Declaration

```
typedef struct
{
    u8 received,
    enum XPmSuspendReason reason,
    u32 latency,
    u32 state,
    u32 timeout
} pm_init_suspend;
```

Table 379: Structure pm_init_suspend member description

Member	Description
received	Has init suspend callback been received/handled
reason	Reason of initializing suspend
latency	Maximum allowed latency
state	Targeted sleep/suspend state
timeout	Period of time the client has to response

XPm_Master

[XPm_Master](#) - Master structure

Declaration

```
typedef struct
{
    enum XPmNodeId node_id,
    const u32 pwrctl,
    const u32 pwrdn_mask,
    XIpiPsu * ipi
} XPm_Master;
```

Table 380: Structure XPm_Master member description

Member	Description
node_id	Node ID
pwrctl	
pwrdn_mask	< Power Control Register Address Power Down Mask
ipi	IPI Instance

XPm_NodeStatus

[XPm_NodeStatus](#) - struct containing node status information

Declaration

```
typedef struct
{
    u32 status,
    u32 requirements,
    u32 usage
} XPm_NodeStatus;
```

Table 381: Structure XPm_NodeStatus member description

Member	Description
status	Node power state
requirements	Current requirements asserted on the node (slaves only)
usage	Usage information (which master is currently using the slave)

XPm_Notifier

XPm_Notifier - Notifier structure registered with a callback by app

Declaration

```
typedef struct
{
    void(*const callback)(struct XPm_Ntfier *const notifier),
    const u32 node,
    enum XPmNotifyEvent event,
    u32 flags,
    u32 oppoint,
    u32 received,
    struct XPm_Ntfier * next
} XPm_Notifier;
```

Table 382: Structure XPm_Notifier member description

Member	Description
callback	Custom callback handler to be called when the notification is received. The custom handler would execute from interrupt context, it shall return quickly and must not block! (enables event-driven notifications)
node	Node argument (the node to receive notifications about)
event	Event argument (the event type to receive notifications about)
flags	Flags
oppoint	Operating point of node in question. Contains the value updated when the last event notification is received. User shall not modify this value while the notifier is registered.
received	How many times the notification has been received - to be used by application (enables polling). User shall not modify this value while the notifier is registered.
next	Pointer to next notifier in linked list. Must not be modified while the notifier is registered. User shall not ever modify this value.

XiIFPGA Library v5.2

Overview

The XiIFPGA library provides an interface to the Linux or bare-metal users for configuring the programmable logic (PL) over PCAP from PS. The library is designed for Zynq UltraScale+ MPSoC to run on top of Xilinx standalone BSPs. It is tested for A53, R5 and MicroBlaze. In the most common use case, we expect users to run this library on the PMU MicroBlaze with PMUFW to serve requests from either Linux or Uboot for Bitstream programming.

Note: XiIFPGA does not support a DDR less system. DDR must be present for use of XiIFPGA.

Supported Features

The following features are supported in Zynq UltraScale+ MPSoC platform.

- Full bitstream loading
- Partial bitstream loading
- Encrypted bitstream loading
- Authenticated bitstream loading
- Authenticated and encrypted bitstream loading
- Readback of configuration registers
- Readback of configuration data

XiIFPGA library Interface modules

XiIFPGA library uses the below major components to configure the PL through PS.

Processor Configuration Access Port (PCAP)

The processor configuration access port (PCAP) is used to configure the programmable logic (PL) through the PS.

CSU DMA driver

The CSU DMA driver is used to transfer the actual bitstream file for the PS to PL after PCAP initialization.

XilSecure Library

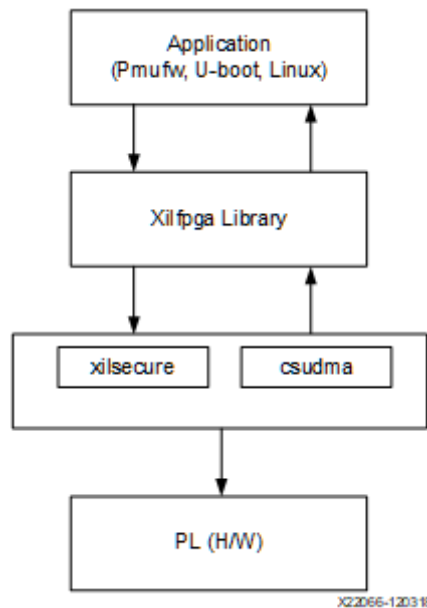
The XilSecure library provides APIs to access secure hardware on the Zynq UltraScale+ MPSoC devices.

Note: The current version of library supports only Zynq UltraScale MPSoC devices.

Design Summary

XilFPGA library acts as a bridge between the user application and the PL device. It provides the required functionality to the user application for configuring the PL Device with the required bitstream. The following figure illustrates an implementation where the XilFPGA library needs the CSU DMA driver APIs to transfer the bitstream from the DDR to the PL region. The XilFPGA library also needs the XilSecure library APIs to support programming authenticated and encrypted bitstream files.

