

Zynq UltraScale+ MPSoC Software Developer Guide

UG1137 (v4.0) May 3, 2017



Revision History

The following table shows the revision history for this document.

Date	Version	Revision
05/03/2017	v4.0	<ul style="list-style-type: none"> • In Chapter 2: <ul style="list-style-type: none"> ◦ Added APU and RPU Executable Memory Regions. • In Chapter 4: <ul style="list-style-type: none"> ◦ Updated Figure 4-2. ◦ Added information about Linux software stack exception levels EL0-EL3. • In Chapter 5: <ul style="list-style-type: none"> ◦ Updated Figure 5-1. • In Chapter 7: <ul style="list-style-type: none"> ◦ Moved Boot Flow here from Chapter 2. ◦ Added QSPI32 Boot Mode and eMMC18 Boot Mode. ◦ Added more information to JTAG Boot Mode. ◦ Added USB Boot Mode. ◦ Updated Figure 7-8. ◦ Added FSBL_USB_EXCLUDE to Table 7-3. • In Chapter 8: <ul style="list-style-type: none"> ◦ Removed figure showing flow diagram for secured booting. ◦ Removed section on library support; this is now covered in Appendix H, XilSKey Library v6.2. ◦ Added examples to Encryption. ◦ Added more information to Authentication. ◦ Added Bitstream Authentication Using External Memory. ◦ Added System Memory Management Unit. ◦ Added A53 Memory Management Unit. ◦ Added R5 Memory Protection Unit. • Added Chapter 9, Introduction to the Power Management Framework. This content was previously in the <i>Power Management Framework User Guide: For Zynq UltraScale+ MPSoC Devices</i> (UG1199). • In Chapter 15: <ul style="list-style-type: none"> ◦ Added parameters and descriptions in Table 15-1. ◦ Added Boot Image Format. ◦ Added additional bit descriptions in Table 15-7. • Added Appendixes for OS & Libraries content (Appendices A-K).

12/15/2016	v3.0	<p>Added content to Introduction in Chapter 1.</p> <p>Corrected text in Boot Modes in Chapter 7.</p> <p>Changed link references to the <i>Zynq UltraScale+ MPSoC Technical Reference Manual</i> (UG1085).</p> <p>Corrected and added links to Appendix L, Additional Resources and Legal Notices.</p>
10/05/2016	v2.0	<p>Chapter 2: Removed JTAG and MDM from Figure 2-2. Clarified Secure and Non-Secure Boot Modes in in Chapter 2. Removed Interrupt Features.</p> <p>Chapter 3: Added Hardware IDE feature list. Added Vivado Design Suite. Modified Supported features in Xilinx Software Development Kit. Added a link to the SDK_Download. Replaced PetaLinux figure with Table 3-3.</p> <p>Chapter 4: Replaced Figure 4-2. Added FreeRTOS Software Stack.</p> <p>Chapter 5: Removed Developing Open Source Software.</p> <p>Chapter 6: Changed the title Chapter 6, Software Design Paradigms. Added Frameworks for Multiprocessor Development section.</p> <p>Chapter 7: Modified SD Mode diagram, Figure 7-5 Modified NAND Mode diagram Figure 7-7. Removed Keys organization in the CSU in Chapter 7, System Boot and Configuration. Removed Wake UP Mechanisms Chapter 7, System Boot and Configuration. Added Pre-Boot Sequence in Chapter 7</p> <p>Chapter 8: Changed the title of Chapter 8, Security Features and reorganized sections. Revised text and renamed Xilinx Peripheral Protection Unit. Revised text in Encryption. Removed Encryption Key Types and Key Registers table. Replaced with a cross-reference to the <i>Zynq UltraScale+ MPSoC Technical Reference Manual</i> (UG1085). Made changes to ARM Trusted Firmware. Removed text from Protecting Memory with XMPU. Removed sections "Protection Checking" and "Error Handling" and "Using Peripheral Protection" from Security Features. Added Library Support.</p> <p>Chapter 9: Added reference to ATF [Ref 37]. Removed text under Wake Up Mechanisms. Added Power Management Framework. Modified Using Custom PMU Firmware.</p> <p>Removed Chapter 12, DMA.</p> <p>Chapter 13: Removed QEMU feature table. Added content to Boards and Kits.</p> <p>Removed Chapter 15, System Coherency.</p> <p>Moved Appendix A to Chapter Chapter 15, Bootgen Image Creation. Removed -interface option from Bootgen Command Options. Removed Virtualization section. Replaced with a reference to the <i>Zynq UltraScale+ MPSoC Technical Reference Manual</i> (UG1085). Fields and Offsets table removed. Replaced with a reference to the <i>Zynq UltraScale+ MPSoC Technical Reference Manual</i> (UG1085). Added that boot access is programmable.</p> <p>Removed several Wiki sites from Appendix L, Additional Resources and Legal Notices</p>
11/18/2015	v1.0	Initial Public Access release.

Table of Contents

Revision History	2
Chapter 1: About This Guide	
Introduction	8
Intended Audience and Scope of this Document	9
Prerequisites	9
Chapter 2: Programming View of Zynq UltraScale+ MPSoC Devices	
Introduction	11
Hardware Architecture Overview.....	12
APU and RPU Executable Memory Regions	14
Boot Process.....	15
Virtualization	17
Reset.....	18
Security Features	18
Safety and Reliability.....	21
Chapter 3: Development Tools	
Introduction	26
Vivado Design Suite	27
Xilinx Software Development Kit	28
PetaLinux Tools	31
Open Source	32
Linux Software Development using Yocto Tools	32
Chapter 4: Software Stack	
Introduction	35
Bare-Metal Software Stack	35
Linux Software Stack	38
Third-Party Software	43
Chapter 5: Software Development Flow	
Overview of Software Development Flow	44

Developing Bare-Metal Applications	45
PetaLinux-Tools-Based Software Development.....	47
Developing a Linux Application Using SDK.....	48

Chapter 6: Software Design Paradigms

Introduction	51
Frameworks for Multiprocessor Development	52
Symmetric Multiprocessing (SMP)	52
Asymmetric Multiprocessing (AMP).....	53

Chapter 7: System Boot and Configuration

Introduction	58
Boot Flow	58
Boot Image Creation	61
Boot Modes	61
Detailed Boot Flow	68
Setting FSBL Compilation Flags.....	70

Chapter 8: Security Features

Introduction	74
Boot Time Security.....	74
Bitstream Authentication Using External Memory	79
Run-Time Security	81
ARM Trusted Firmware	81
Xilinx Memory Protection Unit.....	84
Xilinx Peripheral Protection Unit	85
System Memory Management Unit.....	85
A53 Memory Management Unit.....	85
R5 Memory Protection Unit	86

Chapter 9: Introduction to the Power Management Framework

Introduction	87
Zynq UltraScale+ MPSoC Power Management Overview.....	89
Zynq UltraScale+ MPSoC Power Management Software Architecture	91
Using the API for Power Management.....	99
XIPM Implementation Details	106
Linux	108
ARM Trusted Firmware (ATF)	118
PMU Firmware	121

Chapter 10: Platform Management

Introduction	124
Platform Management in PS	124

Chapter 11: Reset

Introduction	130
System-Level Reset	130
Block-Level Resets	130
APU Reset.....	131
RPU Reset.....	131
FPD Reset	132

Chapter 12: High-Speed Bus Interfaces

Introduction	133
USB 3.0	133
Gigabit Ethernet Controller.....	136
PCI Express	140

Chapter 13: Clock and Frequency Management

Introduction	145
Changing the Peripheral Frequency	145

Chapter 14: Target Development Platforms

Introduction	147
QEMU	147
Boards and Kits	147

Chapter 15: Bootgen Image Creation

Introduction	148
BIF File Parameters	148
Bootgen Command Line Options	152
Bootgen Command Example.....	153
Boot Image Format	153
Boot Header Table	154
Register Initialization Table.....	154
Image Header Table	155
Partition Header Tables.....	156
Authentication Certificate.....	158
Authentication Certificate Header	158

Appendix A: Power Management Framework Appendix

XilPM Argument Value Definitions.....	160
XilPM Error Codes	167

Appendix B: Xilinx Standard C Libraries

Xilinx Standard C Libraries.....	169
----------------------------------	-----

Appendix C: Standalone Library Reference v6.2

Xilinx Hardware Abstraction Layer API.....	175
--	-----

Appendix D: XilFlash Library v4.3

Overview	287
----------------	-----

Appendix E: XilIISf Library v5.8

Overview	298
----------------	-----

Appendix F: XilFFS Library Reference 3.6

Overview	318
----------------	-----

Appendix G: XilRSA Library v1.3

Overview	322
----------------	-----

Appendix H: XilSKey Library v6.2

Overview	327
----------------	-----

Appendix I: XilPM Library v2.1

Overview	396
----------------	-----

Appendix J: XilFPGA Library v2.0

Overview	416
----------------	-----

Appendix K: XilSecure Library Reference

Overview	424
----------------	-----

Appendix L: Additional Resources and Legal Notices

Xilinx Resources	448
Solution Centers.....	448
Documentation Navigator and Design Hubs	448
References	449
Please Read: Important Legal Notices	451

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+™ MPSoC devices. 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 1.5GHz
- Real-time processing unit (RPU):
 - Dual-core ARM Cortex-R5 MPCore
 - CPU frequency up to 600MHz
- Graphics processing unit (GPU):
 - ARM Mali™-400 MP2
 - GPU frequency up to 667MHz
- 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 [\[Ref 5\]](#) and the Product Advantages [\[Ref 6\]](#).

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, and frameworks and example applications
 - Platform management unit firmware (PMUFW), 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 Xilinx software developers kit (SDK), 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. It is recommended that all the users go through this chapter completely to understand each feature.
- [Chapter 3, Development Tools](#): Provides a description of the various software stacks: 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 4, Software Stack](#): Provides a brief description about the Xilinx software development tools, helping you understand all the available features in the software development tools. It is recommended for software developers to go through this chapter to understand the procedure involved in building and debugging software applications.
- [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 ensure security during application boot- and run-time.
- [Chapter 10, Platform Management](#): Describes the features available to manage power consumption, and how to control the various power modes using software.
- [Chapter 11, Reset](#): Explains the system and module-level resets.
- [Chapter 12, High-Speed Bus Interfaces](#): Explains the configuration flow of the high-speed interface protocols.
- [Chapter 13, Clock and Frequency Management](#): Briefly explains the clock and frequency management of peripherals in Zynq UltraScale+ MPSoC devices.
- [Chapter 14, 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 15, Bootgen Image Creation](#): Describes Bootgen, a standalone tool for creating a bootable image for Zynq UltraScale+ MPSoC devices. Bootgen is included in the SDK.
- Appendixes A-K describe the available libraries and board support packages to help you develop a software platform
- [Appendix L, Additional Resources and Legal Notices](#): Provides links to additional information that is cited throughout the document.

Programming View of Zynq UltraScale+ MPSoC Devices

Introduction

The Zynq® UltraScale+™ MPSoC device supports a wide range of applications that require heterogenous multiprocessing. 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 for a heterogeneous processing mix
 - 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 device, and various software modules such as operating systems, heterogeneous system softwares and security management modules.

The Zynq UltraScale+ MPSoC device is a heterogeneous device that includes the ARM® v8-based Cortex™-A53, high-performance, energy-efficient, 64-bit application processor, and also the 32-bit ARM Cortex-R5 MPCore real-time processor.

Hardware Architecture Overview

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

The next-generation programmable engines, security, safety, reliability, and scalability from 32 to 64 bits are illustrated in the Zynq UltraScale+ MPSoC architectural block diagram, as shown in [Figure 2-1](#).

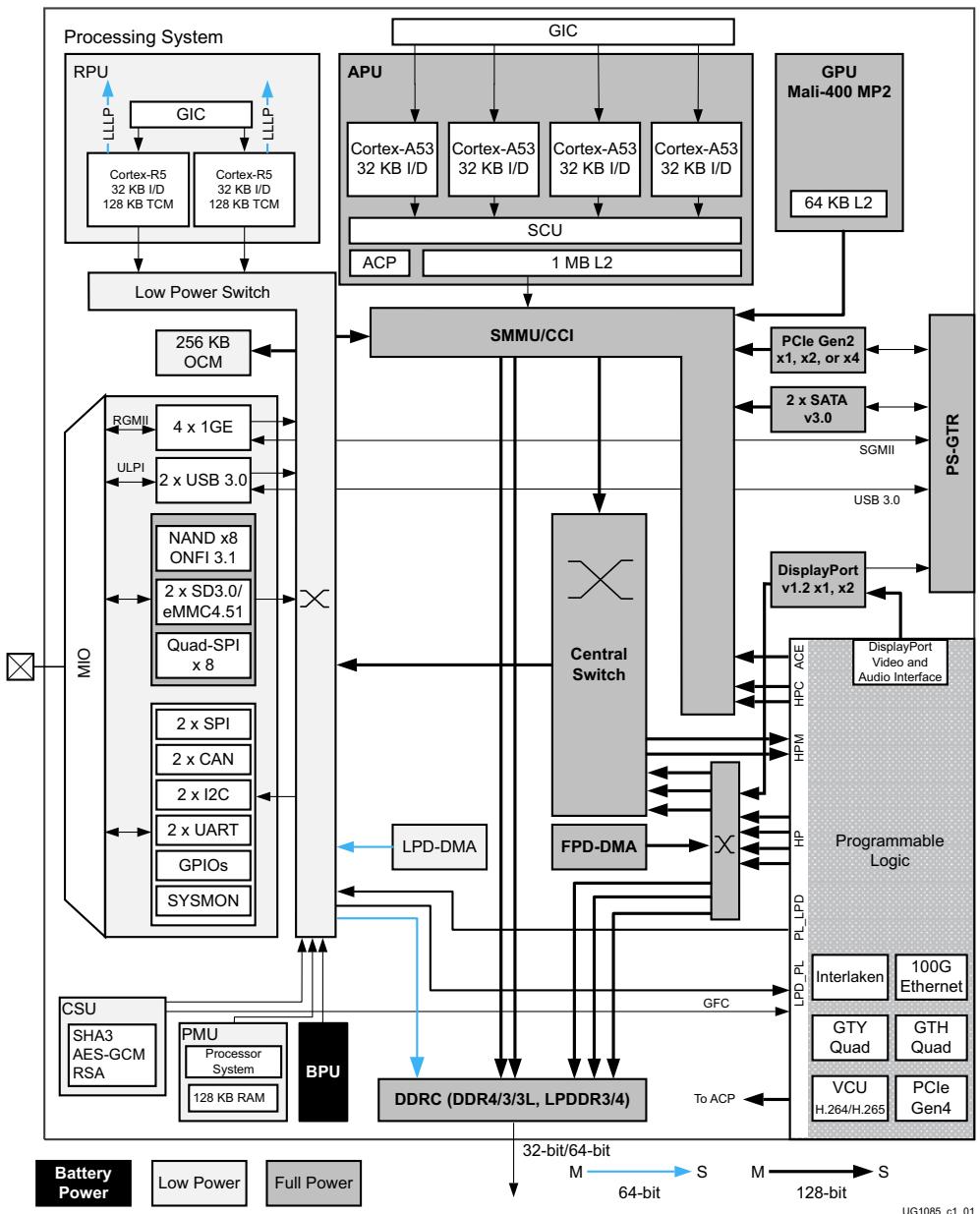


Figure 2-1: Zynq UltraScale+ MPSoC Device Hardware Architecture

[Send Feedback](#)

The Zynq UltraScale+ MPSoC device features are as follows:

- Cortex-R5 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 (I/F): DDR4, LPDDR4, DDR3, DDR3L, LPDDR3, 2x Quad-SPI, and NAND.
- General connectivity: 2x USB 2.0, 2x SD/SDIO, 2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 1GE, and GPIO.
- Security: Advanced encryption security (AES), Rivest Shamir Adleman (RSA), and secure hash algorithm (SHA3)
- AMS system monitor: 10-bit, 1 MSPS ADC, temperature, voltage, and current monitor
- The PS (processor subsystem) has five high-speed serial I/O (HSSIO) interfaces supporting the necessary protocols:
 - Integrated block for the Processor Subsystem (PS) PCIe interface: base specification, version 2.1 compliant, and Gen2x4
 - SATA 3.0 specification-compliant interface
 - Display port interface: Implements a Display port source-only interface with video resolution up to 4k x 2k
 - USB 3.0 interface: Compliant to USB 3.0 specification implementing a 5 Gb/s line rate
 - Serial GMII interface: Supports a 1 Gb/s SGMII interface
- Platform management unit (PMU) for power sequencing, safety, security and debug.

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

For additional components, see the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [\[Ref 10\]](#).

