

Zynq UltraScale+ MPSoC Software Developer Guide

UG1137 (v2020.2) January 5, 2021



Revision History

The following table shows the revision history for this document.

| Section | Revision Summary |
|---|---|
| 01/05/2021 Version 2020.2 | |
| Entire document | Minor updates |
| Appendix A: Libraries | Libraries are currently available in the <i>OS and Libraries Document Collection (UG643)</i> . |
| 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. |
| XilSecure Library v4.3 | Added Additional References. |
| XilFPGA Library v5.3 | Added Additional References. |
| 12/05/2019 Version 11.0 | |
| Vitis Embedded Flow | Updated SDK flows to Vitis Embedded Flow throughout the document. |
| 06/26/2019 Version 10.0 | |
| 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 |

| Section | Revision Summary |
|---|---|
| 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 |
| 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 EL0-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). |

| Section | Revision Summary |
|---|--|
| 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. |
| 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..... | 9 |
| Introduction..... | 9 |
| Intended Audience and Scope of this Document..... | 10 |
| Prerequisites..... | 10 |
| Chapter 2: Programming View of Zynq UltraScale+ MPSoC | |
| Devices..... | 12 |
| Hardware Architecture Overview..... | 12 |
| Boot Process..... | 15 |
| Virtualization..... | 18 |
| System Level Reset Requirements..... | 18 |
| Security..... | 19 |
| Safety and Reliability..... | 22 |
| Memory Overview for APU and RPU Executables..... | 25 |
| Chapter 3: Development Tools..... | 27 |
| Vivado Design Suite..... | 27 |
| Vitis Unified Software Platform..... | 29 |
| Arm GNU Tools..... | 31 |
| Device Tree Generator..... | 32 |
| PetaLinux Tools..... | 32 |
| Linux Software Development using Yocto..... | 33 |
| Chapter 4: Software Stack..... | 36 |
| Bare Metal Software Stack..... | 36 |
| Linux Software Stack..... | 39 |
| 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..... | 76 |
| Chapter 8: Security Features..... | 101 |
| Boot Time Security..... | 101 |
| Bitstream Authentication Using External Memory..... | 112 |
| Run-Time Security..... | 115 |
| 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..... | 249 |
| USB 3.0..... | 249 |
| Gigabit Ethernet Controller..... | 252 |
| PCI Express..... | 255 |

| | |
|--|------------|
| Chapter 14: Clock and Frequency Management..... | 261 |
| Changing the Peripheral Frequency..... | 261 |
| Chapter 15: Target Development Platforms..... | 263 |
| QEMU..... | 263 |
| Boards and Kits..... | 263 |
| Chapter 16: Boot Image Creation..... | 264 |
| Appendix A: Libraries..... | 265 |
| Appendix B: Additional Resources and Legal Notices..... | 266 |
| Xilinx Resources..... | 266 |
| Documentation Navigator and Design Hubs..... | 266 |
| References..... | 266 |
| Please Read: Important Legal Notices..... | 269 |

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 B: 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.

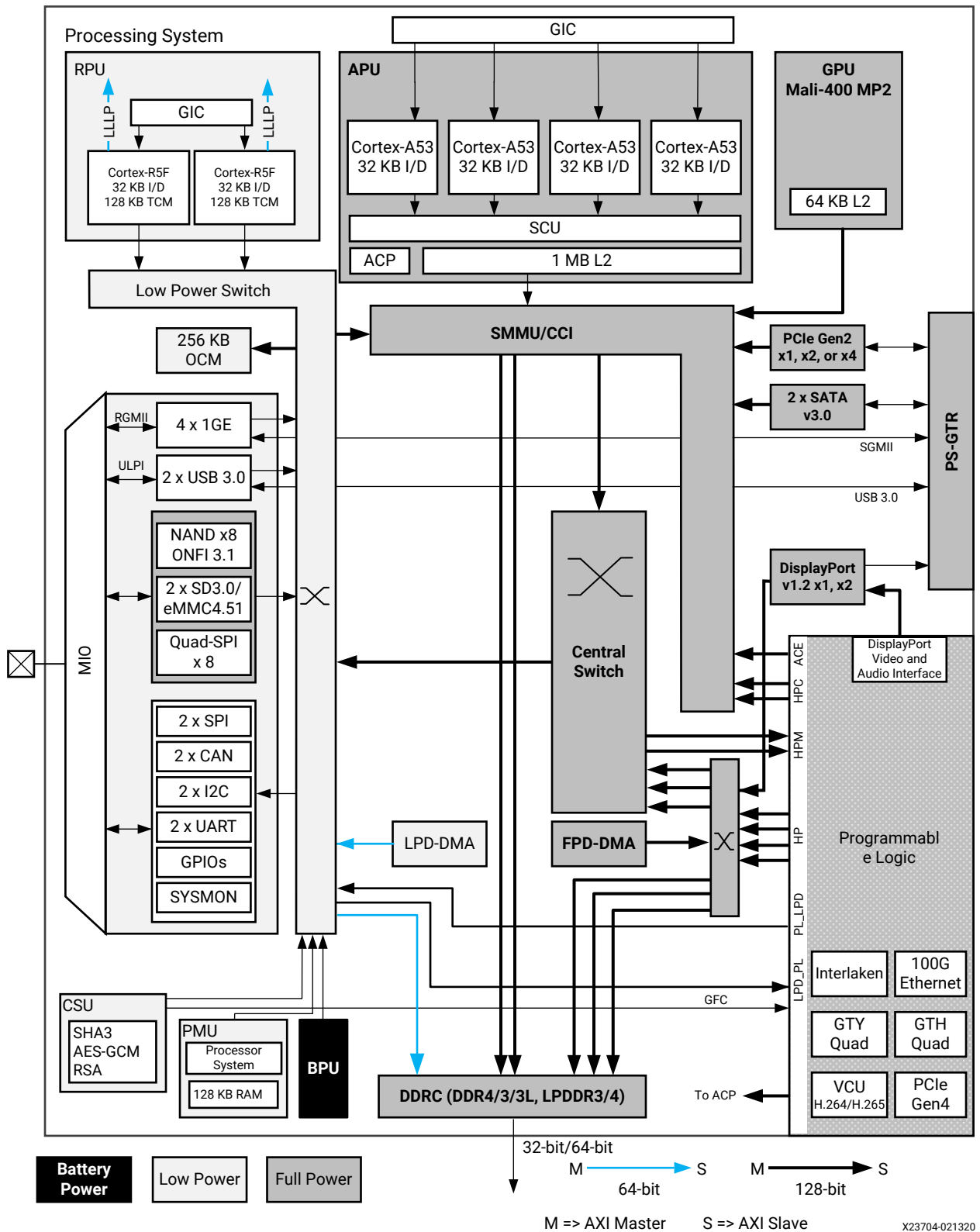
The Zynq UltraScale+ MPSoC is a heterogeneous device that includes the Arm[®] Cortex[™]-A53, high-performance, energy-efficient, 64-bit application processor, and also the 32-bit Arm Cortex[™]-R5F dual-core real-time processor.

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.

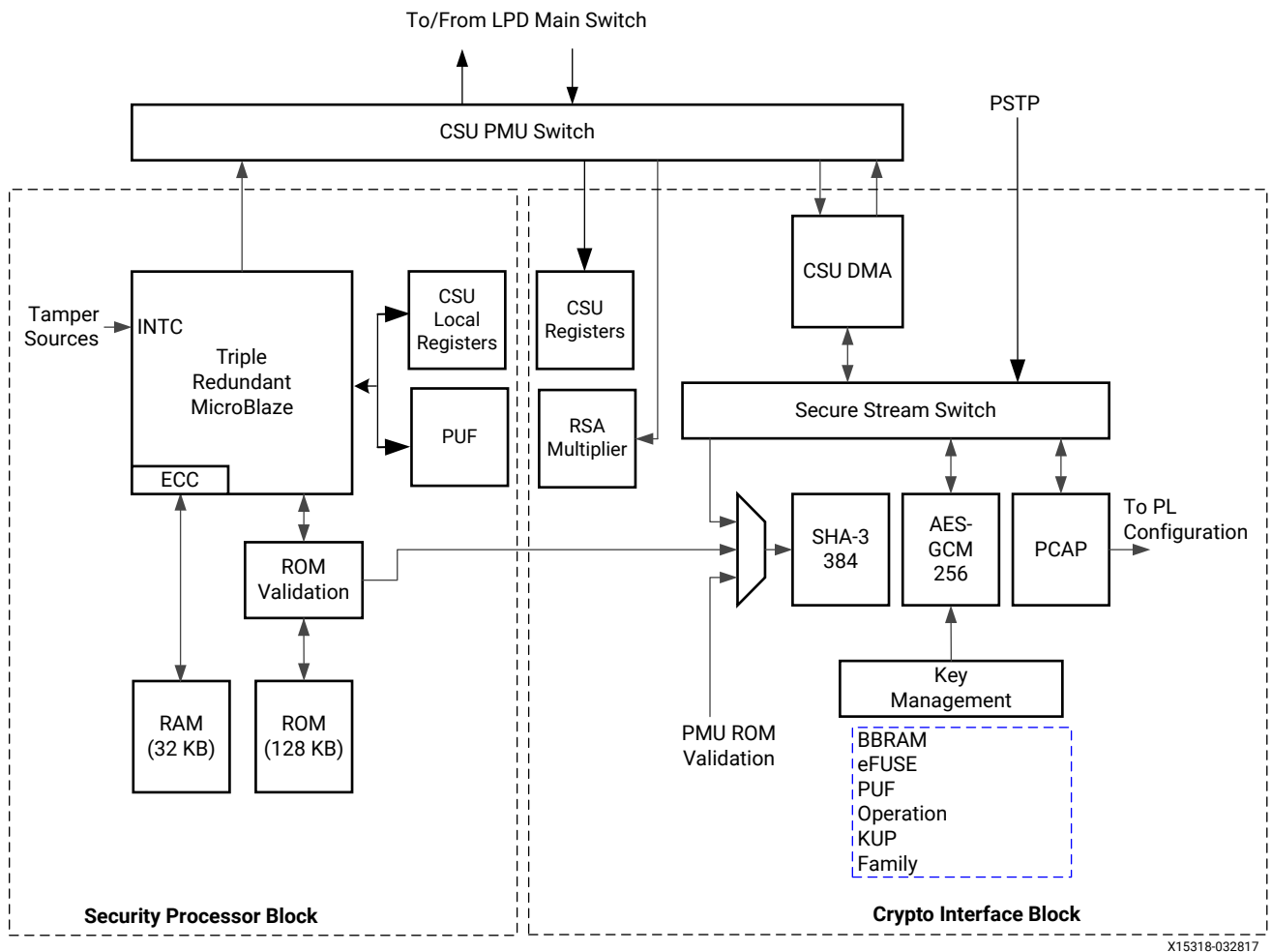
Configuration Security Unit

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 the *XiIFPGA Library v5.3 in the OS and Libraries Document Collection* ([UG643](#)). 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 (SECCDED) 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](#).

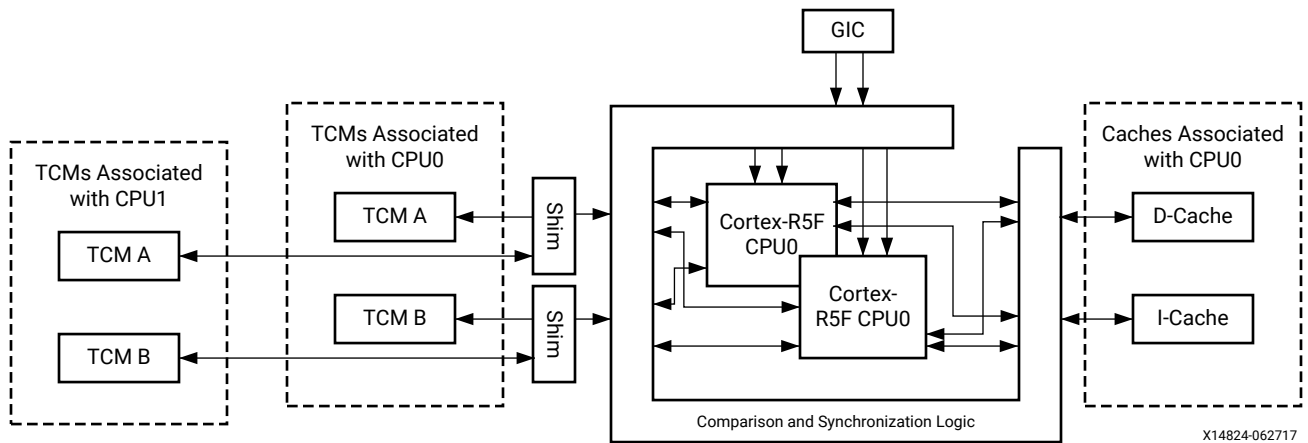
Lock-Step Operation

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 | 0xFFFC0000 | 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 | 0xFFFC0000 | 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 | 0xFFFC0000 | 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 | 0xFFFC0000 | 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 0xFFFC0000 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 (0xFFFC0000). 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:

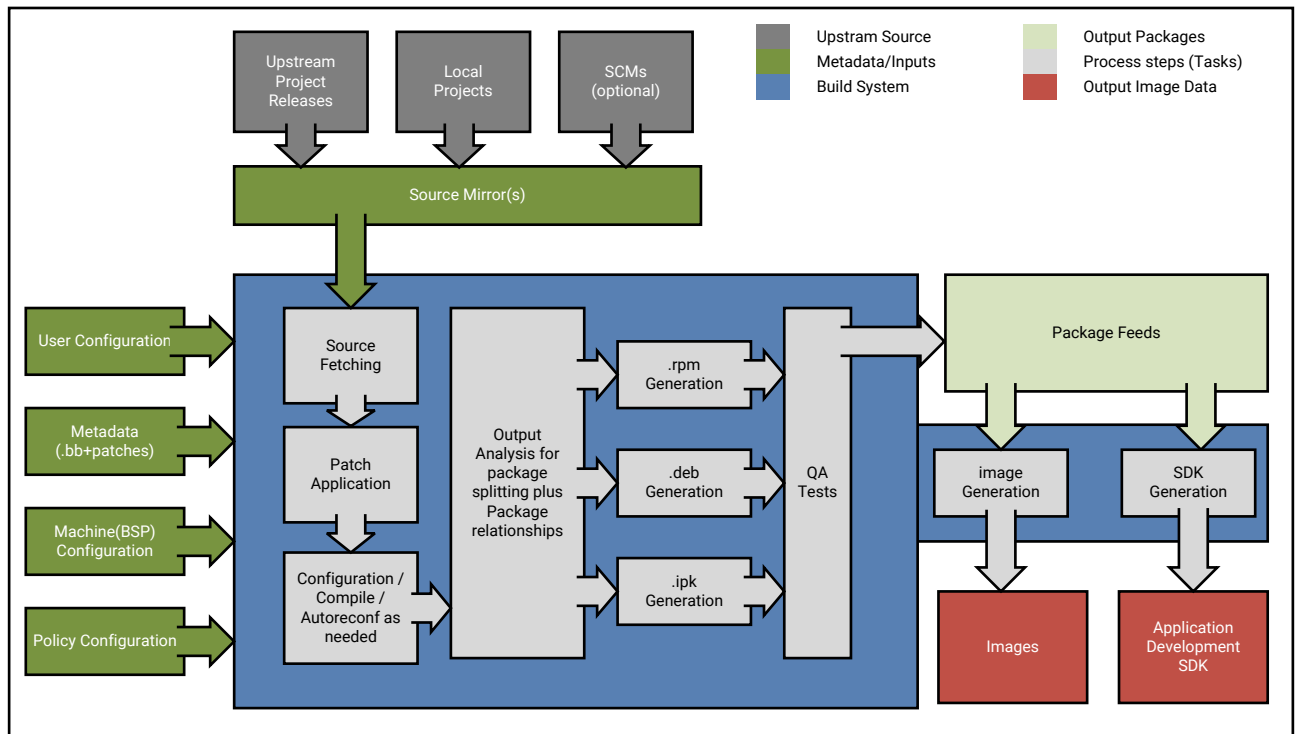
- Provides a recent Linux kernel along with a set of system commands and libraries suitable for the embedded environment.
- Makes available system components such as X11, GTK+, Qt, Clutter, and SDL (among others) so you can create a rich user experience on devices that have display hardware. For devices 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:

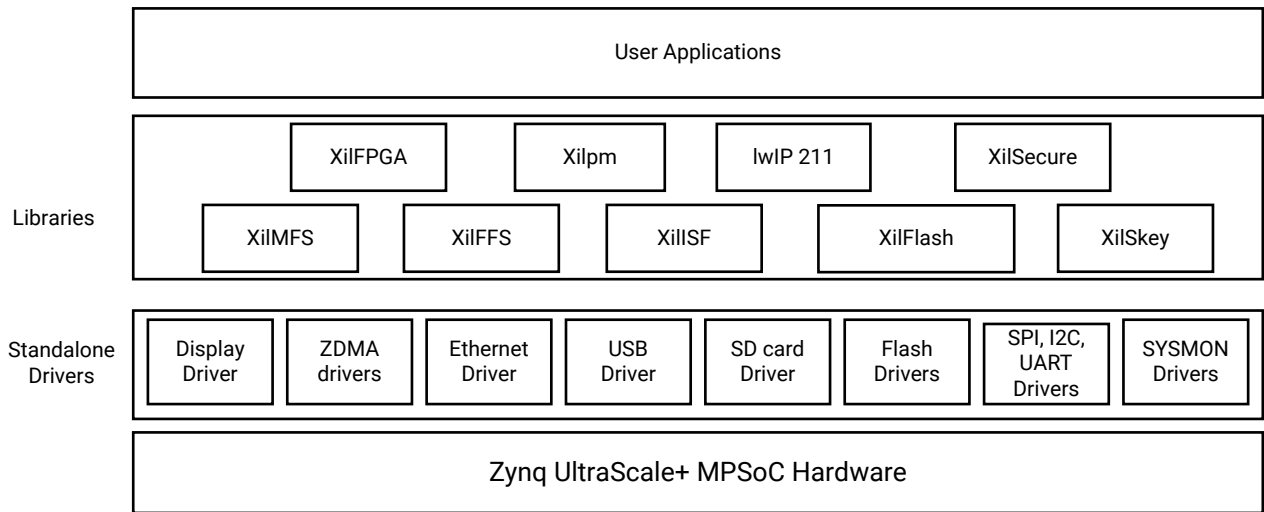
```
<Xilinx Installation Directory>\Vitis\<version>\data\embeddedsw  
\XilinxProcessorIPLib\drivers
```

You can locate libraries at the following path:

```
<Xilinx Installation Directory>\Vitis\<version>\data\embeddedsw\lib  
\sw_services
```

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

Figure 5: Bare-Metal Software Development Stack



X17169-111020

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 |
| inttypes.h | Integer type conversions |

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

| Header File | Description |
|-------------|-----------------------------|
| 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 | cbrt, hypot, sqrt |
| Elementary transcendental | asin, acos, atan, atan2, asinh, acosh, atanh, exp, expm1, pow, log, log1p, log10, sin, cos, tan, sinh, cosh, tanh |
| Higher transcendental | j0, j1, jn, y0, y1, yn, erf, erfc, gamma, lgamma, and gamma_rgamma_r |
| Integral rounding | ceil, 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:** 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.

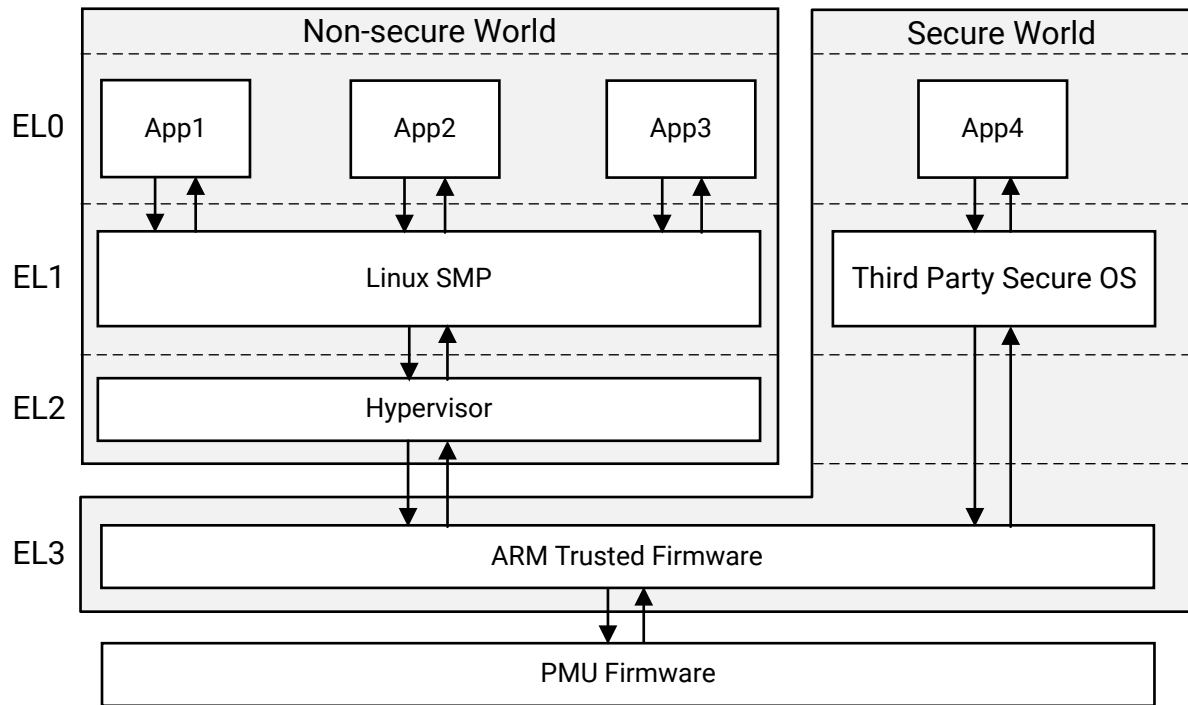
- **XilSKey:** Xilinx secure key library.
- **XilFPGA:** A library that provides an interface to the Linux or bare-metal users for configuring the programmable logic (PL) over PCAP from PS.
- **XilPM:** Xilinx Power Management (XilPM) provides Embedded Energy Management Interface (EEMI) APIs for power management on Zynq UltraScale+ MPSoC.
- **XilMailbox:** 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
- **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 [The C Standard Library \(libc\)](#).

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 secure monitor software. 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

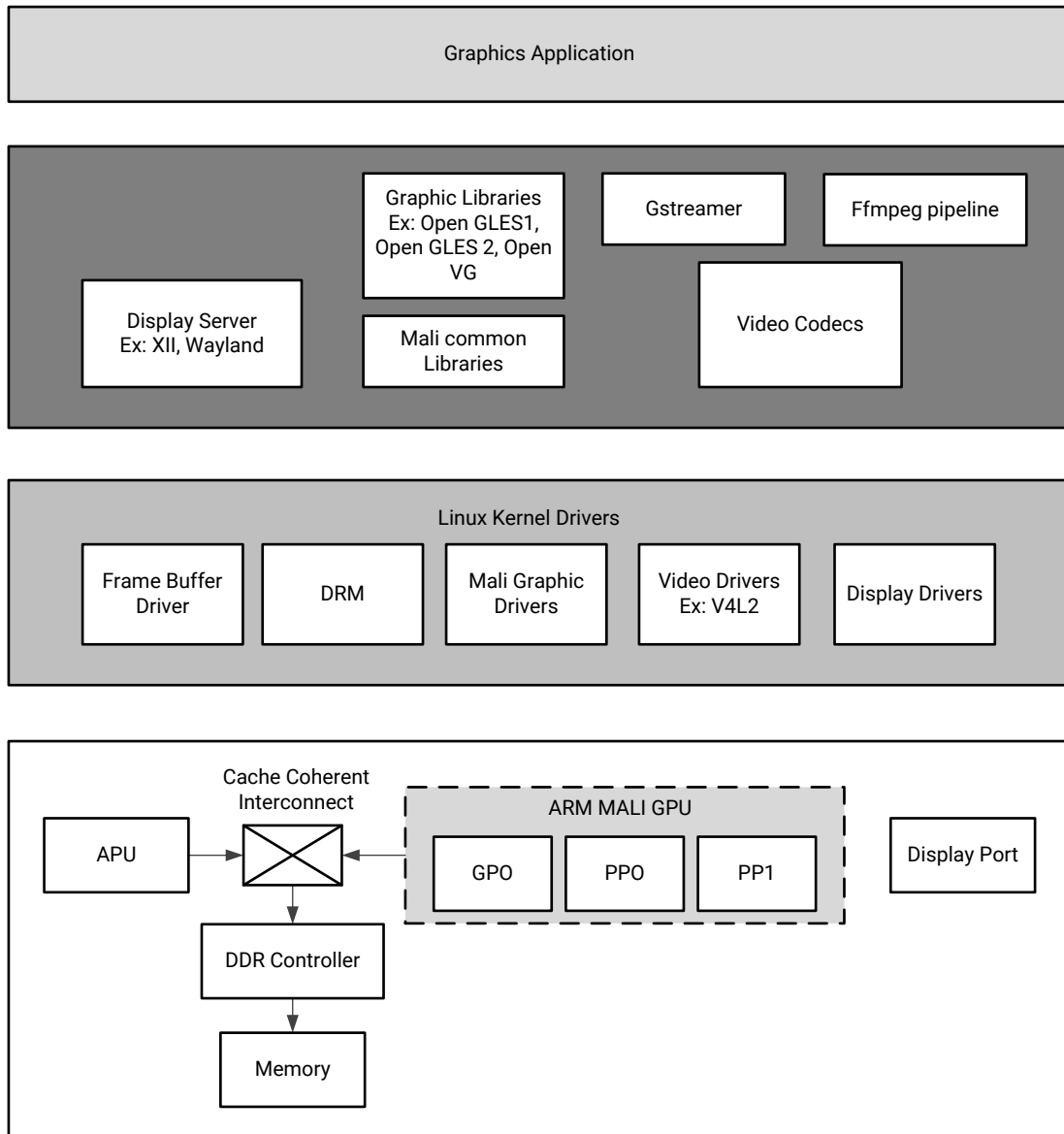
This section describes the multimedia software stack in the Zynq UltraScale+ MPSoC.

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



X14795-110320

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. |

Table 11: Libraries and Framework Components (cont'd)

| Component | Description |
|--------------------------------|---|
| 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. |

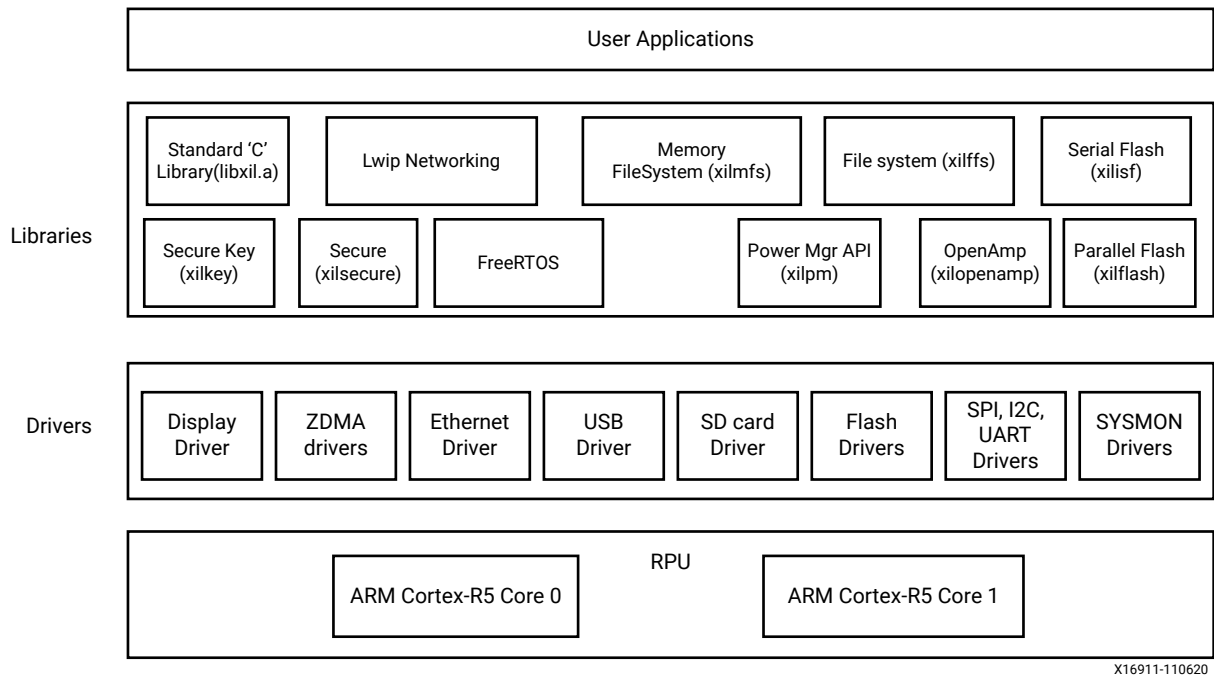
FreeRTOS Software Stack

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

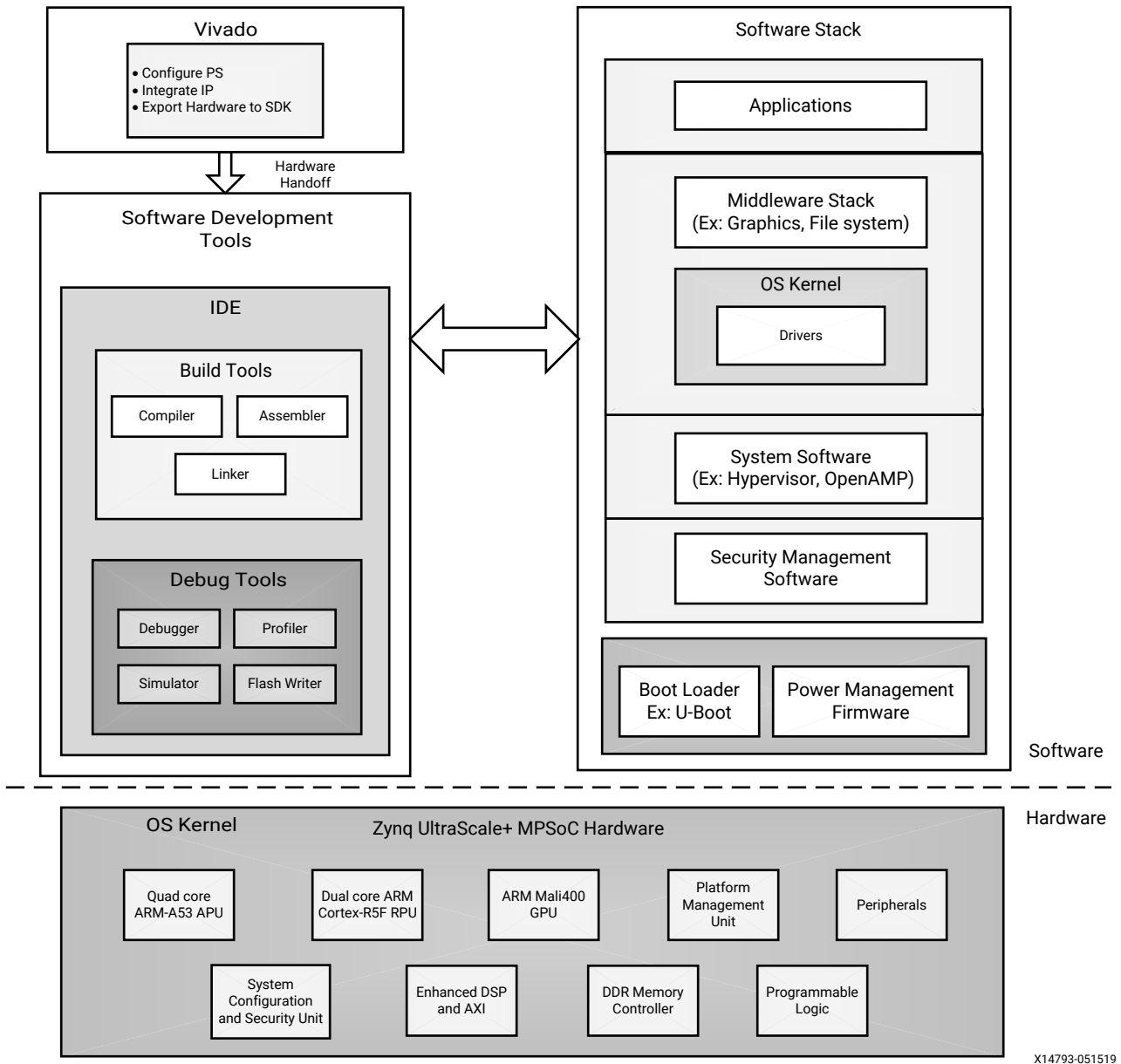
Many other embedded software solutions are also available from the Xilinx partner ecosystem. More information is available from the Xilinx website, [Embedded Computing](#) and the website, [Xilinx Third Party Tools](#).