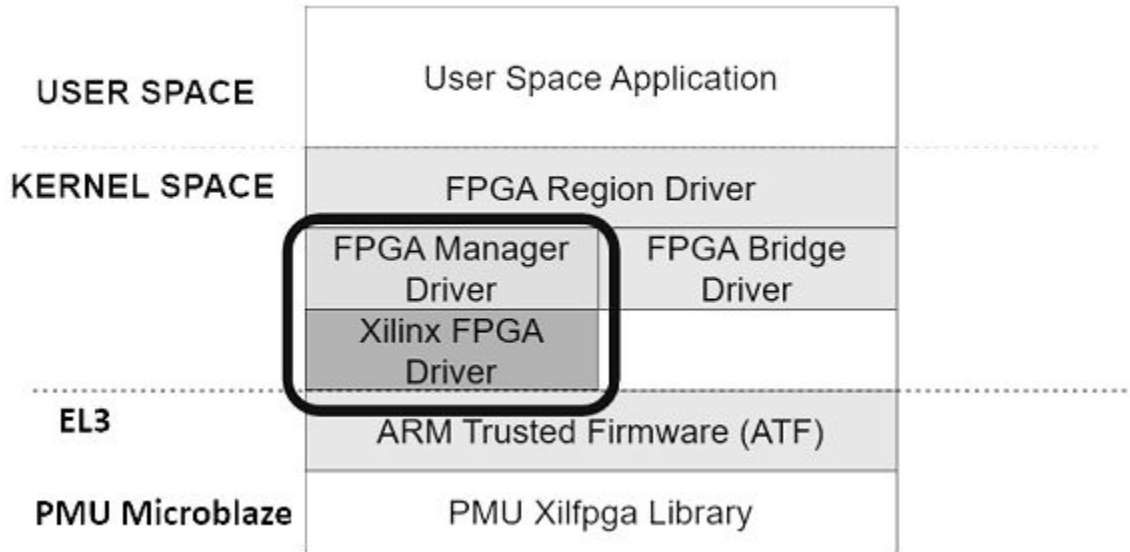
Figure 81: XilFPGA Design Summary



Flow Diagram

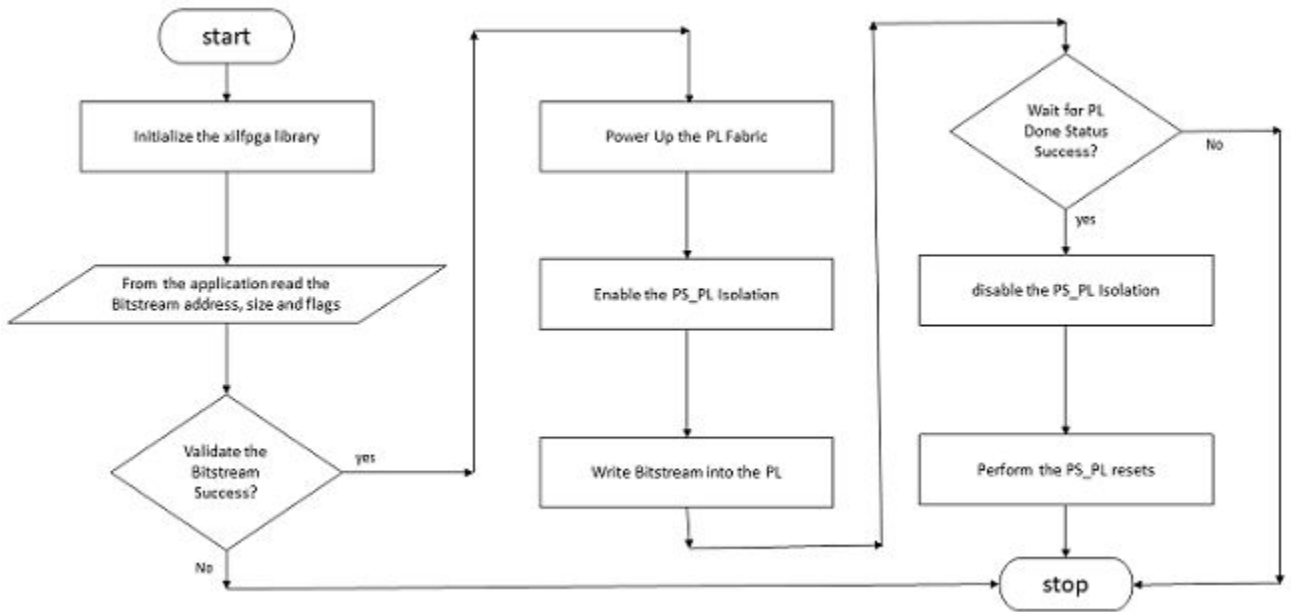
The following figure illustrates the Bitstream loading flow on the Linux operating system.

Figure 82: Bitstream loading on Linux:



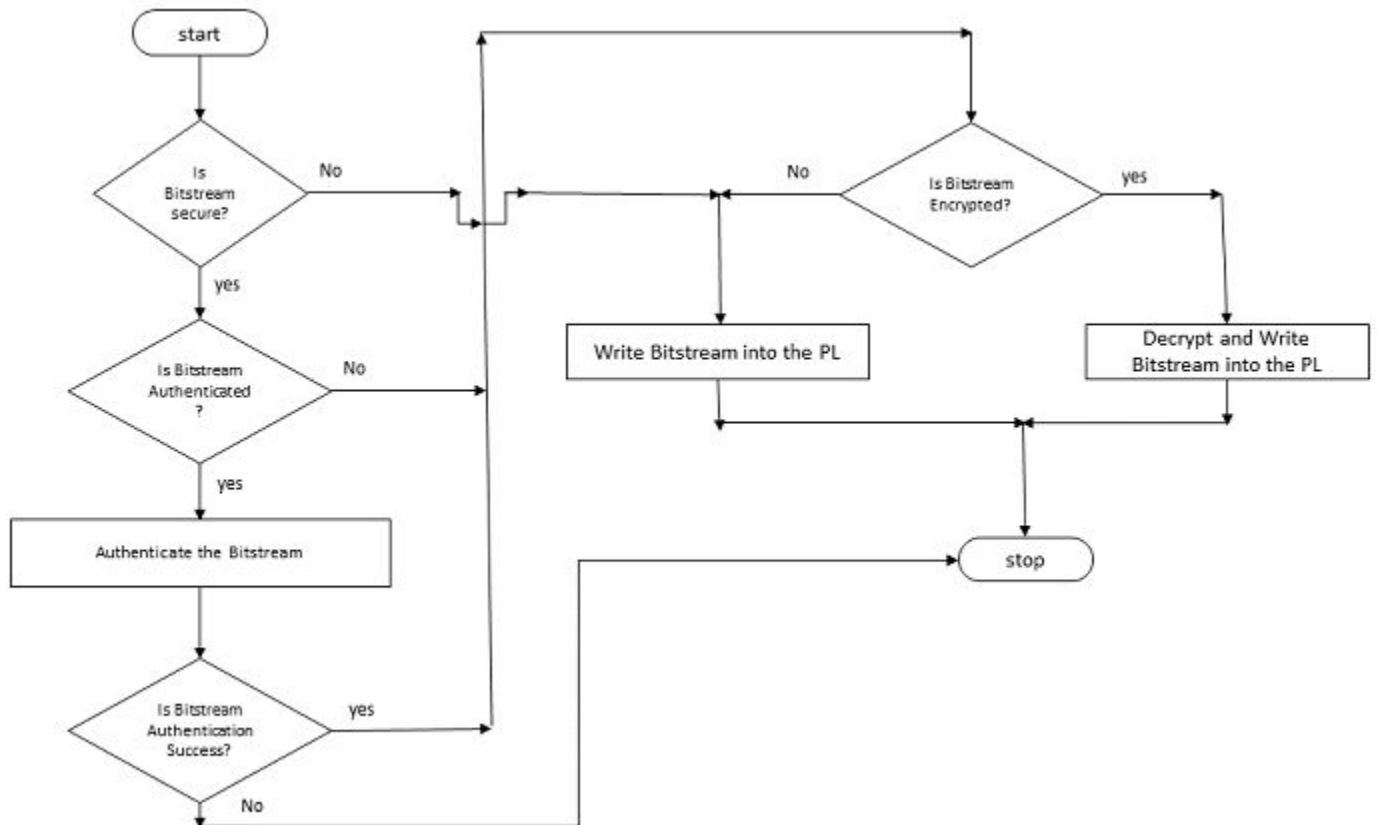
The following figure illustrates the XiIFPGA PL configuration sequence.