APU and RPU Executable Memory Regions

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

Table 2-1: Configurable Memory Regions for APUs

Memory Type	Start Address	Size
DDR Low	0x00000000	2 GB
DDR High	0x800000000	2 GB
OCM	0xFFFFC0000	256 KB
QSPI	0xC0000000	512 MB

Table 2-2: Configurable Memory Regions for RPU Lock-Step Mode

Memory Type	Start Address	Size
DDR Low	0x100000	2047 MB
OCM	0xFFFFC0000	256 KB
QSPI	0xC0000000	512 MB
R5_0_ATCM_MEM_0	0x00000	64 KB
R5_0_BTSM_MEM_0	0x20000	64 KB
R5_TCM_RAM_0_MEM	0x00000	256 KB

Table 2-3: Configurable Memory Regions for RPU Split Mode

Memory Type	Start Address	Size
R5_0		
DDR Low	0x100000	2047 MB
OCM	0xFFFFC0000	256 KB
QSPI	0xC0000000	512 MB
R5_0_ATCM_MEM_0	0x00000	64 KB
R5_0_BTSM_MEM_0	0x20000	64 KB
R5_1		
DDR Low	0x100000	2047 MB
OCM	0xFFFFC0000	256 KB
QSPI	0xC0000000	512 MB
R5_1_ATCM_MEM_0	0x00000	64 KB
R5_1_BTSM_MEM_0	0x20000	64 KB

Note the following:

- In RPU lock-step mode, R5_0_ATCM_MEM_0 and R5_0_BTMC_MEM_0 memory address are mapped to R5_0_ATCM_LSTEP and R5_0_BTMC_LSTEP memory ranges respectively in the system address map.
- In RPU split mode, R5_x_ATCM_MEM_0 and R5_x_BTMC_MEM_0 memory address are mapped to R5_x_ATCM_SPLIT and R5_x_BTMC_SPLIT memory ranges respectively in the system address map.
- QSPI memory is accessible when QSPI controller is in linear mode.

See this [link](#) to the "System Addresses" chapter of the *Zynq UltraScale+ MPSoC Technical Reference Manual* [Ref 10] for more information.

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. 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-R5 processor or the Cortex-A53 processor. In the Cortex-R5 processor, lock-step is also supported.
- Post-configuration stage: After FSBL execution starts, the Zynq UltraScale+ MPSoC device enters the post configuration stage.

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
- SD0/SD1
- JTAG
- USB

The BootROM does not directly support booting from SATA, Ethernet, or PCI Express (PCIe).

To understand more about the boot process in the different boot modes, see this [link](#) to the "Boot and Configuration" chapter of the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085)[Ref 10].

QSPI24

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

Note: I/O mode is not supported in FSBL.

For additional information, see [QSPI24 Boot Mode](#).

QSPI32

The QSPI32 boot mode supports the following:

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

For additional information, see [QSPI32 Boot Mode](#).

eMMC18

The eMMC18 boot mode 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 [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 device in **non-secure mode** only.*

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

USB

USB boot mode supports USB 2.0. It does not support multiboot, image fallback or XIP. It supports both secure and non-secure boot mode. It is not supported for DDR-less systems. 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 device. 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.

The support for virtualization applies only to an implementation that includes ARM exception level-2 (EL2). The ARM v8 supports virtualization extension to achieve full virtualization with near native guest operating system performance.

To understand Virtualization, see this [link](#) to "System Virtualization" in the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [\[Ref 10\]](#).

Reset

Zynq UltraScale+ MPSoC provides multiple system level resets like power-on reset (POR), system reset, and debug system reset; And also block-level resets for PS, APU, RPU, FPD and PL.

The Zynq UltraScale+ MPSoC device reset block is responsible for handling both internal and external reset inputs to the system, and to ensure that the reset requirements are met 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 the processor system (PS) and processor logic (PL) blocks.

Security Features

The following blocks provide the main Zynq UltraScale+ MPSoC device security features.

Configuration Security Unit

The configuration security unit (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.

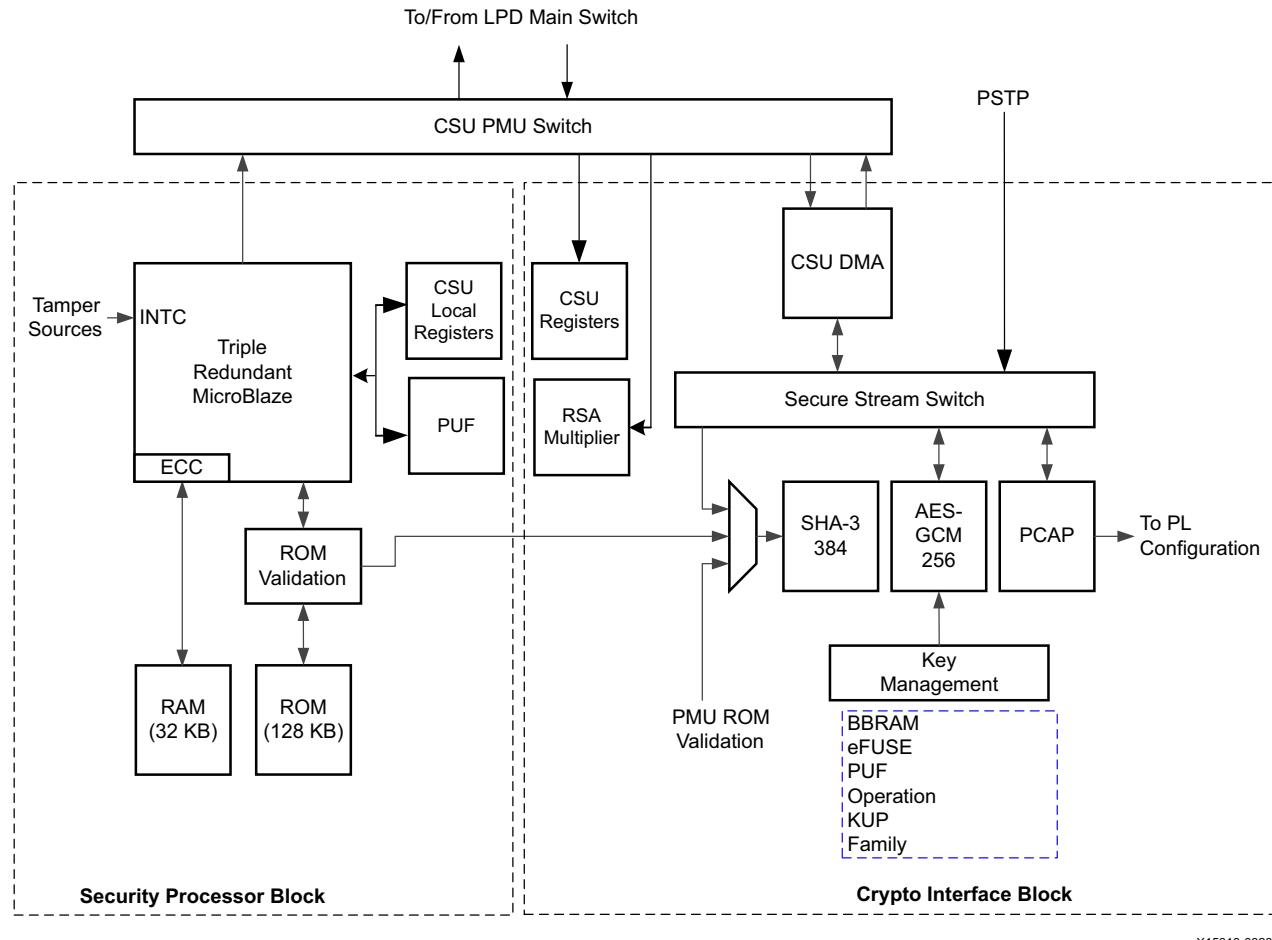


Figure 2-2: Configuration and Security Unit Architecture

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. After boot, the CSU provides tamper response monitoring.

These crypto interfaces are available to the user during runtime. To understand how to use these features, see [Appendix K, XilSecure Library Reference](#). Refer to this [link](#) in the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [Ref 10], for more information.

- **Secure Processor Block:** The triple-redundant processor ensures an error-free device under 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 software or the CSU boot ROM.
- **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, see the following references:

- [Chapter 7, System Boot and Configuration](#)
- [Chapter 15, Bootgen Image Creation](#).
- This [link](#) in the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [\[Ref 10\]](#).
- "Boot and Configuration" information in the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085)[\[Ref 10\]](#).

System-Level Protections

The system-level protection mechanism involves the following areas:

- Zynq UltraScale+ MPSoC system software stack relies on ARM Trusted Firmware (ATF) to provide 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) provides 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 programmable 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 without knowing the address aperture of the peripheral.

Safety and Reliability

The Zynq UltraScale+ MPSoC architecture includes features that ensure the reliability of safety critical applications to give users and designers 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 two separate protection options.

- SCU-L2 cache protection
- CPU cache protection

These options enable the Cortex-A53 MPCore processor to detect and correct a one-bit error in any RAM, and to detect two-bit errors.

Cortex-A53 MPCore RAMs are protected against single-event-upset (SEU) such that the processor system can detect and 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 (SECDED) capability.

The RPU includes two major safety features:

- Lock-step operation, shown in [Figure 2-3](#).
- Error checking and correction, described further in [Error Checking and Correction](#).

Lock-Step Operation

Cortex-R5 processors support lock-step operation mode, which operates both RPU CPU cores as a redundant CPU configuration, called safety mode.

The Cortex-R5 processor set to operate in the lock-step configuration exposes only one CPU interface.

Because Cortex-R5 processor only supports the static split and lock configuration, switching between these modes is permitted only while the processor group is held in power-on reset (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-R5 processors are in the lock-step mode (shown in the following figure), there must be code in the reset handler to ensure that the distributor within the GIC dispatches interrupts only to CPU0.

The RPU includes a dedicated interrupt controller for Cortex™-R5 MPCore processors. The ARM® PL390 generic interrupt controller (GIC) is based on the GICv1 specification.

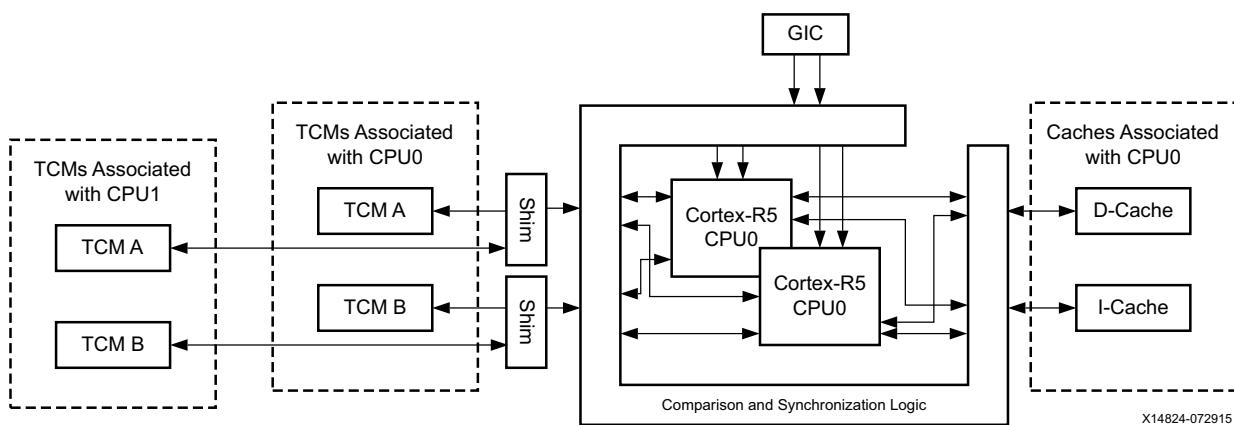


Figure 2-3: RPU Lock-Step Operation

TCMs are mapped in the local address space of each Cortex-R5 processor; however, they are also mapped in global address space where any master can access them. The following table lists the address maps from the RPU point of view.

Table 2-4: 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 BTM (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 (64KB)	N/A	0x0000_0000	0xFFE9_0000
	R5_1 BTM (64KB)	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 (128KB)	0x0000_0000	N/A	0xFFE0_0000
	R5_0 BTM (128KB)	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-R5 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 ensure an error-free operation of the Zynq UltraScale+ MPSoC.

These features include the following:

- Platform management unit
- PMU triple-redundancy

The following sections describe these features.

Platform Management Unit

The platform management unit (PMU) in the Zynq UltraScale+ MPSoC device 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.

Processor 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 the following set of heterogeneous 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 10, 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 device 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+ MPSoC Technical Reference Manual* (UG1085) [Ref 10].

Development Tools

Introduction

This chapter focuses on Xilinx® tools and flows available for programming software for Zynq® UltraScale+™ MPSoC devices. 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 device.

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 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++, SystemC, and OpenCL
- 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* [Ref 3].

Xilinx Software Development Kit

The Xilinx Software Development Kit (SDK) provides a complete environment for creating software applications targeted for Xilinx embedded processors. It includes a GNU-based compiler toolchain, JTAG debugger, flash programmer, middleware libraries, bare-metal BSPs, and drivers for Xilinx IP. SDK also includes a robust IDE for C/C++ bare-metal and Linux application development and debugging. Based upon the open source Eclipse platform, SDK incorporates the C/C++ Development Toolkit (CDT).

SDK lets you create software applications using a unified set of Xilinx tools for the ARM® Cortex™-A53 and Cortex-R5 processors, as well as Xilinx MicroBlaze™ processors*. SDK 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 SDK 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 A53 or R5 in Zynq UltraScale+ MPSoC devices⁽¹⁾)

You can export a block design, hardware design files, and bitstream files to the SDK export directory directly from the Vivado Project Navigator. For more information regarding the Vivado Design Suite, see the *Vivado Design Suite Documentation* [Ref 20].

All processes necessary to successfully complete this export process are run automatically. The SDK process exports the following files to the SDK directory:

- .project: Vivado project file
- psu_init.tcl, psu_init_gpl.c, psu_init_gpl.h, psu_init.c, psu_init.h: Contain information required during FSBL creation
- psu_init.html: Zynq UltraScale+ MPSoC register summary viewer
- system.hdf: Hardware definition file

SDK can also generate:

- First stage boot load (FSBL)
- Boot image header (BOOT.BIN) for secure and non-secure boot for the following processors:
 - ARM Cortex-A53
 - ARM Cortex-R5
 - MicroBlaze

To understand more about FSBL creation, see [Boot Image Creation in Chapter 7](#).

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-A53, Cortex-R5, and Xilinx MicroBlaze devices)

For the list of build procedures, see the *Xilinx Software Developer Kit Help* [Ref 21], where built-in Help content lets you explore further after you launch the SDK tool.

Also, SDK provides the following tools for use in Xilinx embedded software development:

- **Xilinx System Debugger (XSDB):** Provides a system debugger GUI and a command-line interface to the Xilinx hw_server. XSDB also provides various low-level debugging features not directly available in SDK.
- **FPGA programmer:** Programs the Xilinx device with the bitstream.

1. The Zynq-7000 AP SoC device supports the Cortex-A9 device. See the *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821) [Ref 19] for more information.

- **Flash programmer:** Used for burning bitstreams and software application images into external, parallel NOR flash devices.
- **Linker script generator:** Used for mapping your application image across the hardware memory space.
- **Boot image generator:** Used to create a boot image, by combining boot loader, bitstream, user applications, and optional authentication and encryption enabled.
- **Xilinx software command-line tool (XSCT):** The Xilinx software command-line tool is based on Tcl and is delivered in the SDK install. This tool provides the Tcl prompt and you can use all of the supported commands.

The Xilinx software command-line tool is a scriptable command-line interface to run SDK commands, XSDB commands, and HSI commands. XSCT can be started from the Start menu or by executing the `xsct.bat` file, available in the `<SDK installation directory>/bin` folder. All the commands supported by XSCT are grouped under their respective categories.

The following table shows the Xilinx SDK tool chain.

Table 3-1: SDK-Supported Tool Chains

Tools	Description
GNU compiler tool suite	Includes tools like GCC, AS, LD, BIN UTILS used for compilation.
Bootgen	Used for generating BOOT.BIN.
Make	Supports Make build.
Performance analysis tools	Includes SPM, GPROF, OProfile.
Debug/Download Tools	Includes GDB, QEMU, XSDB, Flash writer.

The SDK provides a separate perspective for each task to ease the software development process. Perspectives available for C/C++ developers are, as follows:

- **C/C++ Perspective views:** Helps you to view, create and build the software C/C++ projects. By default, it consists of an editor area and other views, such as SDK projects, C/C++ projects to show the software projects present in the workspace, a navigation console, properties, tasks, make targets, outline, and search.
- **System Debugger:** Helps you debug a software application. The system debugger from open source is customized and integrated with SDK.
- **System Performance Monitor:** Assists you in characterizing and evaluating the performance of hardware and software systems by providing the performance summary views of the MicroBlaze devices, and the PL and PS of the Zynq UltraScale+ MPSoC device.
- **Remote System Explorer:** Let you connect and work with a variety of remote systems.