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



X14793-051519

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).

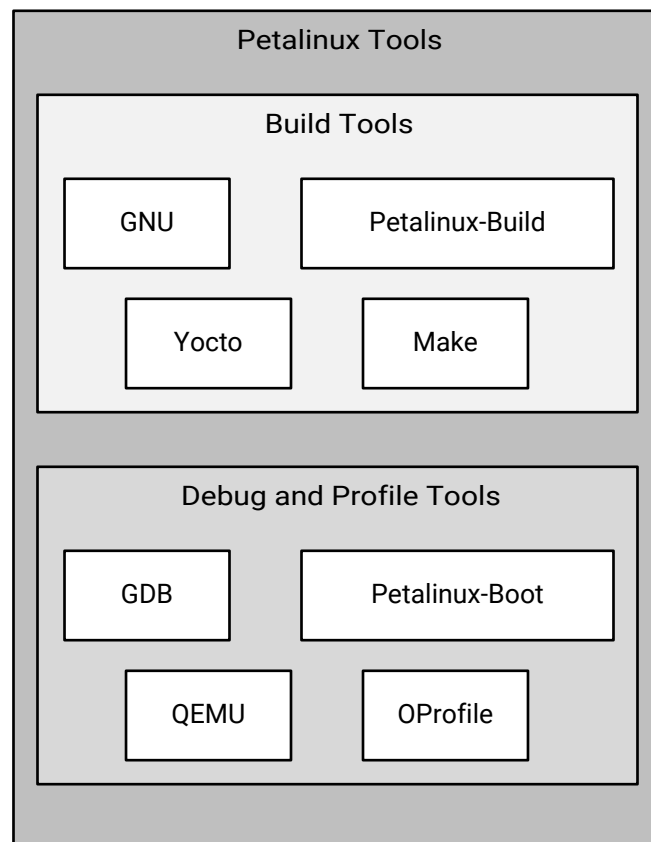
Note:

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



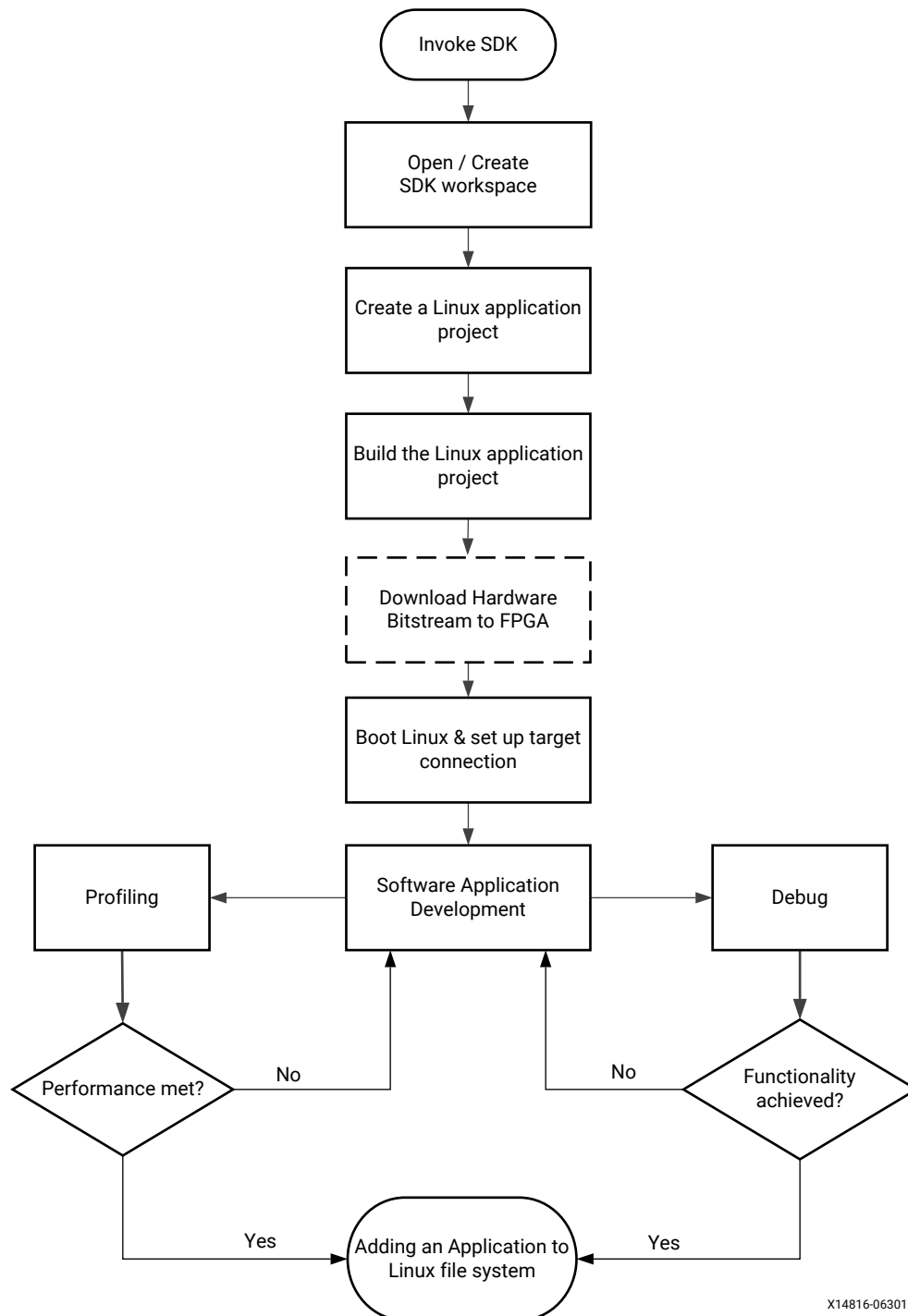
X14815-063017

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



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**.
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. These frameworks are described as follows:

- **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. See [this whitepaper](#) for more information.

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](#).

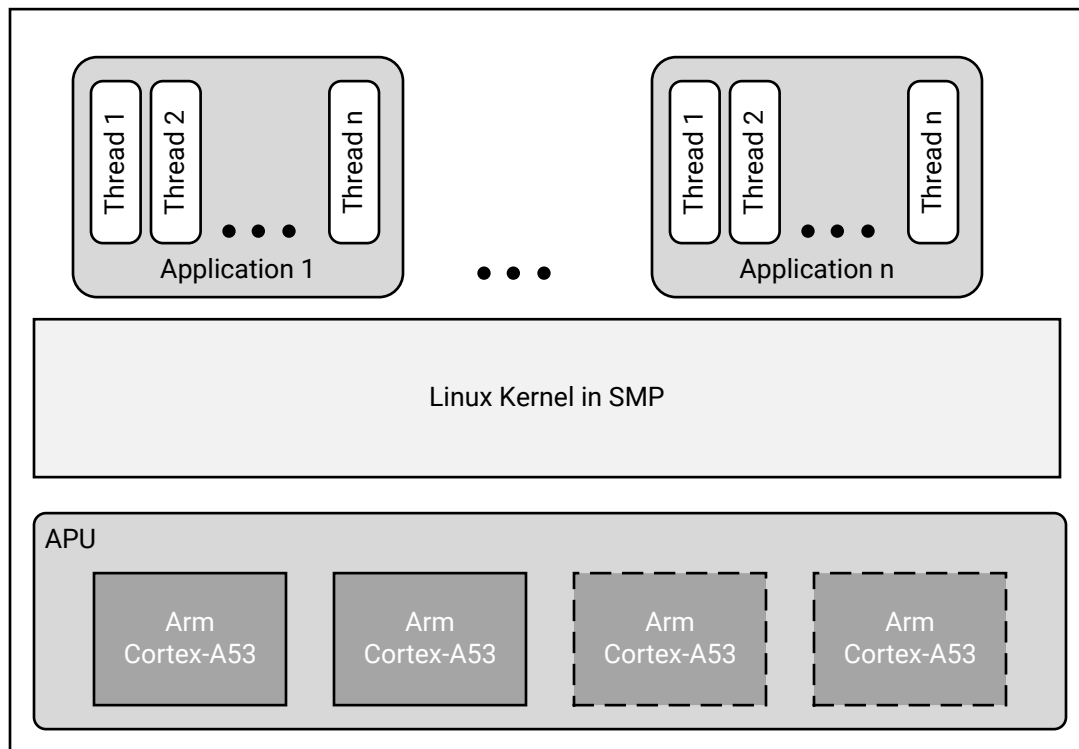
- **Multiprocessor Communication Framework:** Xilinx provides the OpenAMP framework for Zynq UltraScale+ MPSoCs to allow communication between the different processing units. For more details, see the *Xilinx Quick Emulator User Guide: QEMU* ([UG1169](#))
- **Power Management Framework:** The power management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

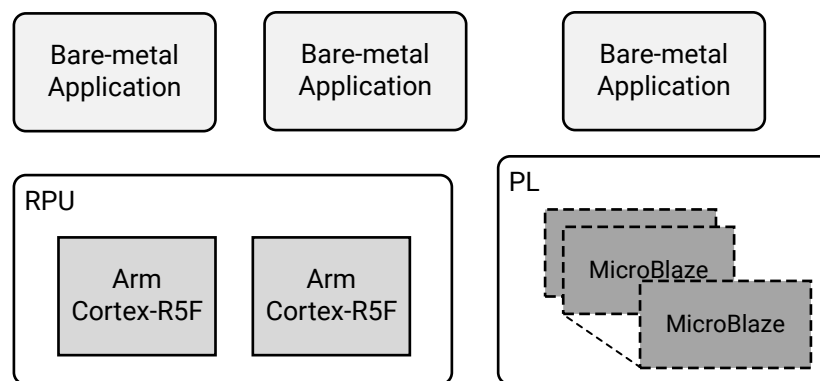
AMP uses multiple processors with precise control over what runs on each processor. Unlike SMP, there are many different ways to use AMP. This section describes two ways of using AMP with varying levels of complexity.

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

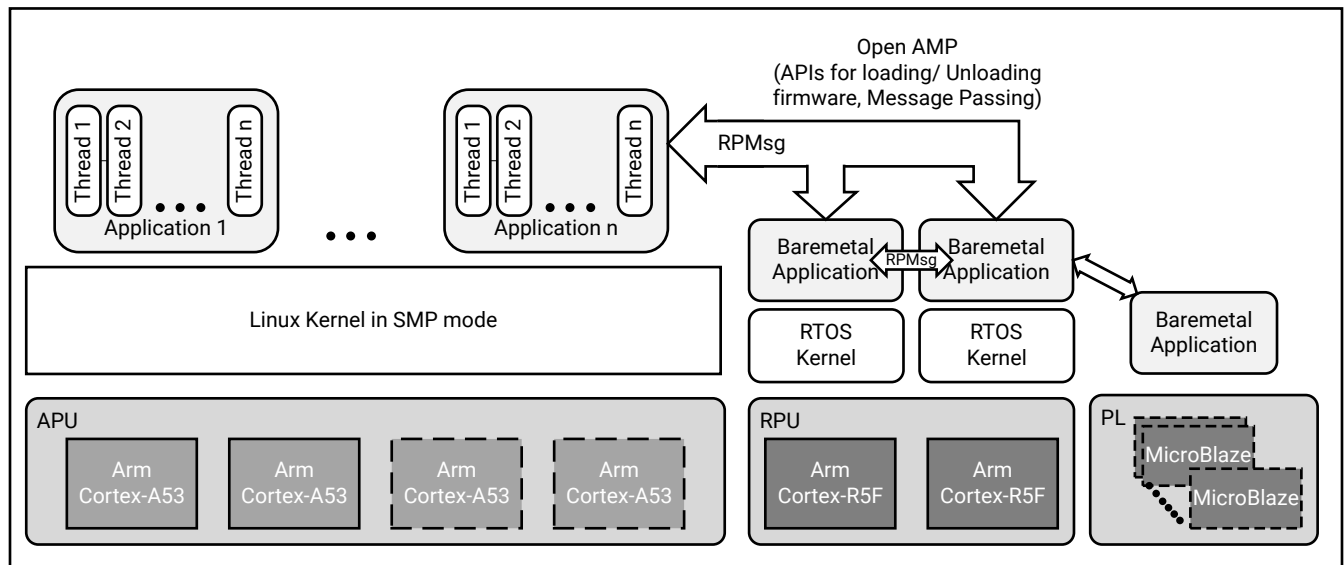
The OpenAMP framework provides mechanisms to do the following:

- 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.

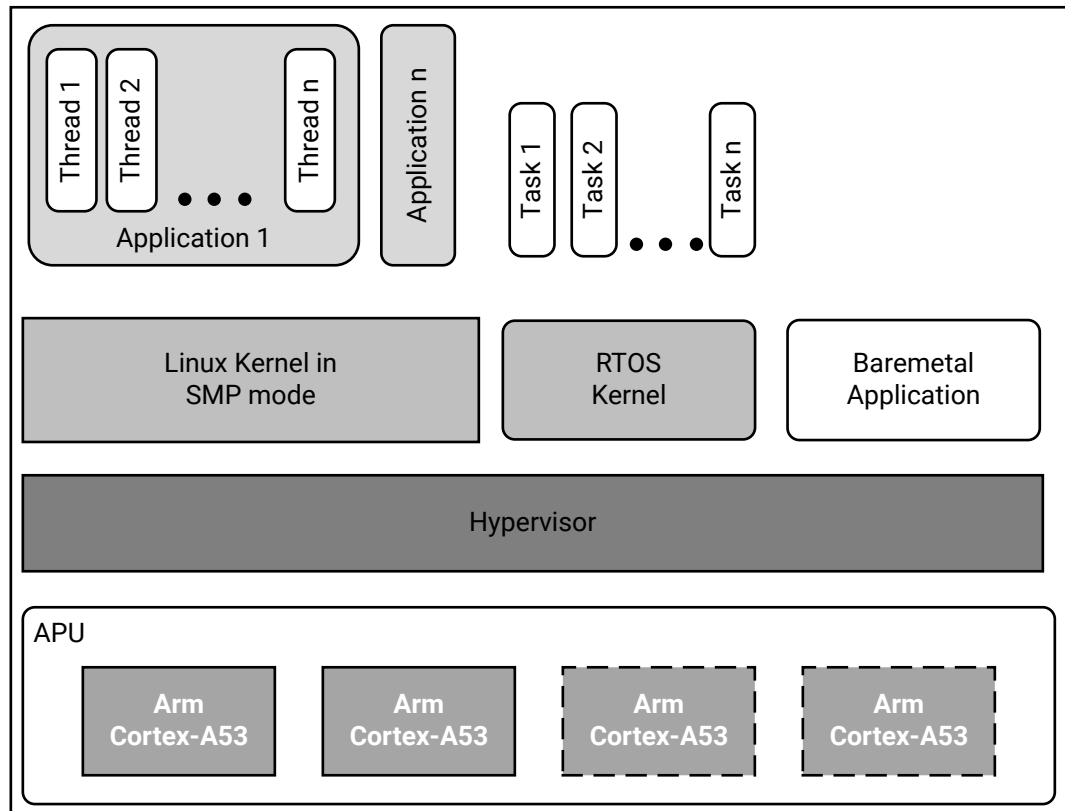
Virtualization with Hypervisor

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:

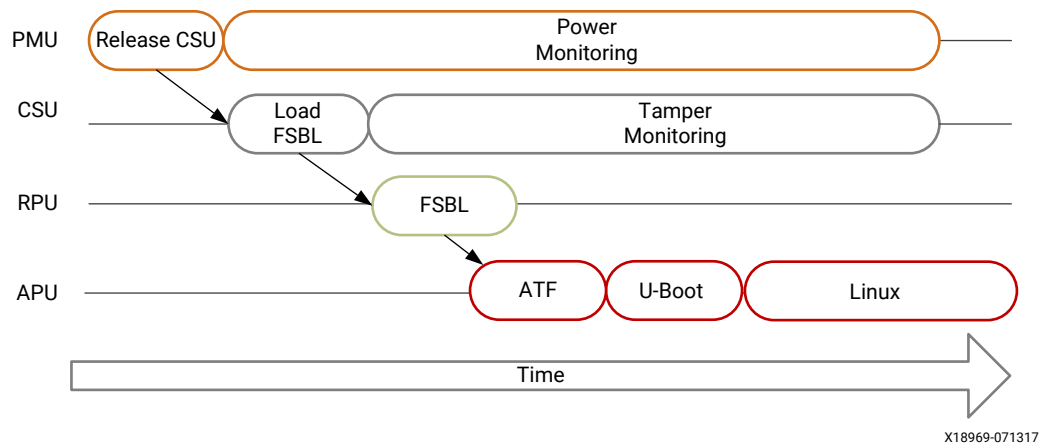
- **Pre-configuration stage:** The PMU primarily controls pre-configuration stage that executes PMU ROM to setup the system. The PMU handles all of the processes related to reset and wake-up.
- **Configuration stage:** This stage is responsible for loading the first-stage boot loader (FSBL) code for 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.

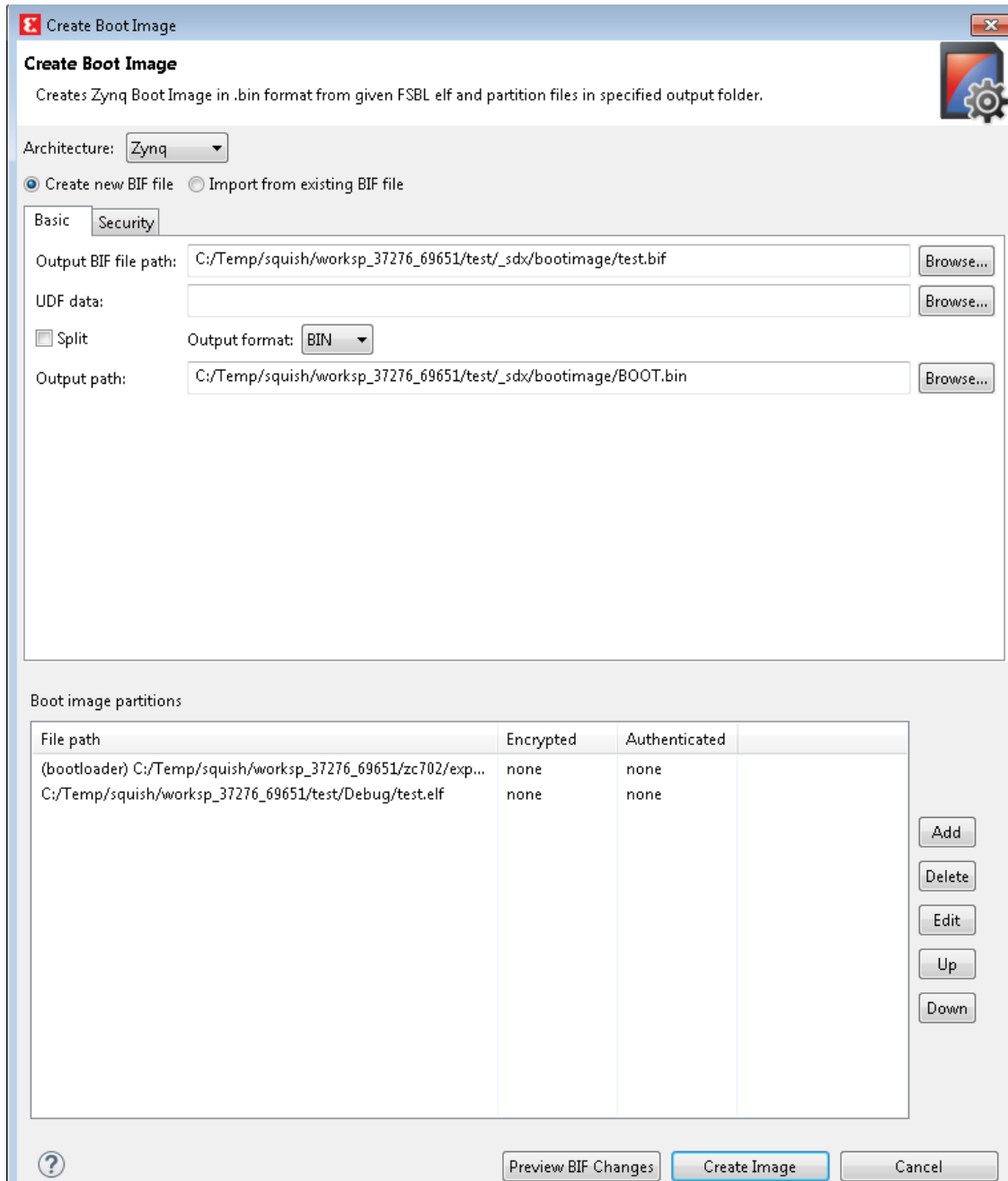
Creating a Bootable Image

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.



- 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

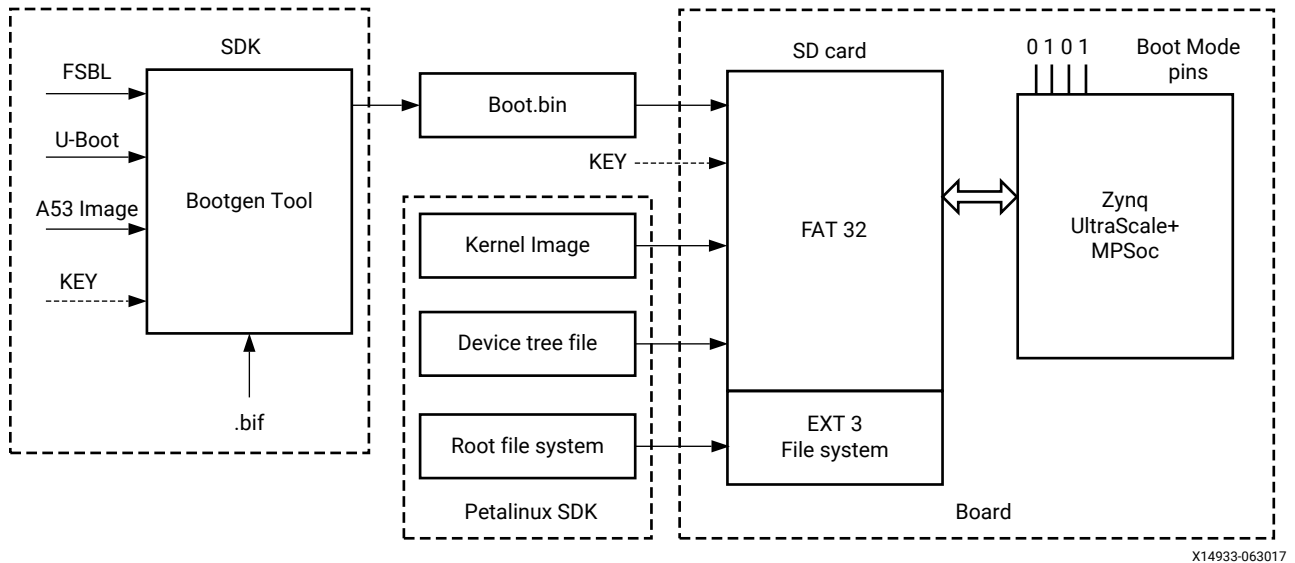
SD boot (version 3.0) supports the following:

- 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:

- Build the Linux kernel image, device tree file, and the root file system.
- 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`.

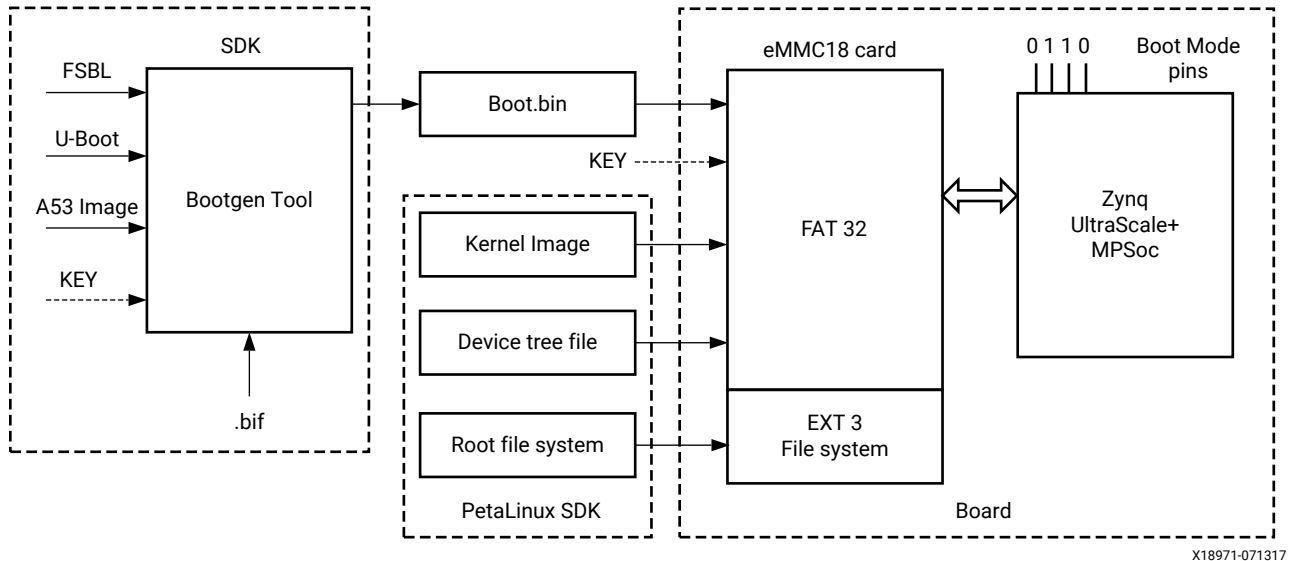
eMMC18 Boot Mode

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:

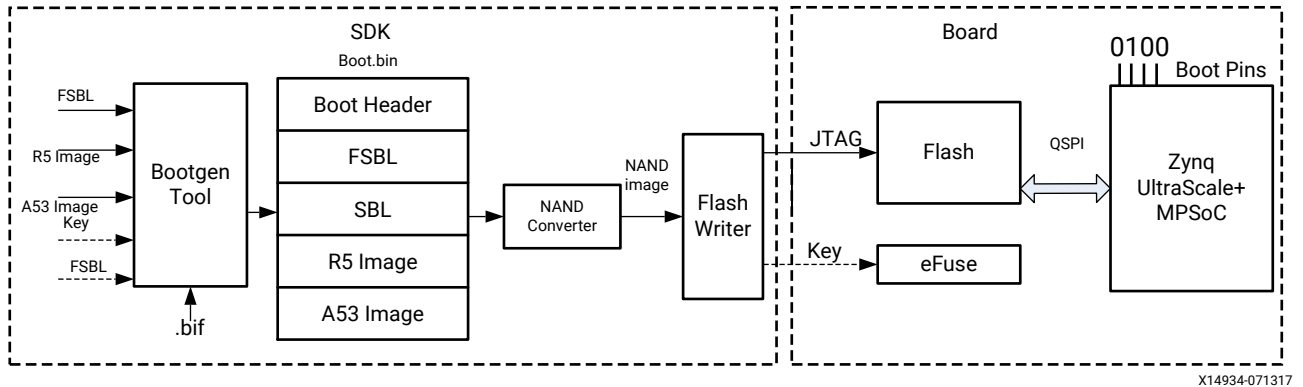
- Build the Linux kernel image, device tree file, and the root file system.
- Copy the files into the eMMC18 card.