Figure 83: XiIFPGA PL Configuration Sequence



The following figure illustrates the Bitstream write sequence.

Figure 84: Bitstream write Sequence



XiIFPGA BSP Configuration Settings

The XiIFPGA library provides user configuration BSP settings. The following table describes the parameters and their default value:

Parameter Name	Type	Default Value	Description
secure_mode	bool	TRUE	Enables secure Bitstream loading support.
debug_mode	bool	FALSE	Enables the Debug messages in the library.
ocm_address	int	0xffffc0000	Address used for the Bitstream authentication.
base_address	int	0x80000	Holds the Bitstream Image address. This flag is valid only for the Cortex-A53 or the Cortex-R5 processors.
secure_readback	bool	FALSE	Should be set to TRUE to allow the secure Bitstream configuration data read back. The application environment should be secure and trusted to enable this flag.

Parameter Name	Type	Default Value	Description
secure_environment	bool	FALSE	Enable the secure PL configuration using the IPI. This flag is valid only for the Cortex-A53 or the Cortex-R5 processors.

Setting up the Software System

To use XilFPGA in a software application, you must first compile the XilFPGA library as part of software application.

1. Click **File > New > Platform Project**.
2. Click **Specify** to create a new Hardware Platform Specification.
3. Provide a new name for the domain in the **Project name** field if you wish to override the default value.
4. Select the location for the board support project files. To use the default location, as displayed in the **Location** field, leave the **Use default location** check box selected. Otherwise, deselect the checkbox and then type or browse to the directory location.
5. From the **Hardware Platform** drop-down choose the appropriate platform for your application or click the **New** button to browse to an existing Hardware Platform.
6. Select the target CPU from the drop-down list.
7. From the **Board Support Package OS** list box, select the type of board support package to create. A description of the platform types displays in the box below the drop-down list.
8. Click **Finish**. The wizard creates a new software platform and displays it in the Vitis Navigator pane.
9. Select **Project > Build Automatically** to automatically build the board support package. The Board Support Package Settings dialog box opens. Here you can customize the settings for the domain.
10. Click **OK** to accept the settings, build the platform, and close the dialog box.
11. From the Explorer, double-click platform.spr file and select the appropriate domain/board support package. The overview page opens.
12. In the overview page, click **Modify BSP Settings**.
13. Using the Board Support Package Settings page, you can select the OS Version and which of the Supported Libraries are to be enabled in this domain/BSP.
14. Select the **xilfpga** library from the list of **Supported Libraries**.
15. Expand the **Overview** tree and select **xilfpga**. The configuration options for xilfpga are listed.
16. Configure the xilfpga by providing the base address of the Bit-stream file (DDR address) and the size (in bytes).

17. Click **OK**. The board support package automatically builds with XilFPGA library included in it.
18. Double-click the **system.mss** file to open it in the **Editor** view.
19. Scroll-down and locate the **Libraries** section.
20. Click **Import Examples** adjacent to the XilFPGA entry.

Enabling Security

To support encrypted and/or authenticated bitstream loading, you must enable security in PMUFW.

1. Click **File > New > Platform Project**.
2. Click **Specify** to create a new Hardware Platform Specification.
3. Provide a new name for the domain in the **Project name** field if you wish to override the default value.
4. Select the location for the board support project files. To use the default location, as displayed in the **Location** field, leave the **Use default location** check box selected. Otherwise, deselect the checkbox and then type or browse to the directory location.
5. From the **Hardware Platform** drop-down choose the appropriate platform for your application or click the **New** button to browse to an existing Hardware Platform.
6. Select the target CPU from the drop-down list.
7. From the **Board Support Package OS** list box, select the type of board support package to create. A description of the platform types displays in the box below the drop-down list.
8. Click **Finish**. The wizard creates a new software platform and displays it in the Vitis Navigator pane.
9. Select **Project > Build Automatically** to automatically build the board support package. The Board Support Package Settings dialog box opens. Here you can customize the settings for the domain.
10. Click **OK** to accept the settings, build the platform, and close the dialog box.
11. From the Explorer, double-click platform.spr file and select the appropriate domain/board support package. The overview page opens.
12. In the overview page, click **Modify BSP Settings**.
13. Using the Board Support Package Settings page, you can select the OS Version and which of the Supported Libraries are to be enabled in this domain/BSP.
14. Expand the **Overview** tree and select **Standalone**.
15. Select a supported hardware platform.
16. Select **psu_pmu_0** from the **Processor** drop-down list.
17. Click Next. The **Templates** page appears.

18. Select **ZynqMP PMU Firmware** from the **Available Templates** list.
19. Click **Finish**. A PMUFW application project is created with the required BSPs.
20. Double-click the **system.mss** file to open it in the **Editor** view.
21. Click the **Modify this BSP's Settings** button. The **Board Support Package Settings** dialog box appears.
22. Select **xilfpga**. Various settings related to the library appears.
23. Select **secure_mode** and modify its value to **true** .
24. Click **OK** to save the configuration.

Note: By default the secure mode is enabled. To disable modify the `secure_mode` value to `false`.