SDK supports Linux application development but does not explicitly target Linux Kernel development and debug; however, both Xilinx PetaLinux tools and third-party partners provide such tools and capabilities.

For a detailed explanation on the SDK features, and to understand the SDK design flow with a "Hello World" example see the *Xilinx Software Developer Kit Help*, [Ref 21].

You can download the SDK tool from the *Embedded Design Tools Download* [Ref 23] link.

PetaLinux Tools

The PetaLinux tools offer everything necessary to customize, build, and deploy embedded Linux solutions on Xilinx processing systems. Tailored to accelerate design productivity for MPSoC-like devices, the solution works with the Xilinx hardware design tools to facilitate the development of Linux systems for the Zynq UltraScale+ MPSoC device.

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 tool-chain.

Table 3-2: PetaLinux Supported Tools

Tools	Description
GNU	Xilinx ARM GNU tools.
PetaLinux-Build	PetaLinux build commands used to build software image files.
Make	Make build for compiling the applications.
GDB	GDB tools for debugging.
PetaLinux-Boot	PetaLinux-Boot command used to boot the linux.
QEMU	Emulator platform for the Zynq UltraScale+ MPSoC device.
OProfile	Used for profiling.

See the following documentation for more details:

- *PetaLinux Tools* documentation [Ref 2]
- *Zynq Embedded Design Tutorial* (UG1165) [Ref 17]
- *Zynq UltraScale+ MPSoC OpenAMP Getting Started Guide* (UG1186) [Ref 13]

Open Source

The ARM GNU open source tool chain is adopted for the Xilinx software development platform. The GNU tools for Linux hosts are available as part of Xilinx software development kit (SDK). This section details the open source GNU tools and Linux tools available for the processing clusters in the Zynq UltraScale+ MPSoC device.

Xilinx ARM GNU Tools

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

Table 3-3: Xilinx ARM GNU Tools

Tool	Description
aarch64-linux-gnu-gcc	GNU C/C++ compiler.
aarch64-linux-gnu-g++	
aarch64-linux-gnu-as	GNU assembler.
aarch64-linux-gnu-ld	GNU linker.
aarch64-linux-gnu-ar	A utility for creating, modifying and extracting from archives.
aarch64-linux-gnu-objcopy	Copies and translates object files.
aarch64-linux-gnu-objdump	Displays information from object files.
aarch64-linux-gnu-size	Lists the section sizes of an object or archive file.
aarch64-linux-gnu-gprof	Displays profiling information.
aarch64-linux-gnu-gdb	The GNU debugger.

Linux Software Development using Yocto Tools

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+ MPSoC devices.

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 the `meta-xilinx` link [\[Ref 30\]](#).

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 3-4: 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 <code>ftrace</code> 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

The Yocto project development environment supports the developing Linux software for Zynq UltraScale+ MPSoC devices 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.

[Figure 3-1](#) 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 rich compiler and quality analyzing 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 SDK 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 *Zynq UltraScale+ MPSoC QEMU User Guide* (UG1169) [Ref 8] for more information.

IMPORTANT: Full Yocto enablement of Xilinx QEMU is not available.

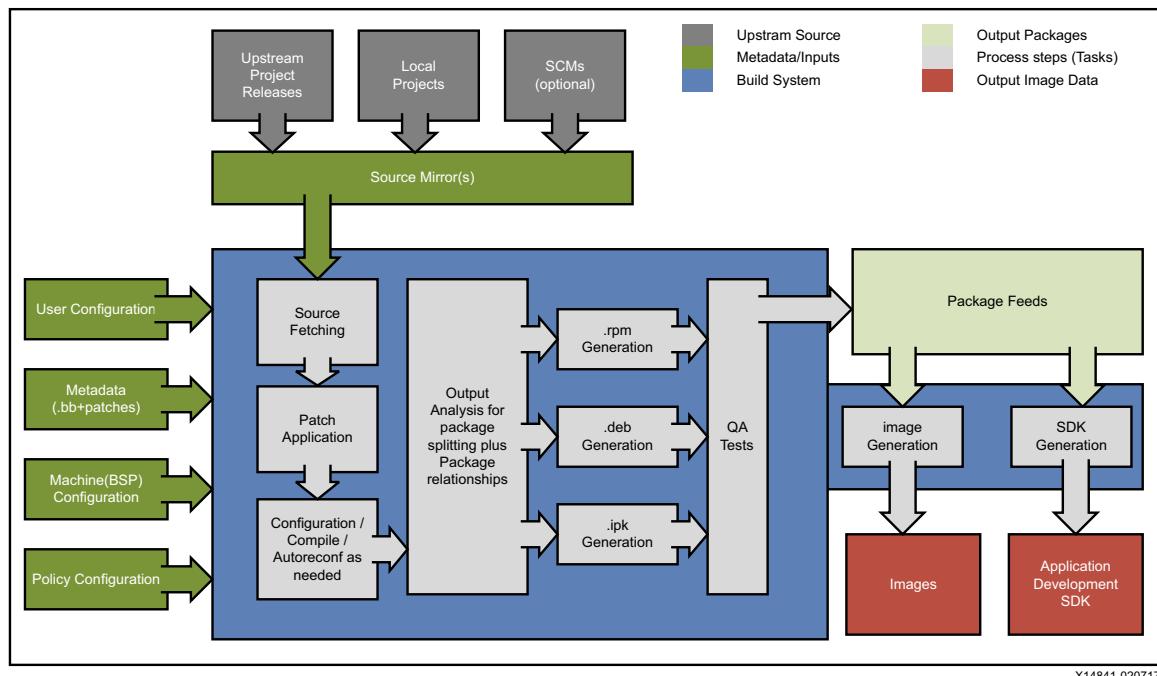


Figure 3-1: Yocto Project Development Environment

You can download the Yocto tools and the Yocto project development environment from the *Yocto Project Organization* [Ref 40].

For more information about Xilinx-supported Yocto features, see this [link](#) in the *PetaLinux Tools Documentation: Reference Guide* (UG1144) [Ref 24].

Software Stack

Introduction

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

To understand the various software development tools used with this device, see [Chapter 3, Development Tools](#). To understand more 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 Xilinx® SDK tools. 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. The [Xilinx Software Development Kit](#) in [Chapter 3, Development Tools](#) provides more overview information. You can locate the standalone drivers at the following path:

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

You can locate libraries at the following path:

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

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

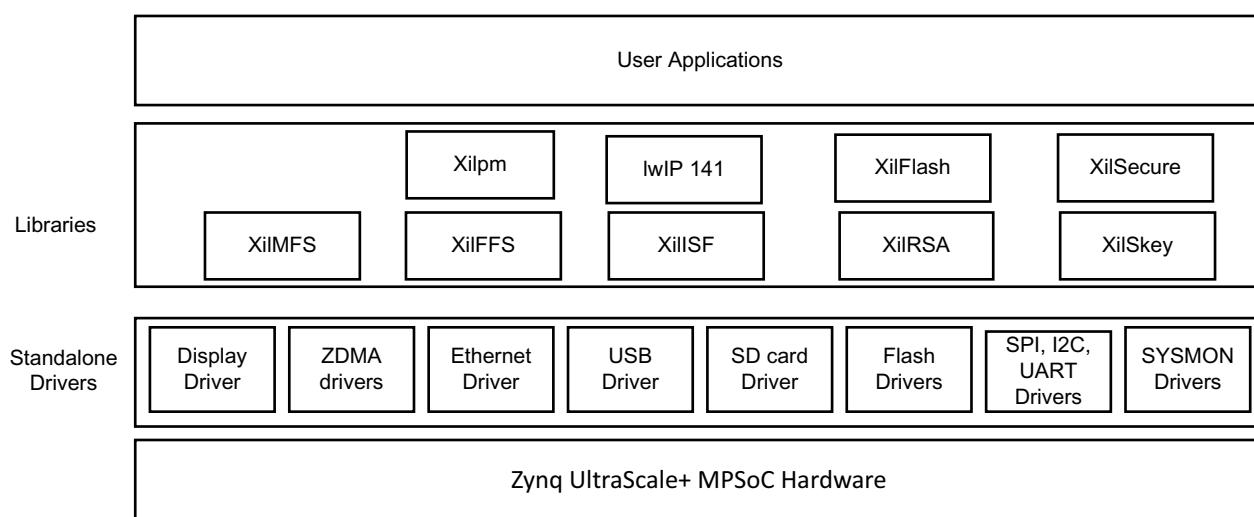


Figure 4-1: Bare-Metal Software Development Stack

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-R5 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-R5, and the MicroBlaze processors.
- Additional middleware libraries that provide networking, file system, and encryption support.
- Application examples including the first stage bootloader (FSBL) and test applications.

Libc

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

Table 4-1: 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.
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.

libm

The following table lists the libm mathematical C modules:

Table 4-2: 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 transcendentals	j0, j1, jn, y0, y1, yn, erf, erfc, gamma, lgamma, and gamma_ramma_r
Integral Rounding	eil, floor, rint
IEEE standard recommended	copysign, fmod, ilogb, nextafter, remainder, scalbn, and fabs
IEEE Classification	isnan
Floating Point	logb, scalb, significand
User defined Error handling routine	matherr

Standalone BSP

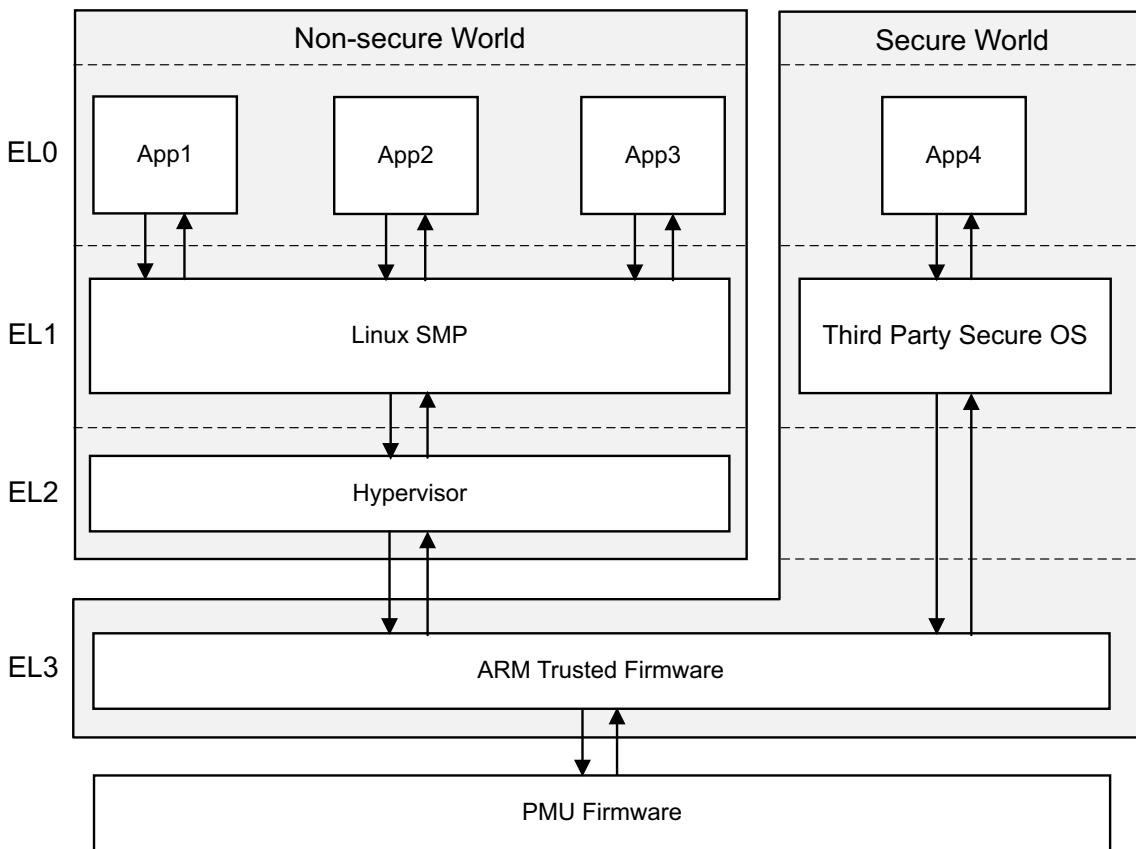
The libraries available with the Standalone BSP are as follows:

- XilFatFS: Is a LibXil FATFile system and provides read/write access to files stored on a Xilinx system ACE compact flash.
- XilFFS: Generic Fat File System Library.
- XilFlash: Xilinx flash library for Intel/AMD CFI compliant parallel flash.
- XilISF: In-System Flash library that supports the Xilinx in-system flash hardware.
- XilMFS: Memory file system.
- XilRSA: Xilinx RSA library.
- XilSkey: Xilinx secure key library.
- lwIP Library: An open source TCP/IP protocol suite that provides access to the core lwIP stack and BSD (Berkeley Software Distribution) sockets style interface to the stack

These libraries are documented in [Appendix B, Xilinx Standard C Libraries](#).

Linux Software Stack

The Linux OS supports the Zynq UltraScale+ MPSoC device. 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 Linux software stack in APU.



X18968-032717

Figure 4-2: Linux Software Development Stack

The ARM v8 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.
- EL3 provides support for a secure state. The Cortex-A53 MPCore processor implements all the exception levels (EL0–EL3) and supports both execution states (AArch64 and AArch32) at each exception level.

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

- **PetaLinux Tools:** PetaLinux tools includes a well-tested 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. See the *PetaLinux Product Page* [Ref 2].
- **Open Source Linux and U-Boot:** The Linux Kernel updates including drivers, board configurations, and U-Boot updates for the Zynq UltraScale+ MPSoC device, and are available from the Xilinx *Github* link [Ref 28], and on a continuing basis from the main Linux Kernel and U-Boot trees as well. Yocto board support packages are also available 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 [Ref 29].

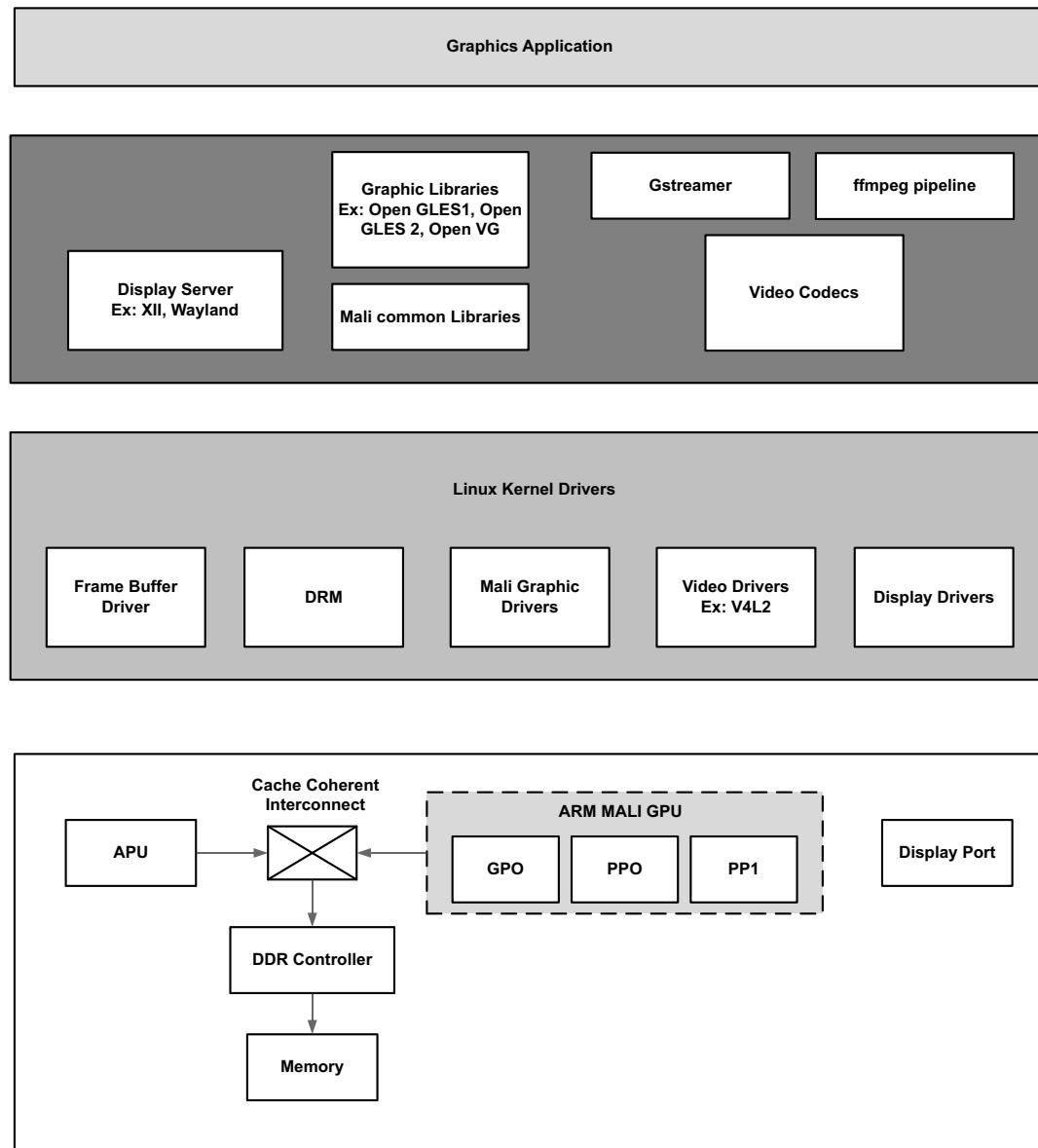
Multimedia Stack Overview

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

The GPU and a high performance DisplayPort accelerate the graphics application. 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 APU and GPU is achieved by cache-coherent interconnect (CCI), which supports AXI coherency extension (ACE) only.