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 `xfsb1_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

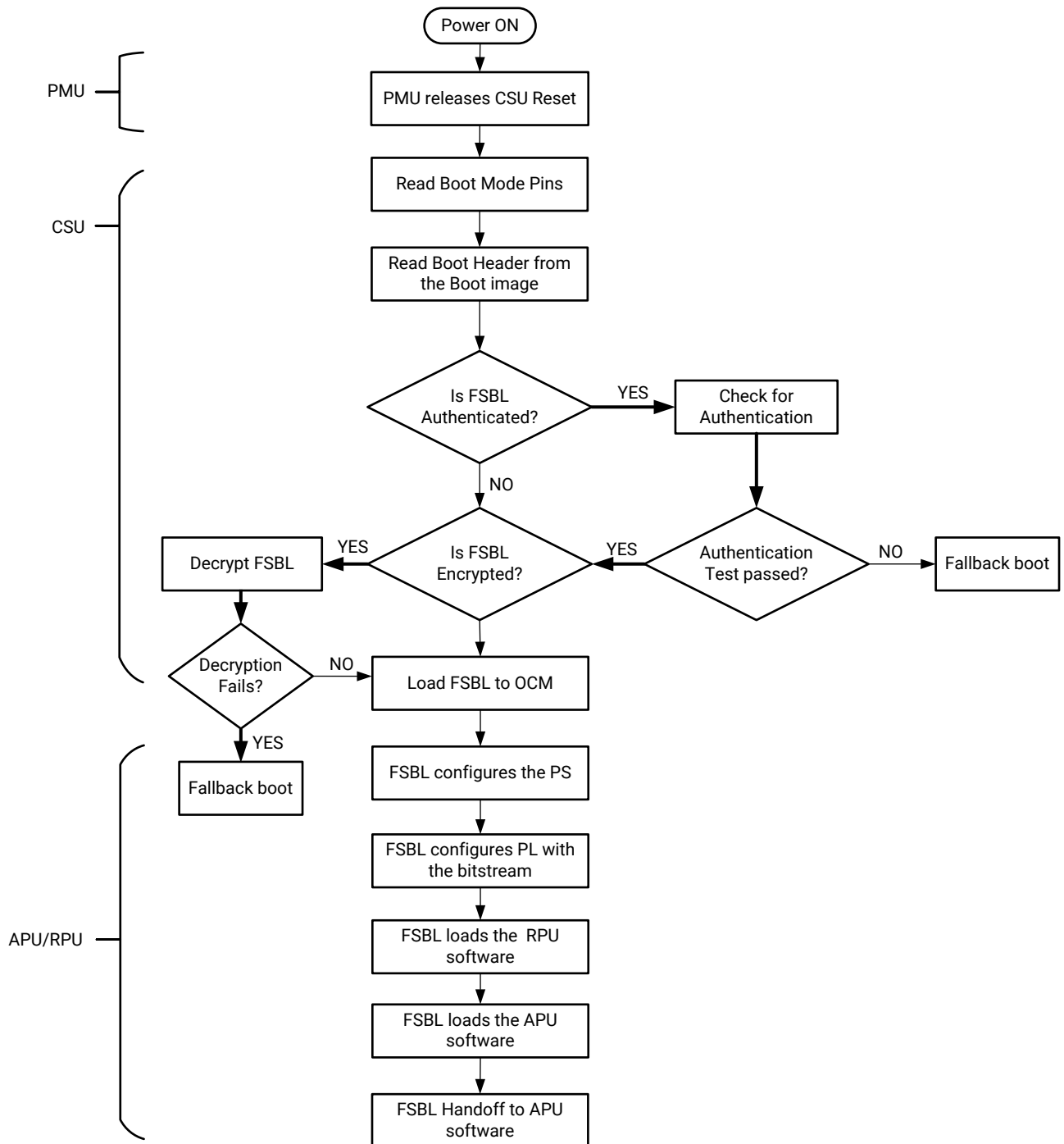
| Pre-Boot Task | Description |
|---------------|--|
| 0 | Initialize MicroBlaze™ processor. Capture key states. |
| 1 | Scan, and clear LPD and FPD. |
| 2 | Initialize the System Monitor. |
| 3 | Initialize the PLL used for MBIST clocks. |
| 4 | Zero out the PMU RAM. |
| 5 | Validate the PLL. Configure 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

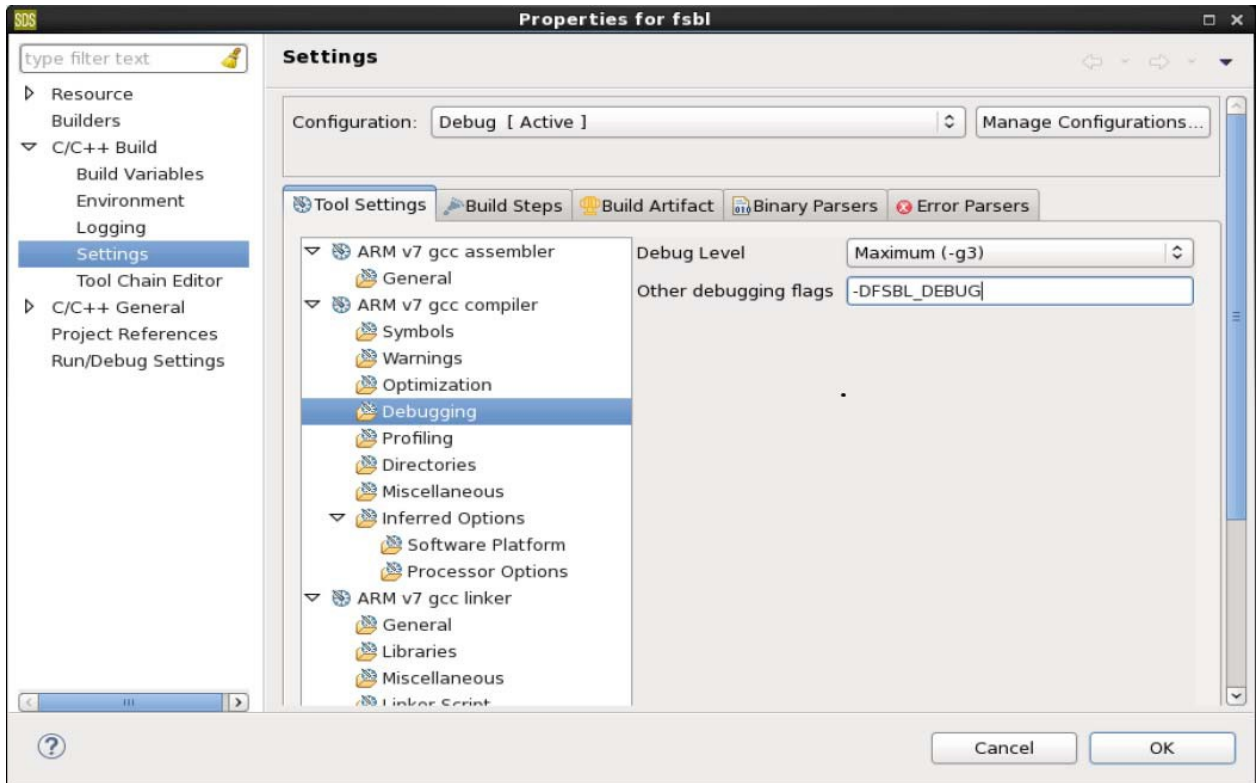
1. Power up the R5.
2. A register is set indicating the FPD is locked pending POR as the reset or clear sequence cannot execute on the FPD.
3. R5 can read the FP locked status from PMU_GLOBAL REQ_ISO_STATUS register bit '4'.
4. At this stage, PMU_GLOBAL REQ_PWRUP_STATUS bit '22' will not be set.
5. To 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. |
| SBL_SECURE_EXCLUDE | Excludes authentication code and encryption code but not checksum 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:

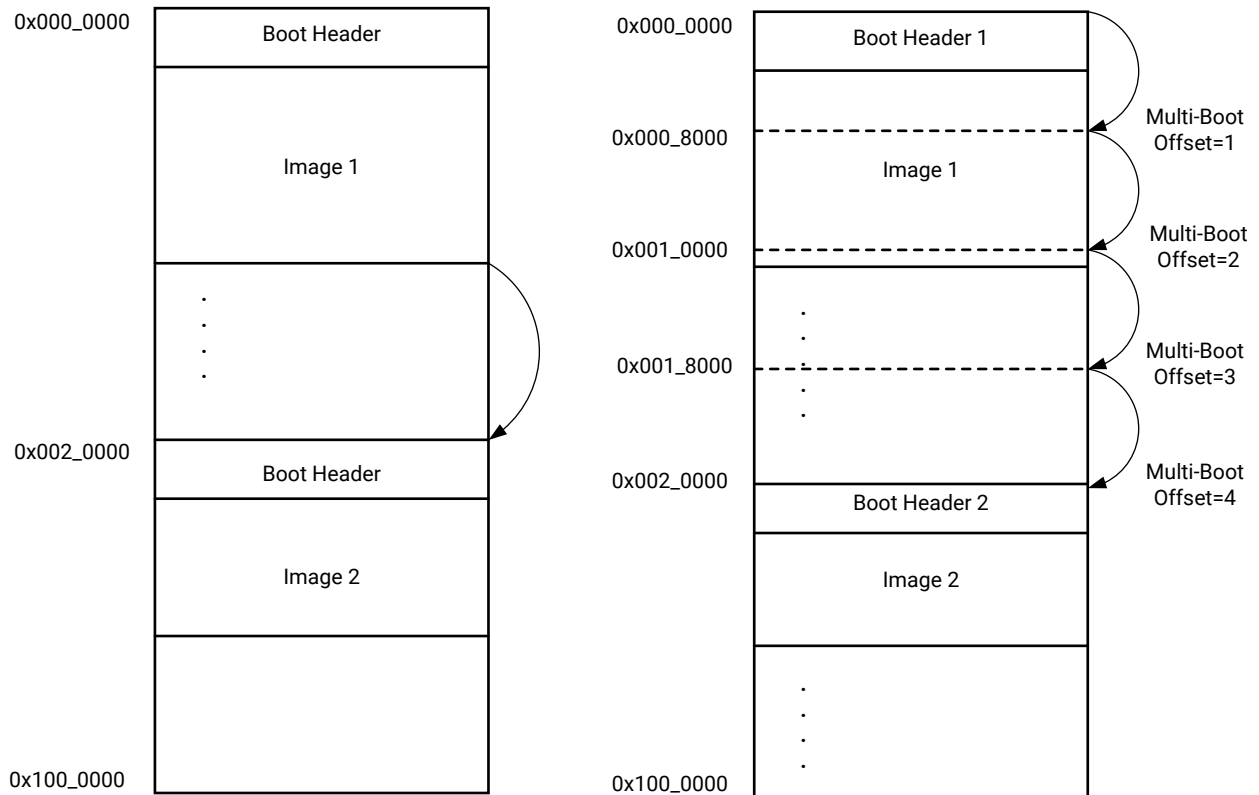
- BootROM searches for a valid image identification string (XLNX as image ID) at offsets of 32 KB in the flash.
- After finding a valid identification value, validates the checksum for the header.
- If the checksum is valid, the bootROM loads the image. This allows for more than one image in the flash.

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 using the MultiBoot flow.

Figure 22: Fallback using the 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).

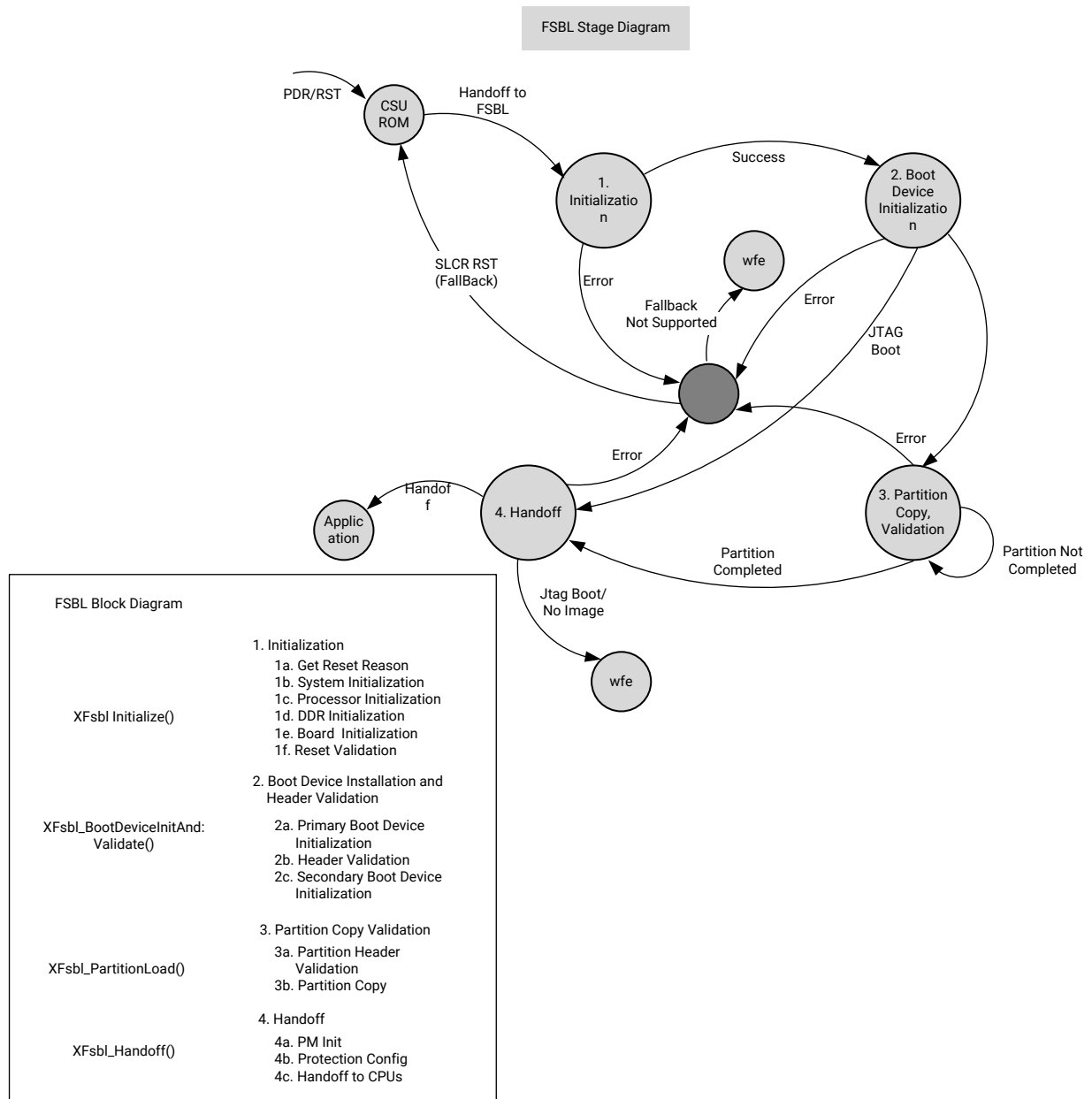
Phases of FSBL Operation

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

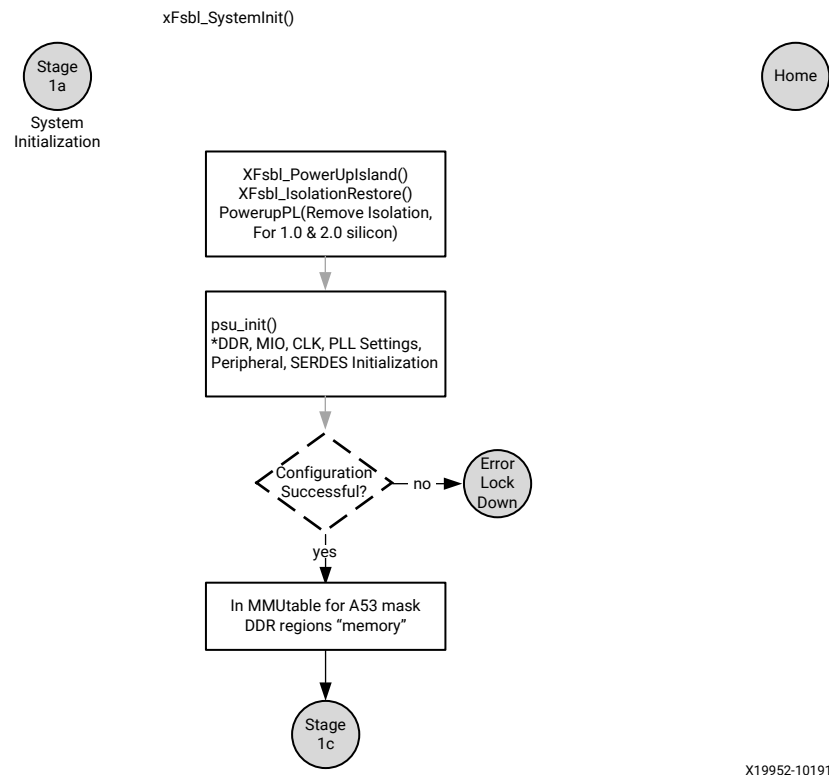
Initialization

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.

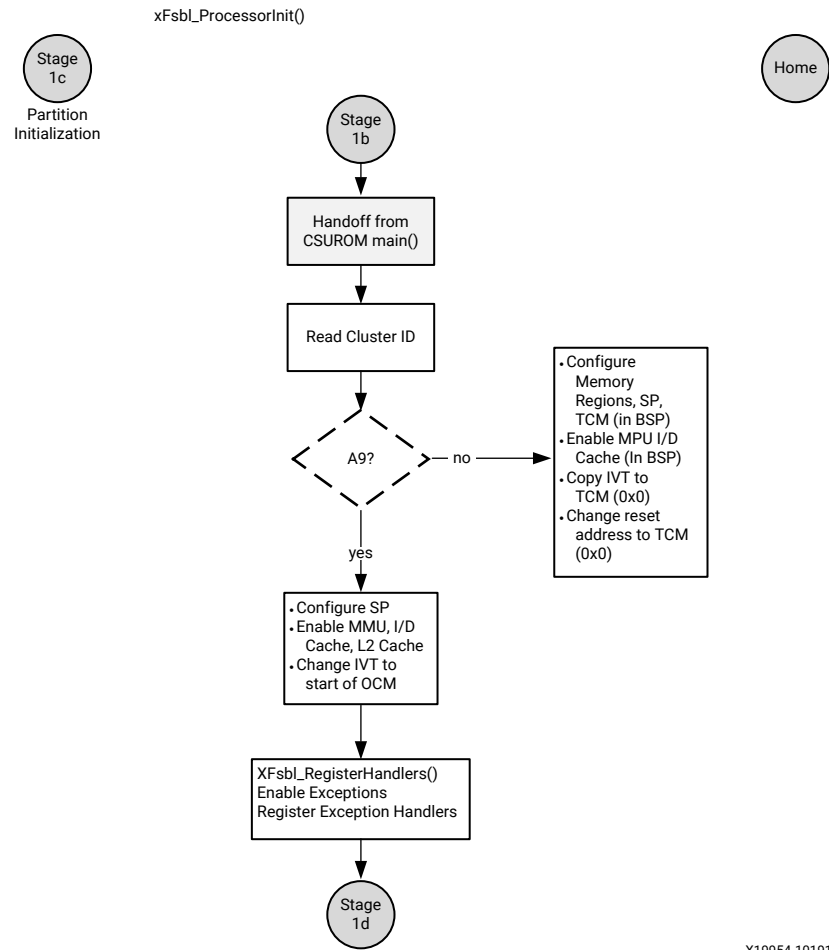
Figure 24: FSBL System Initialization



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.

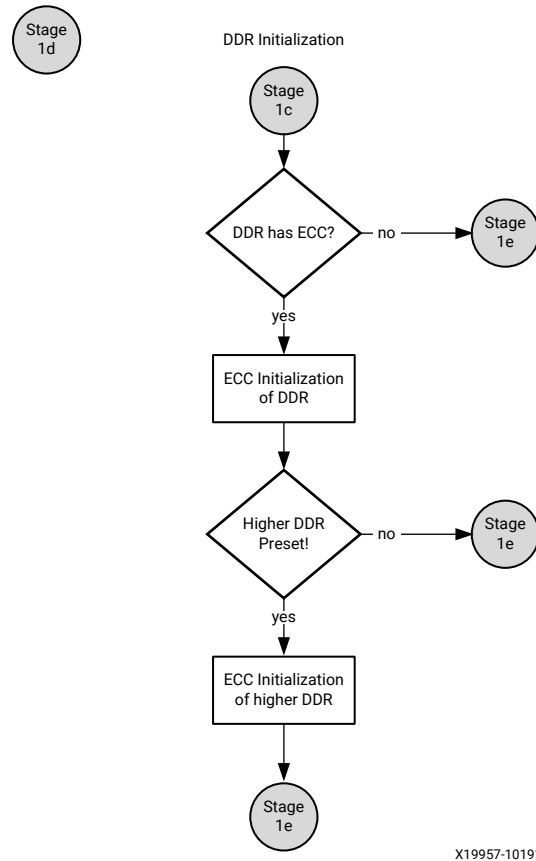
Figure 25: Processor Initialization



Initialize DDR

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

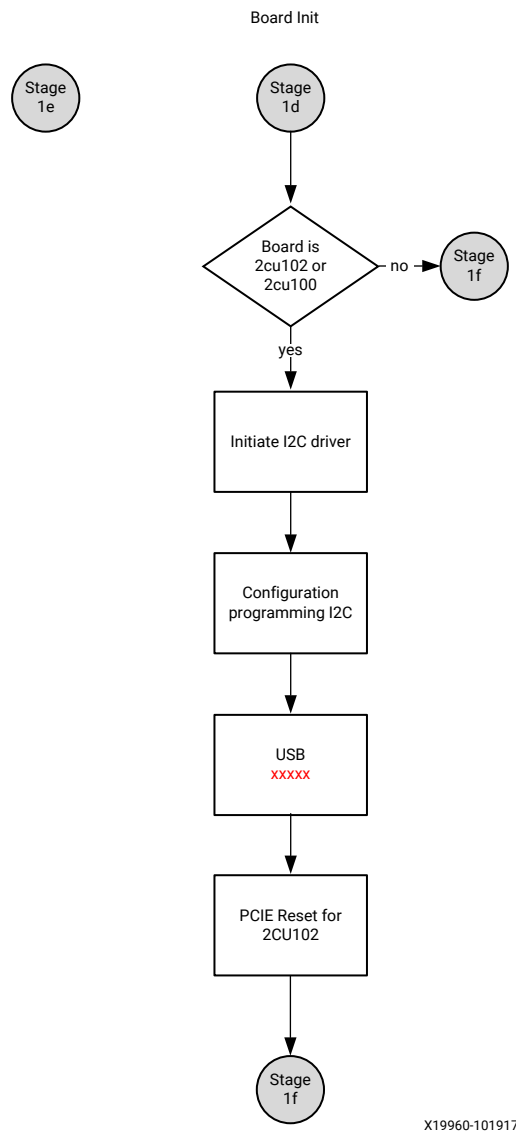
Figure 26: DDR Initialization



XFsbI_BoardInit

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

Figure 27: Board Initialization

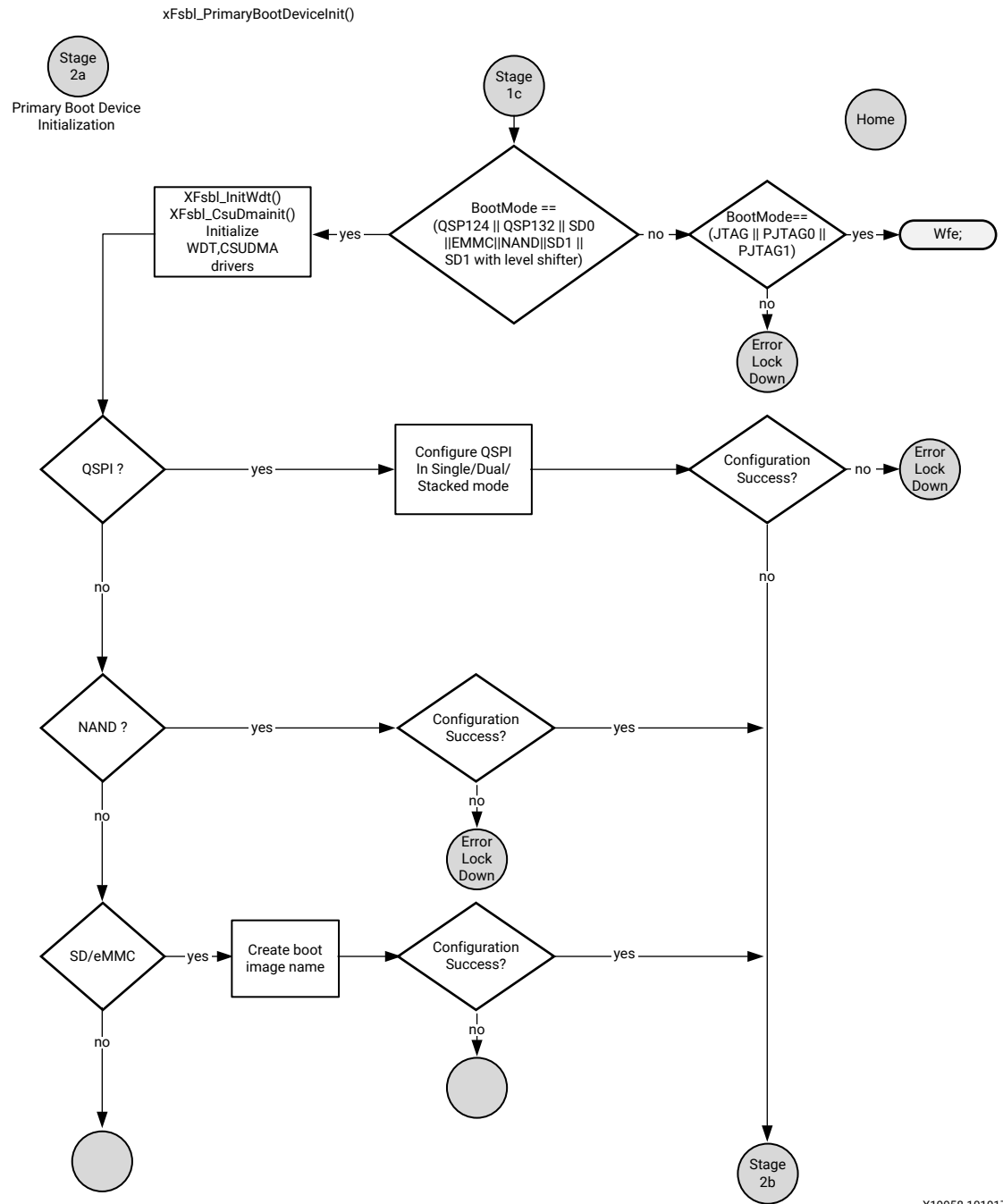


Boot Device Initialization

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

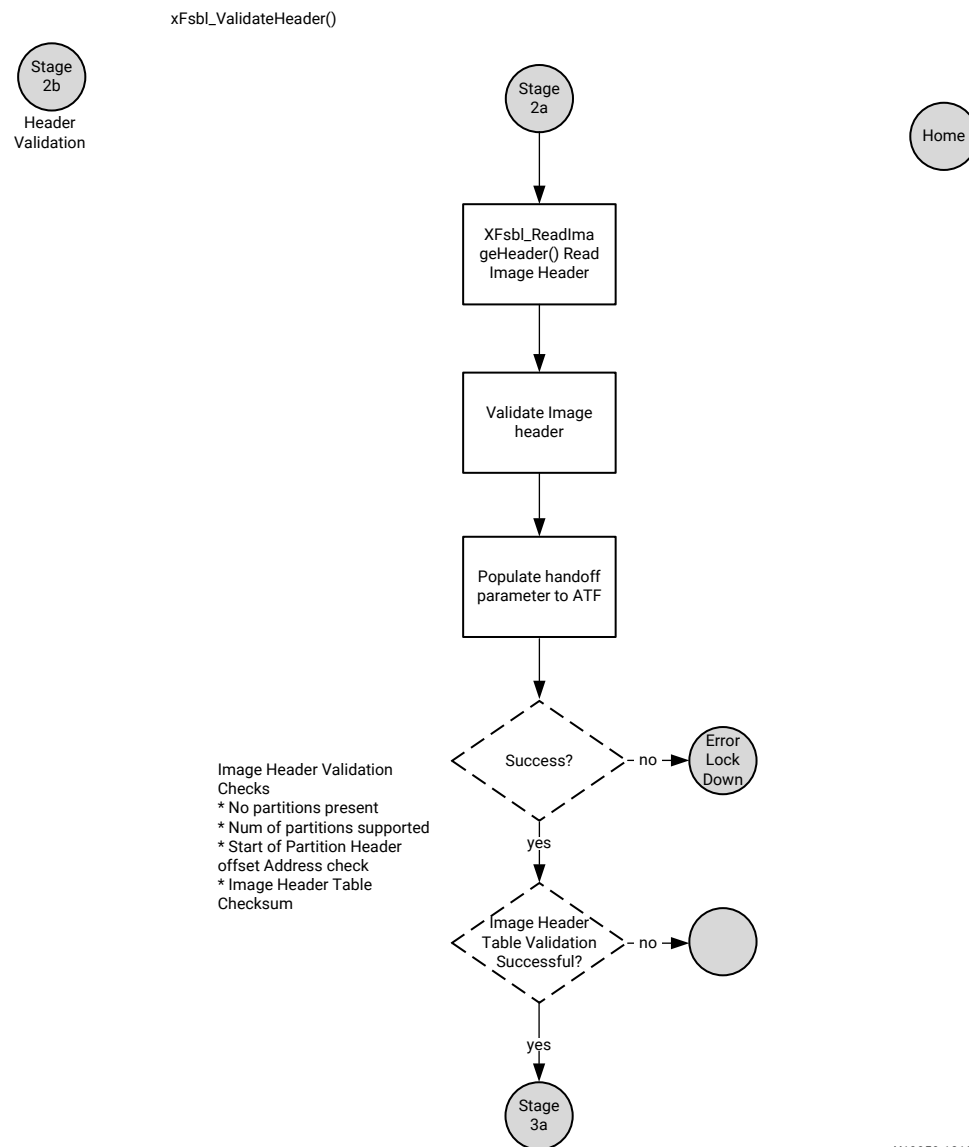


X19958-101917

XFsbL_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.