Bitstream Authentication Using External Memory

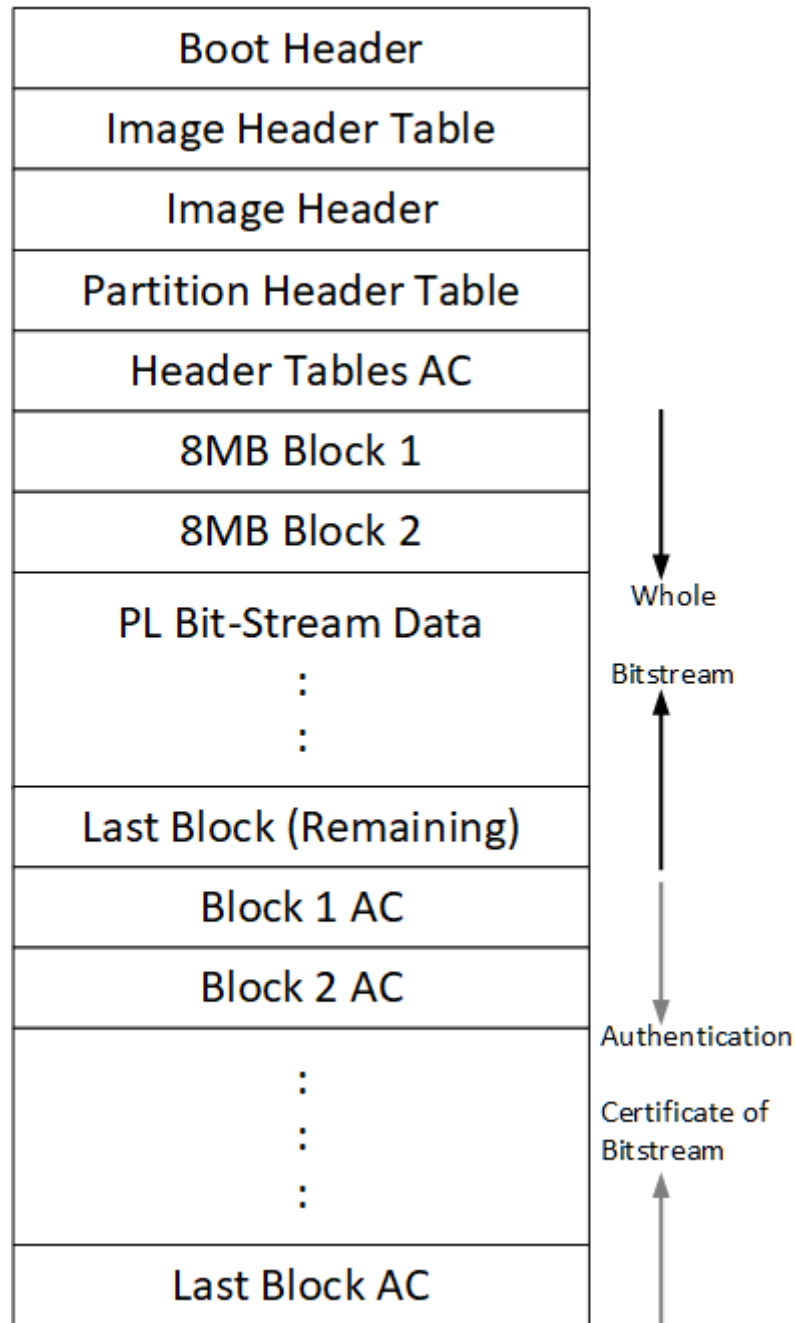
The size of the Bitstream is too large to be contained inside the device, therefore external memory must be used. The use of external memory could create a security risk. Therefore, two methods are provided to authenticate and decrypt a Bitstream.

- The first method uses the internal OCM as temporary buffer for all cryptographic operations. For details, see [Authenticated and Encrypted Bitstream Loading Using OCM](#). This method does not require trust in external DDR.
- The second method uses external DDR for authentication prior to sending the data to the decryptor, there by requiring trust in the external DDR. For details, see [Authenticated and Encrypted Bitstream Loading Using DDR](#).

Bootgen

When a Bitstream is requested for authentication, Bootgen divides the Bitstream into blocks of 8MB each and assigns an authentication certificate for each block. If the size of a Bitstream is not in multiples of 8 MB, the last block contains the remaining Bitstream data.

Figure 85: Bitstream Blocks



When both authentication and encryption are enabled, encryption is first done on the Bitstream. Bootgen then divides the encrypted data into blocks and assigns an Authentication certificate for each block.

Authenticated and Encrypted Bitstream Loading Using OCM

To authenticate the Bitstream partition securely, XilFPGA uses the FSBL section's OCM memory to copy the bitstream in chunks from DDR. This method does not require trust in the external DDR to securely authenticate and decrypt a Bitstream.

The software workflow for authenticating Bitstream is as follows:

1. XilFPGA identifies DDR secure Bitstream image base address. XilFPGA has two buffers in OCM, the Read Buffer is of size 56KB and hash of chunks to store intermediate hashes calculated for each 56 KB of every 8MB block.
2. XilFPGA copies a 56KB chunk from the first 8MB block to Read Buffer.
3. XilFPGA calculates hash on 56 KB and stores in HashsOfChunks.
4. XilFPGA repeats steps 1 to 3 until the entire 8MB of block is completed.

Note: The chunk that XilFPGA copies can be of any size. A 56KB chunk is taken for better performance.
5. XilFPGA authenticates the 8MB Bitstream chunk.
6. Once the authentication is successful, XilFPGA starts copying information in batches of 56KB starting from the first block which is located in DDR to Read Buffer, calculates the hash, and then compares it with the hash stored at HashsOfChunks.
7. If the hash comparison is successful, FSBL transmits data to PCAP using DMA (for un-encrypted Bitstream) or AES (if encryption is enabled).
8. XilFPGA repeats steps 6 and 7 until the entire 8MB block is completed.
9. Repeats steps 1 through 8 for all the blocks of Bitstream.

Note: You can perform warm restart even when the FSBL OCM memory is used to authenticate the Bitstream. PMU stores the FSBL image in the PMU reserved DDR memory which is visible and accessible only to the PMU and restores back to the OCM when APU-only restart needs to be performed. PMU uses the SHA3 hash to validate the FSBL image integrity before restoring the image to OCM (PMU takes care of only image integrity and not confidentiality). PMU checks if FSBL image is encrypted and skips copying FSBL from OCM to reserved DDR memory. In this case, If XilFPGA uses OCM memory for authenticating bitstream, APU restart feature will not work.

Also, copying FSBL to DDR for APU restart feature can be disabled by setting `USE_DDR_FOR_APU_RESTART_VAL` macro value in `xpfw_config.h` file to 0. If XilFPGA uses OCM memory for authenticating bitstream, APU restart feature will not work.

Authenticated and Encrypted Bitstream Loading Using DDR

The software workflow for authenticating Bitstream is as follows:

1. XiIFPGA identifies DDR secure Bitstream image base address.
2. XiIFPGA calculates hash for the first 8MB block.
3. XiIFPGA authenticates the 8MB block while stored in the external DDR.
4. If Authentication is successful, XiIFPGA transmits data to PCAP via DMA (for unencrypted Bitstream) or AES (if encryption is enabled).
5. Repeats steps 1 through 4 for all the blocks of Bitstream.