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

The following figure shows the multimedia stack.



X19224-050217

Figure 4-3: Multimedia Stack

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 4-3: Libraries and Framework Components

Component	Description
Display server	Coordinates the input and output from the applications to the operating system.
Graphics library	The Zynq UltraScale+ MPSoC device architecture supports OpenGL ES 1.1 and 2.2, and Open VG 1.1.
Mali-400 MP2 common libraries	Mali-400 MP2 graphic libraries.
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 4-4: Linux Kernel Drivers

Drivers	Description
Frame buffer driver	Kernel graphics driver exposing its interface through <code>/dev/fb*</code> . 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 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 Xilinx SDK tool. 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.

The following figure illustrates the FreeRTOS software stack for RPU.

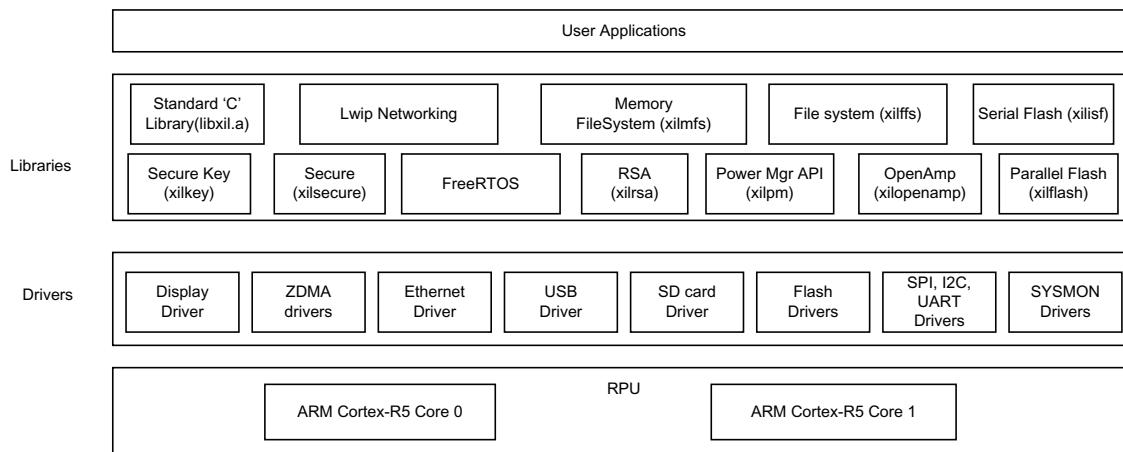

X16911-061516

Figure 4-4: 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

Many other embedded software solutions are also available from the Xilinx partner ecosystem. More information is available from the Xilinx website, *Embedded Computing* [Ref 29] and the website, *Xilinx Third Party Tools* [Ref 4].

Software Development Flow

Overview of Software Development Flow

This chapter explains the bare-metal software development for RPU and APU using the Xilinx® Software Development Kit (SDK), as well as Linux software development for APU using PetaLinux tools and SDK.

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

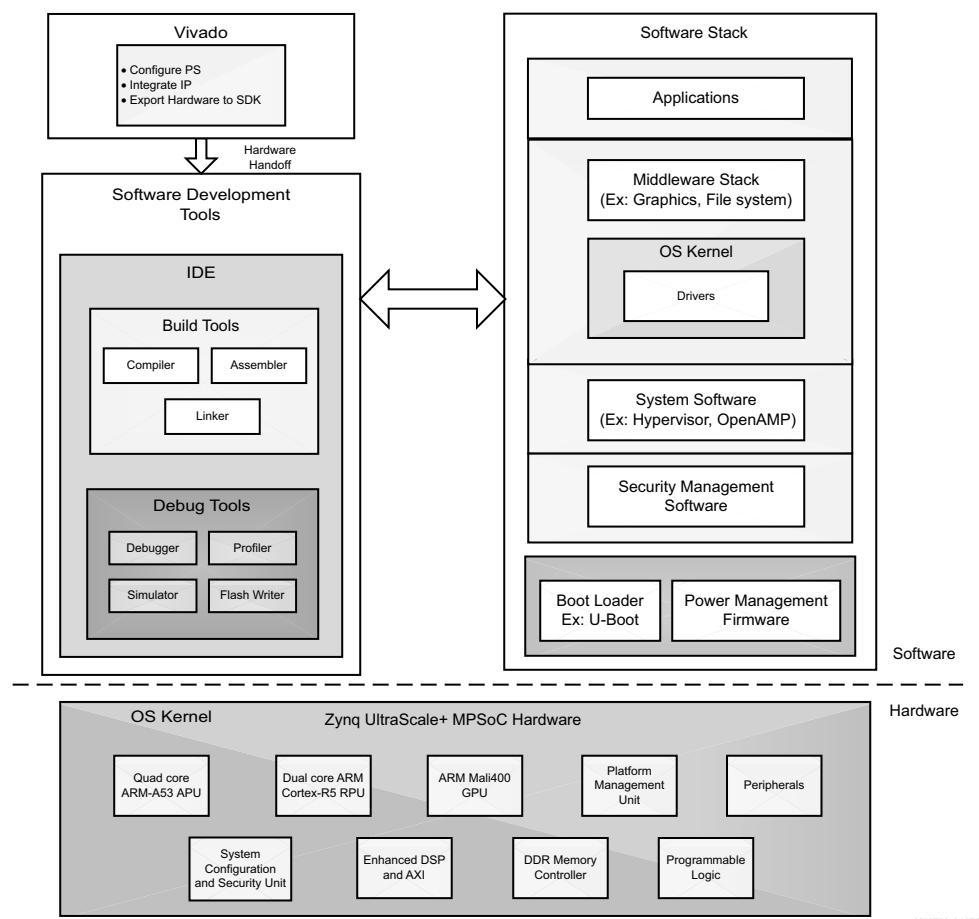


Figure 5-1: Software Development Architecture

Developing Bare-Metal Applications

This section assists you in understanding the design flow of bare-metal application development for APU and RPU using SDK.

The following figure shows the top-level design flow in SDK.

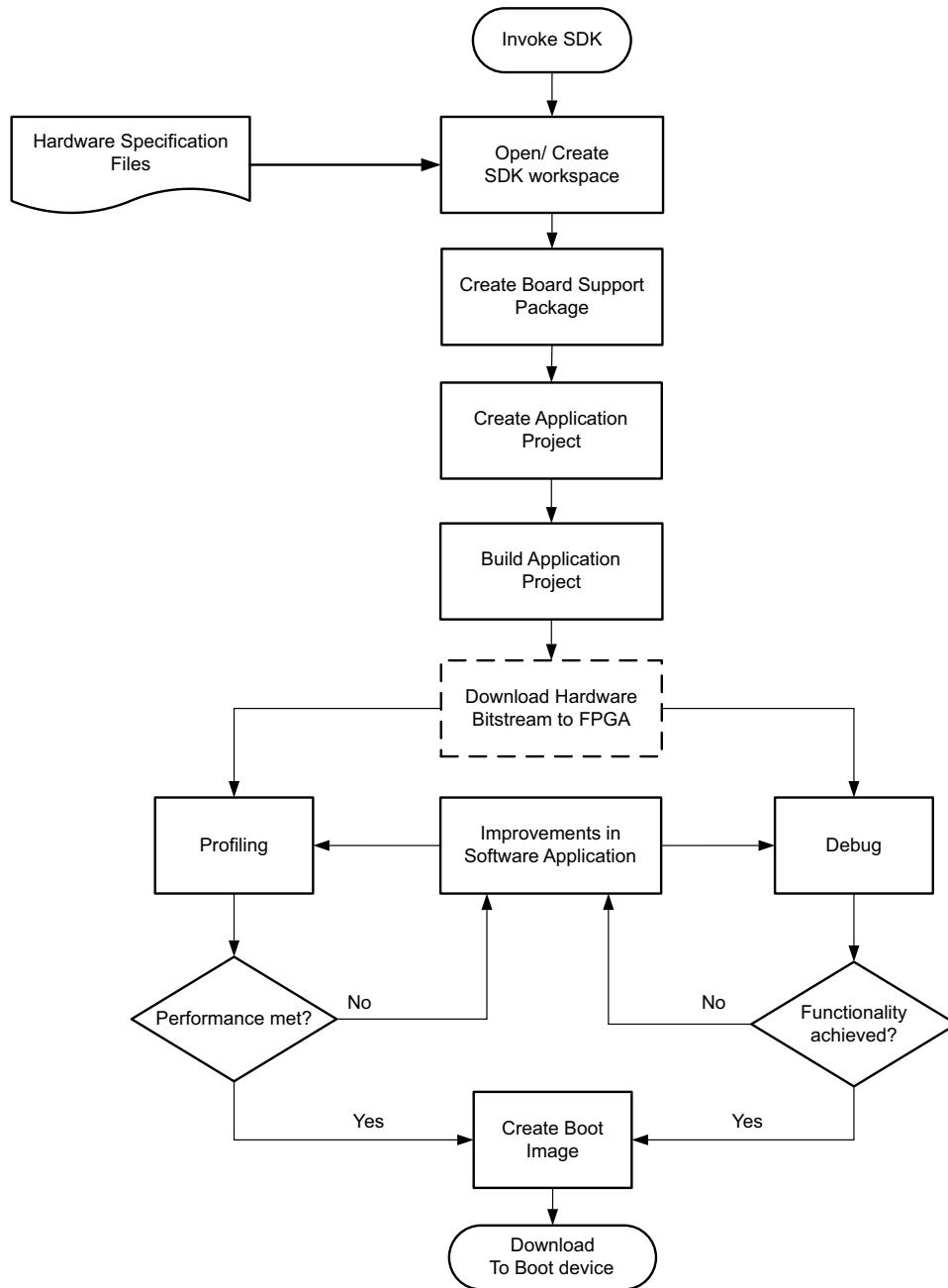

X14817-050317

Figure 5-2: Bare-Metal Application Development Flow

Developing bare-metal applications involves the following steps:

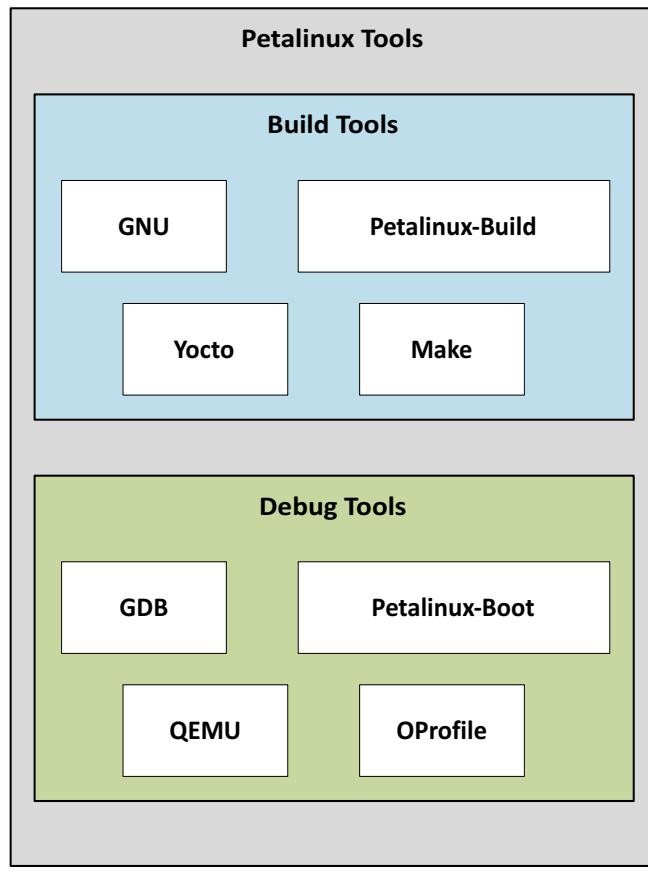
1. Opening and creating an SDK workspace for bare-metal applications: See [Creating a Standalone Application Project](#).
2. Importing the hardware platform information: See [Importing a Hardware Platform Specification File](#).
3. Select target processor (A53|R5|PMU MicroBlazeTM).
4. Creating a board support package (BSP): See [Creating a Board Support Package](#).
5. Alter the BSP configuration settings (optional): See [Changing Build Configuration](#).
6. Adding custom IP driver support (optional): See [Using the Board Support Package Drivers Page](#).
7. Creating application projects: See [Creating Application Projects](#).
8. Building application projects: See [Building Projects](#).
9. Debugging user applications: See [Debugging Projects](#).
10. Running user applications: See [Running Projects](#)
11. Profiling user applications: See [Software Profiling](#).
12. Modeling system performance: See [System Performance Modeling](#).
13. Creating boot image: See [Creating a Boot Image](#).

For more details on these steps, see the *MPSoC PetaLinux Software Development link* [Ref 31].

For more details on QEMU, see the *Zynq UltraScale+ MPSoC QEMU User Guide (UG1169)* [Ref 8].

PetaLinux-Tools-Based Software Development

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.



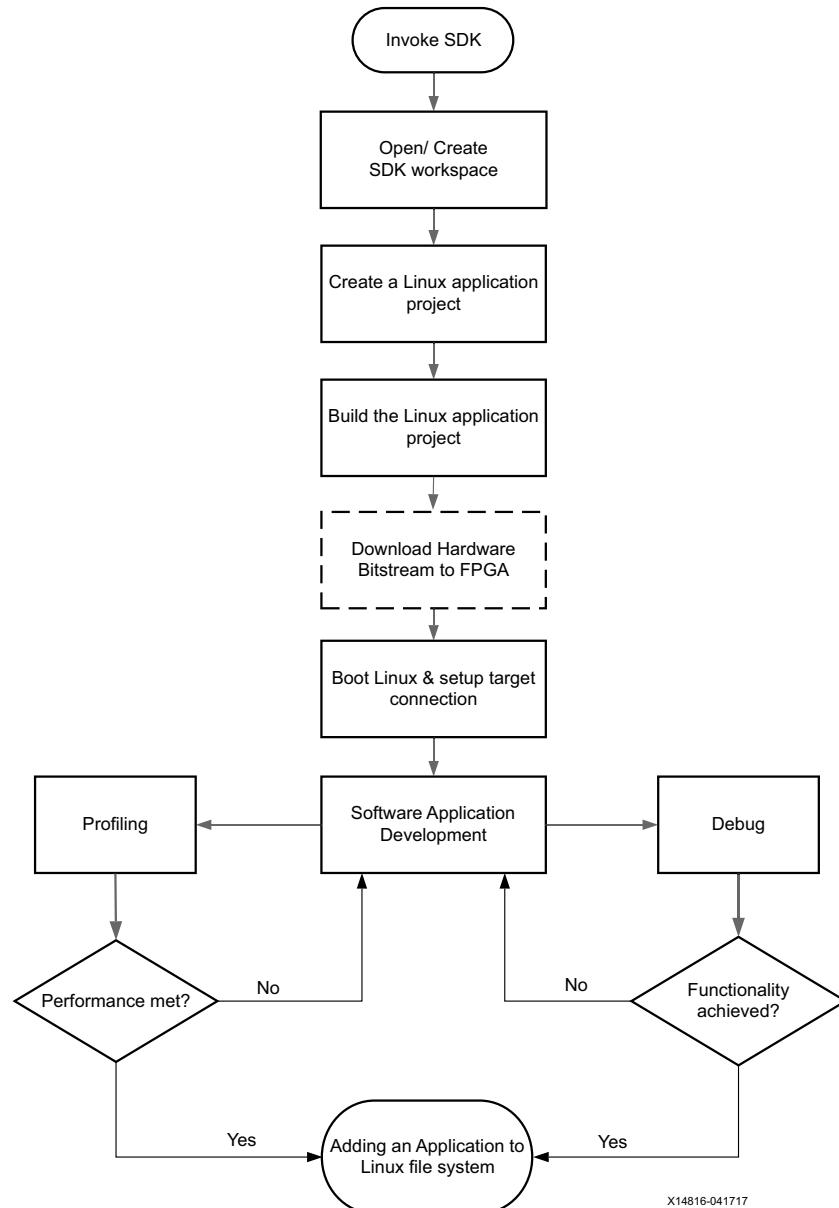
X14815-072115

Figure 5-3: PetaLinux Tool-Based Software Development Flow

Developing a Linux Application Using SDK

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 SDK.