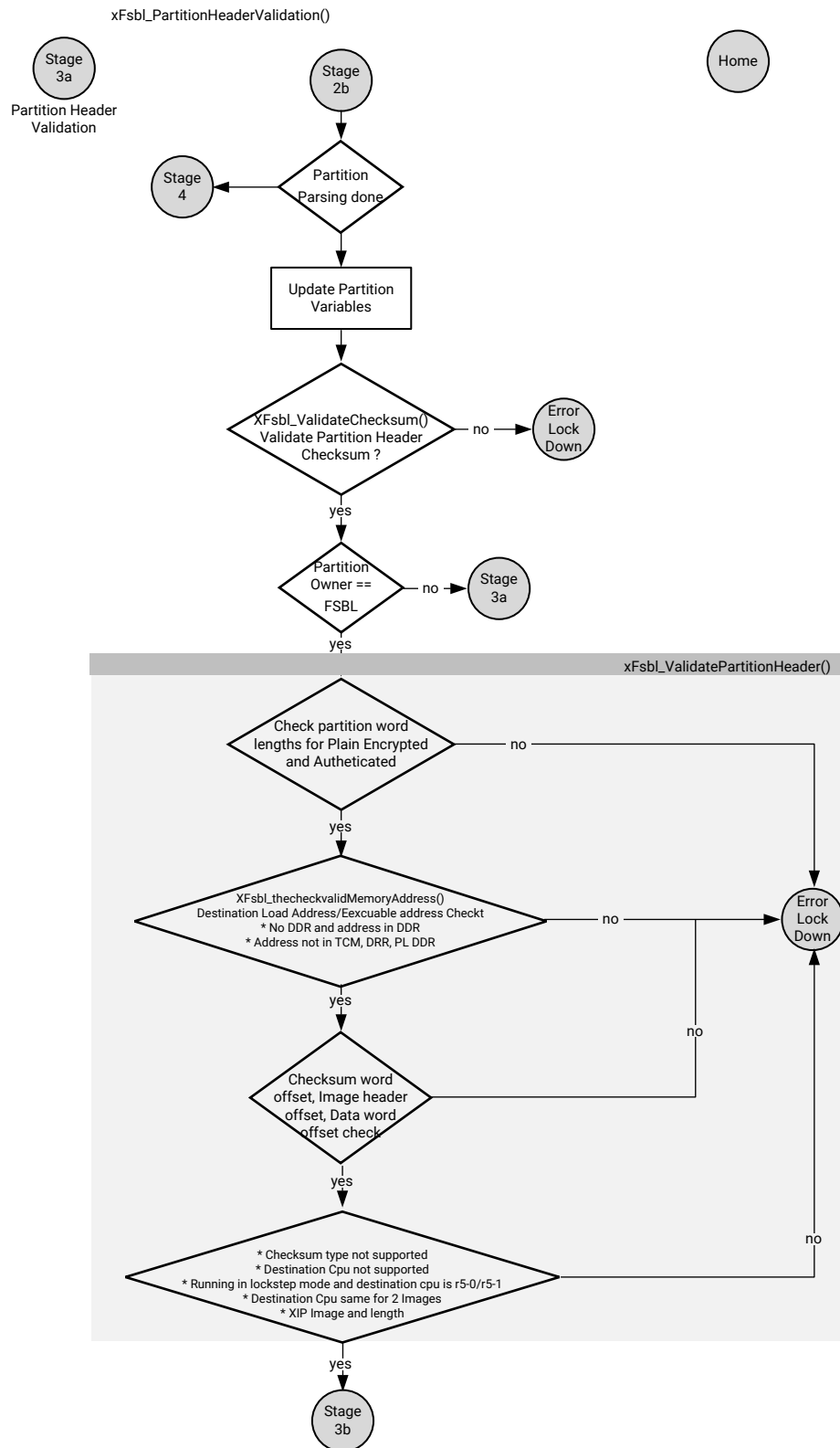
This function sets the handoff parameters to the Arm Trusted Firmware (ATF). The first argument is taken from the FSBL partition header. A pointer to the handoff structure containing these parameters is stored in the PMU_GLOBAL.GLOBAL_GEN_STORAGE6 register, which the ATF reads. The structure is filled 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

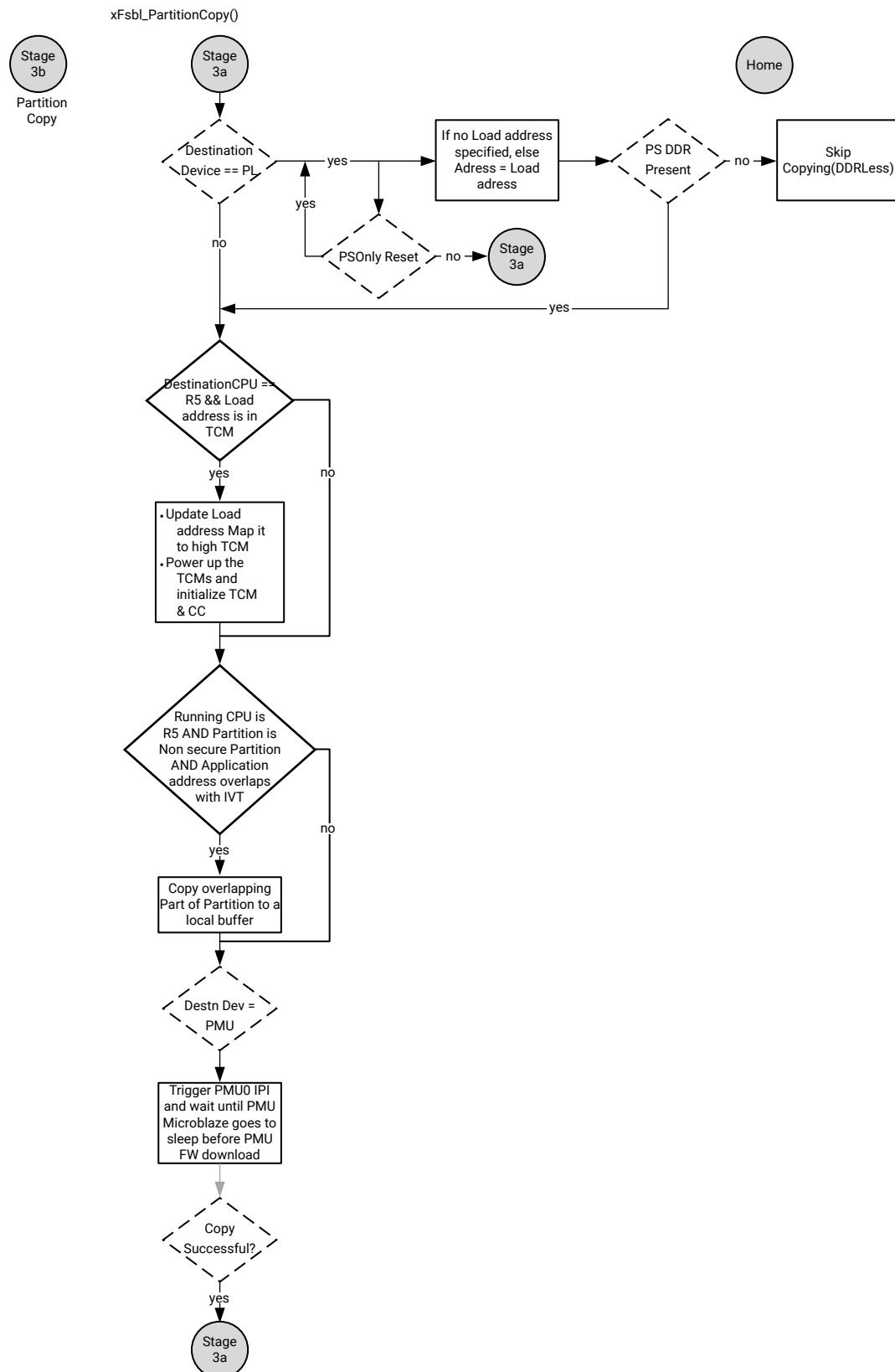


X19951-101917

XFsbI_PartitionCopy

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

Figure 31: Partition Copy

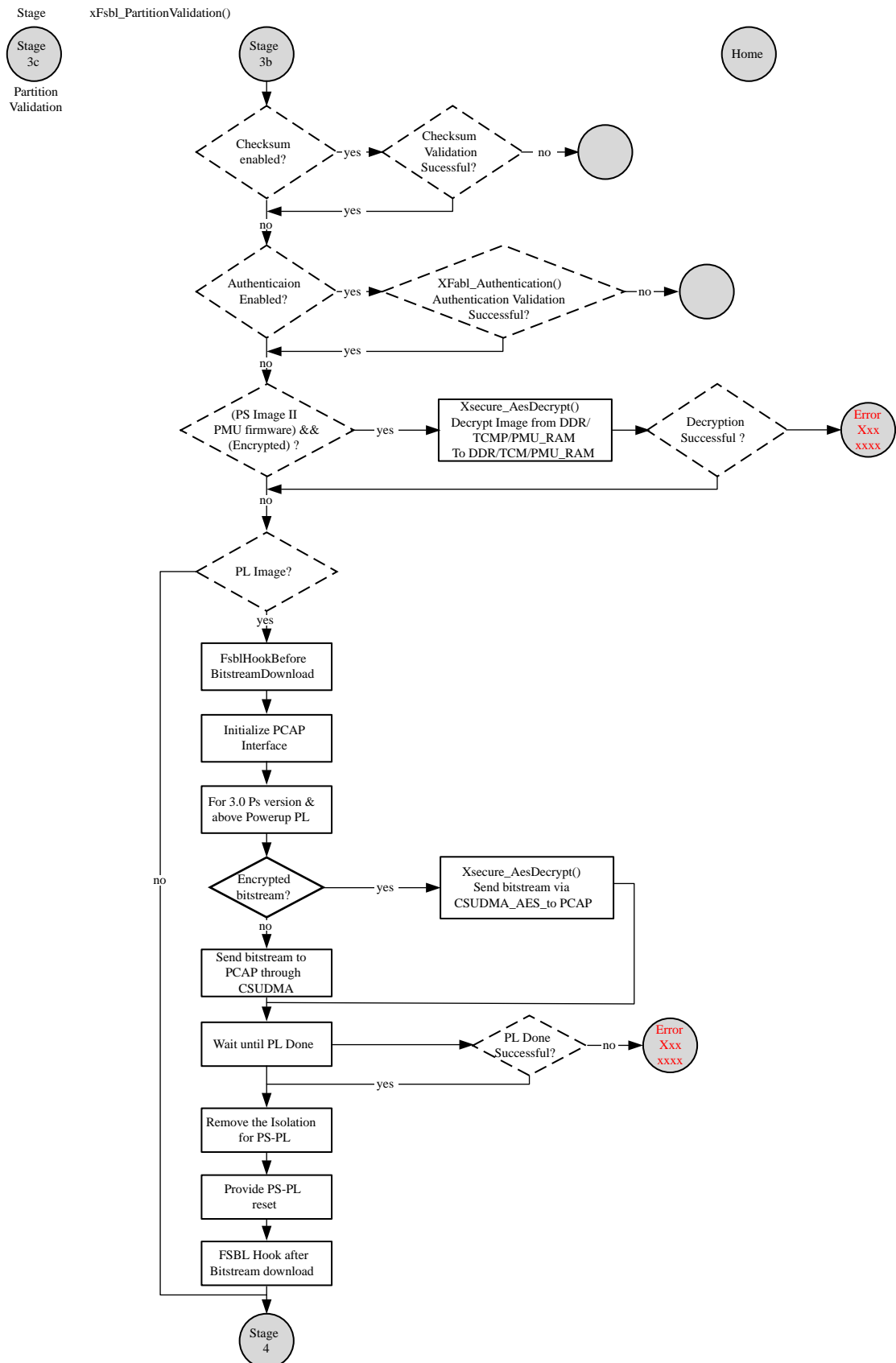


X19950-101917

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.

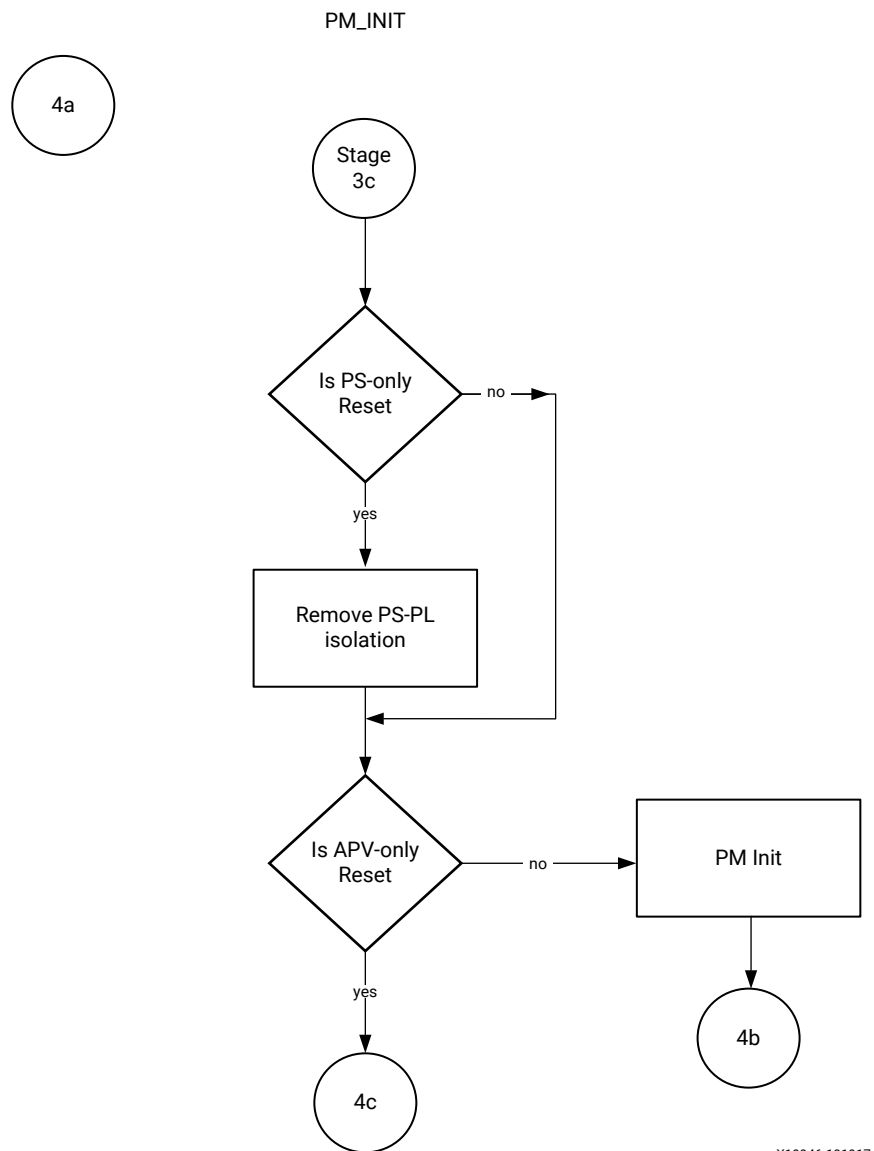
Any running processor cannot pass any parameters to any other processor. Any communication between various partitions can happen by reading from (or writing to) the PMU global registers.

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.

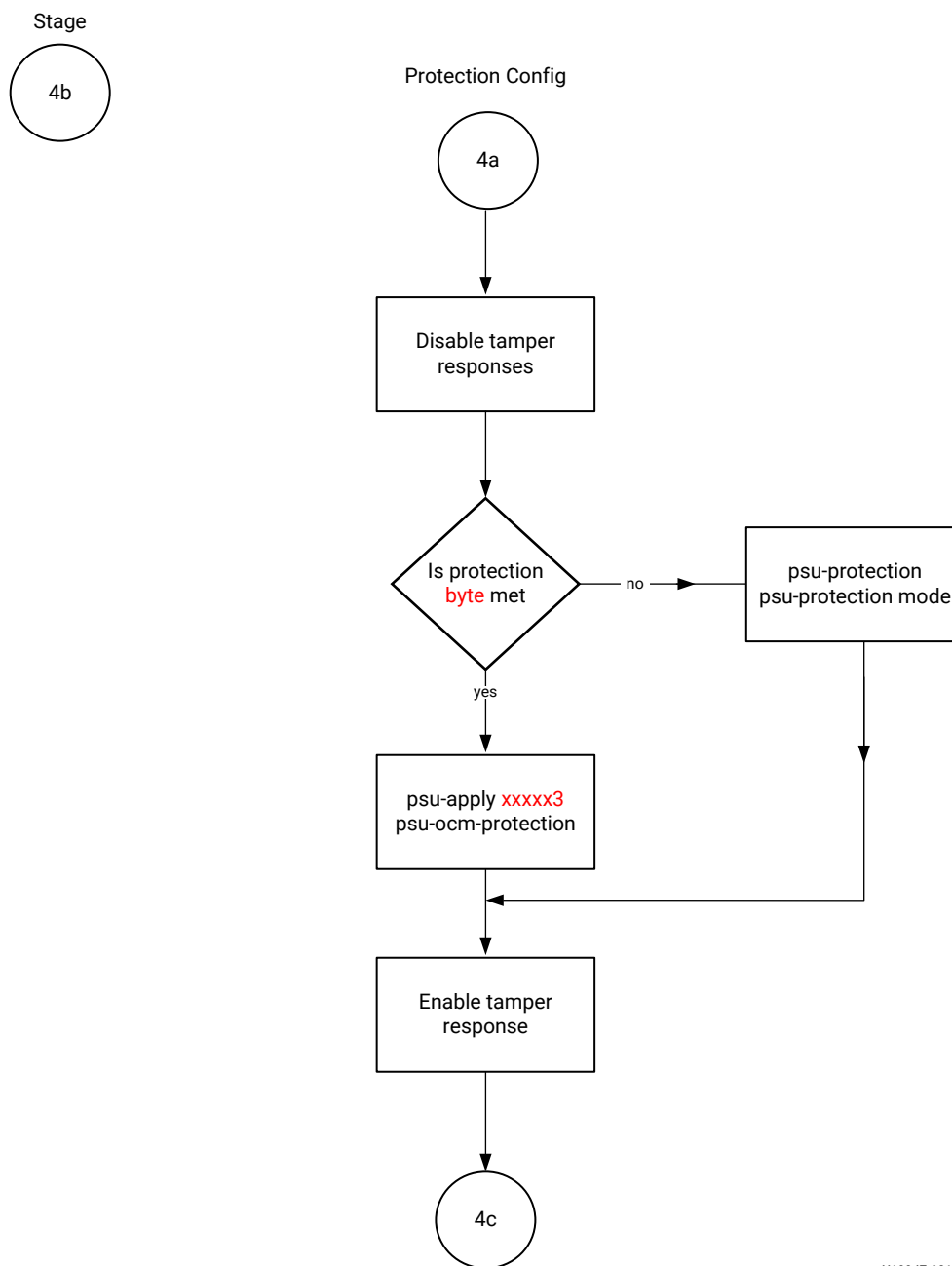
Figure 33: PM Initialization



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



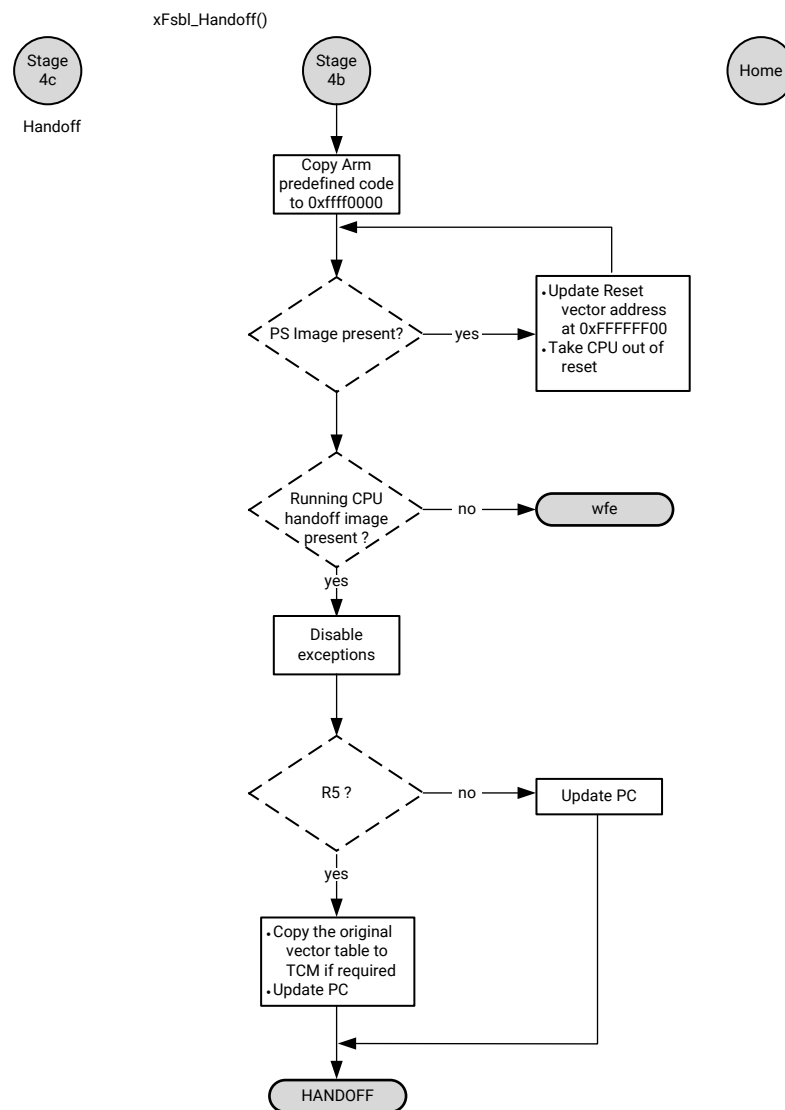
X19947-101917

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

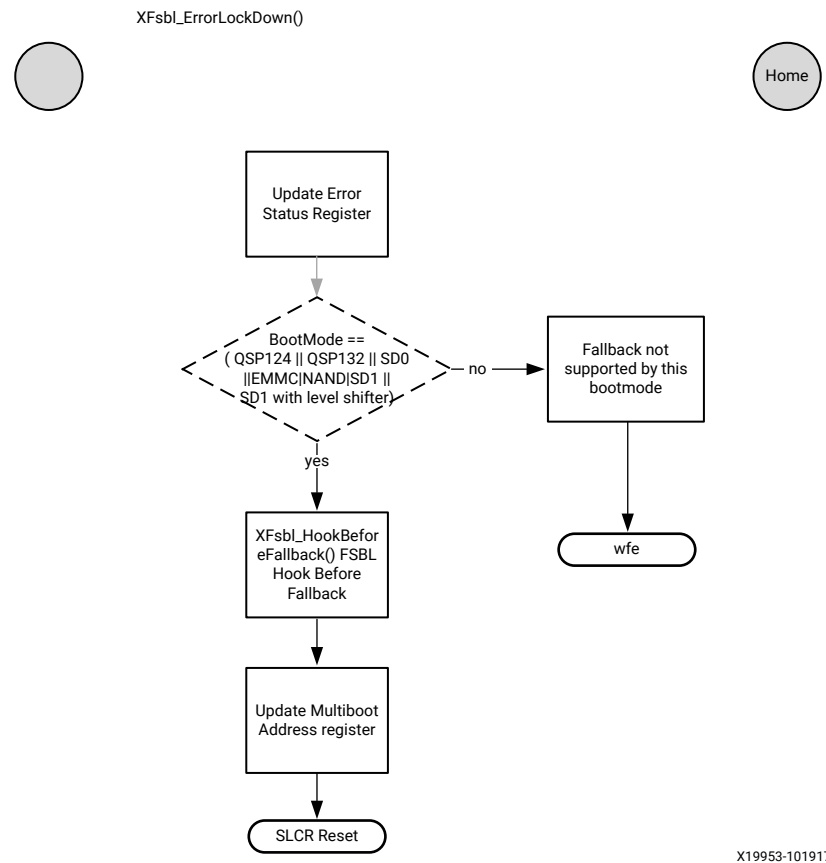
| FSBL | Application | Processor Cores | Execution Address |
|--------|-------------|---------------------------------------|-------------------|
| 64-bit | 64-bit | All (i.e. A53-0, A53-1, A53-2, A53-3) | Any Address |
| 64-bit | 32-bit | A53-1, A53-2, A53-3 | 0x0 |
| 32-bit | 32-bit | A53-0 | Any Address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



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 a 256-bit 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 cryptographic 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]      ZynqMP_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] bbam_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] bbam_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 the eFUSEs with the back key, see Zynq eFUSE PS API in the *OS and Libraries Document Collection* ([UG643](#)).

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 Zynq UltraScale+ MPSoCs 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] 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. 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` file across multiple partitions, reuses 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. These blocks (SHA-3/384, and RSA) are hardened and part of crypto interface block (CIB). You can use authentication by itself or in conjunction with encryption.

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](#)).

As with all asymmetric algorithms, the private (secret) keys (PSK and SSK) are used to sign while the public versions (PPK and SPK) are used to verify (authenticate). 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, Xilinx uses 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. The difference between the two is padding (10*1 vs 0110*1). Request XPT475 from your Xilinx representative for details on the NIST variances.

Bootgen supports RSA-4096 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=0x00000001
[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 =
x000000001, sskfile = ssk2.pem]zynqmp-fsbl.elf
[destination_cpu =a53-0, authentication = rsa, spk_select = user-
efuse,spk_id = 0x1, sskfile = ssk3.pem]Application1.elf
[destination_cpu =a53-0, authentication = rsa, spk_select = spk-efuse,
spk_id = 0x00000001, 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 =
0x000000001, sskfile = ssk2.pem]zynqmp-fsbl.elf
[destination_cpu =a53-0, authentication = rsa, spk_select = user-efuse,
spk_id = 1, sskfile = ssk3.pem]Application1.elf
[destination_cpu =a53-0, authentication = rsa, spk_select = spk-efuse,
spk_id = 0x00000001, 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 a 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.

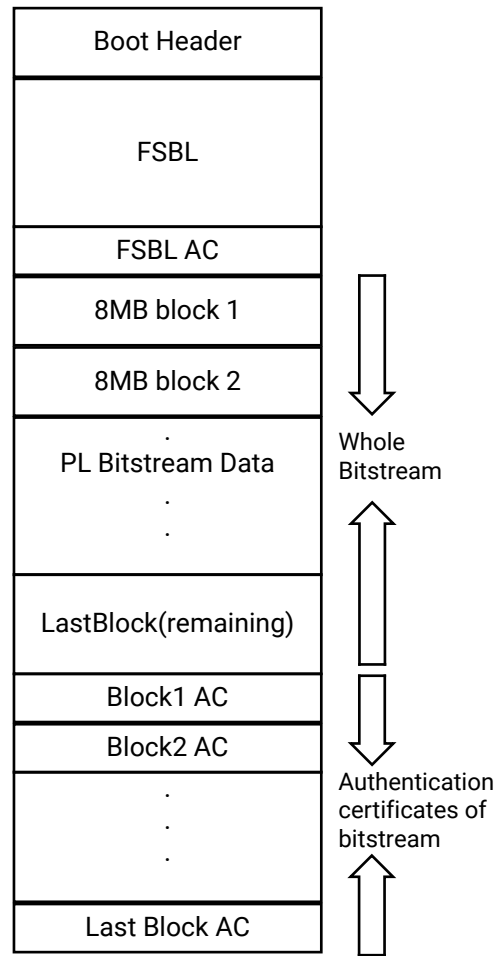
The following section describes how the bitstream is authenticated securely using external memory.

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



X19220-110320

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.



IMPORTANT! While creating a boot image, the bitstream partition should be before the 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 a secure monitor. The specific secure monitor 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. However, the ATF provided by Xilinx does include hooks that allow customers to add their own Trusted OS and Trusted applications in order to implement a Trusted Execution Environment. ATF is the secure monitor that provides switching between the secure and the non-secure world. See [this whitepaper](#) for more information.

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.

Another usage of ATF is to prevent any user space application from directly accessing the hardware cryptographic engine. Instead, a user space application can make a call to the kernel where the data to be processed is copied to kernel space. Afterwards, the driver will copy the data from the kernel's virtual memory to physical address. Later, the driver will make a call to ATF and then to the PMU/CSU to perform cryptographic operations on the physical address.