Additional References

Linux Interface

To know more about linux interface driver to program bitstream, see <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842505/Zynq+UltraScale+MPSoC+Secure+bitstream+programming+from+Linux>.

U-boot Interface

To know more about how to program bitstream from u-boot interface, see <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842432/Loading+authenticated+and+or+encrypted+image+partitions+from+u-boot>.

XiIFPGA APIs

This section provides detailed descriptions of the XiIFPGA library APIs.

Table 383: Quick Function Reference

Type	Name	Arguments
u32	XFpga_Initialize	void
u32	XFpga_PL_BitStream_Load	XFpga * InstancePtr UINTPTR BitstreamImageAddr UINTPTR AddrPtr_Size u32 Flags
u32	XFpga_PL_Preconfig	void
u32	XFpga_PL_Write	void

Table 383: Quick Function Reference (cont'd)

Type	Name	Arguments
u32	XFpga_PL_PostConfig	XFpga * InstancePtr
u32	XFpga_PL_ValidateImage	XFpga * InstancePtr UINTPTR BitstreamImageAddr UINTPTR AddrPtr_Size u32 Flags
u32	XFpga_GetPConfigData	XFpga * InstancePtr
u32	XFpga_GetPConfigReg	XFpga * InstancePtr ConfigReg Address
u32	XFpga_InterfaceStatus	XFpga * InstancePtr

Functions

XFpga_Initialize

Prototype

```
u32 XFpga_Initialize(XFpga *InstancePtr);
```

XFpga_PL_BitStream_Load

The API is used to load the bitstream file into the PL region.

It supports vivado generated Bitstream(*.bit, *.bin) and bootgen generated Bitstream(*.bin) loading, Passing valid Bitstream size (AddrPtr_Size) info is mandatory for vivado * generated Bitstream, For bootgen generated Bitstreams it will take Bitstream size from the Bitstream Header.

Prototype

```
u32 XFpga_PL_BitStream_Load(XFpga *InstancePtr, UINTPTR BitstreamImageAddr,
UINTPTR AddrPtr_Size, u32 Flags);
```

Parameters

The following table lists the XFpga_PL_BitStream_Load function arguments.

Table 384: Xfpga_PL_BitStream_Load Arguments

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFpga structure.
UINTPTR	BitstreamImageAddr	Linear memory Bitstream image base address
UINTPTR	AddrPtr_Size	Aes key address which is used for Decryption (or) In none Secure Bitstream used it is used to store size of Bitstream Image.
u32	Flags	Flags are used to specify the type of Bitstream file. <ul style="list-style-type: none"> • BIT(0) - Bitstream type <ul style="list-style-type: none"> ◦ 0 - Full Bitstream ◦ 1 - Partial Bitstream • BIT(1) - Authentication using DDR <ul style="list-style-type: none"> ◦ 1 - Enable ◦ 0 - Disable • BIT(2) - Authentication using OCM <ul style="list-style-type: none"> ◦ 1 - Enable ◦ 0 - Disable • BIT(3) - User-key Encryption <ul style="list-style-type: none"> ◦ 1 - Enable ◦ 0 - Disable • BIT(4) - Device-key Encryption <ul style="list-style-type: none"> ◦ 1 - Enable ◦ 0 - Disable

Returns

- XFPGA_SUCCESS on success
- Error code on failure.
- XFPGA_VALIDATE_ERROR.
- XFPGA_PRE_CONFIG_ERROR.
- XFPGA_WRITE_BITSTREAM_ERROR.
- XFPGA_POST_CONFIG_ERROR.

XFpga_PL_Preconfig

Prototype

```
u32 XFpga_PL_Preconfig(XFpga *InstancePtr);
```

XFpga_PL_Write

Prototype

```
u32 XFpga_PL_Write(XFpga *InstancePtr, UINTPTR BitstreamImageAddr, UINTPTR
AddrPtr_Size, u32 Flags);
```

XFpga_PL_PostConfig

This function set FPGA to operating state after writing.

Prototype

```
u32 XFpga_PL_PostConfig(XFpga *InstancePtr);
```

Parameters

The following table lists the XFpga_PL_PostConfig function arguments.

Table 385: XFpga_PL_PostConfig Arguments

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFpga structure

Returns

Codes as mentioned in xilfpga.h

XFpga_PL_ValidateImage

This function is used to validate the Bitstream Image.

Prototype

```
u32 XFpga_PL_ValidateImage(XFpga *InstancePtr, UINTPTR BitstreamImageAddr,
UINTPTR AddrPtr_Size, u32 Flags);
```

Parameters

The following table lists the XFpga_PL_ValidateImage function arguments.

Table 386: XFpga_PL_ValidateImage Arguments

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFpga structure
UINTPTR	BitstreamImageAddr	Linear memory Bitstream image base address
UINTPTR	AddrPtr_Size	Aes key address which is used for Decryption (or) In none Secure Bitstream used it is used to store size of Bitstream Image.
u32	Flags	Flags are used to specify the type of Bitstream file. <ul style="list-style-type: none"> • BIT(0) - Bitstream type <ul style="list-style-type: none"> ◦ 0 - Full Bitstream ◦ 1 - Partial Bitstream • BIT(1) - Authentication using DDR <ul style="list-style-type: none"> ◦ 1 - Enable ◦ 0 - Disable • BIT(2) - Authentication using OCM <ul style="list-style-type: none"> ◦ 1 - Enable ◦ 0 - Disable • BIT(3) - User-key Encryption <ul style="list-style-type: none"> ◦ 1 - Enable ◦ 0 - Disable • BIT(4) - Device-key Encryption <ul style="list-style-type: none"> ◦ 1 - Enable ◦ 0 - Disable

Returns

Codes as mentioned in xilfpga.h

XFpga_GetPlConfigData

Provides functionality to read back the PL configuration data.

Prototype

```
u32 XFpga_GetPlConfigData(XFpga *InstancePtr, UINTPTR ReadbackAddr, u32 NumFrames);
```

Parameters

The following table lists the XFpga_GetPlConfigData function arguments.

Table 387: XFpga_GetPlConfigData Arguments

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFpga structure
UINTPTR	ReadbackAddr	Address which is used to store the PL readback data.
u32	NumFrames	The number of Fpga configuration frames to read.