X14816-041717

Figure 5-4: Linux Application Development Flow

The development and execution of PetaLinux application involve the following steps:

1. Setting up PetaLinux tools working environment: See [PetaLinux Working Environment Setup](#).
2. Creating PetaLinux project or a User application: See [Create a New PetaLinux Project](#).
3. Configuring and customizing the Petalinux Tools for your project
4. Importing SDK Linux C/C++ application project into PetaLinux workspace: [See PetaLinux Tools Installation Steps](#).
5. Building a PetaLinux image: See [Build System Image](#).
6. Running a PetaLinux image on a platform (QEMU or Board): See [Building User Applications](#).
7. Debugging a petalinux image. There are several options for Debugging. See the *PetaLinux Tools Documentation Reference Guide* (UG1144) [\[Ref 24\]](#).

For more details on each of above steps, see the *MPSoC PetaLinux Software Development* link [\[Ref 31\]](#).

For more details on QEMU, see the *Zynq UltraScale+ MPSoC QEMU User Guide* (UG1169) [\[Ref 8\]](#).

For a detailed explanation of the SDK features, and to understand the SDK design flow with a "Hello World" example, see the *SDK Help* [\[Ref 21\]](#). Also, see *Xilinx Software Development Kit: System Performance* (UG1145) [\[Ref 25\]](#).

Creating an Application Project

SDK provides a template-based application generator for included sample programs, from a basic "Hello World," an empty application, or an FSBL application. The Xilinx C or C++ application wizard invokes the application generator.

You can also create an empty application or import existing Linux applications for porting. Code development tools include editors, search, re-factoring, and features available in the base Eclipse platform and CDT plug-in.

Building the Application

SDK application projects can be user-managed (user-created makefiles) or automatically managed (SDK created makefiles). For user-managed projects, the user maintains the makefile and initiates application builds. For automatically managed projects, SDK updates the makefile as needed when source files are added or removed, source files are compiled when changes are saved and the ELF is built automatically; in Eclipse CDT terminology, the application project is a managed makefile project. Where possible, SDK infers or sets default build options based on the hardware platform and BSP used, including compiler, linker, and library path options.

Running the Application

You can create an SDK run configuration to copy the compiled application to the file system and run the application. With Linux running on the Zynq UltraScale+ MPSoC device platform, the run configuration copies the executable to the file system using `sftp` if the Linux environment includes SSH. A terminal view is available to interact with the application using `STDIN` and `STDOUT`.

You can also run the application using a command line shell. Use the following commands as needed:

- `sftp` to copy the executable
- `ssh` in Linux to run the executable

Adding Driver Support for Custom IP in the PL

SDK supports Linux BSP generation for peripherals in the PS as well as custom IP in the PL. When generating a Linux BSP, SDK 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.

Adding an Application to a Linux File System

You can add the compiled user application, and the required shared libraries to the Linux file system, as follows:

- While Linux is running on the Zynq UltraScale+ MPSoC device platform, you can copy the files using `sftp`, if the Linux environment includes SSH.
- In SDK, a remote system explorer (RSE) plug-in lets you copy files using a drag-and-drop mechanism.
- In workflows outside of SDK, add the application and libraries to the file system folder before creating the file system image and programming it to flash.

For more details on developing a Linux application, see “Developing a Linux Application” in the *SDK Help* [\[Ref 21\]](#).

Software Design Paradigms

Introduction

The Xilinx® Zynq® UltraScale+™ MPSoC device architecture supports heterogeneous multiprocessor engines targeted at different tasks. The main approaches to develop 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 device.
- **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 device.
- **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+ MPSoC devices to facilitate the application development on the heterogeneous processors and FPGA. The following bullets explain these frameworks:

- **Hypervisor Framework:** Xilinx provides the Xen hypervisor, a critical item of many needed to support virtualization on APU of Zynq UltraScale+ MPSoC. The [Use of Hypervisors](#) section covers more details.
- **Authentication Framework:** The Zynq UltraScale+ MPSoC device supports authentication and encryption features as a part of authentication framework. To understand more about the authentication framework, see [Boot Time Security in Chapter 8](#).
- **TrustZone Framework:** The TrustZone technology allows and maintains isolation between secure and non-secure processes within the same system.

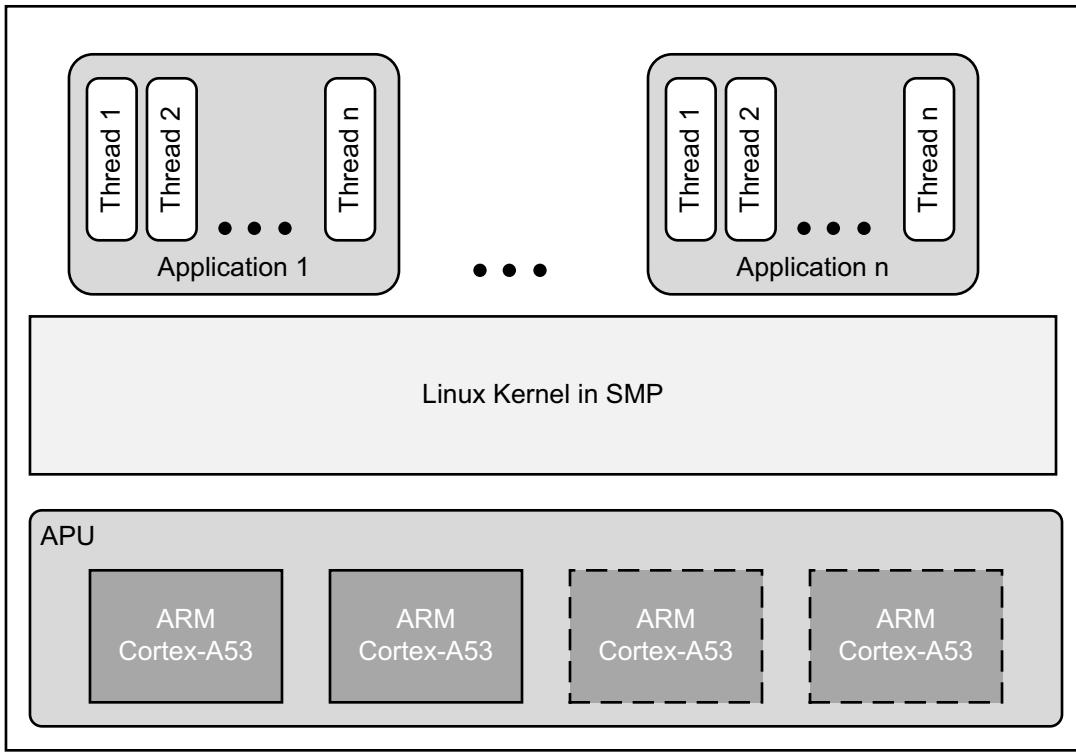
Xilinx provides the trustzone support through the ARM Trusted Firmware (ATF) to maintain the isolation between secure and non-secure world. To understand more about ATF, see [ARM Trusted Firmware in Chapter 8](#).

- **Multiprocessor Communication Framework:** Xilinx provides the OpenAMP framework for Zynq UltraScale+ MPSoC devices to allow communication between the different processing units. For more details, see the [Zynq UltraScale+ MPSoC OpenAMP Framework for Zynq Devices: Getting Started Guide \(UG1169\)](#) [Ref 13].
 - **Power Management Framework:** The power management framework allows software components running across different processing units to communicate with the power management unit. For more details, see the [Power Management Framework \(UG1199\)](#) [Ref 14].
-

Symmetric Multiprocessing (SMP)

SMP lets software development on multi-processors with relatively less development effort. A standard OS, such as Linux, handles most of the complexity in managing the multiple processors, caches, peripheral interrupts, and load balancing.

The APU in the Zynq UltraScale+ MPSoC devices contain four homogeneous cache coherent ARM Cortex™-A53 processors that support SMP mode of operation using an OS (Linux or VxWorks). Xilinx and 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.



X14837-050317

Figure 6-1: Example SMP Using Linux

This would not be the best mode of operation when there are hard, real-time requirements because it ignores Linux application core affinity which should be available to developers with existing Xilinx software. This is why the realtime processing unit (RPU) is typically not be used in SMP mode even though it has two cache coherent ARM Cortex-R5 processors.

Asymmetric Multiprocessing (AMP)

AMP uses 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.

In AMP, a software developer has to decide what code has to run on each processor before compiling and create a boot image that includes the software executable for each CPU. Using AMP with the ARM Cortex-R5 processors (SMP is not supported in Cortex-R5) in the RPU enables developers to meet highly demanding, hard, 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+ MPSoC Technical Reference Manual* (UG1085) [Ref 10] 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.

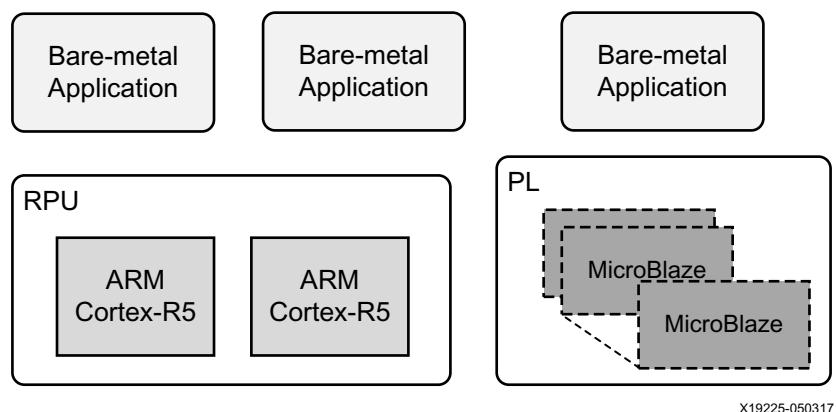


Figure 6-2: AMP Example using Bare-Metal Applications Running on RPU and PL

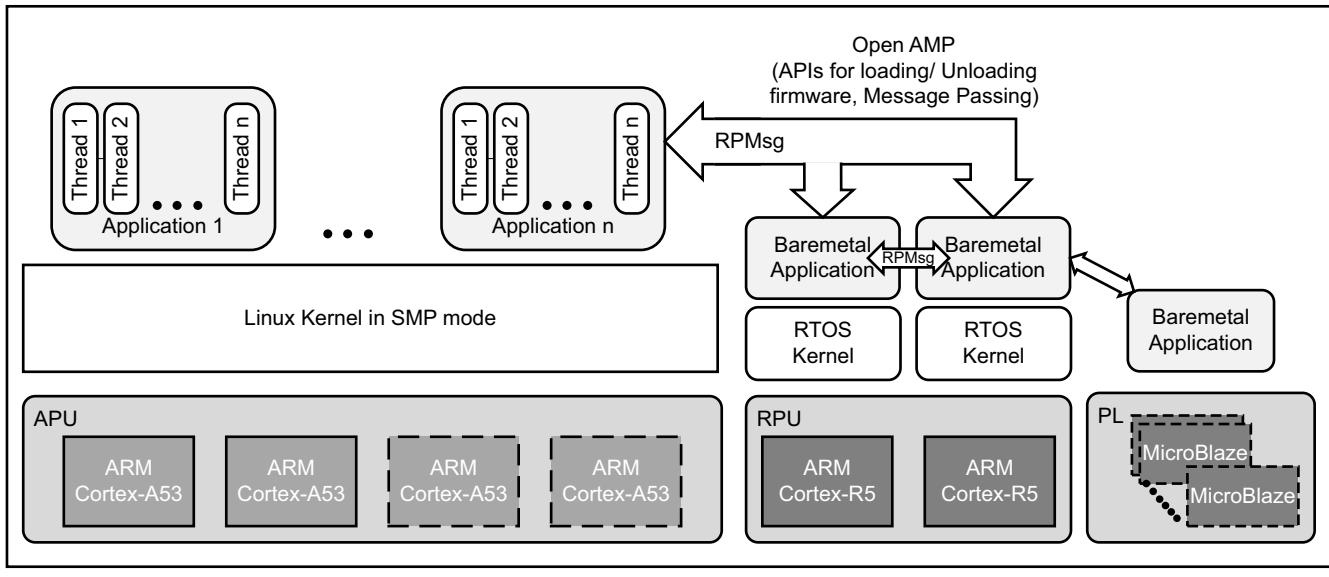
OpenAMP

The OpenAMP framework provides mechanisms to do the following:

- Load and unload firmware
- Communicate between applications using standard APIs

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.



X14839-050317

Figure 6-3: Example with SMP and AMP using OpenAMP Framework

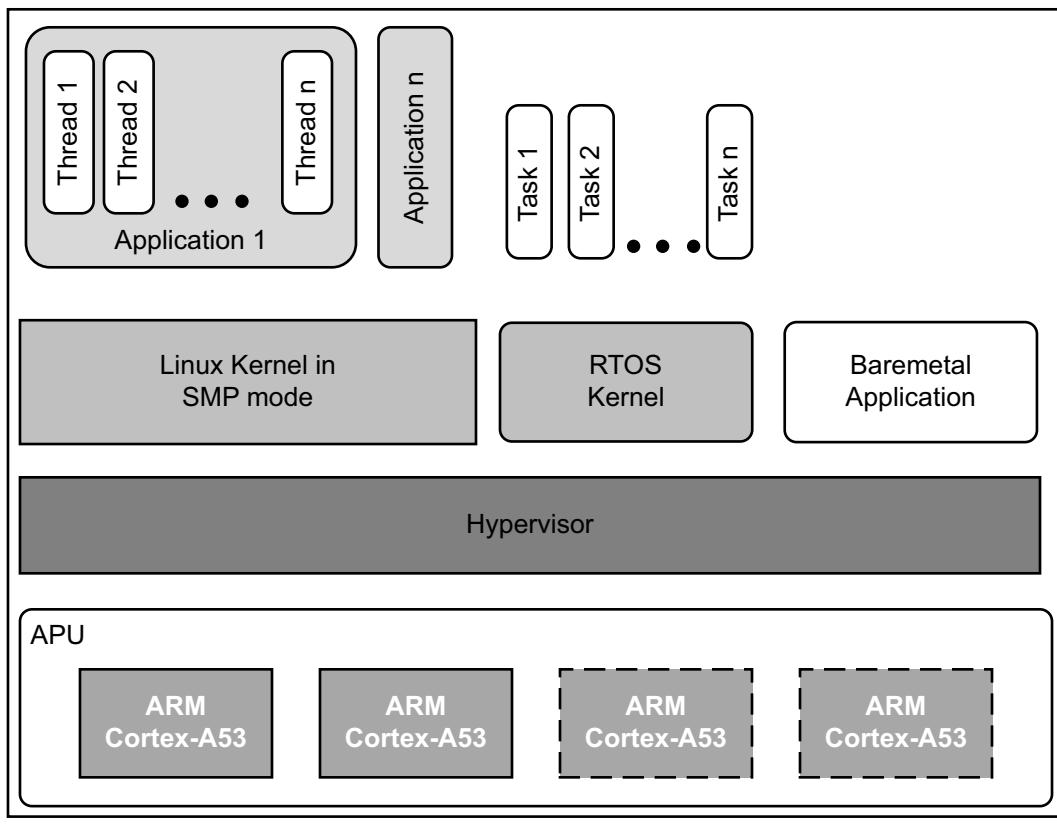
See the *Zynq UltraScale+ MPSoC OpenAMP Getting Started Guide* (UG1186) [Ref 13] for more information about the OpenAMP Framework.

Virtualization with Hypervisor

The Zynq UltraScale+ MPSoC devices 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 OS (RTOS) as well as a bare-metal application using a single hypervisor.

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



X14840-050317

Figure 6-4: Example with Hypervisor

For more details on using Hypervisor like the Xen Hypervisor, see *MPSoC Xen Hypervisor* website [Ref 37].

Use of Hypervisors

Xen is the port for the Xen open source hypervisor in the Xilinx Zynq UltraScale+ MPSoC device. 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 OS are called *domains* (also called as *virtual machines* (VMs)).

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

Note: Xen hypervisor available as a selectable component within the PetaLinux tools; Xen Hypervisor can also be downloaded from Xilinx GIT for those who use alternative work flow. 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* [\[Ref 2\]](#).