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, are 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>dcsw_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 <code>SCR_EL3</code> which contains the routing model (expressed through the <code>IRQ</code> and <code>FIQ</code> bits) for a security state that is stored through a previous call to <code>set_routing_model()</code> . |

Table 31: ATF Functions (cont'd)

| ATF Functions | Description |
|---------------------------------|--|
| cm_prepare_el3_exit() | 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 SCTLR_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 cpu_context. |
| cm_get_context(security_state); | Gets the context of the security state. |
| el1_sysregs_context_restore | Restores the context of the system registers. |
| cm_set_next_context | Programs the context used for exception return. This initializes the SP_EL3 to a pointer to a cpu_context set for the required security state. |
| 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

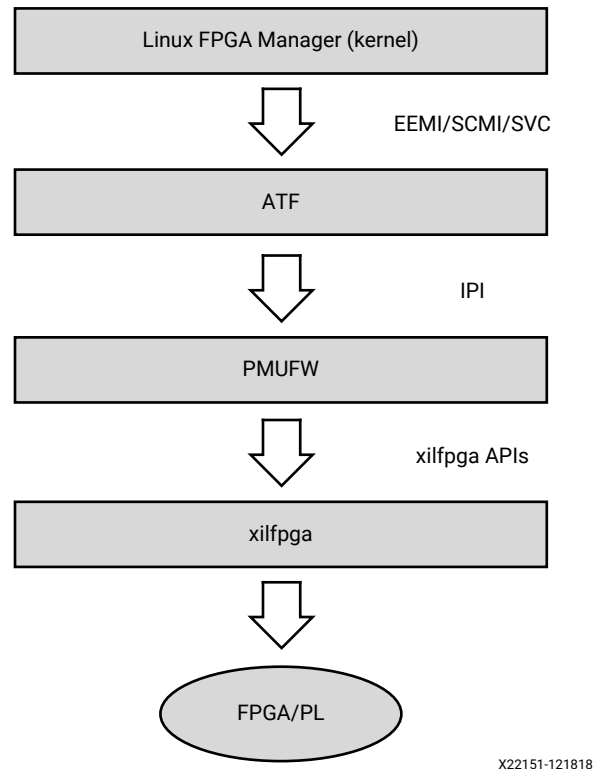
The FPGA Manager in the Zynq UltraScale+ MPSoC provides an interface to download different types of bitstreams (full, partial, authenticated, encrypted and so on) during runtime from Linux environment. The key features of the FPGA Manager are as follows:

- 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



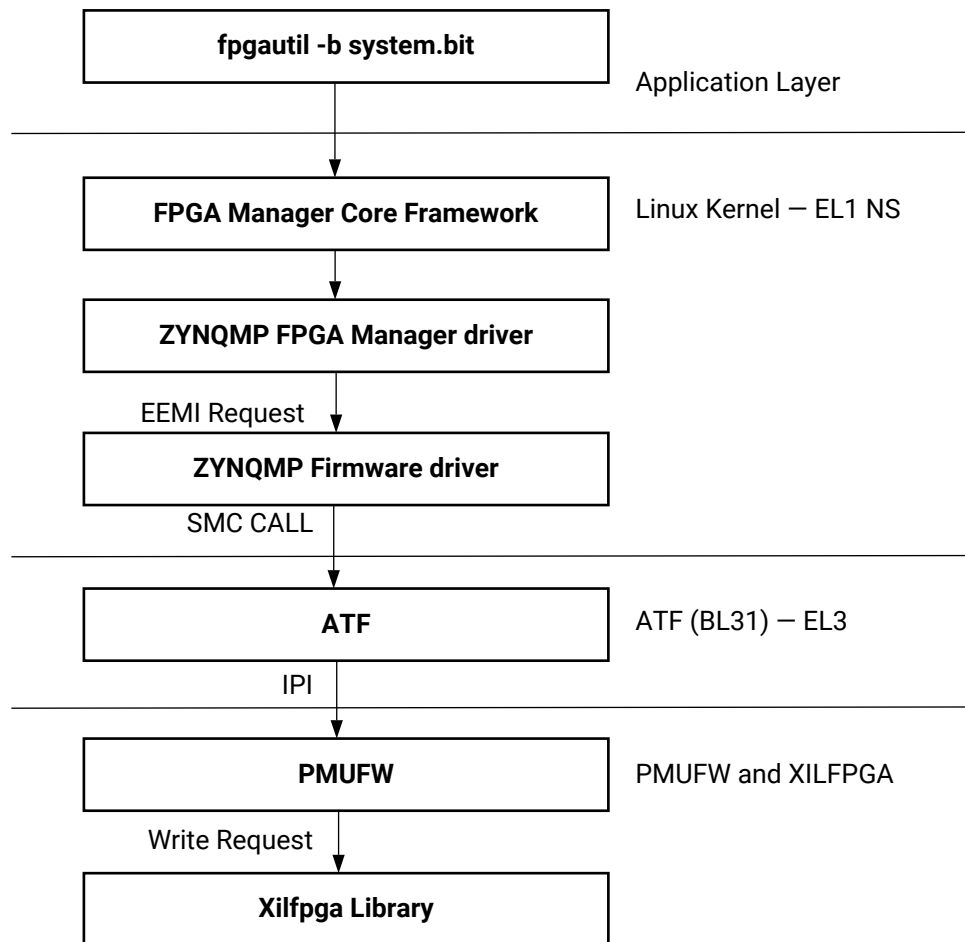
X22151-121818

Execution Flow

FPGA manager provides an abstraction for the user to load bitstream using Linux. The xilfpga library initializes the PCAP, CSUDMA and other hardware. For more details about xilfpga, see the XilFPGA section in the *OS and Libraries Document Collection* ([UG643](#)).

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 Manager 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



X22152-110320

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](#).

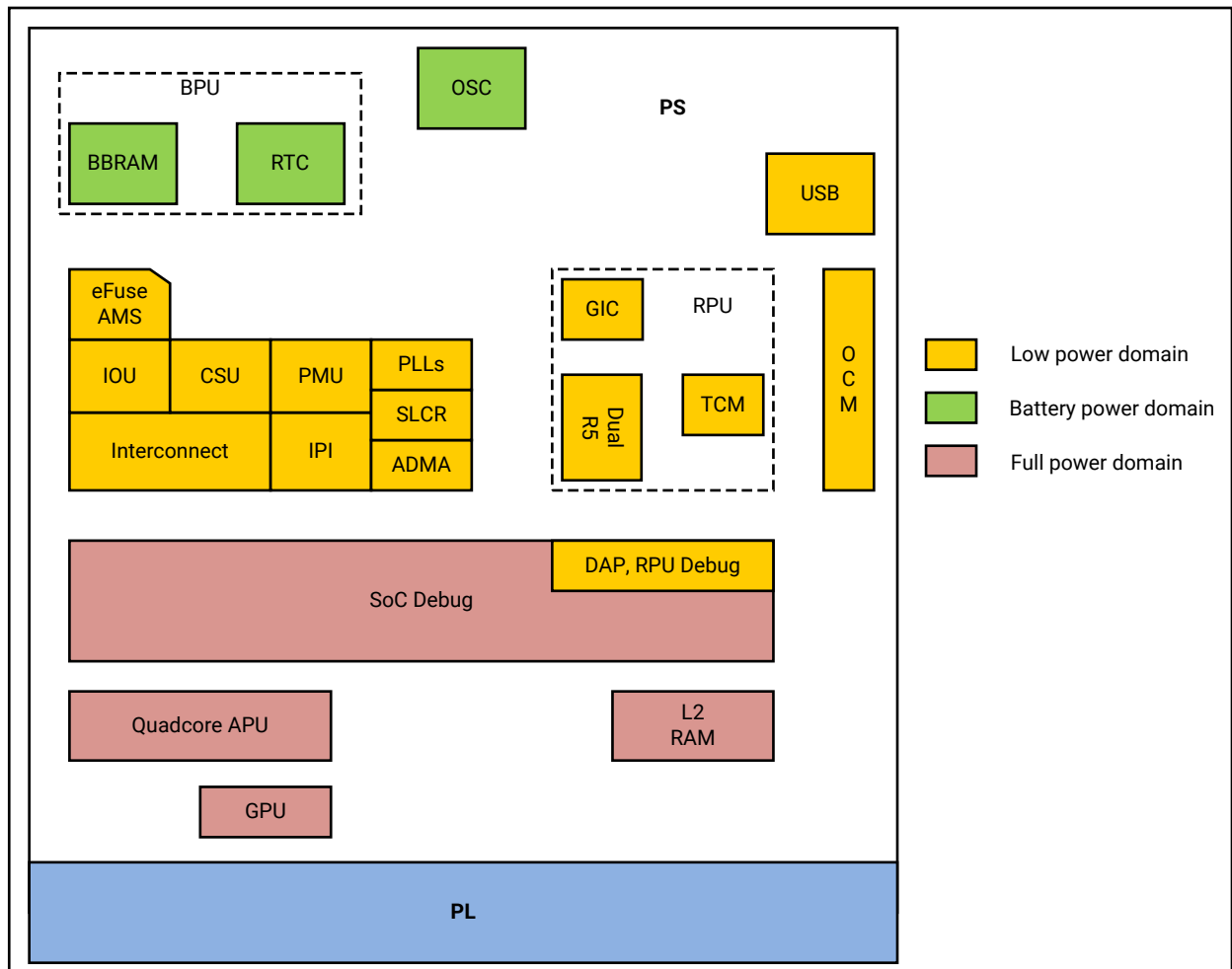
The PS block offers high levels of functionality and performance. At the same time, there is a strong need to optimize the power consumption of this block with respect to the functionality and performance that is necessary at each stage of the operation.

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.

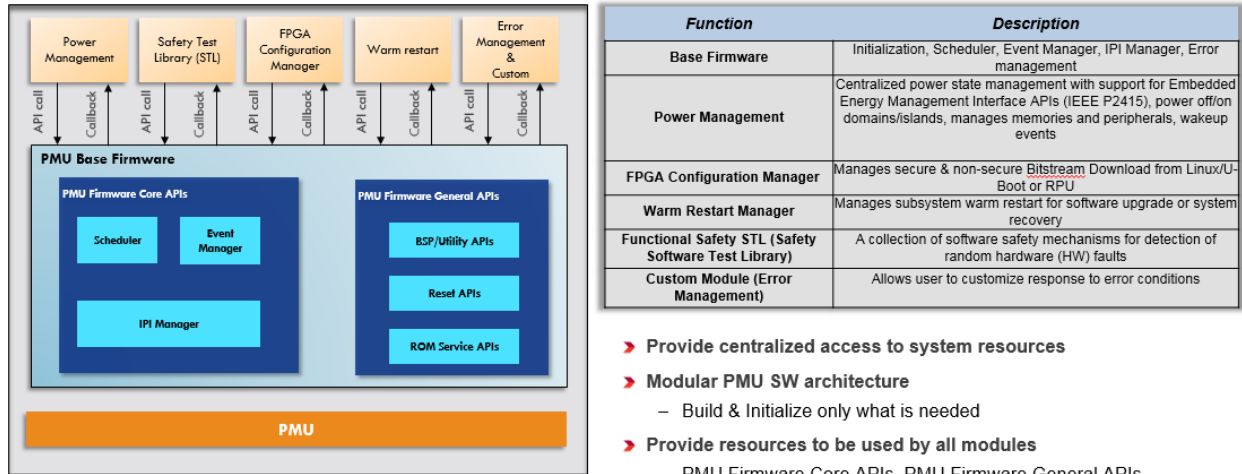
PMU firmware can be divided into base firmware and modules. PMU Base Firmware does initialization of modules, registering events for the modules, and provides all the common functions that may be required by the modules. These common functions can be categorized into the following APIs:

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

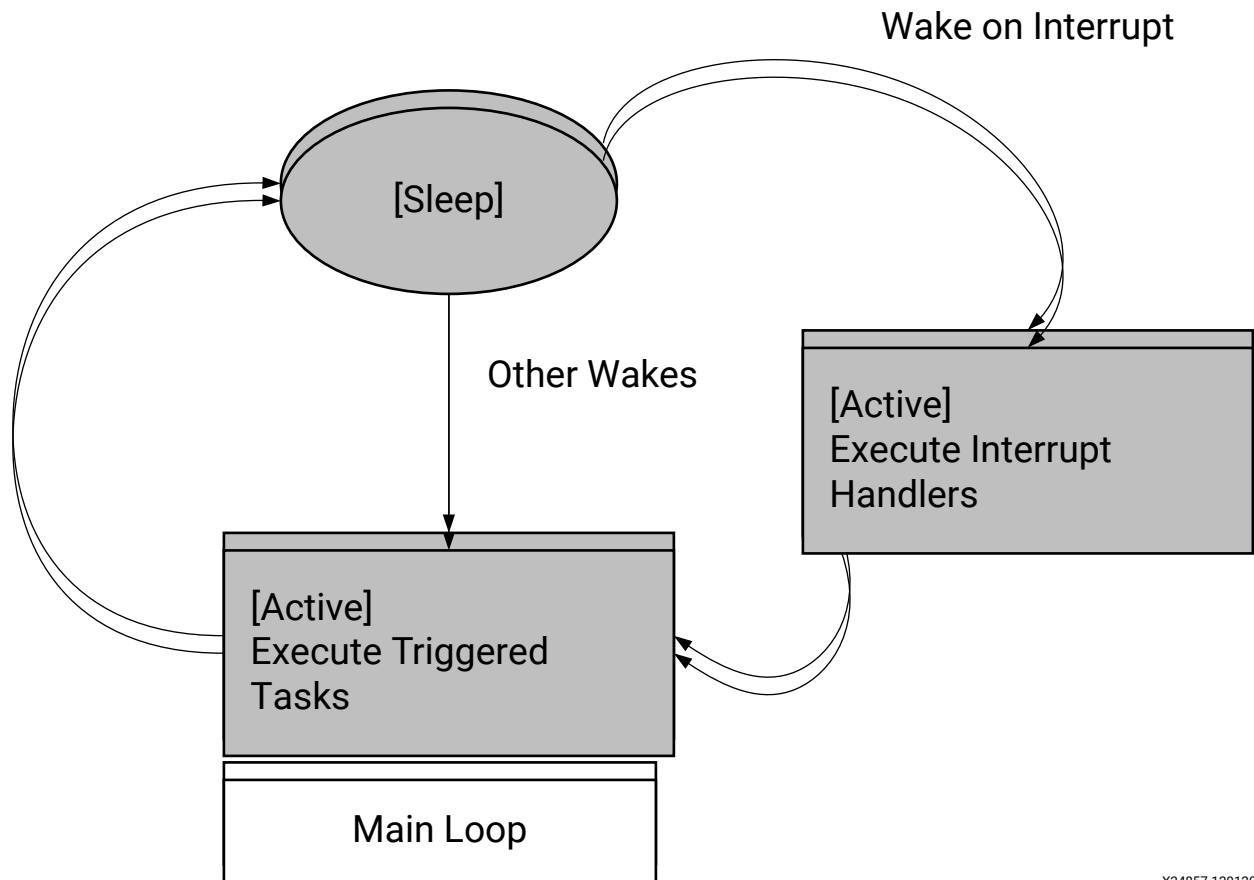
Platform Management Overview



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



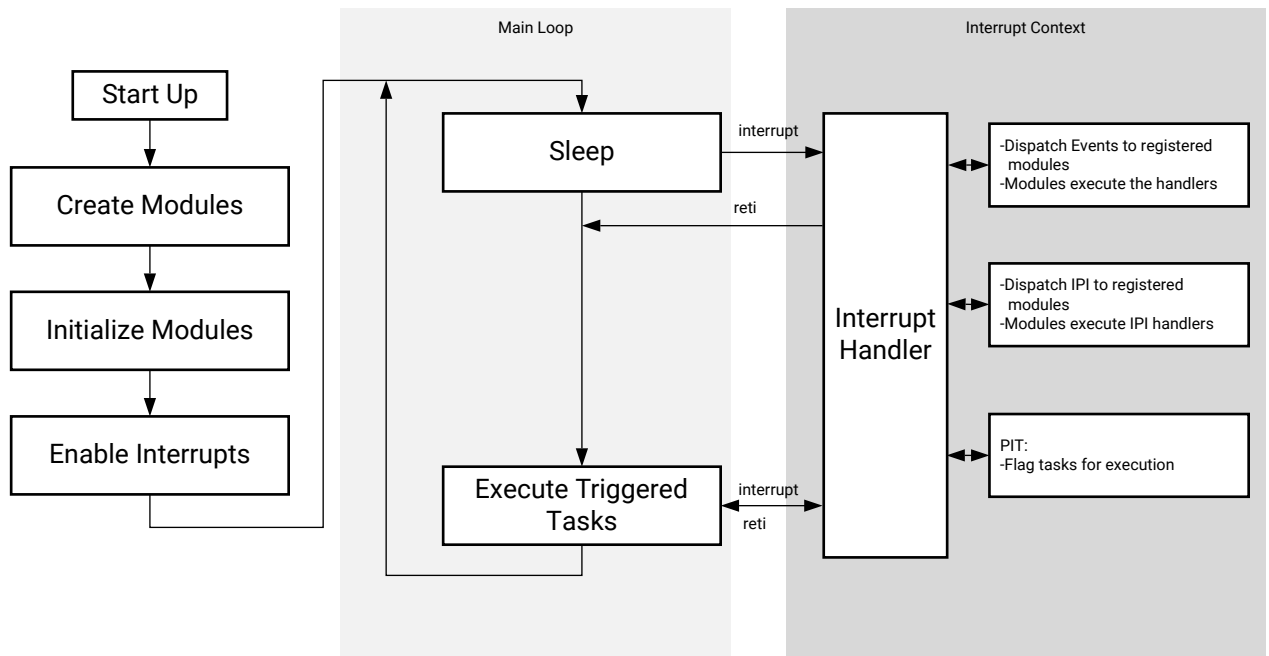
X24857-120120

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



X24856-120120

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

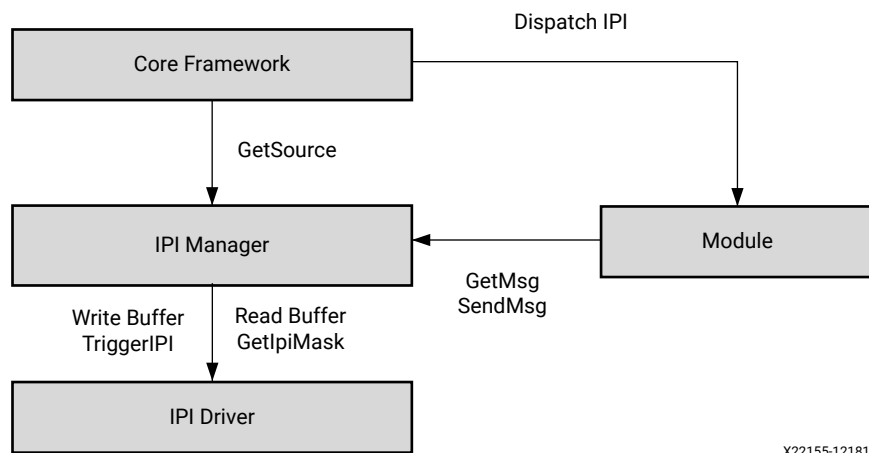
| Word | Content | Description |
|------|---------|----------------------------|
| 0 | Header | <target_module_id, api_id> |

Table 32: IPI Message Format (cont'd)

| Word | Content | Description |
|------|----------|---------------------------|
| 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, target_module_id 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 |
|---------------------------|--|--|-----------------|
| XPFW_DEBUG_DETAILED | Enables detailed debug prints in PMU firmware. This feature is supported in 2017.3 release and above. | | Disabled |
| 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 |

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

| Flag | Description | Prerequisites | Default Setting |
|-------------------------|---|---------------|-----------------|
| 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

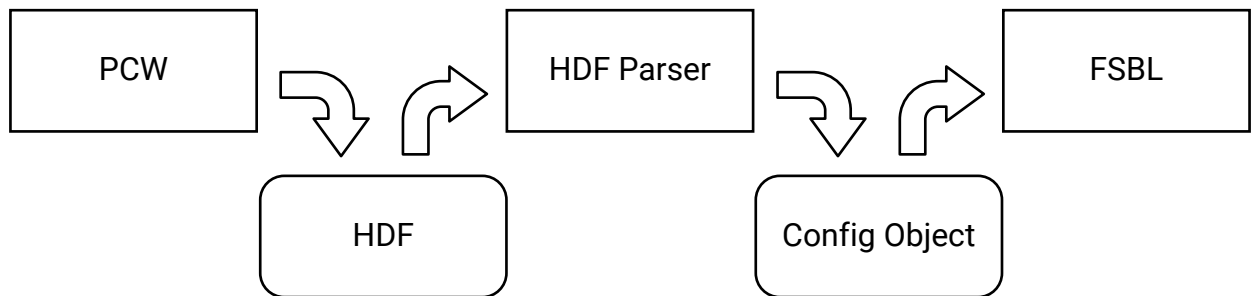
- Which masters are allowed to set configuration after the configuration is already set upon the system boot by the FSBL

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.

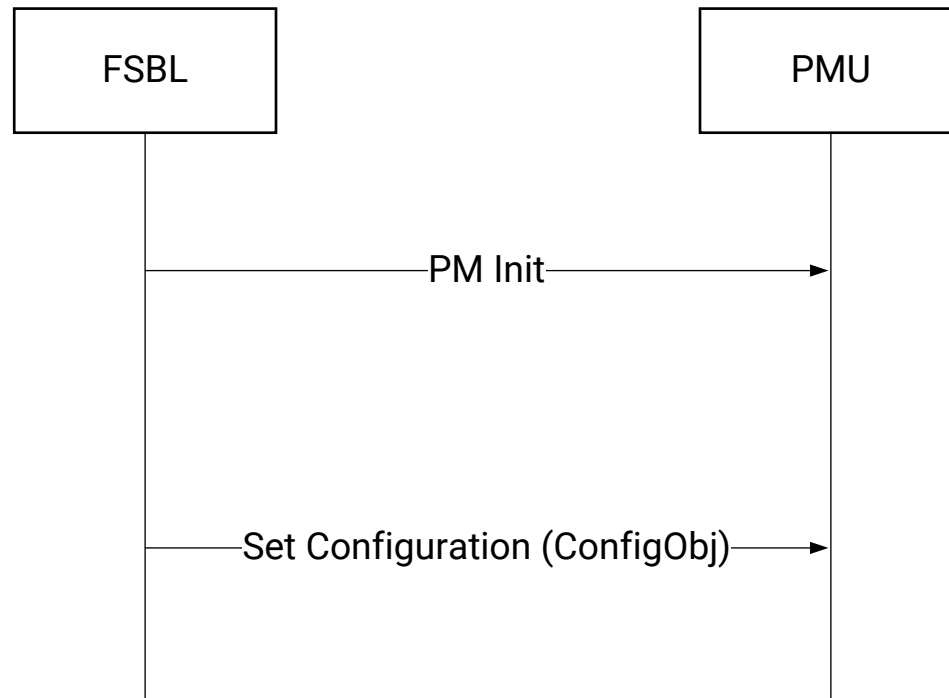
Following are the steps at boot level:

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

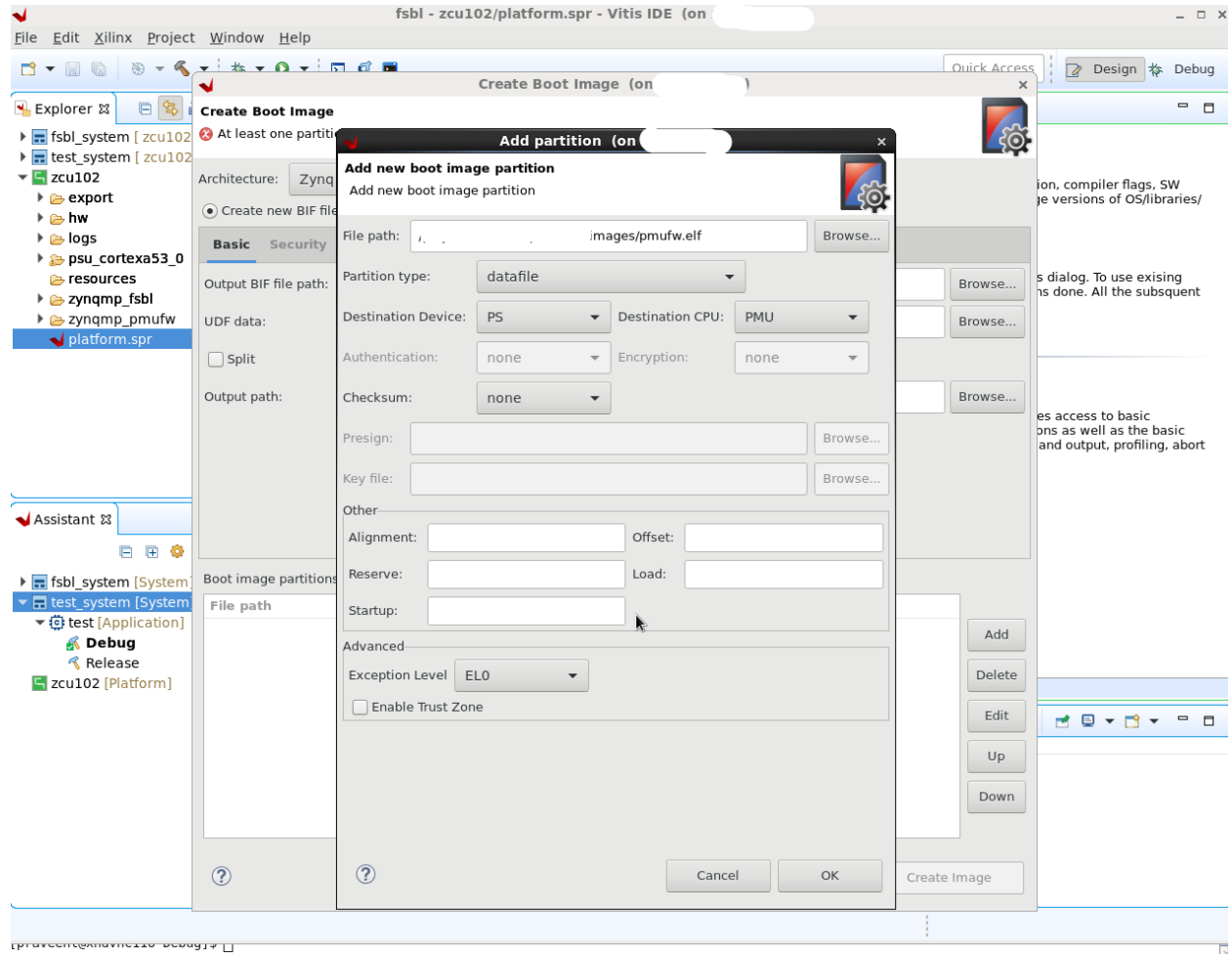
#Loading bitstream to PL
Targets -set -nocase -filter {name =~ "*PL*"}
fpga download.bit
```

Loading PMU Firmware in NON-JTAG Boot Mode

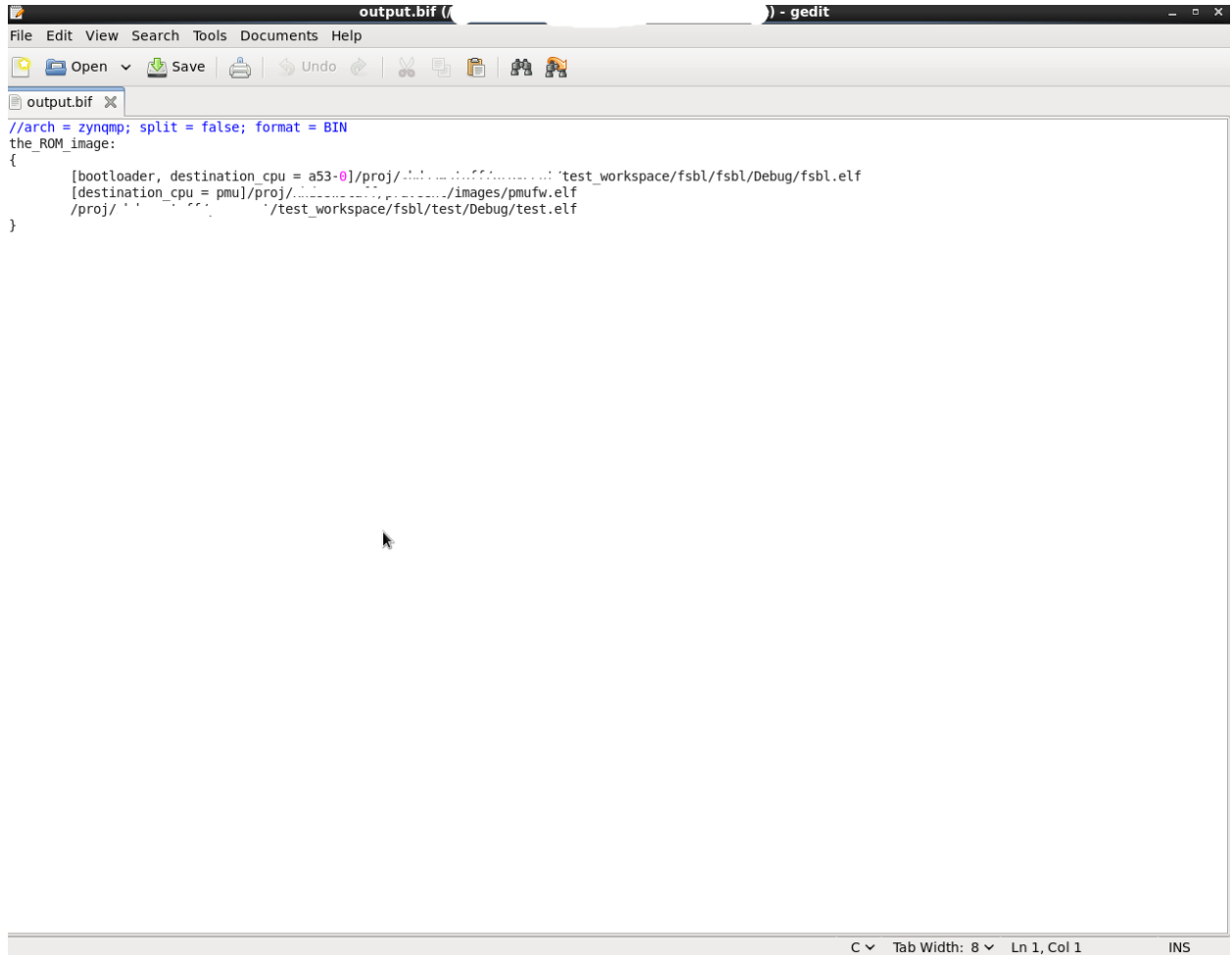
When PMU firmware is loaded in a non-JTAG Boot mode on a 1.0 Silicon, an error message 'Error: Unhandled IPI received' may be logged by PMU firmware at startup, which can be safely ignored. This is due to the IPI0 ISR not being cleared by PMU ROM. This is fixed in 2.0 and later versions of silicon.

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.