Returns

- XFPGA_SUCCESS if successful
- XFPGA_FAILURE if unsuccessful
- XFPGA_OPS_NOT_IMPLEMENTED if implementation not exists.

XFpga_GetPlConfigReg

Provides PL specific configuration register values.

Prototype

```
u32 XFpga_GetPlConfigReg(XFpga *InstancePtr, UINTPTR ReadbackAddr, u32 ConfigRegAddr);
```

Parameters

The following table lists the XFpga_GetPlConfigReg function arguments.

Table 388: XFpga_GetPlConfigReg Arguments

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFpga structure
UINTPTR	ReadbackAddr	Address which is used to store the PL Configuration register data.
u32	ConfigRegAddr	Configuration register address. For more information, see, UG570 - UltraScale Architecture Configuration User Guide .

Returns

- XFPGA_SUCCESS if successful
- XFPGA_FAILURE if unsuccessful
- XFPGA_OPS_NOT_IMPLEMENTED if implementation not exists.

XFpga_InterfaceStatus

This function provides the STATUS of PL programming interface.

Prototype

```
u32 XFpga_InterfaceStatus(XFpga *InstancePtr);
```

Parameters

The following table lists the `XFpga_InterfaceStatus` function arguments.

Table 389: XFpga_InterfaceStatus Arguments

Type	Name	Description
XFpga *	InstancePtr	Pointer to the XFpga structure

Returns

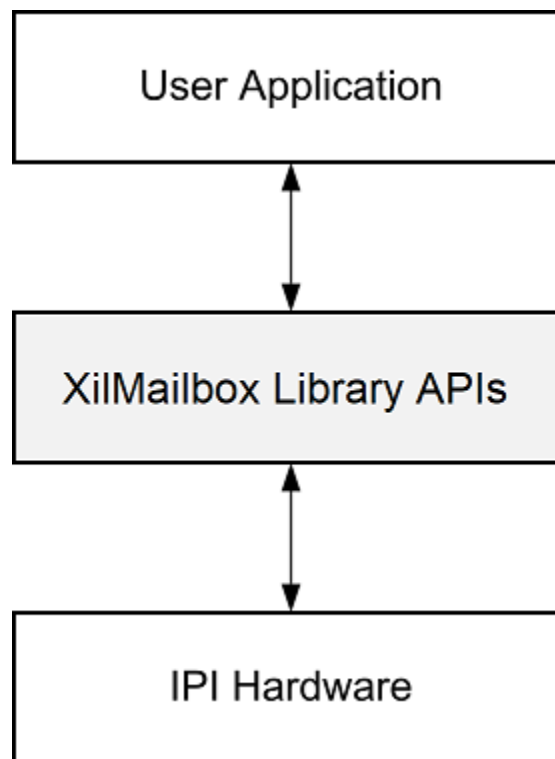
Status of the PL programming interface.

XilMailbox v1.2

Overview

The XilMailbox library provides the top-level hooks for sending or receiving an inter-processor interrupt (IPI) message using the Zynq® UltraScale+ MPSoC IPI hardware.

Figure 86: Overview



For more details on the IPI interrupts, see the Zynq UltraScale+ MPSoC Technical Reference Manual ([UG1085](#)).

This library supports the following features:

- Triggering an IPI to a remote agent.
- Sending an IPI message to a remote agent.

- Callbacks for error and recv IPI events.
- Reading an IPI message.

Software Initialization

The following is a list of software initialization events for a given IPI channel:

1. `XMailbox_Initialize()` function initializes a library instance for the given IPI channel.
2. `XMailbox_Send()` function triggers an IPI to a remote agent.
3. `XMailbox_SendData()` function sends an IPI message to a remote agent, message type should be either `XILMBOX_MSG_TYPE_REQ` (OR) `XILMBOX_MSG_TYPE_RESP`.
4. `XMailbox_Recv()` function reads an IPI message from a specified source agent, message type should be either `XILMBOX_MSG_TYPE_REQ` (OR) `XILMBOX_MSG_TYPE_RESP`.
5. `XMailbox_SetCallBack()` using this function user can register call backs for receive and error events.

Table 390: Quick Function Reference

Type	Name	Arguments
u32	<code>XMailbox_Send</code>	<code>XMailbox * InstancePtr</code> u32 RemoteId u8 Is_Blocking
u32	<code>XMailbox_SendData</code>	<code>XMailbox * InstancePtr</code> u32 RemoteId void * BufferPtr u32 MsgLen u8 BufferType u8 Is_Blocking
u32	<code>XMailbox_Recv</code>	<code>XMailbox * InstancePtr</code> u32 SourceId void * BufferPtr u32 MsgLen u8 BufferType
s32	<code>XMailbox_SetCallBack</code>	<code>XMailbox * InstancePtr</code> <code>XMailbox_Handler</code> HandlerType CallbackFunc CallbackRef
u32	<code>XMailbox_Initialize</code>	<code>XMailbox * InstancePtr</code> u8 DeviceId

Table 390: Quick Function Reference (cont'd)

Type	Name	Arguments
u32	XIpiPs_Init	XMailbox * InstancePtr u8 DeviceId
u32	XIpiPs_Send	XMailbox * InstancePtr u8 Is_Blocking
u32	XIpiPs_SendData	XMailbox * InstancePtr void * MsgBufferPtr u32 MsgLen u8 BufferType u8 Is_Blocking
u32	XIpiPs_PollforDone	XMailbox * InstancePtr
u32	XIpiPs_RecvData	XMailbox * InstancePtr void * MsgBufferPtr u32 MsgLen u8 BufferType
XStatus	XIpiPs_RegisterIrq	void
void	XIpiPs_ErrorIntrHandler	void
void	XIpiPs_IntrHandler	void

Functions

XMailbox_Send

This function triggers an IPI to a destination CPU.

Prototype

```
u32 XMailbox_Send(XMailbox *InstancePtr, u32 RemoteId, u8 Is_Blocking);
```

Parameters

The following table lists the `XMailbox_Send` function arguments.

Table 391: XMailbox_Send Arguments

Type	Name	Description
XMailbox *	InstancePtr	Pointer to the XMailbox instance
u32	RemoteId	is the Mask of the CPU to which IPI is to be triggered
u8	Is_Blocking	if set trigger the notification in blocking mode

Returns

- XST_SUCCESS if successful
- XST_FAILURE if unsuccessful

XMailbox_SendData

This function sends an IPI message to a destination CPU.