System Boot and Configuration

Introduction

Zynq® UltraScale+™ MPSoC devices support the ability to boot from different devices such as a QSPI flash, an SD card, USB Device Firmware Upgrade (DFU) host, and the NAND flash drive. This chapter details the booting process using different booting devices in both secure and non-secure modes.

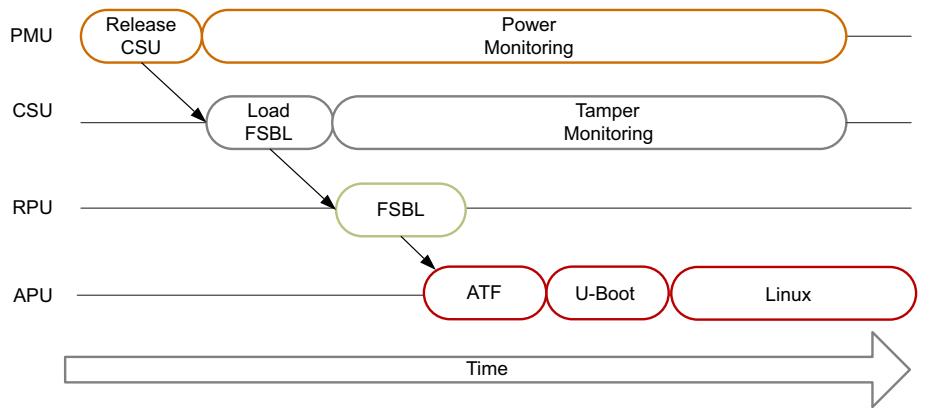
Boot Flow

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.

Non-Secure Boot Flow

The following figure shows an example of non-secure boot flow in the Zynq UltraScale+ MPSoC device with each color representing a boot entity.



X18969-050117

Figure 7-1: Non-Secure Boot Flow Example

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 as shown in Figure 1. After the PMU releases the CSU from its reset, it loads the FSBL into OCM. In this example, FSBL is loaded into RPU and U-boot and Linux are loaded into APU. You can load the FSBL into either RPU or APU. Other boot configurations allow the RPU to start and operate wholly independent of the APU and vice-versa.

- In APU, after the FSBL handoff, typically a second stage boot loader like U-Boot executes and loads an OS, such as Linux.
- In RPU, FSBL hands off to a software application.
- Linux, in turn, loads the executable software.

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

Secure Boot Flow

The following figure shows an example of secure boot flow with each color representing a boot entity.

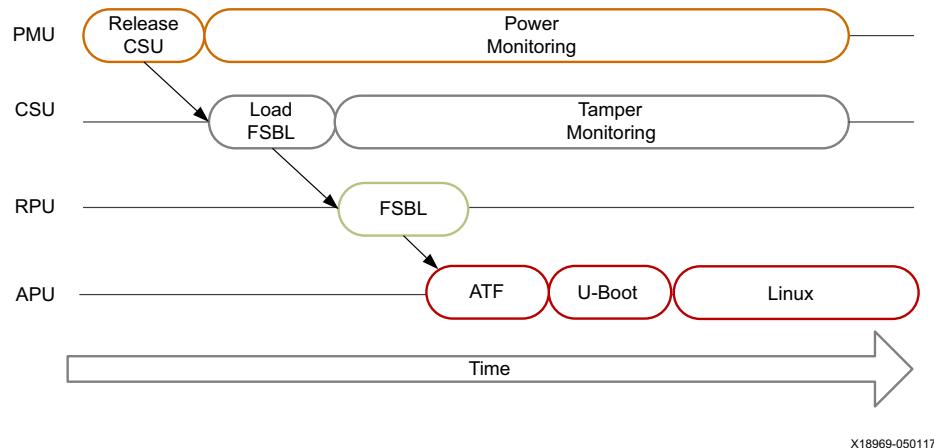


Figure 7-2: Secure Boot Flow Example

In 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 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.

In APU, FSBL hands off to ATF. 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).

Boot Image Creation

The Bootgen utility, which is available as part of SDK, creates a single boot image file suitable for booting applications developed in the Zynq UltraScale+ MPSoC device.

Bootgen creates the image by building the required boot header, appending tables that describe the following partitions, and processing the input data files (ELF files, FPGA bitstreams, and other binary files) to partitions. Bootgen has features for assigning specific destination memory addresses or imposing alignment requirements for each partition.

Bootgen also supports the encryption, authentication, and performing checksums on each partition.

The utility is driven by a configuration file known as the boot image format (BIF) file with a file extension of *.bif .



IMPORTANT: *The .bif file must contain the bitstream above ATF.*

For advanced authentication flows, you can use the Bootgen utility to output intermediate hash files that can be signed offline. Otherwise, Bootgen uses the provided private keys to sign the authentication certificates included in the boot image.

Building a boot image involves the following steps:

1. Create a BIF file.
2. Run the Bootgen utility to create a binary file.
3. (For QEMU): Convert the binary file to an image format corresponding to the boot device.

For more information regarding Bootgen, see [Chapter 15, Bootgen Image Creation](#).

Boot Modes

See [Table 7-4](#) for a brief list of available boot modes. See this [link](#) to the "Boot and Configuration" chapter of the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [\[Ref 10\]](#) for a comprehensive table of the available boot modes.

QSPI24 Boot Mode

The QSPI24 boot mode supports the following:

- x1, x2, and x4 read modes for single Quad SPI flash memory 24 (QSPI24)
- 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 boot mode (where the QSPI24 device is >128Mb), to use MultiBoot, place the multiple images so that they fit in memory locations less than 128Mb.

The following figure shows an example for booting in QSPI24 mode.

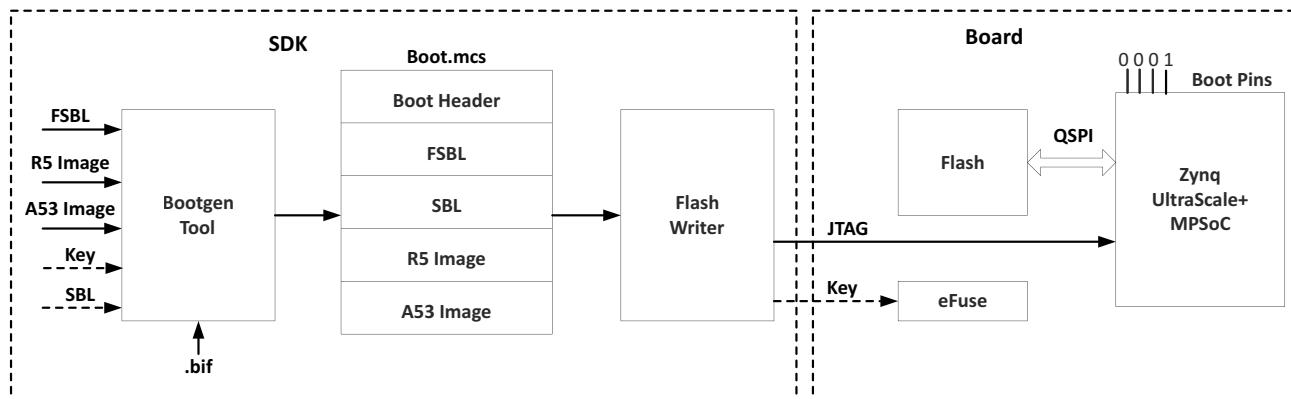


Figure 7-3: Booting in QSPI24 Mode

To create a QSPI24 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-R5 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 QSPI24 flash using the flash writer. MCS is an Intel hex-formatted file that includes an checksum for reliability.



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.

QSPI32 Boot Mode

The QSPI32 boot mode supports the following:

- x1, x2, and x4 read modes for 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 QSPI32 boot mode (where the QSPI32 device is >128Mb), to use MultiBoot, place the multiple images so that they fit in memory locations less than 128Mb.

The following figure shows an example for booting in QSPI32 mode.

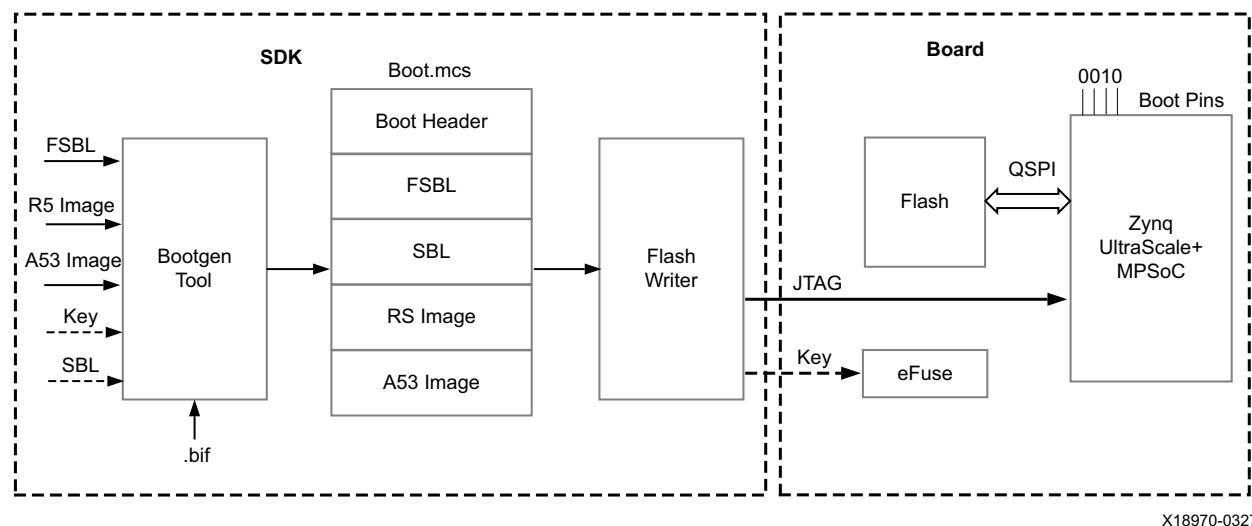


Figure 7-4: Booting in QSPI32 Mode

To create a 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-R5 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 QSPI32 flash using the flash writer. MCS is an Intel hex-formatted file that includes a checksum for reliability.

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 of files for MultiBoot are 8,192.

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

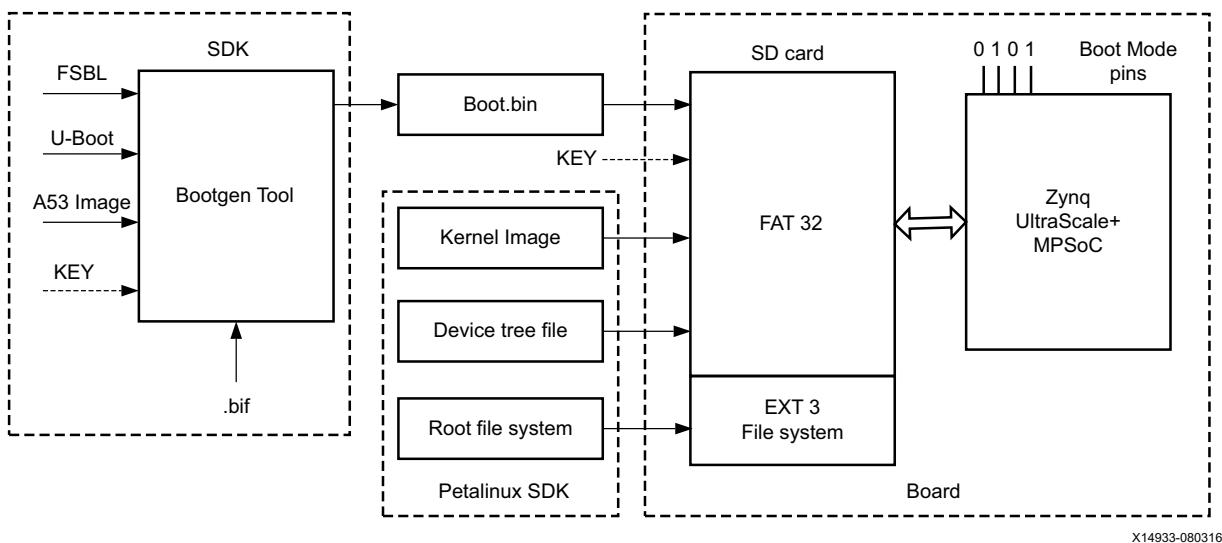


Figure 7-5: Booting in SD Mode

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

- An FSBL ELF
- A Cortex-R5 and/or an Cortex-A53 application ELF
- Optional authentication and encryption key

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

In PetaLinux, do the following:

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

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



IMPORTANT: To boot from SD in QEMU, configure the boot pins to 0x5.

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 files for MultiBoot are 8,192.

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

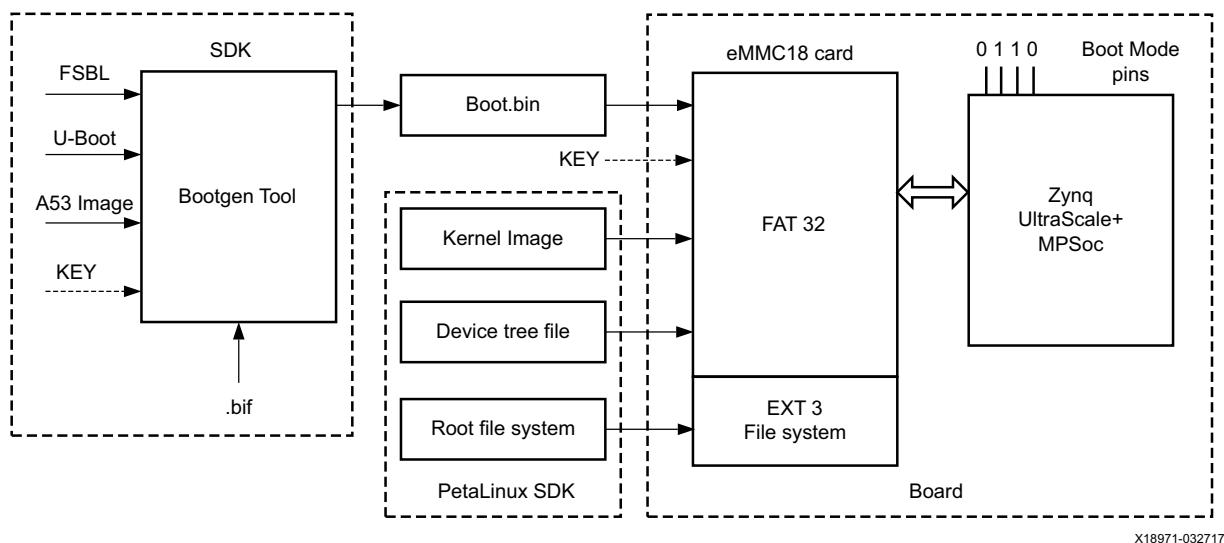


Figure 7-6: Booting in eMMC18 Mode

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

- An FSBL ELF
- A Cortex-R5 and/or a Cortex-A53 application ELF
- Optional authentication and encryption key

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.

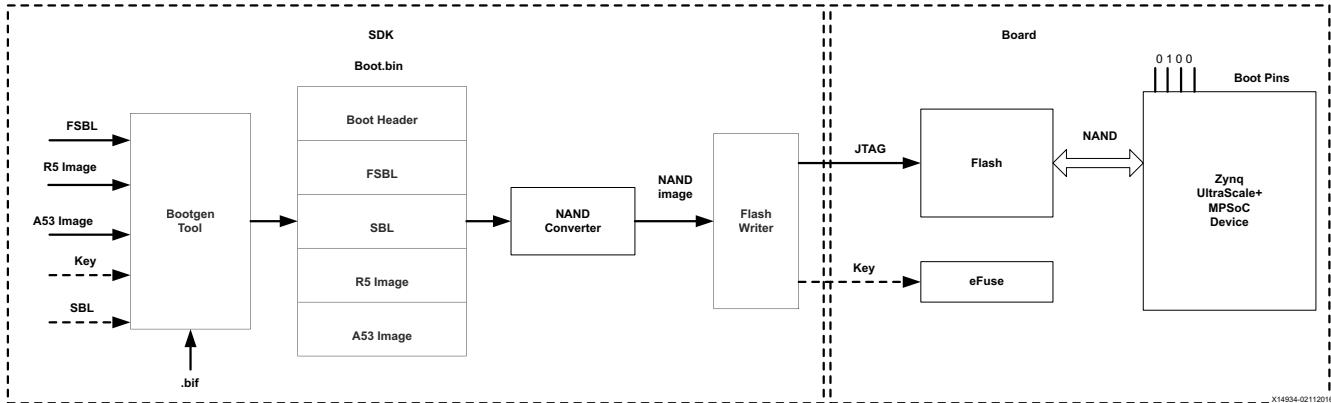


Figure 7-7: **Booting in NAND Mode**

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

- An FSBL ELF
- A Cortex-R5 application ELF and/or an Cortex-A53 application ELF
- Optional authentication/encryption key

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 the boot pins to 0x4.*

JTAG Boot Mode

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

For JTAG boot mode settings, see this [link](#) in the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [Ref 10]



IMPORTANT: *Secure boot is not supported in the JTAG mode.*

USB Boot Mode

USB boot mode supports USB 2.0. It does not support multiboot, image fallback or XIP. It supports both secure and non-secure boot mode. It is not supported for DDR-less systems.