```
//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.

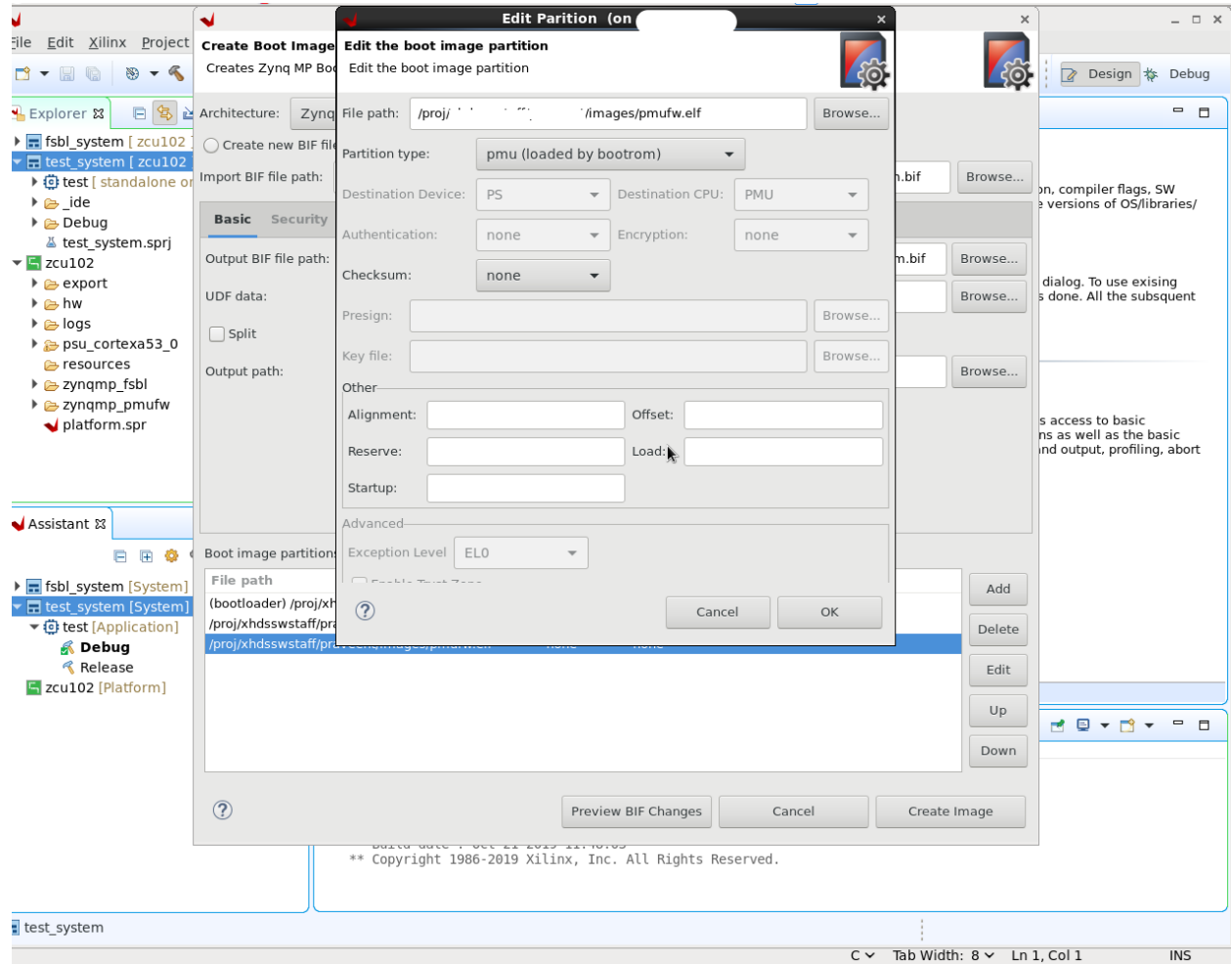
Using CBR to load PMU Firmware

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.

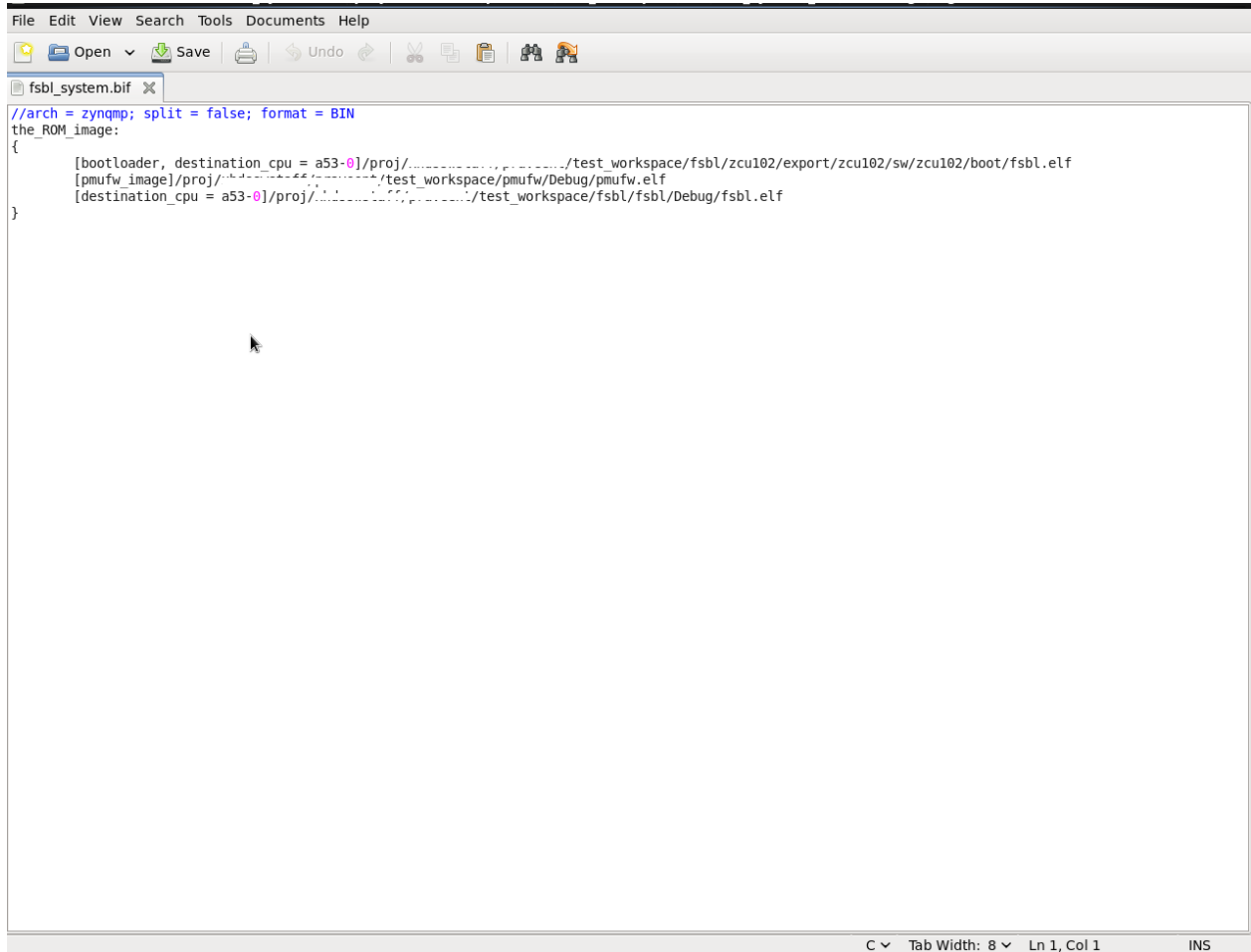
To make the CBR load PMU firmware, follow these steps:

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, Undo, Redo, Find, etc.]
fsbl_system.bif
//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");
}

```

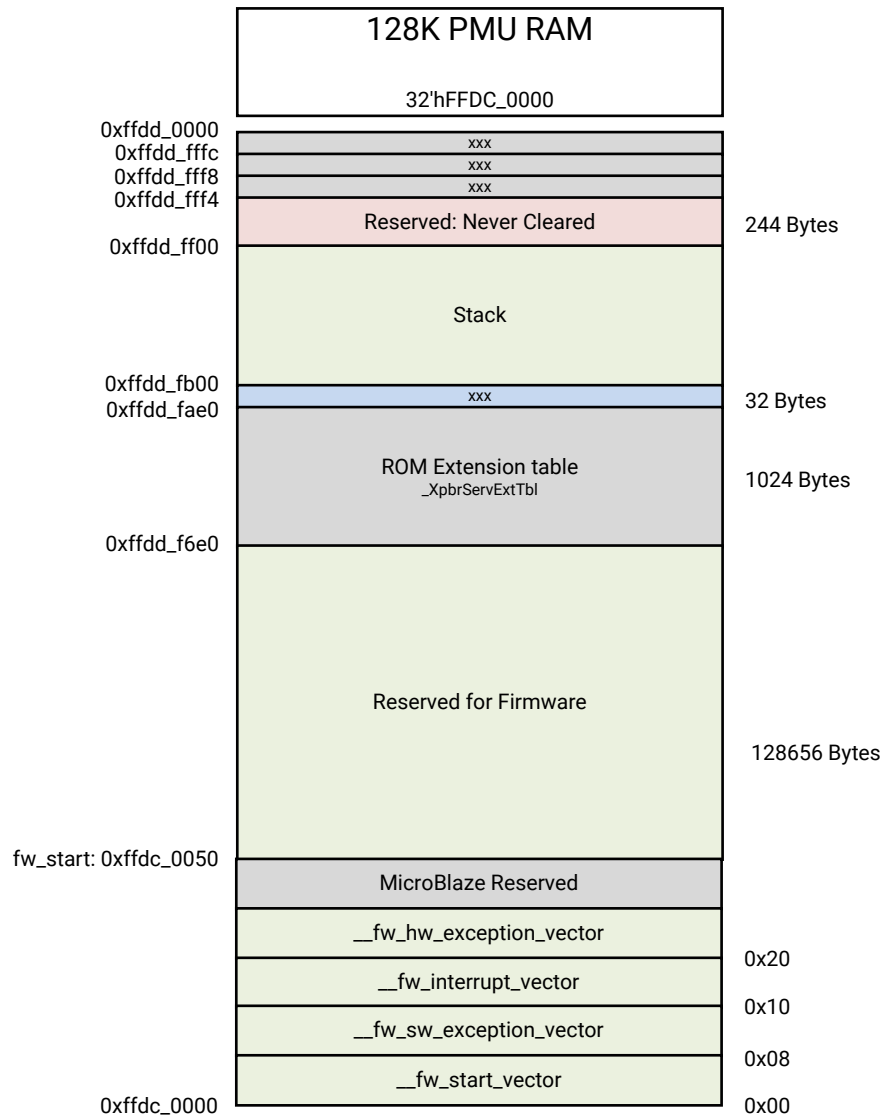
Note: This application must be run after FSBL is successfully executed. This application cannot run successfully, if FSBL fails to send configuration object to PMU firmware.

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.

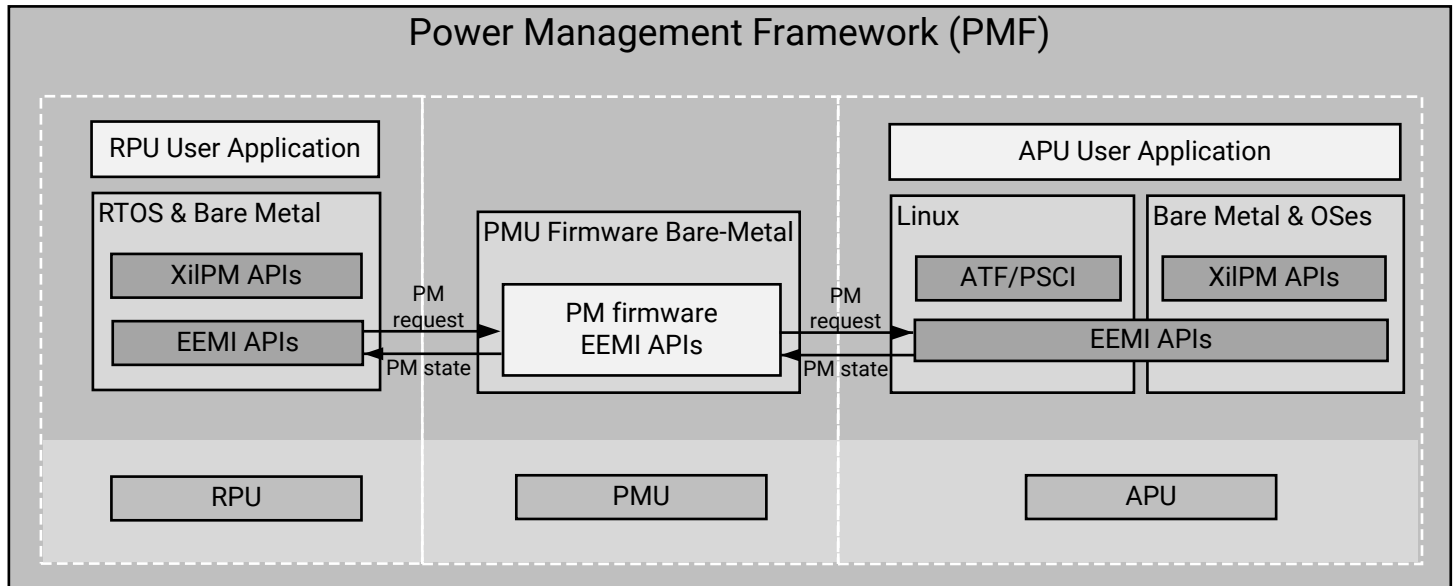
- Provides centralized power state information through use of a Power Management Unit (PMU)

- 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



X19504-100620

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 underlining 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

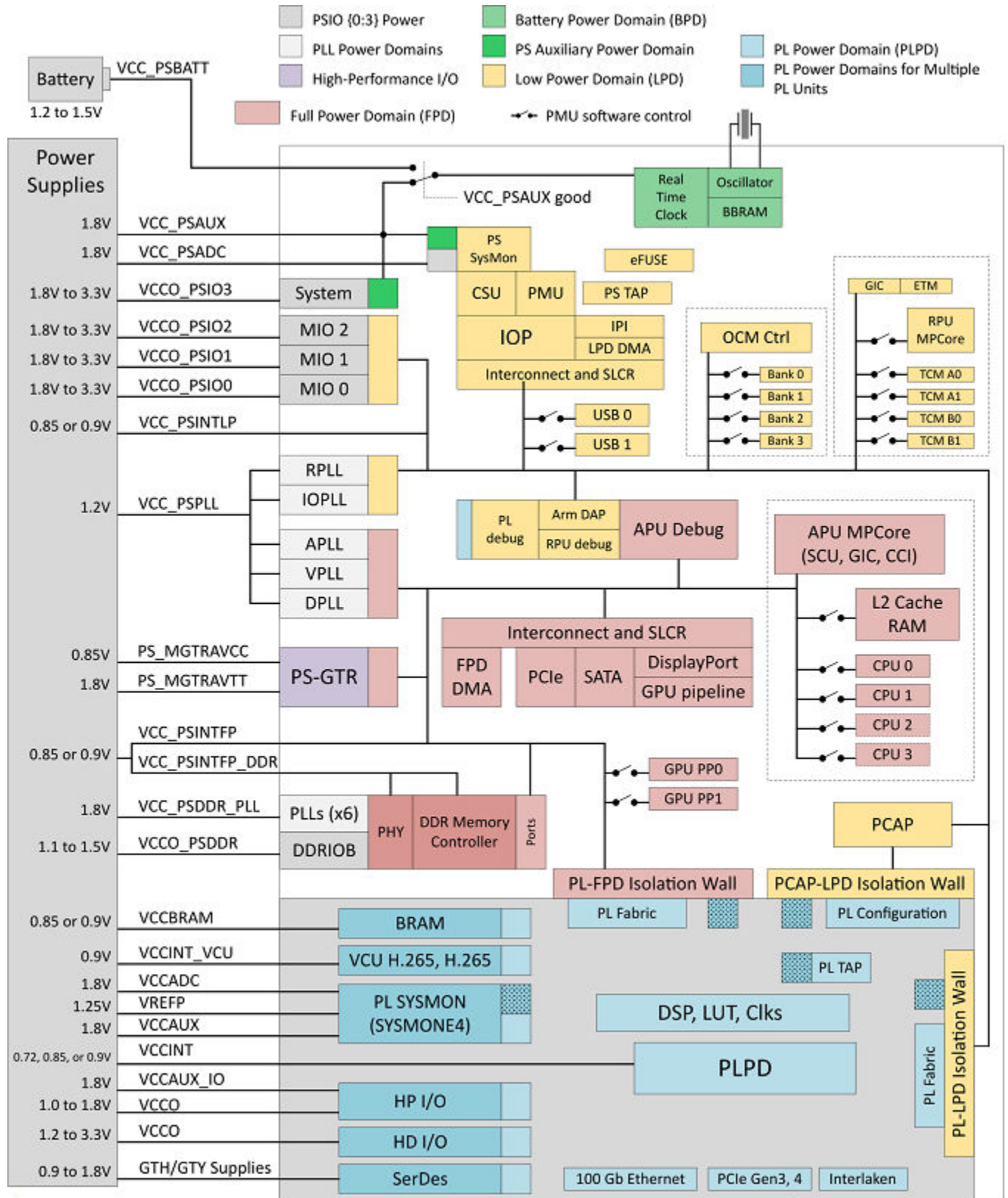
The Zynq UltraScale+ MPSoC is divided into four major power domains:

- 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



X199327-120418

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.*

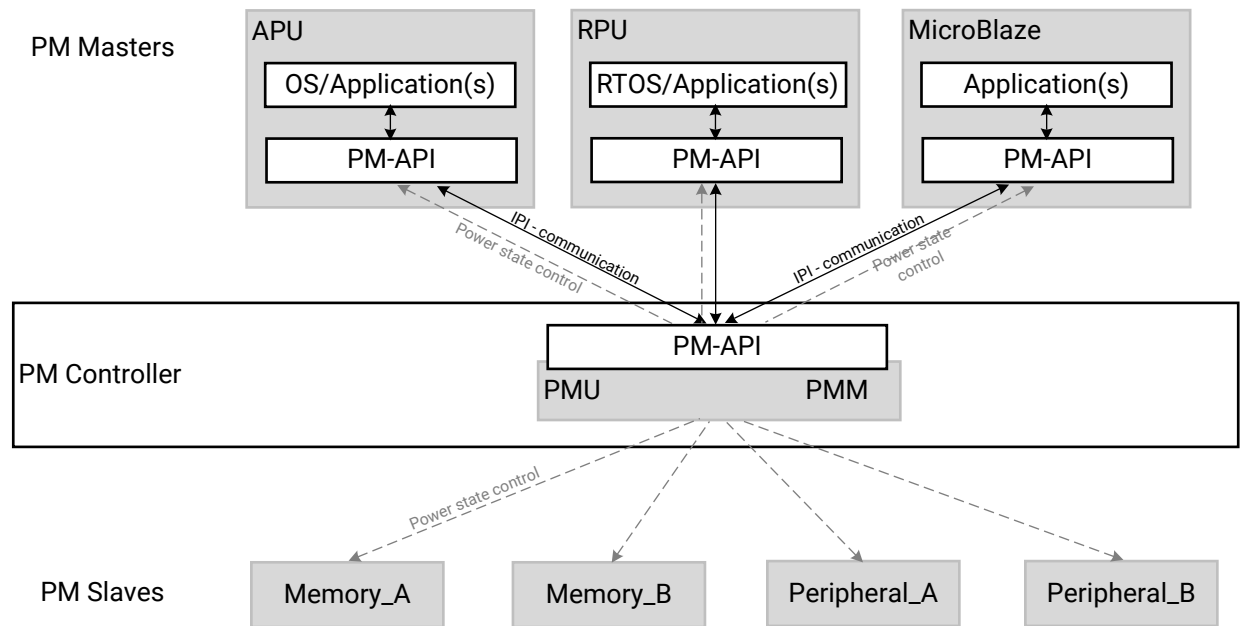
The Zynq UltraScale+ MPSoC also supports inter-processor interrupts (IPIs), which are used as the basis for power-management related communication between the different processors. See this [link](#) to the “Interrupts” chapter of the *Zynq UltraScale+ Device Technical Reference Manual* (UG1085) for 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 *XilPM Library in the OS and Libraries Document Collection* ([UG643](#)).

Power Management Framework Layers

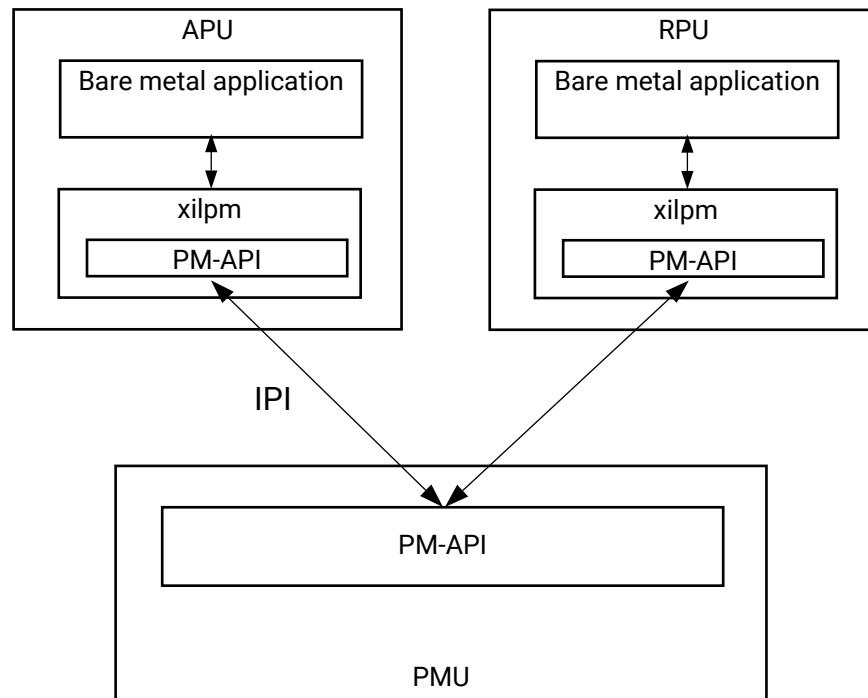
There are different API layers in the power management framework (PMF) implementation for Zynq UltraScale+ MPSoCs, which are, as follows:

- **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

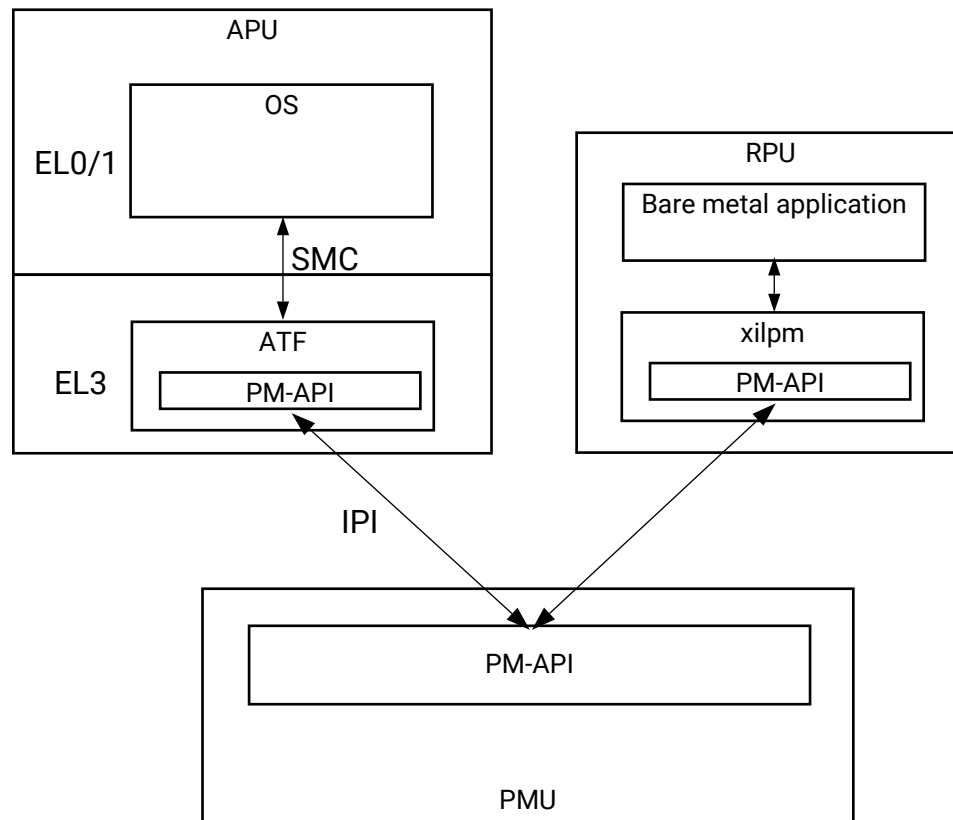


X19094-071317

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 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 Setti... | 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 | 0xFE2FFFFF | LPD |
| Coresight | 0xFE800000 | 8192 | KB | NonSecure | Read/Write | 0xFEFFFFFF | 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_IOW | 0xFFA20000 | 64 | KB | NonSecure | Read/Write | 0xFFA2FFFF | LPD |
| IOW_GPV | 0xFE000000 | 1024 | KB | NonSecure | Read/Write | 0xFE0FFFFF | LPD |
| IPI_CTRL | 0xFF380000 | 512 | KB | NonSecure | Read/Write | 0xFF3FFFFF | LPD |
| LPD_GPV | 0xFE100000 | 1024 | KB | NonSecure | Read/Write | 0xFE1FFFFF | LPD |
| RTC | 0xFFA60000 | 64 | KB | NonSecure | Read/Write | 0xFFA6FFFF | LPD |

Figure 57: PCW Configuration Contd

| | | | | | | | | | |
|------------------------------|------------|------|----|--|-----------|------------|------------|-----|--|
| APU_secure | | | | | | | | | |
| Masters | | | | | | | | | |
| SD1 | | | | | Secure | | | | |
| APU | | | | | | | | | |
| Slaves | | | | | | | | | |
| Memory | | | | | | | | | |
| OCM | 0xFFFC0000 | 256 | KB | | Secure | Read/Write | 0xFFFFFFFF | OCM | |
| 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 | |
| PMU Firmware | | | | | | | | | |
| Masters | | | | | | | | | |
| PMU | | | | | | | | | |
| Slaves | | | | | | | | | |
| Peripherals | | | | | | | | | |
| UART0 | 0xFF000000 | 64 | KB | | NonSecure | Read/Write | 0xFF00FFFF | LPD | |
| 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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

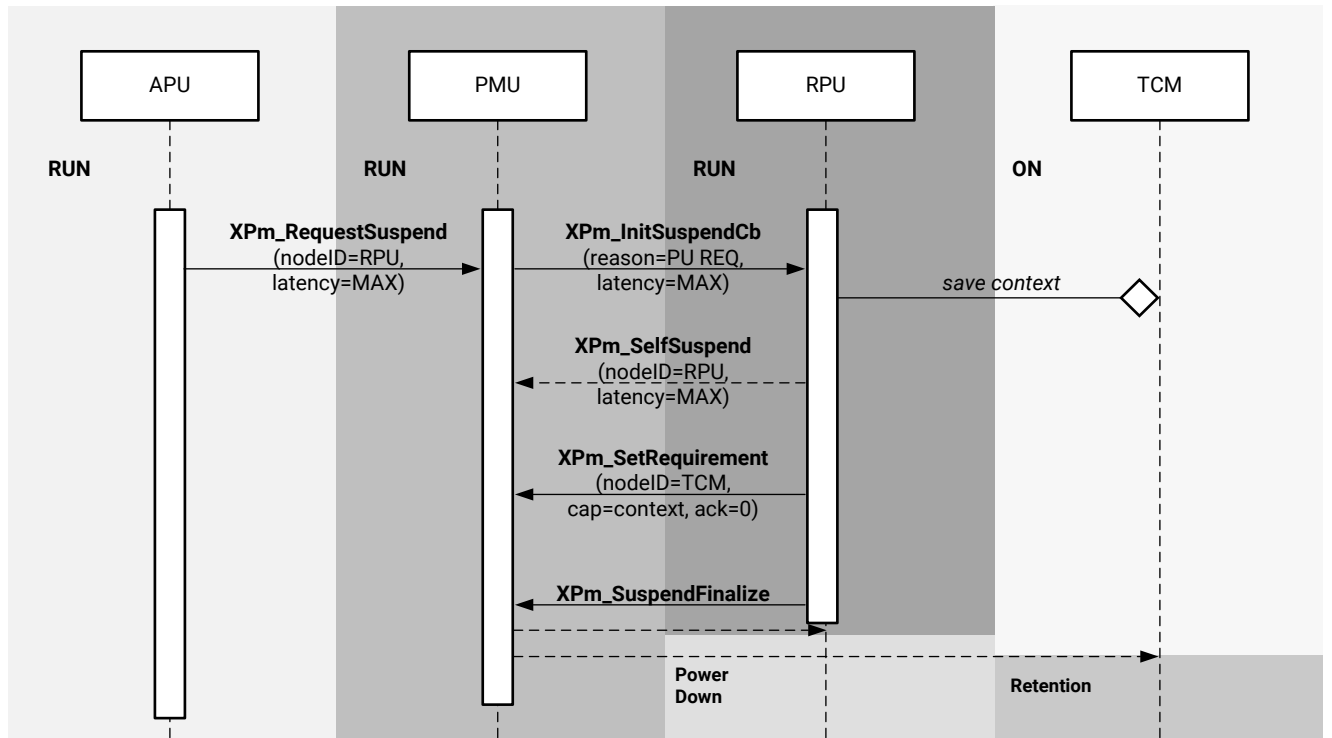
This causes the power management controller to call `XPm_InitSuspendCb()`, which is a callback function implemented in the target PU. The target PU then initiates its own suspend procedure, or call `XPm_AbortSuspend` and specify the abort reason. For example, you can request an APU to suspend with the 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 *XiIPM Library in the OS and Libraries Document Collection* ([UG643](#)).

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 *XiIPM Library in the OS and Libraries Document Collection* ([UG643](#)).

DDR Self-refresh over Warm Restart

In most systems, the RAM of a computing system is cleared when the system resets or powers down. Any data that needs to be retained, such as settings and logs, are usually stored in a non-volatile memory such as flash and battery backed-up RAM. These non-volatile memories are slower, especially when the amount of data is huge. For some systems, a more preferred solution is to retain the data in the DRAM, thus effectively using it as a non-volatile memory.

The Zynq UltraScale+ MPSoC software solution supports a feature to put DDR into self-refresh mode during warm restart (system reset, or PS only reset). This makes the DDR a non-volatile memory and its contents remain as it is even after a reset.

By default, this feature is disabled. You can enable this feature by enabling the following build flags during PMUFW and FSBL compilation:

- **PMUFW:** `ENABLE_DDR_SR_WR`
- **FSBL:** `XFSBL_ENABLE_DDR_SR`

After these build flags are enabled, the PMUFW puts the DDR in self-refresh mode during a warm restart (PS only or System restart).

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

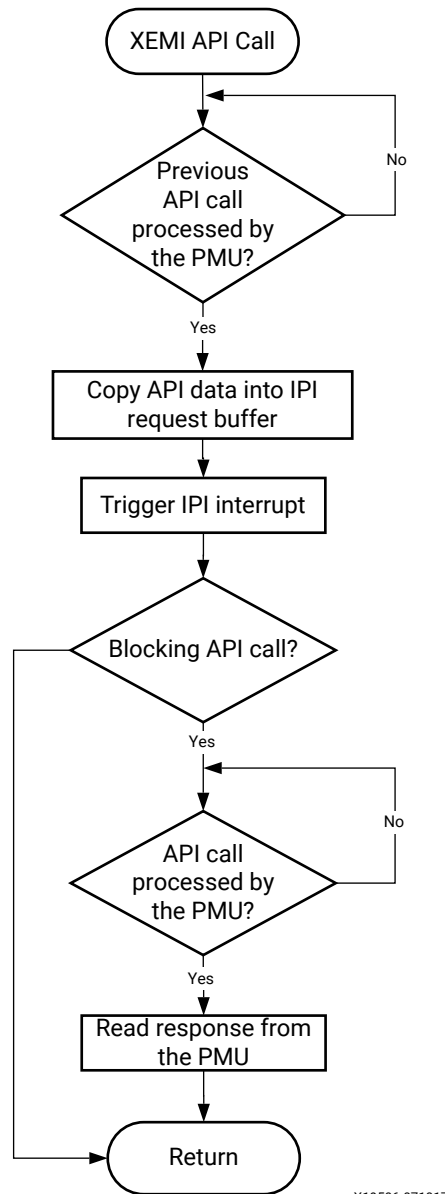
Refer to the XilPM Library in the *OS and Libraries Document Collection* ([UG643](#)) for a list of all API identifiers as well as API argument values.

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_PGGGS | writes value to PGGGS 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 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.

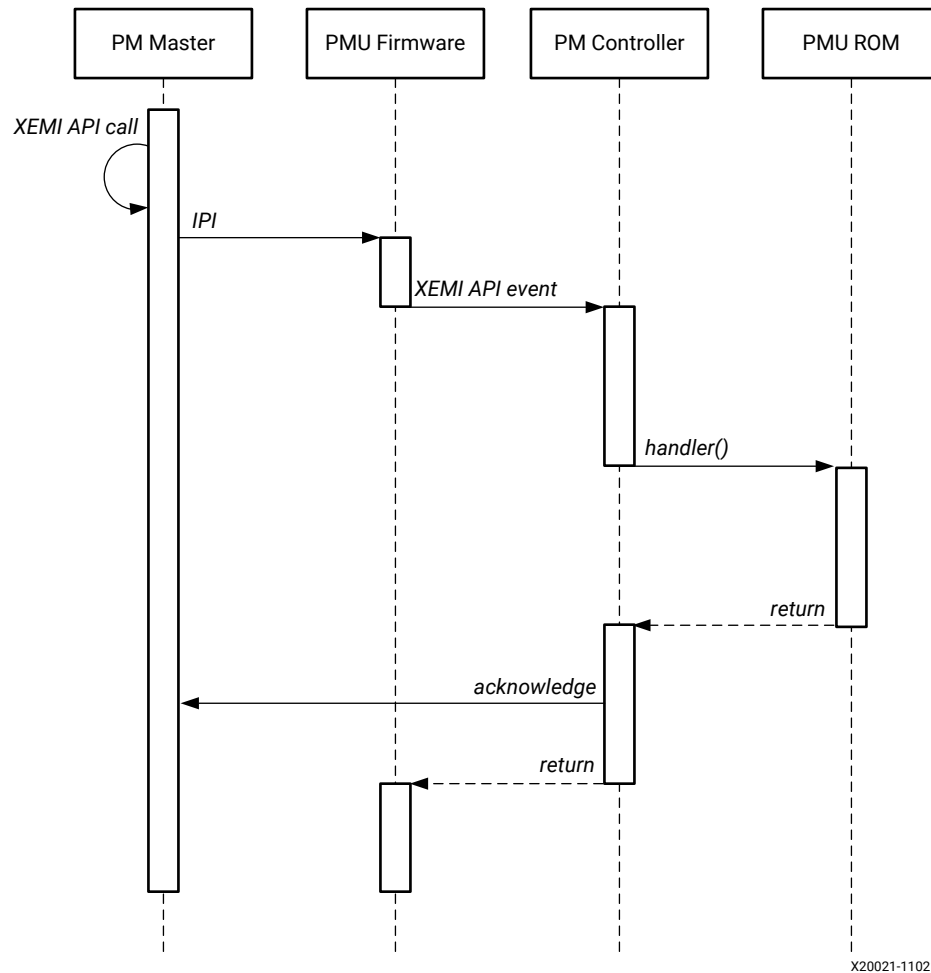
Timer Event

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



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 [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.

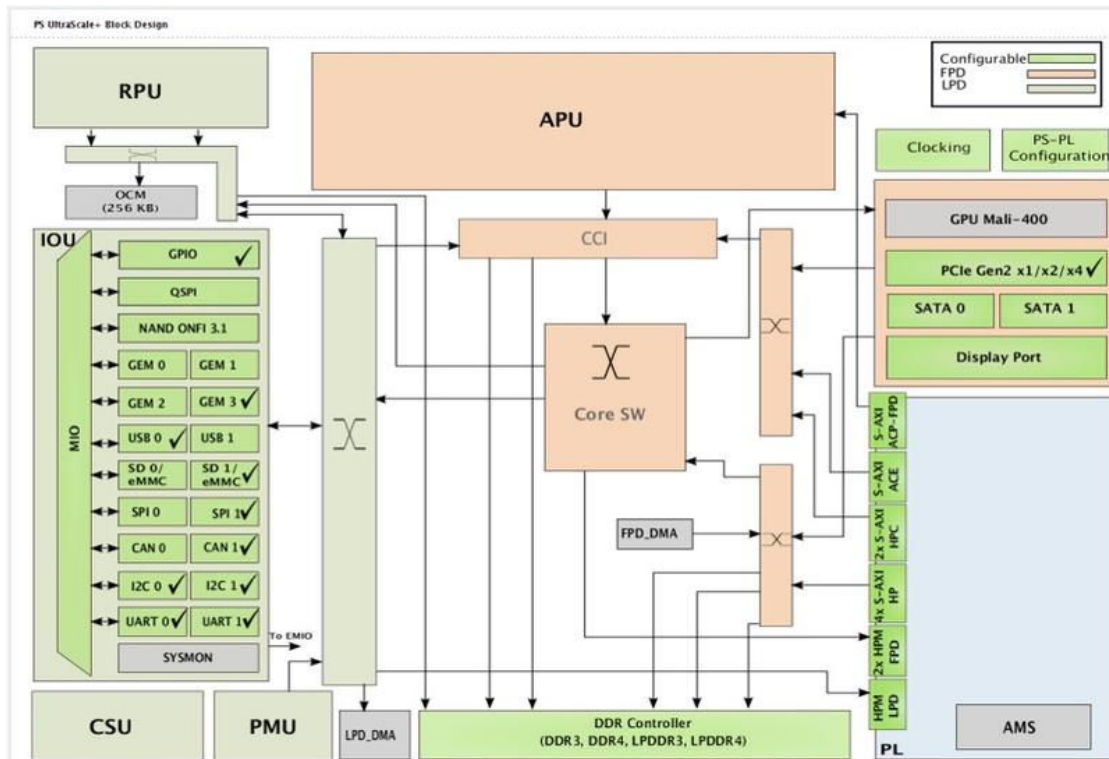
Full system resets are quite straight forward. Hardware is brought back to the reset state and software starts executing ROM code, with a minor behavior difference between the reset types. There are subtleties to PS-only reset which will be discussed in later sections.

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.

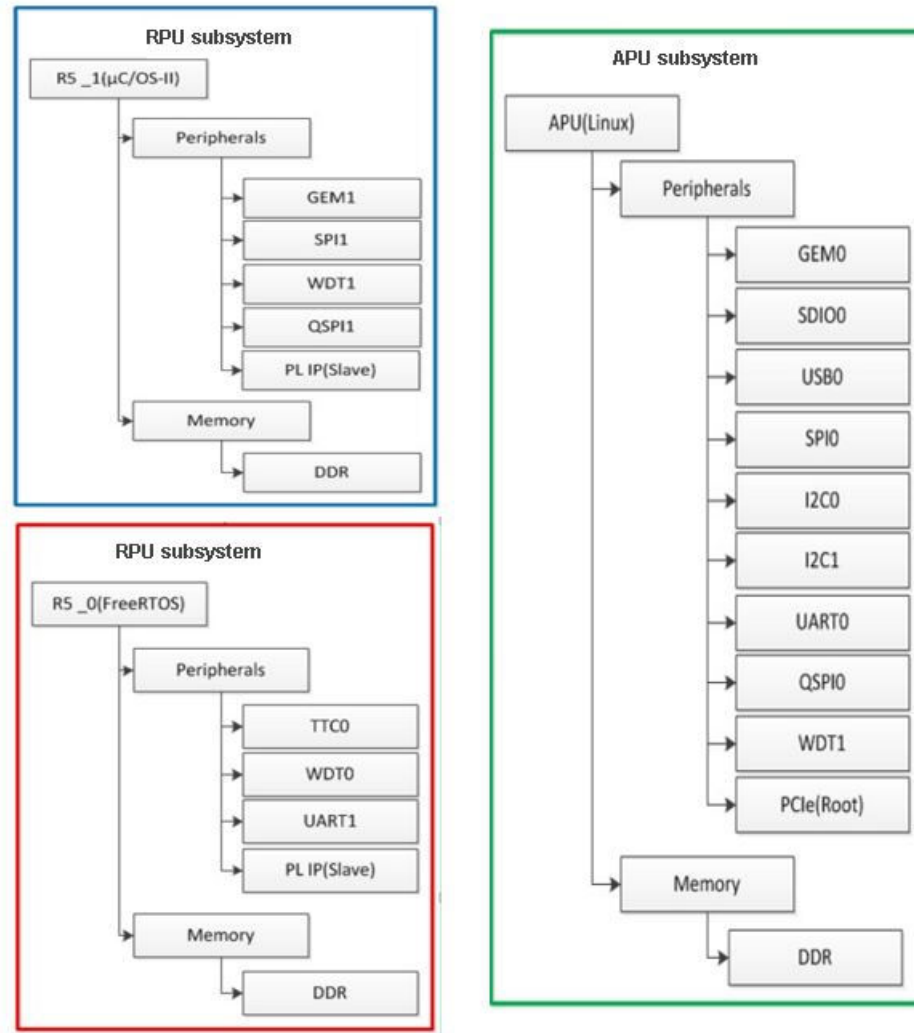
Figure 61: Vivado IP Configuration Menu



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

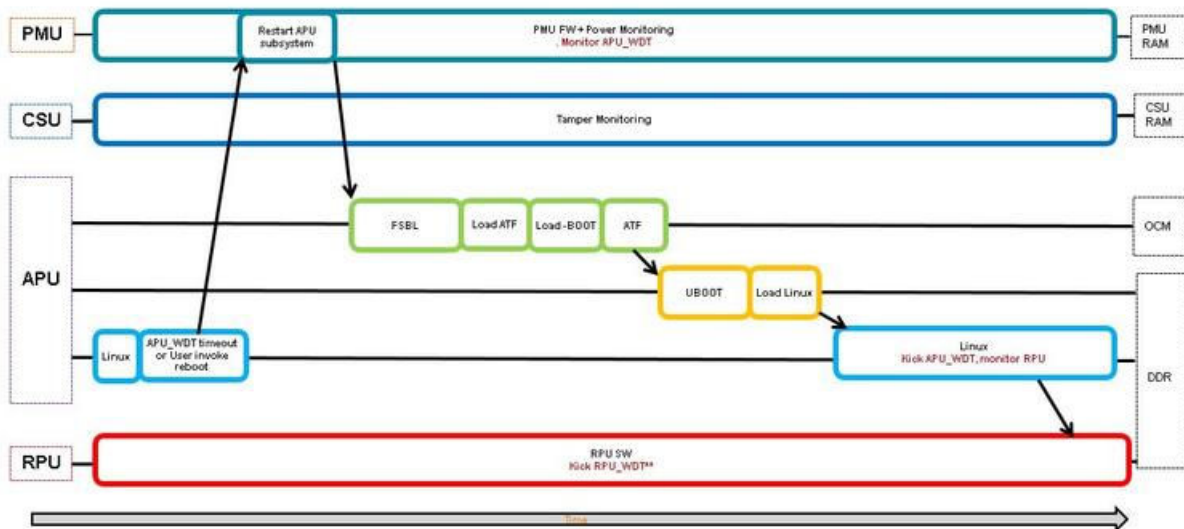
APU Subsystem Restart

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

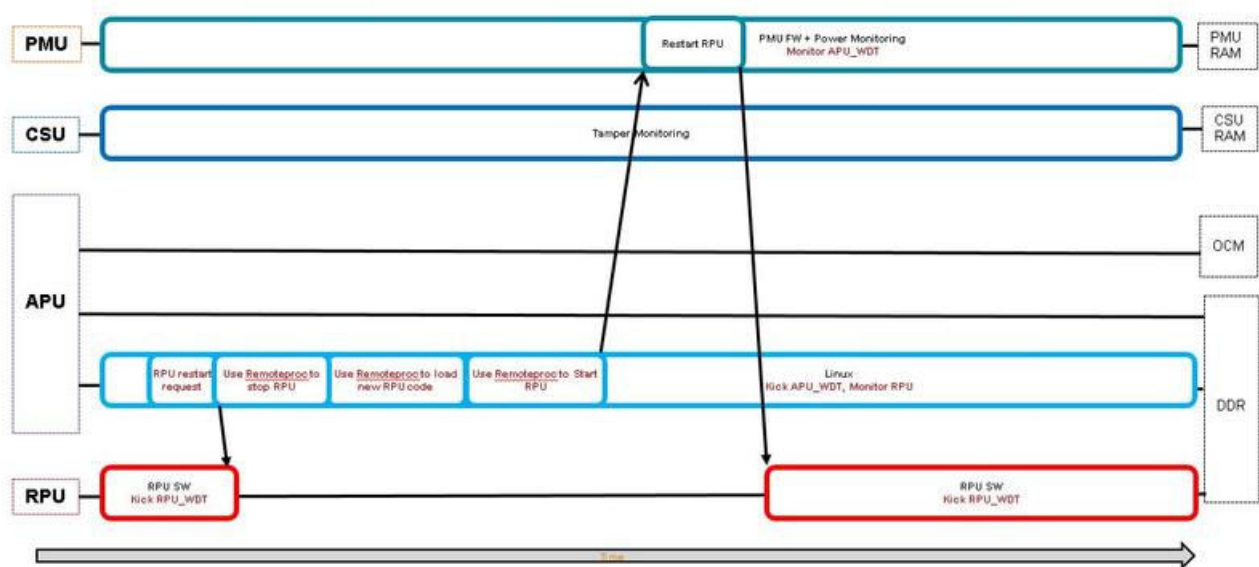
RPU subsystem restart requires the APU subsystem and one or more RPU subsystems running in lock-step or split mode. The APU subsystem running Linux is the master of the RPU subsystems and manages the life cycle of the subsystem using the remoteproc feature of OpenAMP. APU uses remoteproc to load, start, 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:

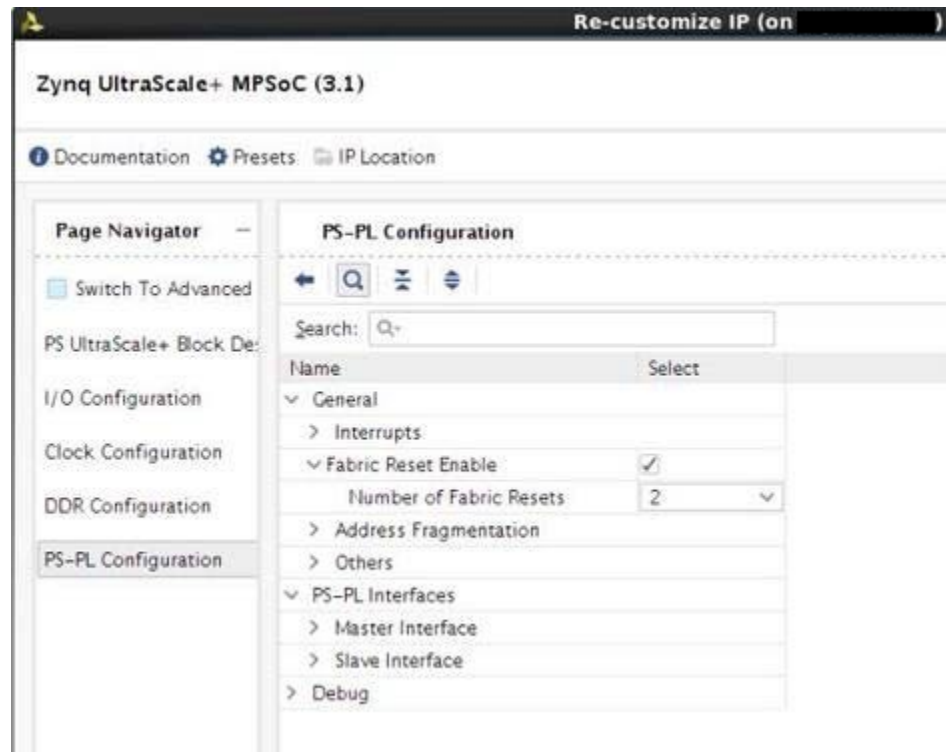
- Write AXI registers of PL IPs to bring them to idle state and reset. This is the preferred and a graceful way to idle PL peripherals.
- Implement a signal based handshake where PMU firmware signals PL to idle all PL IPs. This implementation should be used when there is no direct control to gracefully 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 `p1_resetn0` and `p1_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
```