Prototype

```
u32 XMailbox_SendData(XMailbox *InstancePtr, u32 RemoteId, void *BufferPtr,
u32 MsgLen, u8 BufferType, u8 Is_Blocking);
```

Parameters

The following table lists the XMailbox_SendData function arguments.

Table 392: XMailbox_SendData Arguments

Type	Name	Description
XMailbox *	InstancePtr	Pointer to the XMailbox instance
u32	RemoteId	is the Mask of the CPU to which IPI is to be triggered
void *	BufferPtr	is the pointer to Buffer which contains the message to be sent
u32	MsgLen	is the length of the buffer/message
u8	BufferType	is the type of buffer (XILMBOX_MSG_TYPE_REQ (OR) XILMBOX_MSG_TYPE_RESP)
u8	Is_Blocking	if set trigger the notification in blocking mode

Returns

- XST_SUCCESS if successful
- XST_FAILURE if unsuccessful

XMailbox_Recv

This function reads an IPI message.

Prototype

```
u32 XMailbox_Recv(XMailbox *InstancePtr, u32 SourceId, void *BufferPtr, u32
MsgLen, u8 BufferType);
```

Parameters

The following table lists the `XMailbox_Recv` function arguments.

Table 393: XMailbox_Recv Arguments

Type	Name	Description
<code>XMailbox *</code>	InstancePtr	Pointer to the <code>XMailbox</code> instance
u32	SourceId	is the Mask for the CPU which has sent the message
void *	BufferPtr	is the pointer to Buffer to which the read message needs to be stored
u32	MsgLen	is the length of the buffer/message
u8	BufferType	is the type of buffer (XILMBOX_MSG_TYPE_REQ or XILMBOX_MSG_TYPE_RESP)

Returns

- XST_SUCCESS if successful
- XST_FAILURE if unsuccessful

XMailbox_SetCallback

This routine installs an asynchronous callback function for the given HandlerType.

Note: Invoking this function for a handler that already has been installed replaces it with the new handler.

Prototype

```
s32 XMailbox_SetCallBack(XMailbox *InstancePtr, XMailbox_Handler
HandlerType, void *CallbackFuncPtr, void *CallbackRefPtr);
```

Parameters

The following table lists the `XMailbox_SetCallBack` function arguments.

Table 394: XMailbox_SetCallBack Arguments

Type	Name	Description
<code>XMailbox *</code>	InstancePtr	is a pointer to the <code>XMailbox</code> instance.
<code>XMailbox_Handler</code>	HandlerType	specifies which callback is to be attached.

Table 394: XMailbox_SetCallback Arguments (cont'd)

Type	Name	Description
Commented parameter CallbackFunc does not exist in function XMailbox_SetCallback.	CallbackFunc	is the address of the callback function.
Commented parameter CallbackRef does not exist in function XMailbox_SetCallback.	CallbackRef	is a user data item that will be passed to the callback function when it is invoked.

Returns

- XST_SUCCESS when handler is installed.
- XST_INVALID_PARAM when HandlerType is invalid.

XMailbox_Initialize

Initialize the XMailbox Instance.

Prototype

```
u32 XMailbox_Initialize(XMailbox *InstancePtr, u8 DeviceId);
```

Parameters

The following table lists the XMailbox_Initialize function arguments.

Table 395: XMailbox_Initialize Arguments

Type	Name	Description
XMailbox *	InstancePtr	is a pointer to the instance to be worked on
u8	DeviceId	is the IPI Instance to be worked on

Returns

XST_SUCCESS if initialization was successful XST_FAILURE in case of failure

XIpiPs_Init

Initialize the ZynqMP Mailbox Instance.

Prototype

```
u32 XIpiPs_Init(XMailbox *InstancePtr, u8 DeviceId);
```

Parameters

The following table lists the `XIpiPs_Init` function arguments.

Table 396: **XIpiPs_Init Arguments**

Type	Name	Description
<code>XMailbox *</code>	<code>InstancePtr</code>	is a pointer to the instance to be worked on
<code>u8</code>	<code>DeviceId</code>	is the IPI Instance to be worked on

Returns

`XST_SUCCESS` if initialization was successful `XST_FAILURE` in case of failure

XIpiPs_Send

This function triggers an IPI to a destination CPU.

Prototype

```
u32 XIpiPs_Send(XMailbox *InstancePtr, u8 Is_Blocking);
```

Parameters

The following table lists the `XIpiPs_Send` function arguments.

Table 397: **XIpiPs_Send Arguments**

Type	Name	Description
<code>XMailbox *</code>	<code>InstancePtr</code>	Pointer to the <code>XMailbox</code> instance.
<code>u8</code>	<code>Is_Blocking</code>	if set trigger the notification in blocking mode

Returns

`XST_SUCCESS` in case of success `XST_FAILURE` in case of failure

XIpiPs_SendData

This function sends an IPI message to a destination CPU.

Prototype

```
u32 XIpiPs_SendData(XMailbox *InstancePtr, void *MsgBufferPtr, u32 MsgLen,
u8 BufferType, u8 Is_Blocking);
```

Parameters

The following table lists the `XIpiPs_SendData` function arguments.

Table 398: **XIpiPs_SendData Arguments**

Type	Name	Description
<code>XMailbox *</code>	<code>InstancePtr</code>	Pointer to the <code>XMailbox</code> instance
<code>void *</code>	<code>MsgBufferPtr</code>	is the pointer to Buffer which contains the message to be sent
<code>u32</code>	<code>MsgLen</code>	is the length of the buffer/message
<code>u8</code>	<code>BufferType</code>	is the type of buffer
<code>u8</code>	<code>Is_Blocking</code>	if set trigger the notification in blocking mode

Returns

`XST_SUCCESS` in case of success `XST_FAILURE` in case of failure

XIpiPs_PollforDone

Poll for an acknowledgement using Observation Register.

Prototype

```
u32 XIpiPs_PollforDone(XMailbox *InstancePtr);
```

Parameters

The following table lists the `XIpiPs_PollforDone` function arguments.

Table 399: **XIpiPs_PollforDone Arguments**

Type	Name	Description
<code>XMailbox *</code>	<code>InstancePtr</code>	Pointer to the <code>XMailbox</code> instance

Returns

`XST_SUCCESS` in case of success `XST_FAILURE` in case of failure

XIpiPs_RecvData

This function reads an IPI message.

Prototype

```
u32 XIpiPs_RecvData(XMailbox *InstancePtr, void *MsgBufferPtr, u32 MsgLen, u8 BufferType);
```

Parameters

The following table lists the `XIpiPs_RecvData` function arguments.