USB boot mode is disabled by default.

Table 7-1: USB Boot Mode Details

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

USB boot mode requires a host with dfu-utils installed in it. The host and device need to be connected through a USB 3.0 cable. The host must contain one boot.bin for BootROM code, which contains only fsbl.elf and another boot_all.bin for FSBL. On powering on the board in USB bootmode, issue the following commands:

- For Linux:

- `dfu-util -D boot.bin`

This downloads the file to the device, which then runs FSBL.

- `dfu-util -D boot_all.bin`

This downloads the file to the device. FSBL carries out the required processing.

- For Windows:

- `sudo dfu-util.exe -D boot.bin`

This downloads the file to the device, which then runs FSBL.

- `sudo dfu-util.exe -D boot_all.bin`

This downloads the file to the device. FSBL carries out the required processing.

The size limit of boot.bin and boot_all.bin are the sizes of OCM and DDR, available respectively.

Detailed Boot Flow

The platform management unit (PMU) in the Zynq UltraScale+ MPSoC device 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 7-2: 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	Zero 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.

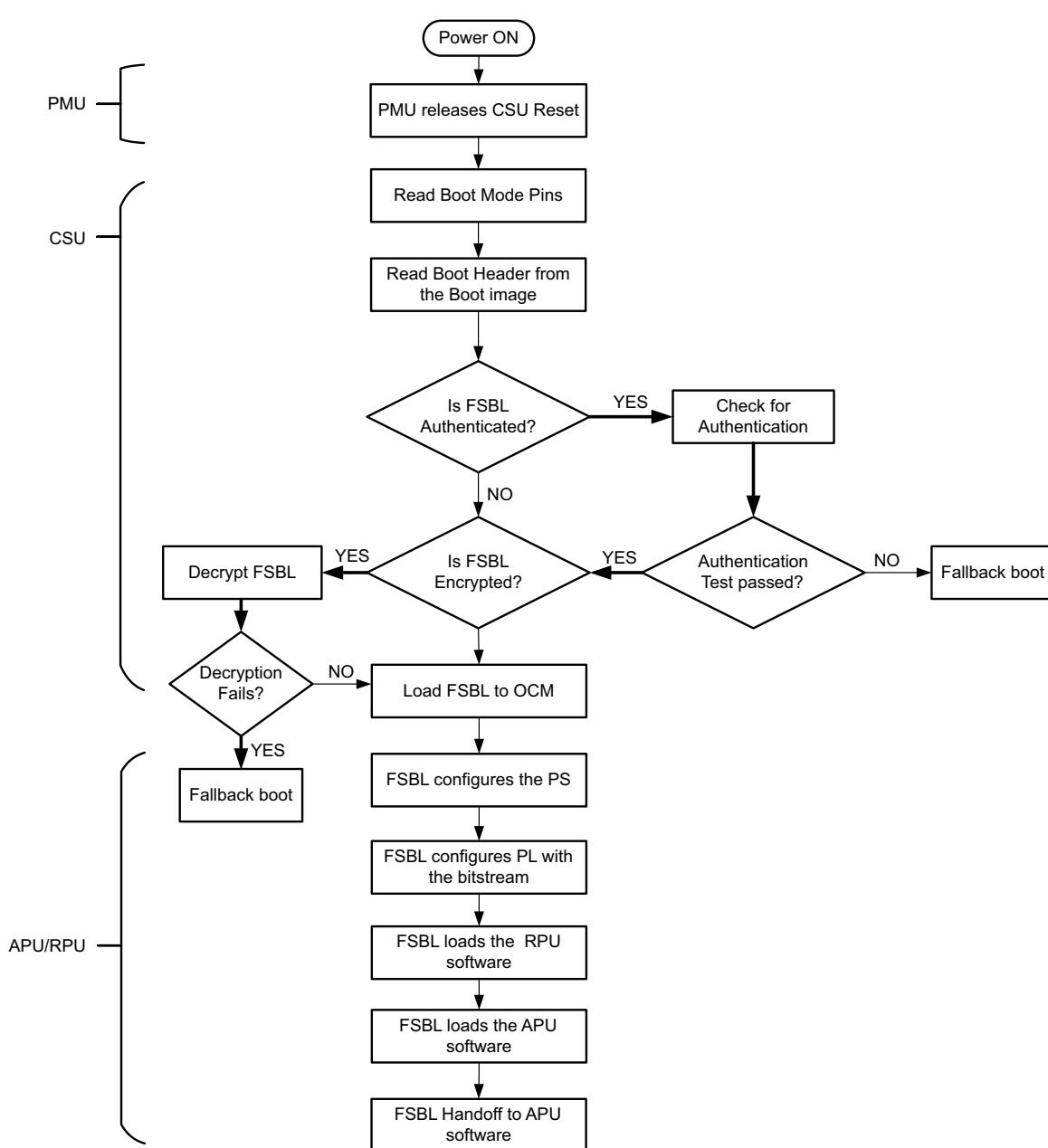
As soon as the CSU reset is released, it executes the CSU bootROM and performs the following sequence:

1. Determines the boot mode by reading the boot mode register, which captures the boot-mode pin strapping at the power on reset (POR).
2. Initializes the OCM.
3. Reads the boot header.

4. If the FSBL boot image must be authenticated, the CSU BootROM authenticates it.

- If the FSBL passes the authentication test, the CSU BootROM checks to determine whether the FSBL is encrypted.
- If FSBL is encrypted, the CSU BootROM decrypts the FSBL, and then loads it into the OCM.

Figure 7-8 shows the detailed boot flow diagram.



X14935-041717

Figure 7-8: Detailed Boot Flow Example

Setting FSBL Compilation Flags

You can set compilation flags using the C/C++ settings in SDK 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.

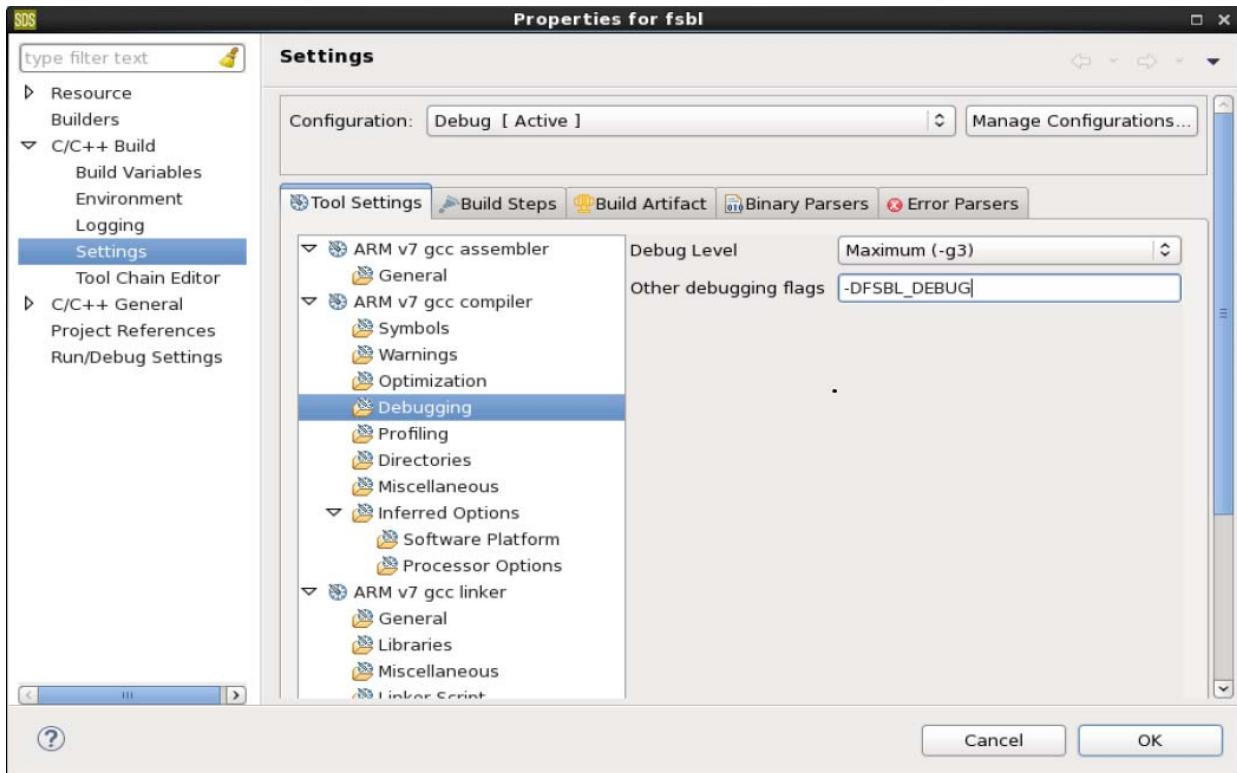


Figure 7-9: FSBL Debug Flags

The following table lists the FSBL compilation flags.

Table 7-3: FSBL Compilation Flags

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

Table 7-3: FSBL Compilation Flags (Cont'd)

Flag	Description
FSBL_AES_EXCLUDE AES	Excludes decryption code.
FSBL_BS_EXCLUDE PL	Excludes bitstream code.
FSBL_SHA2_EXCLUDE SHA2	Excludes SHA2 code.
FSBL_WDT_EXCLUDE WDT	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.

Fallback and MultiBoot Flow

In the Zynq UltraScale+ MPSoC device, the CSU boot ROM supports MultiBoot and fallback boot image search where the configuration security unit (CSU) boot ROM 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 (XILNX as image ID) at offsets of 32KB in the flash.
- After finding a valid identification value, validates the checksum for the header.
- If the checksum is valid, the boot ROM loads the image. This allows for more than one image in the flash.

In MultiBoot:

- 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, FSBL or the user application updates the MultiBoot offset to point the intended boot image, and generates a soft reset by writing into the CRL_APB register.

[Figure 7-10](#) shows an example of the fallback and MultiBoot flow.

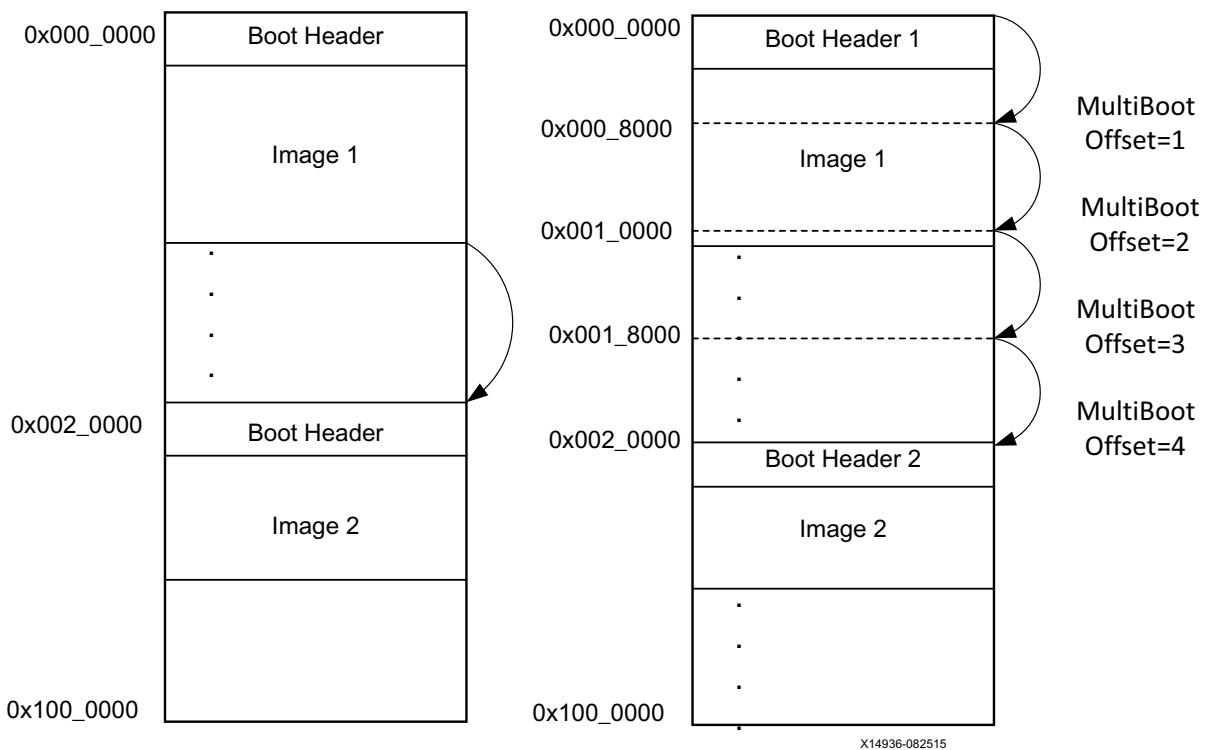


Figure 7-10: MultiBoot Flow (left): Example Fallback (right)

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 boot ROM loads the boot image found at 0x000_0000.
- If this image is found to be corrupted or the decryption and authentication fails, CSU boot ROM increments the MultiBoot offset by 1 and searches for a valid boot image at 0x000_8000 (32 KB offset).
- If the CSU boot ROM 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 boot ROM 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 where the MultiBoot offset is 4.

In the example MultiBoot flow, to load the second image that is at the address 0x002_0000, and set the MultiBoot offset to 4 with the FSBL or the user application. When the MultiBoot offset is updated, soft reset the system.

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

Table 7-4: Boot Devices and Image Search Range

Boot Device	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

Security Features

Introduction

This chapter details the Zynq® UltraScale+™ MPSoC device features that you can leverage to ensure security during boot time and run time of an application. The Secure Boot mechanism is described in detail in this [link](#) to the “Boot and Configuration” chapter of the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [Ref 10].

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

- [Xilinx Memory Protection Unit \(XMPU\)](#)
 - [Xilinx Peripheral Protection Unit \(XPPU\)](#)
 - [System Memory Management Unit \(SMMU\)](#)
 - [A53 Memory Management Unit](#)
 - [R5 Memory Protection Unit](#)
-

Boot Time Security

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

Encryption

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

The AES crypto engine has access to a diverse set of key sources. The key sources are described in [Table 8-1](#).

Table 8-1: Types of Keys

Key Name	Description
BBRAM	The battery backed RAM (BBRAM) key is stored in plain text form in a 256-bit SRAM array. To extend battery life, the SRAM array receives its power from VCCAUX when VCCAUX is powered. Otherwise the SRAM array is powered by VCCBAT.
Boot	The boot key register holds the decrypted key while the key is in use.
eFUSE	The eFUSE key is stored in eFUSES. It can be either plain text, obfuscated (i.e., encrypted with the family key), or encrypted with the PUF KEK.
Family	The family key is a constant AES key value hard-coded into the devices. The same key is used across all devices in the Zynq UltraScale+ MPSoC family. This key is only used by the CSU ROM to decrypt an obfuscated key. The decrypted obfuscated key is used to decrypt the boot images. The obfuscated key can be stored in either eFUSE or the authenticated boot header. Because the family key is the same across all devices, the term obfuscated is used rather than encrypted to reflect the relative strength of the security mechanism.
Operational	The operational key is obtained by decrypting the secure header using a plain text key obtained from the other device key sources. For secure boot, this key is optional. The operational key is specified in the boot header and minimizes the use of the device key, thus limiting its exposure.
PUF KEK	The PUF KEK is a key-encryption key that is generated by the PUF.
Key update register	User provided key source. After boot, a user selected key can be used with the hardened AES accelerator.

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.

BIF File with BBRAM Red Key

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

```
the_ROM_image: {
    [aeskeyfile] bbram.nky
    [fsbl_config] a5x_x64
    [keysrc_encryption] bbram_red_key
    [bootloader, encryption=aes] 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
    [fsbl_config] a5x_x64
    [keysrc_encryption] efuse_red_key
    [bootloader, encryption=aes] 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, the user 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+ MPSoC Technical Reference Manual* [Ref 10].