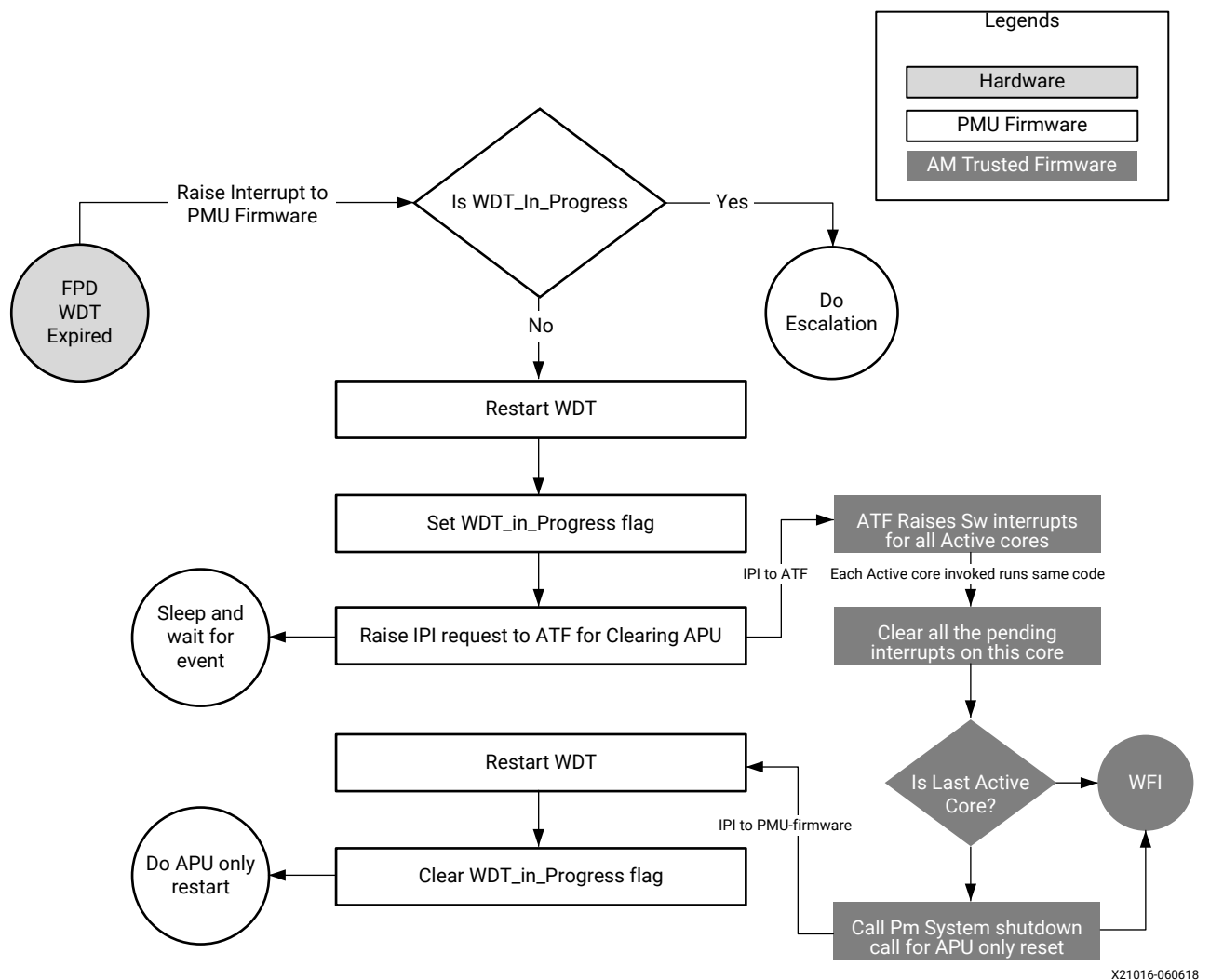
Default Scheme

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: Flow of Control for Default Escalation Scheme



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):

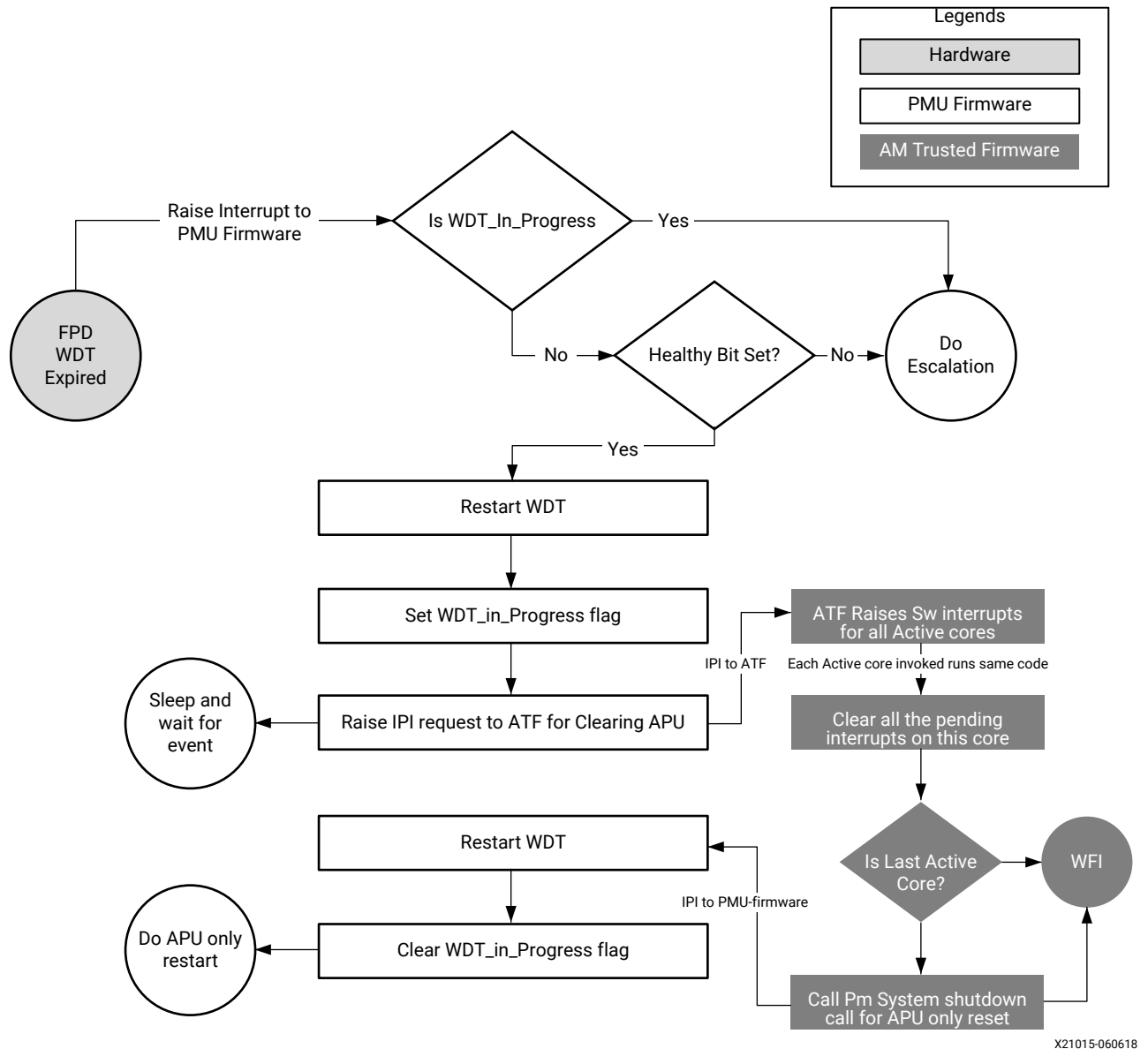
```
# echo "0x20000000 0x20000000" > "/sys/devices/platform/firmware/ggs0"
```

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 `xpfw_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 `xpfw_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 |
|--------------|-------------------------------------|--|
| 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_pmu_fw`. 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.

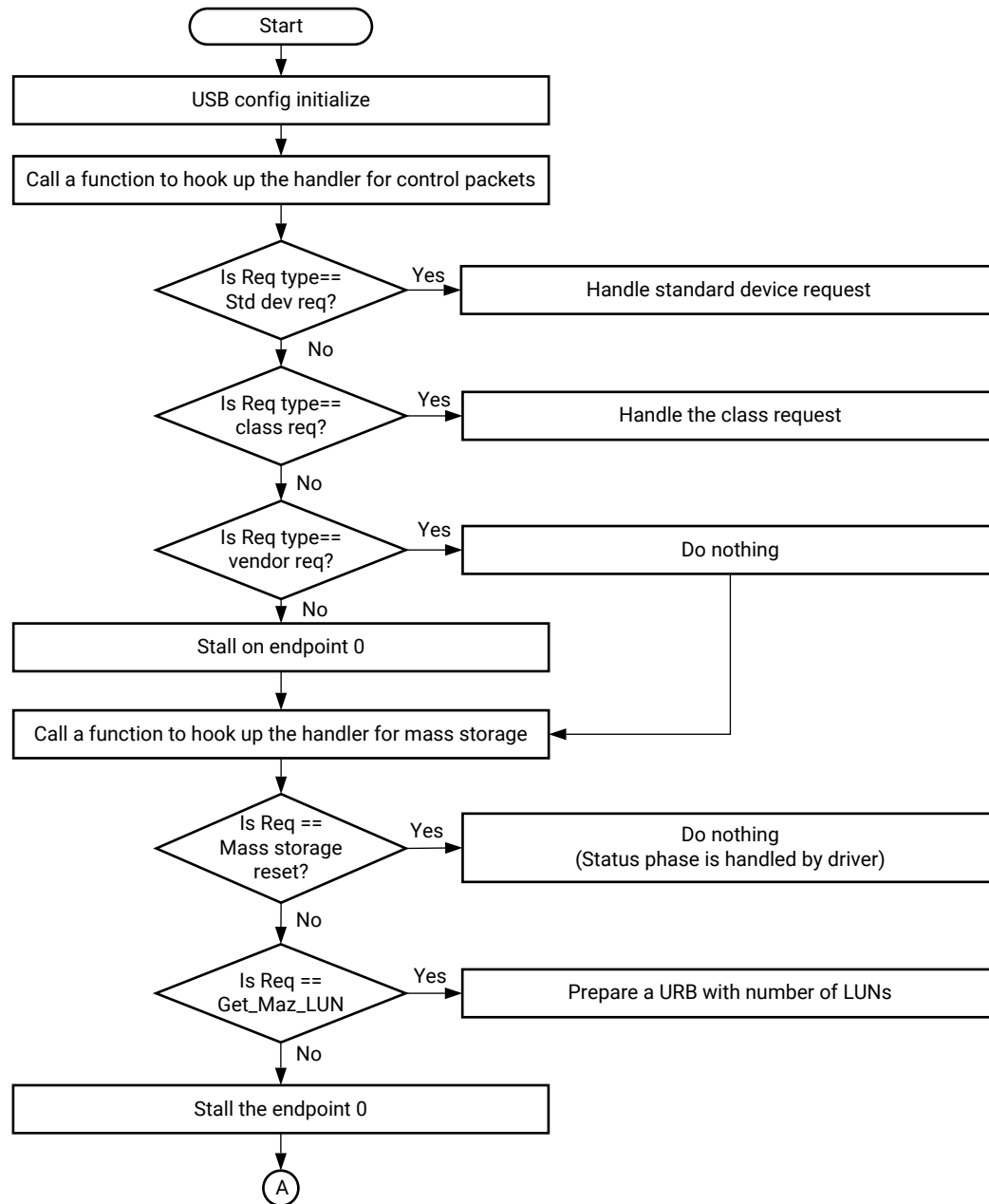
USB 3.0

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.