Table 400: **XIpiPs_RecvData Arguments**

Type	Name	Description
<code>XMailbox *</code>	<code>InstancePtr</code>	Pointer to the <code>XMailbox</code> instance
<code>void *</code>	<code>MsgBufferPtr</code>	is the pointer to Buffer to which the read message needs to be stored
<code>u32</code>	<code>MsgLen</code>	is the length of the buffer/message
<code>u8</code>	<code>BufferType</code>	is the type of buffer

Returns

- `XST_SUCCESS` if successful
- `XST_FAILURE` if unsuccessful

XIpiPs_RegisterIrq

Prototype

```
XStatus XIpiPs_RegisterIrq(XScuGic *IntcInstancePtr, XMailbox *InstancePtr,
u32 IpiIntrId);
```

XIpiPs_ErrorIntrHandler

Prototype

```
void XIpiPs_ErrorIntrHandler(void *XMailboxPtr);
```

XIpiPs_IntrHandler

Prototype

```
void XIpiPs_IntrHandler(void *XMailboxPtr);
```

Enumerations

Enumeration XMailbox_Handler

Contains XMAILBOX Handler Types.

Table 401: Enumeration XMailbox_Handler Values

Value	Description
XMAILBOX_RECV_HANDLER	For Recv Handler.
XMAILBOX_ERROR_HANDLER	For Error Handler.

Data Structure Index

The following is a list of data structures:

- [XMailbox](#)

XMailbox

Holds the function pointers for the operations that can be performed.

Declaration

```
typedef struct
{
    u32 (* XMbox_IPI_Send)(struct XMboxTag *InstancePtr, u8 Is_Blocking),
    u32 (* XMbox_IPI_SendData)(struct XMboxTag *InstancePtr, void *BufferPtr,
    u32 MsgLen, u8 BufferType, u8 Is_Blocking),
    u32 (* XMbox_IPI_Recv)(struct XMboxTag *InstancePtr, void *BufferPtr, u32
    MsgLen, u8 BufferType),
    XMailbox_RecvHandler RecvHandler,
    XMailbox_ErrorHandler ErrorHandler,
    void * ErrorRefPtr,
    void * RecvRefPtr,
    XMailbox_Agent Agent
} XMailbox;
```

Table 402: Structure XMailbox member description

Member	Description
XMbox_IPI_Send	Triggers an IPI to a destination CPU.
XMbox_IPI_SendData	Sends an IPI message to a destination CPU.
XMbox_IPI_Recv	Reads an IPI message.
RecvHandler	Callback for rx IPI event.
ErrorHandler	Callback for error event.
ErrorRefPtr	To be passed to the error interrupt callback.
RecvRefPtr	To be passed to the receive interrupt callback.
Agent	Used to store IPI Channel information.

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see [Xilinx Support](#).

Documentation Navigator and Design Hubs

Xilinx[®] Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado[®] IDE, select **Help** → **Documentation and Tutorials**.
- On Windows, select **Start** → **All Programs** → **Xilinx Design Tools** → **DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the [Design Hubs](#) page.

Note: For more information on DocNav, see the [Documentation Navigator](#) page on the Xilinx website.

References

Xilinx References

1. Xilinx Third-Party Licensing Solution Center
2. [PetaLinux Product Page](#)
3. [Xilinx Vivado Design Suite – HLx Editions](#)
4. [Xilinx Third-Party Tools](#)
5. [Zynq UltraScale+ MPSoC Product Table](#)
6. [Zynq UltraScale+ MPSoC Product Advantages](#)
7. [Zynq UltraScale+ MPSoC Products Page](#)

Zynq Devices Documentation

1. *Xilinx Quick Emulator User Guide: QEMU* ([UG1169](#))
2. *UltraScale Architecture and Product Data Sheet: Overview* ([DS890](#))
3. *Isolation Methods in Zynq UltraScale+ MPSoCs* ([XAPP1320](#))
4. *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#))
5. *Zynq UltraScale+ Device Register Reference* ([UG1087](#))
6. *Zynq UltraScale+ MPSoC: Embedded Design Tutorial* ([UG1209](#))
7. *Zynq UltraScale+ MPSoC Processing System LogiCORE IP Product Guide* ([PG201](#))
8. *UltraScale Architecture System Monitor User Guide* ([UG580](#))
9. *Libmetal and OpenAMP for Zynq Devices User Guide* ([UG1186](#))
10. *Embedded Energy Management Interface Specification* ([UG1200](#))
11. *UltraFast Embedded Design Methodology Guide* ([UG1046](#))
12. *Zynq-7000 SoC: Embedded Design Tutorial* ([UG1165](#))
13. *Zynq-7000 SoC Software Developers Guide* ([UG821](#))
14. *UltraScale Architecture PCB Design User Guide* ([UG583](#))
15. [Vivado Design Suite Documentation](#)
16. *Bootgen User Guide* ([UG1283](#))

Vitis software platform and PetaLinux Documents

1. Vitis Unified Software Platform Documentation

2. *OS and Libraries Document Collection* ([UG643](#))
3. Embedded Design Tools [Download](#)
4. *PetaLinux Tools Documentation: Reference Guide* ([UG1144](#))
5. *Xilinx Software Development Kit: System Performance* ([UG1145](#))

Xilinx IP Documents

1. *AXI Central Direct Memory Access LogiCORE IP Product Guide* ([PG034](#))
2. *AXI Video Direct Memory Access LogiCORE IP Product Guide* ([PG020](#))

Miscellaneous Links

1. [Xilinx Github](#)
2. [Embedded Development](#)
3. [meta-xilinx](#)
4. [PetaLinux Software Development](#)
5. [Zynq UltraScale+ Silicon Devices Page](#)
6. Xilinx Answer: [66249](#)
7. [Vivado Quick Take Video: Vivado PS Configuration Wizard Overview](#)
8. Xilinx Wiki

Third-Party References

1. [Lauterbach Technologies](#)
2. [Arm Trusted Firmware](#)
3. [Xen Hypervisor](#)
4. [Arm Developer Center](#)
5. [Arm Cortex-A53 MPCore Processor Technical Reference Manual](#)
6. [Yocto Product Development](#)
7. [GNU FTP](#)
8. [Power State Coordination Interface – Arm DEN 0022B.b, 6/25/2013](#)

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright

© Copyright 2015-2020 Xilinx, Inc. Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Spartan, Versal, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.