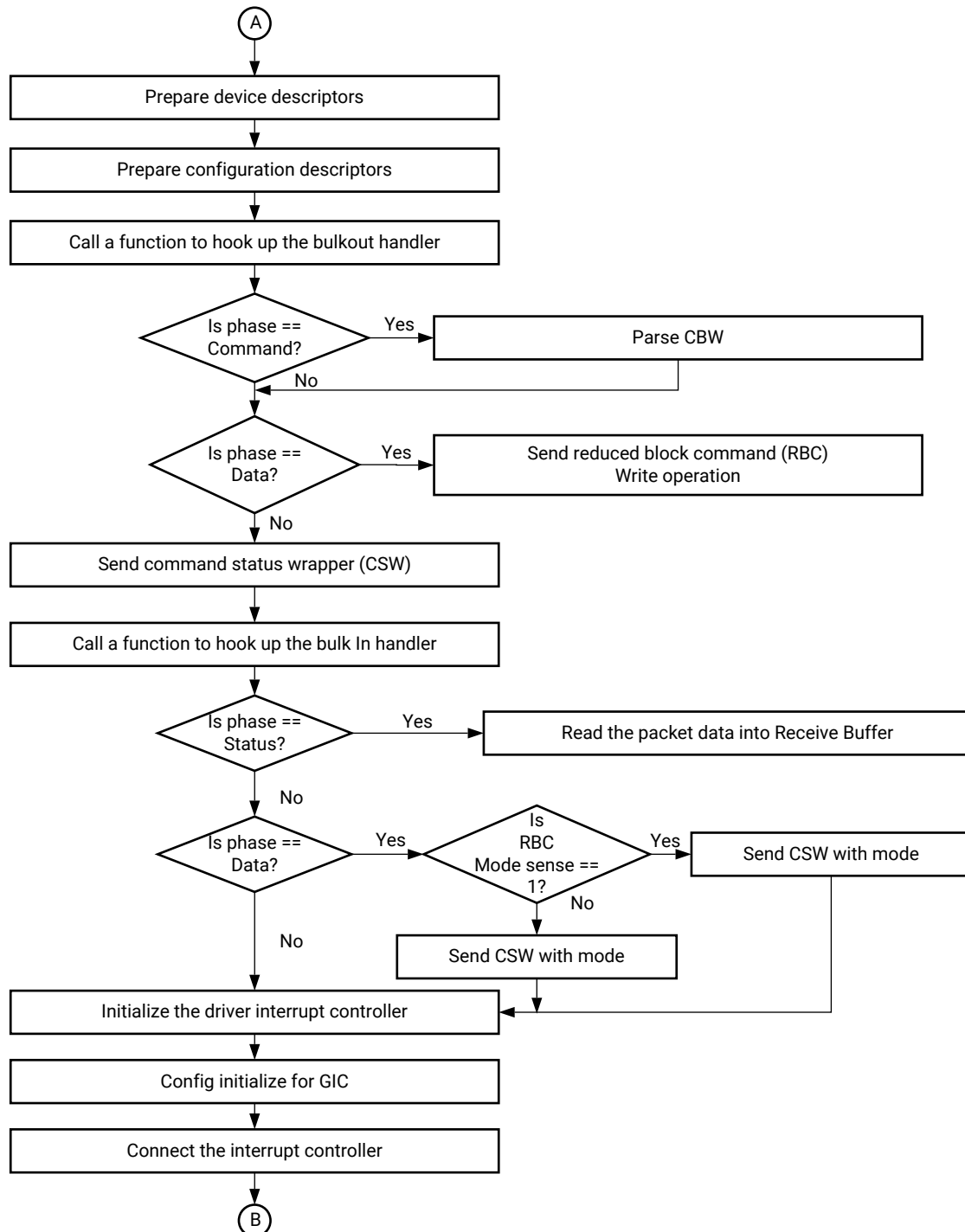
The following flow diagrams illustrate how to configure USB as mass storage device.

Figure 69: USB Example Flow: USB Initialization



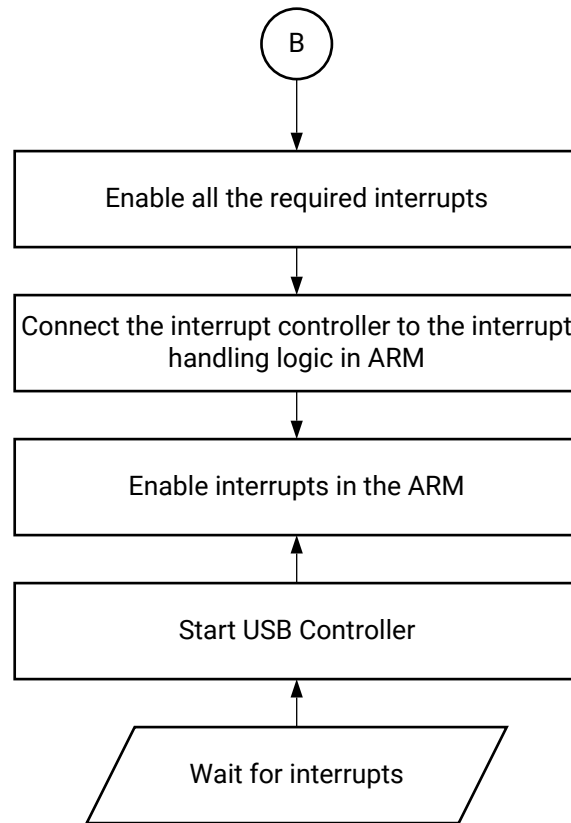
X15463-111020

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



X15478-021317

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).

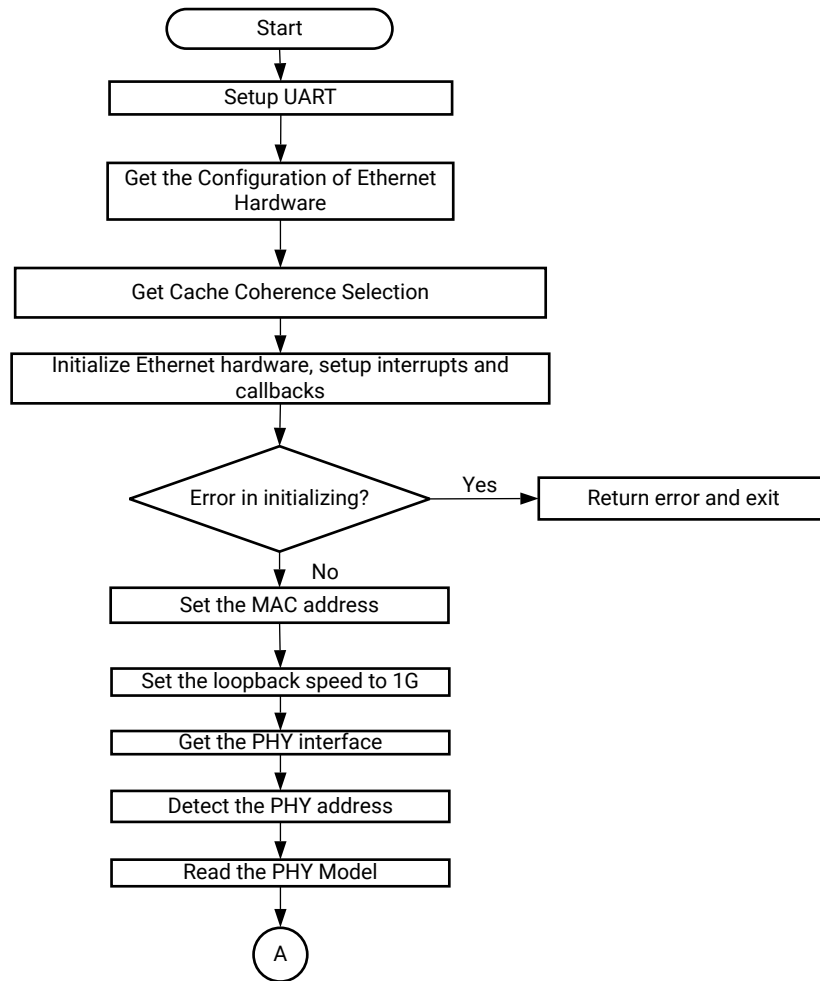
Gigabit Ethernet Controller

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.

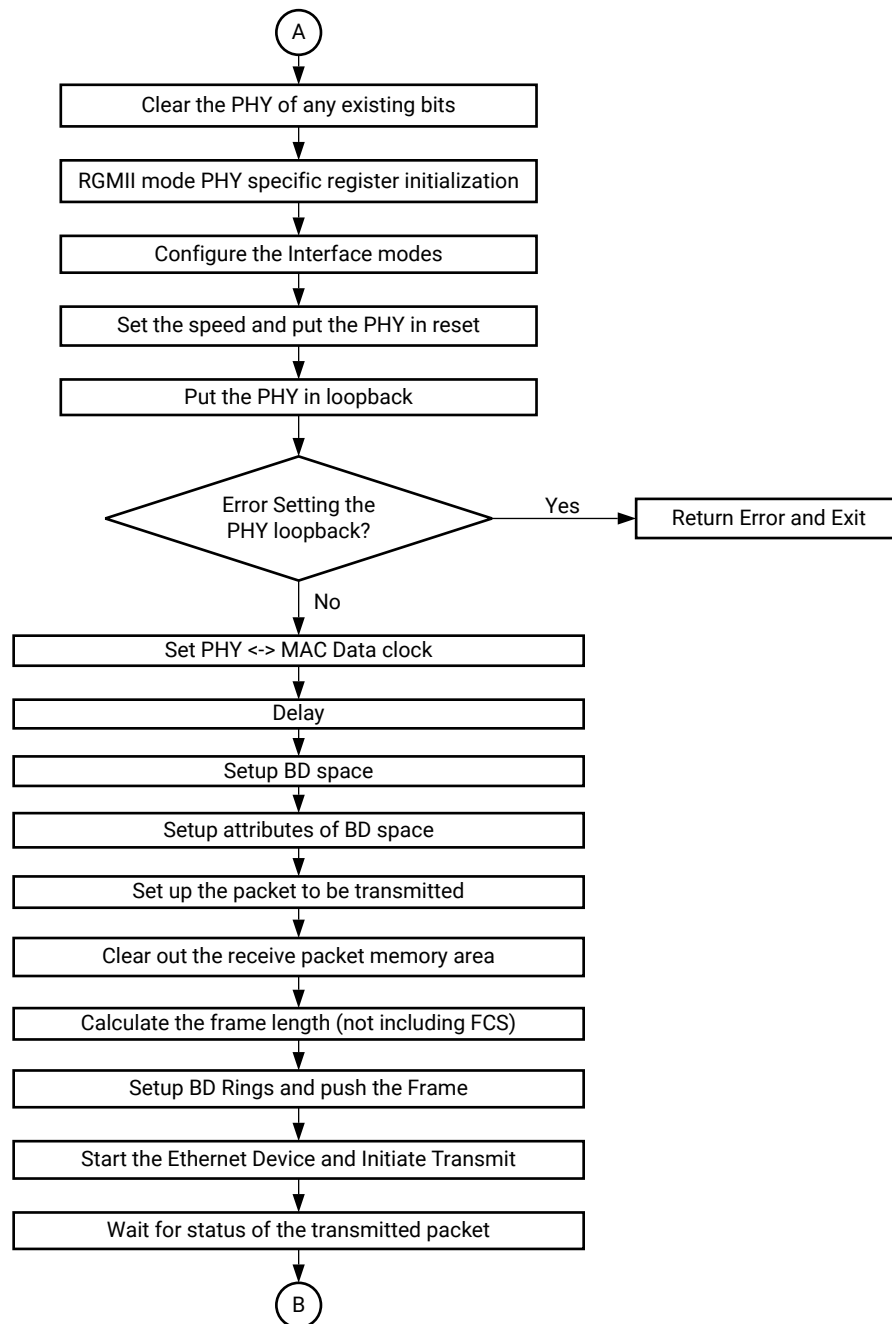
The following figures illustrate an example for configuring an Ethernet controller to send a single packet of data in RGMII mode.

Figure 72: **Example Ethernet Flow: Initialize Ethernet Controller**



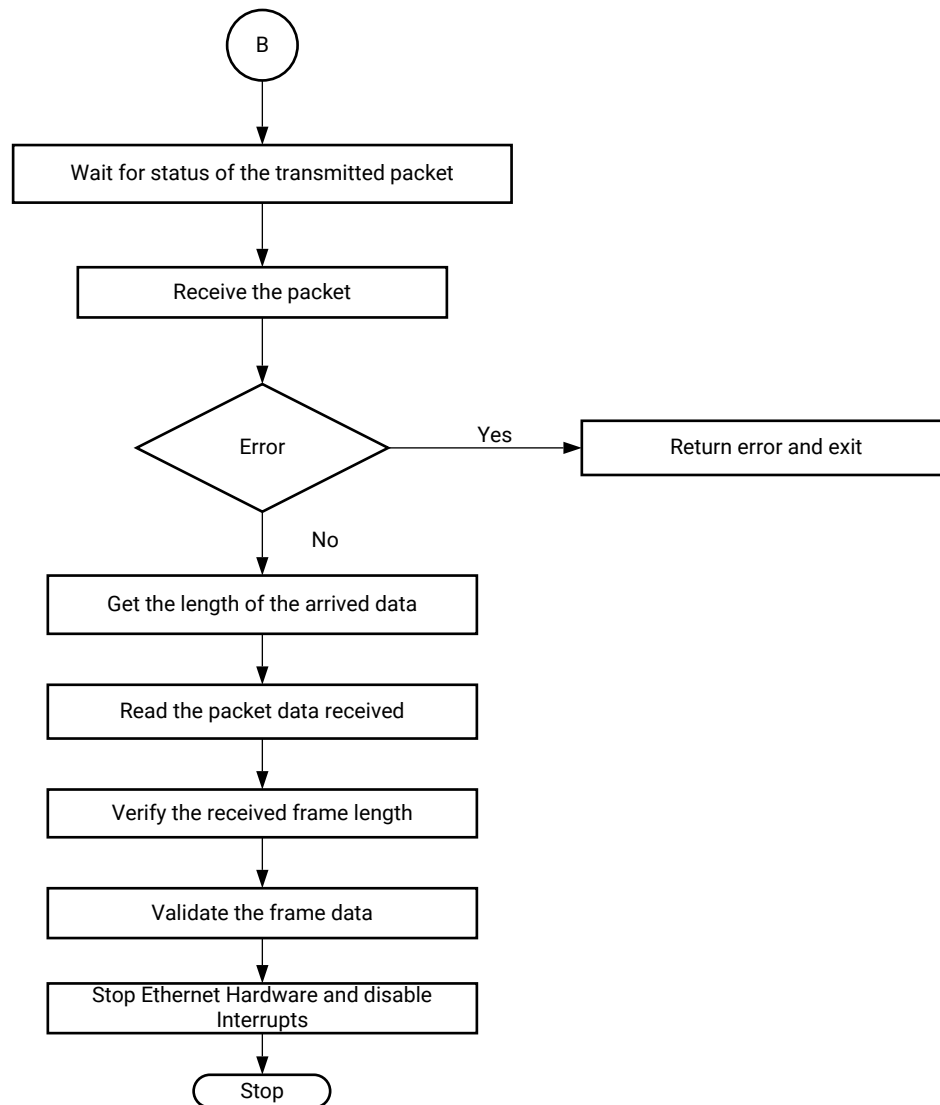
X15462-071017

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

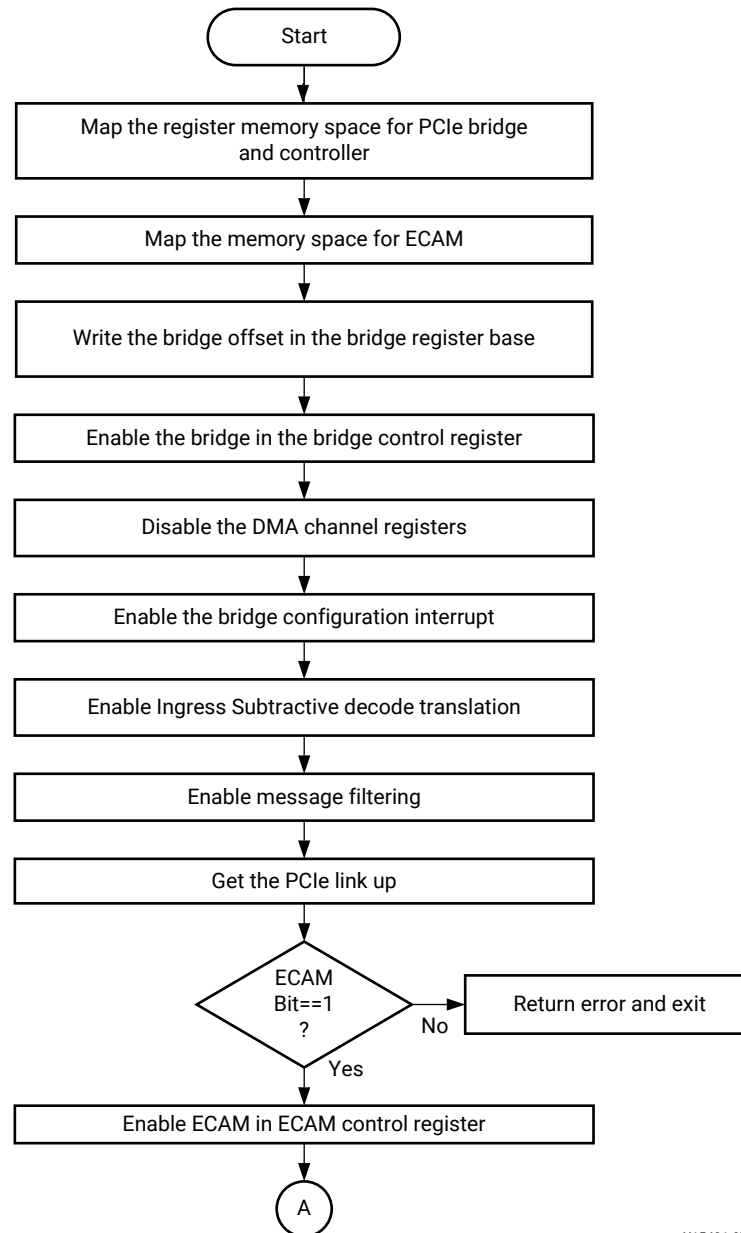
For more information on Ethernet Controller, see this [link](#) to the “Gigabit Ethernet Controller” chapter in the *Zynq UltraScale+ Device Technical Reference Manual* (UG1085).

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.

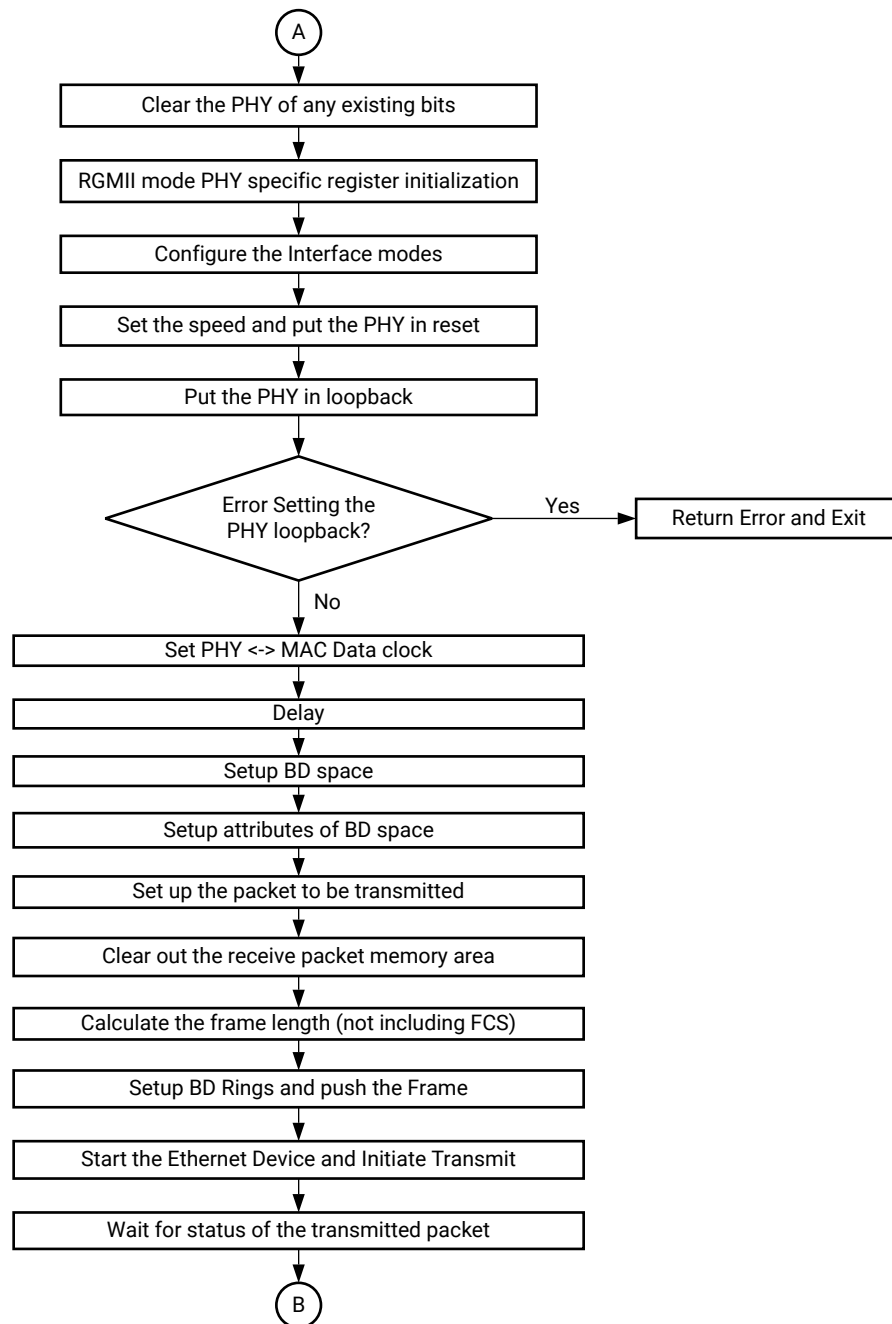
The following flow diagrams illustrate an example for configuring PCIe root complex for a data transfer.

Figure 75: Example PCIe Flow: Enable the Legacy Interrupts and Create PCIe Root Bus



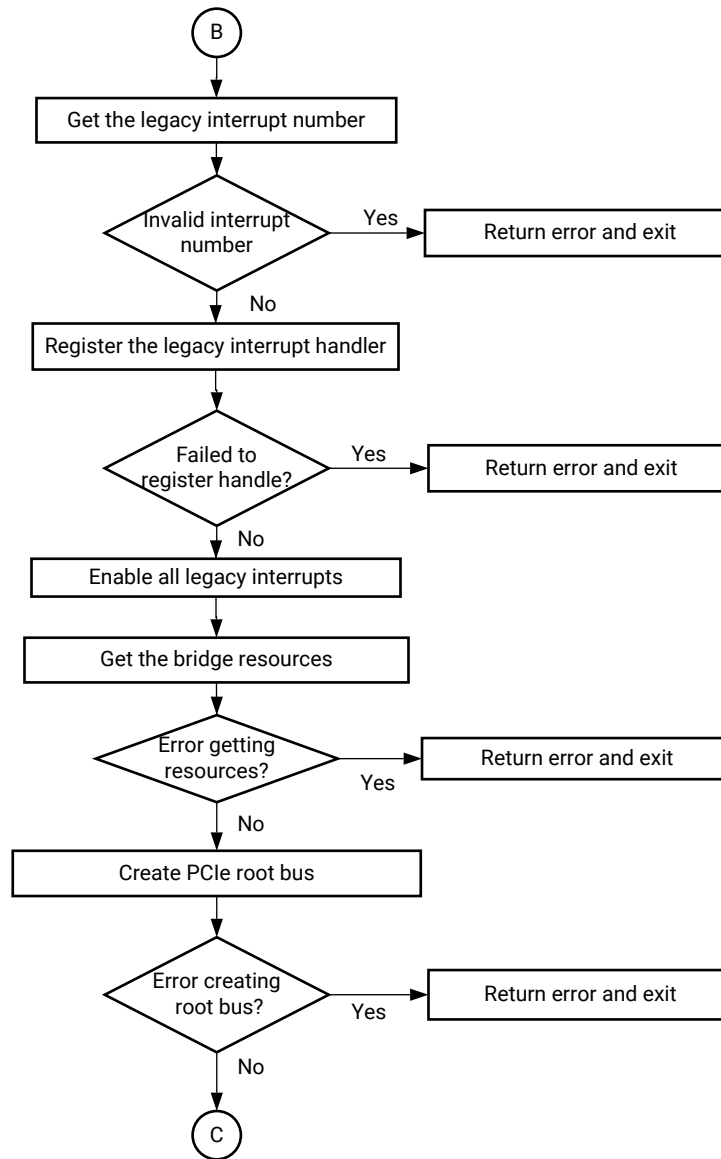
X15481-071217

Figure 76: Example PCIe Flow: Configure the PCIe Parameters and Initialize the Transmit



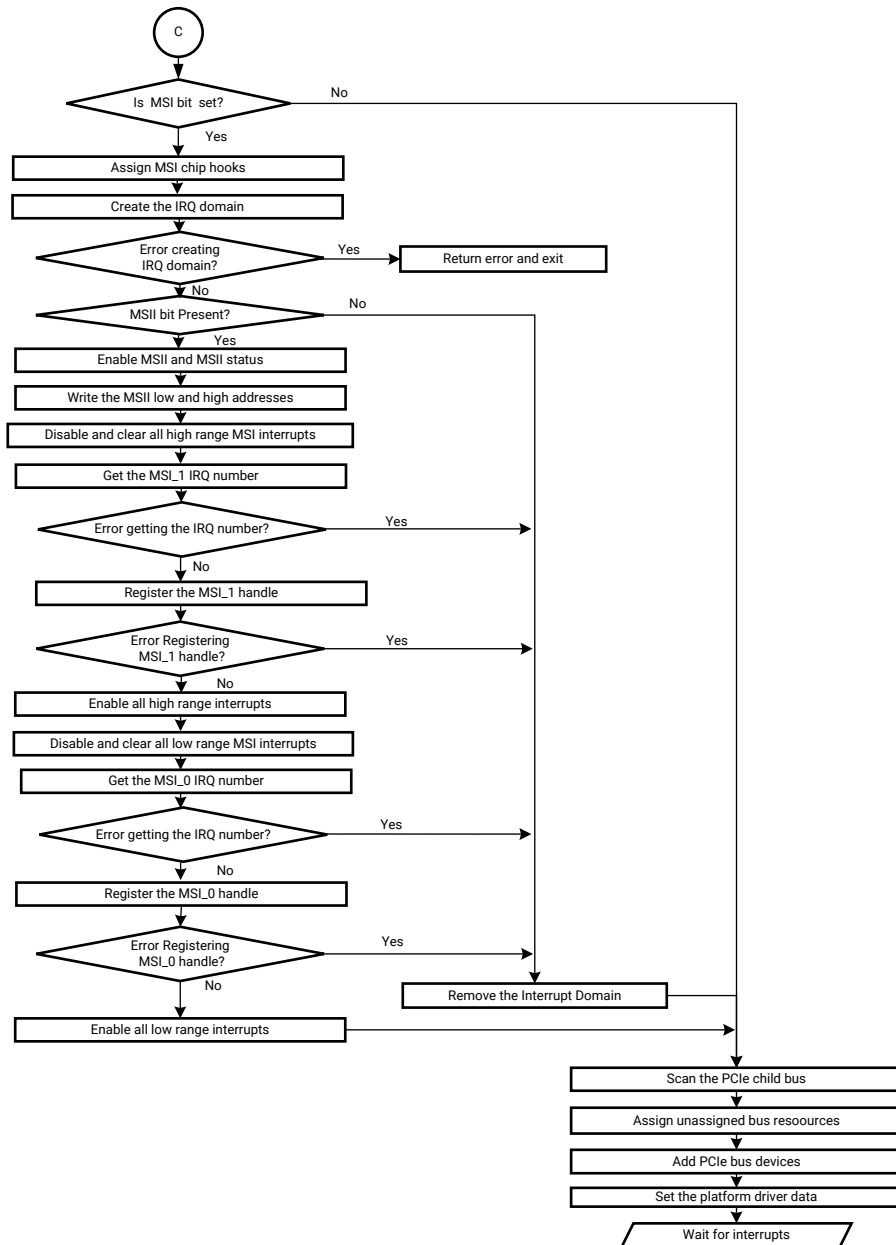
X15479-071017

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



X15484-071217

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. |
| XQSIPSU_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](#)).

Libraries

See *OS and Libraries Document Collection* ([UG643](#)) for information on API reference for the following libraries.

- Standalone Library
- LwIP 2.1.1 Library
- XilIFS
- XilFFS
- XilSecure
- XilSkey
- XilPM
- XilFPGA
- XilMailbox
- XilPM

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 2015-2021 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.