```
the_ROM_image: {
    [aeskeyfile] bbram.nky
    [fsbl_config] a5x_x64, opt_key
    [keysrc_encryption] bbram_red_key
    [bootloader, encryption=aes] ZynqMP_Fsbl.elf
    [destination_cpu = a53-0, encryption=aes] App_A53_0.elf
}
```

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 we encrypt the actual red key 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. 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] a53_x64, shutter=0x0100005E
    [auth_params] ppk_select=0
    [bootloader, encryption = aes, authentication = rsa] 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
    [keysrccryption] bh_blk_key
    [bh_keyfile] blackkey.txt
    [bh_key_iv] black_key_iv.txt
    [fsbl_config] a5x_x64, pufhd_bh , puf4kmode , shutter=0x0100005E, bh_auth_enable
    [pskfile] PSK.pem
    [sskfile] SSK.pem
    [bootloader,authentication=rsa , encryption=aes ]fsbl.elf
    [puf_file]hlprdata4k.txt
}
```

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

Authentication

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

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

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

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

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



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

To better understand the format of the authentication certificate, see [Authentication Certificate in Chapter 15](#).

The PPK and SPK keys authenticate a partition.

PSK and SSK are used to sign the partition.

The equations for each signature (SPK, boot header, and boot image) are listed here:

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

$$\text{SPK signature} = \text{RSA}(\text{PSK}, \text{padding} \parallel \text{SHA}(\text{SPK} + \text{auth_header})).$$
- Boot header signature. The 512 bytes of the boot header signature is generated by the following calculation:

$$\text{Boot header signature} = \text{RSA}(\text{SSK}, \text{padding} \parallel \text{SHA}(\text{boot header})).$$
- Boot image signature. The 512 bytes of the boot image signature is generated by the following calculation:

$$\text{BI signature} = \text{RSA}(\text{SSK}, \text{padding} \parallel \text{SHA}(\text{PFW} + \text{FSBL} + \text{authentication certificate})).$$

Bootgen supports RSA signature generation only. The modulus, exponentiation and precalculated $R^2 \bmod N$ are required.

Software is supported only for RSA decryption, for decrypting the signature RSA engine requires modulus, exponentiation and pre-calculated $R^2 \bmod N$, all these are extracted from keys.

BIF File with SHA3 Boot Header Authentication and PPK0

The following BIF file sample supports the BH RSA option. This option, documented in the "Security" chapter (available at this [link](#)) of the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [Ref 10], supports integration and test prior to the system being fielded. The BIF file is for SHA3 boot header authentication, where actual PPK hash is not compared with the eFUSE stored value.

```
the_ROM_image: {
    [fsbl_config] a5x_x64, bh_auth_enable
    [auth_params] ppk_select=0; spk_id=0x00000000
    [pskfile] primary_4096.pem
    [sskfile] secondary_4096.pem
    [bootloader, authentication=rsa] fsbl.elf
    [pmufw_image, authentication=rsa] xpfw.elf
}
```

BIF File with SHA3 eFUSE RSA Authenction and PPK0

The following BIF file sample shows eFUSE RSA authentication using PPK0 and SHA3.

```
the_ROM_image:
{
    [fsbl_config]a53_x64
    [auth_params]ppk_select=0;spk_id=0x584C4E58
    [pskfile]psk.pem
    [sskfile]ssk.pem
    [bootloader, authentication = rsa]zynqmp_fsbl.elf
    [destination_cpu = a53-0, authentication = rsa]Application.elf
}
```

Bitstream Authentication Using External Memory

Authentication of bitstream is different from all other partitions. The FSBL can be wholly contained within the OCM, and therefore authenticated and decrypted inside of the device. For the bitstream, the size of the file is so large that it cannot be wholly contained inside the device and external memory must be used. The use of external memory creates a challenge to maintain security because an adversary has 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 8MB blocks and has an authentication certificate for each block.

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

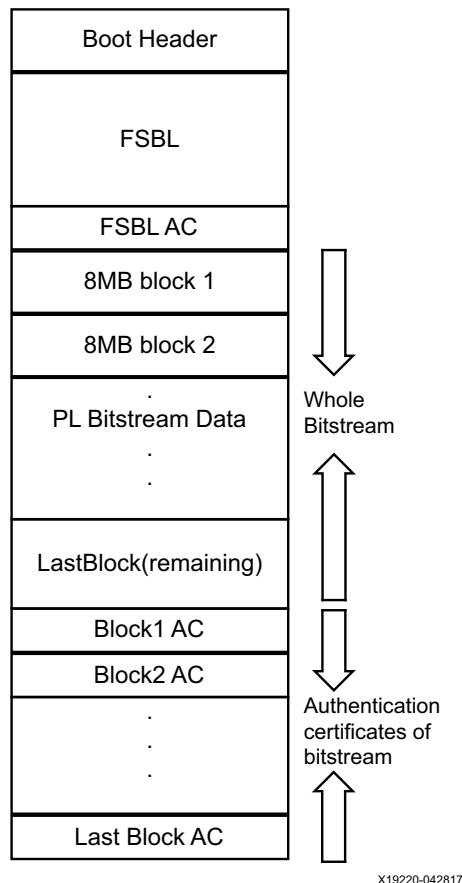


Figure 8-1: Bitstream Blocks

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

Software

To securely authenticate the bitstream partition, FSBL uses the ATF section's OCM memory to copy the bitstream in chunks from FLASH or DDR.

Therefore, while creating a boot image, the bitstream partition should be before ATF partition. Otherwise ATF memory is over-written while processing the bitstream partition.

The software does the following to authenticate the bitstream:

1. FSBL has two buffers in OCM, buffer of size 56KB `ReadBuffer` and another to store intermediate hashes calculated for each 56 KB of 8MB blocks `HashsOfChunks` [].
2. If the system has DDR enabled, copies the entire bitstream partition (bitstream and authentication certificates) to DDR from FLASH devices, because DDR is faster to access.
3. Copies a 56KB chunk from the first 8MB block to `ReadBuffer`.
4. Calculates hash on 56 KB and stores in `HashsOfChunks`.
5. Repeats the previous steps until the entire 8MB of block is completed.

Note: 56KB is taken for performance; it can be of any size.

6. Authenticates the bitstream.
7. Once authentication is successful, starts copying 56KB from starting of the first block which is located at DDR/flash to `ReadBuffer` and calculates the hash, then compares it with the hash stored at `HashsOfChunks`.
8. If hash comparison is successful, transmits data to PCAP via DMA (for unencrypted bitstream) or AES (if encryption is enabled).
9. Repeat the previous two steps until the entire 8MB block is completed.
10. 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 sectio) in chunks.

Note: Xilinx recommends that you have a bitstream partition immediately after 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+ MPSoC devices 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
-

ARM Trusted Firmware

The Zynq UltraScale+ MPSoC device incorporates the standard execution model advocated for ARMv8 cores. This model runs the normal operating system at a less privileged state, requiring it to request access to security-sensitive hardware or registers using a proxy software called as secure monitor code (SMC). The specific SMC provided by Xilinx for the Zynq UltraScale+ MPSoC device is a part of Linaro ARM Trusted Firmware (ATF) which incorporates both the secure world and the non-secure world. The secure application runs on a trusted OS. ATF includes a secure monitor for switching between the secure and the non-secure world.

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

PSCI is the interface from non-secure software to firmware implementing power management use-cases (for example, secondary CPU boot, hotplug, and idle).

It might be necessary for supervisory systems running at exception levels to perform actions, such as restoring context and switches to the power state of core.

Non-secure software can access ATF runtime services using the ARM secure monitor call (SMC) instruction.

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

Two types of calls are defined:

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

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

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

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

- Call type
- Calling convention
- Secure function to invoke

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

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

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

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

The following table lists the ATF APIs.

Table 8-2: ATF APIs

ATF APIs	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>cm_get_context_by_mpidr()</code>	Returns a pointer to the most recent <code>cpu_context</code> structure for the CPU identified by MPIDR that was set as the context for the specified Security state. NULL is returned if no such structure has been specified.
<code>get_scr_el3_from_routing_model()</code>	Returns the cached copy of the SCR_EL3 which contains the routing model (expressed through the IRQ and FIQ bits) for a security state that is stored through a previous call to <code>set_routing_model()</code> .
<code>get_el3state_ctx()</code>	Populates EL3 state so that ERET jumps to the correct entry.
<code>get_gpregs_ctx()</code>	Stores the X0-X7 value from the entry point into the context.

Table 8-2: ATF APIs (Cont'd)

ATF APIs	Description
<code>cm_prepare_el3_exit()</code>	Prepares the CPU system registers for first entry into the secure or the non-secure software world. <ul style="list-style-type: none"> If execution is requested to EL2 or hyp mode, <code>SCTLR_EL2</code> is initialized. If execution is requested to the non-secure EL1 or svc mode, and the CPU supports EL2; then EL2 is disabled by configuring all necessary EL2 registers. For all entries, the EL1 registers are initialized from the <code>cpu_context</code> .
<code>cm_get_context(security_state);</code>	Gets the context of the security state.
<code>el1_sysregs_context_restore</code>	Restores the context of the system registers.
<code>cm_set_next_context</code>	Programs the context used for exception return. This initializes the <code>SP_EL3</code> to a pointer to a <code>cpu_context</code> set for the required security state.
<code>bl31_late_platform_setup();</code>	Sets up the platform.
<code>bl31_register_bl32_init</code>	Initializes the pointer to <code>BL32 init</code> function.
<code>bl31_set_next_image_type</code>	Accessor function to help runtime services determine which image to execute after <code>BL31</code> .

For more information about ATF, see the *ARM ATF* link [\[Ref 37\]](#).

Xilinx Memory Protection Unit

The Xilinx® memory protection unit (XMPU) is a region-based memory protection unit. See this [link](#) to the “System Protection Unit” chapter in the of the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [\[Ref 10\]](#).

Protecting Memory with XMPU

To understand more about XMPU features and functionality, see the this [link](#) to the “System Protection Unit” chapter of the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [\[Ref 10\]](#).

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.

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+ MPSoC Technical Reference Manual* (UG1085) [Ref 10].

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+ MPSoC Technical Reference Manual* (UG1085) [Ref 10].

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+ MPSoC Technical Reference Manual* (UG1085) [Ref 10].

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+ MPSoC Technical Reference Manual* (UG1085) [\[Ref 10\]](#).

Introduction to the 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 (IIEEP2415) and is able to handle all requests coming from multiple CPUs running different operative 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 cases scenarios. For example, Linux 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, 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
 - Management memories and peripherals

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.

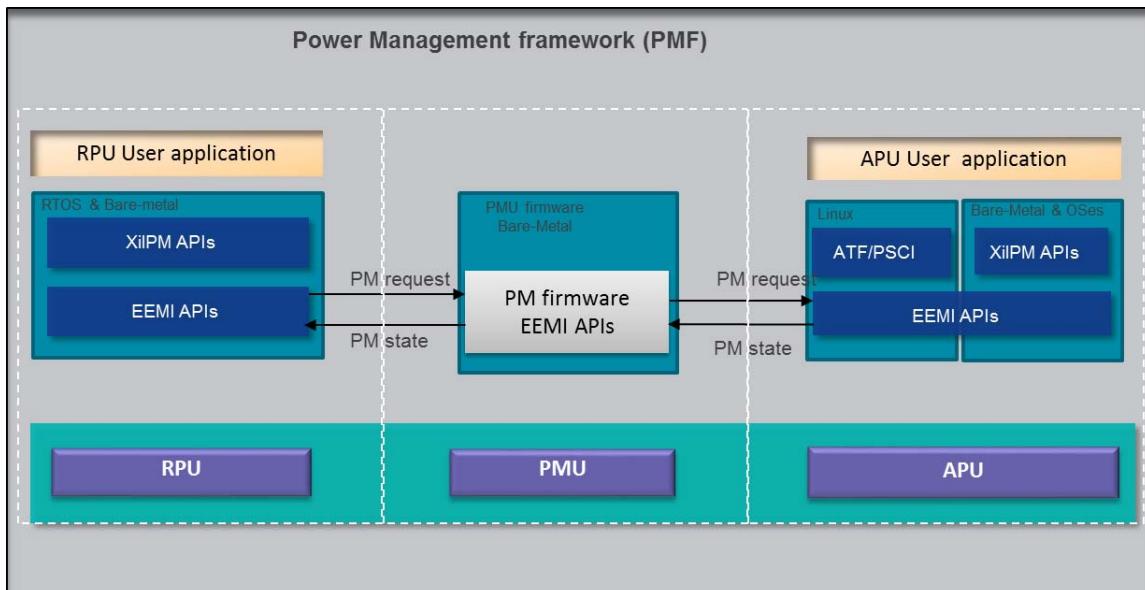


Figure 9-1: Power Management Framework

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 the user 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.

Users can also create their own power management applications using the XiIPM library, which provides access to more than 24 APIs.

Zynq UltraScale+ MPSoC Power Management Overview

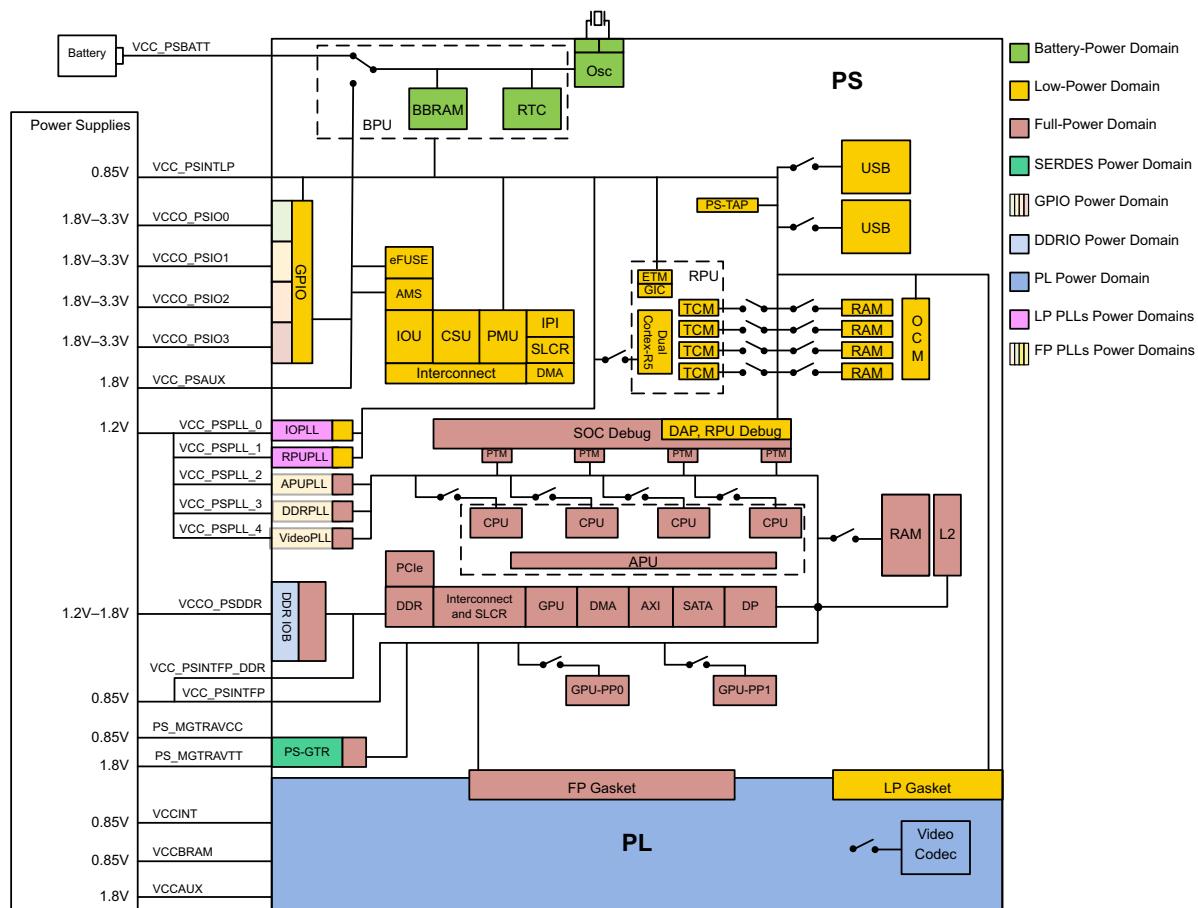
The Zynq UltraScale+ MPSoC power management framework is a set of power management options, based upon an implementation of the extensible energy management interface (EEMI). The power management framework allows software components running across different processing units (PUs) on a chip or device to issue or respond to requests for power management.

Zynq UltraScale+ MPSoC Power Management Hardware Architecture

The Zynq UltraScale+ MPSoC device 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-R5 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. The following is a diagram of the Zynq UltraScale+ MPSoC device power domains and islands.



X16958-101616

Figure 9-2: Zynq UltraScale+ MPSoC Power Domain and Islands

Because of the heterogeneous multi-core architecture of the Zynq UltraScale+ MPSoC device, 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.*

The Zynq UltraScale+ MPSoC device 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+ MPSoC Technical Reference Manual* (UG1085) [\[Ref 10\]](#) 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 device provides an implementation of the power management API for managing heterogeneous multiprocessor systems.

The following figure illustrates the API-based power management software architecture.

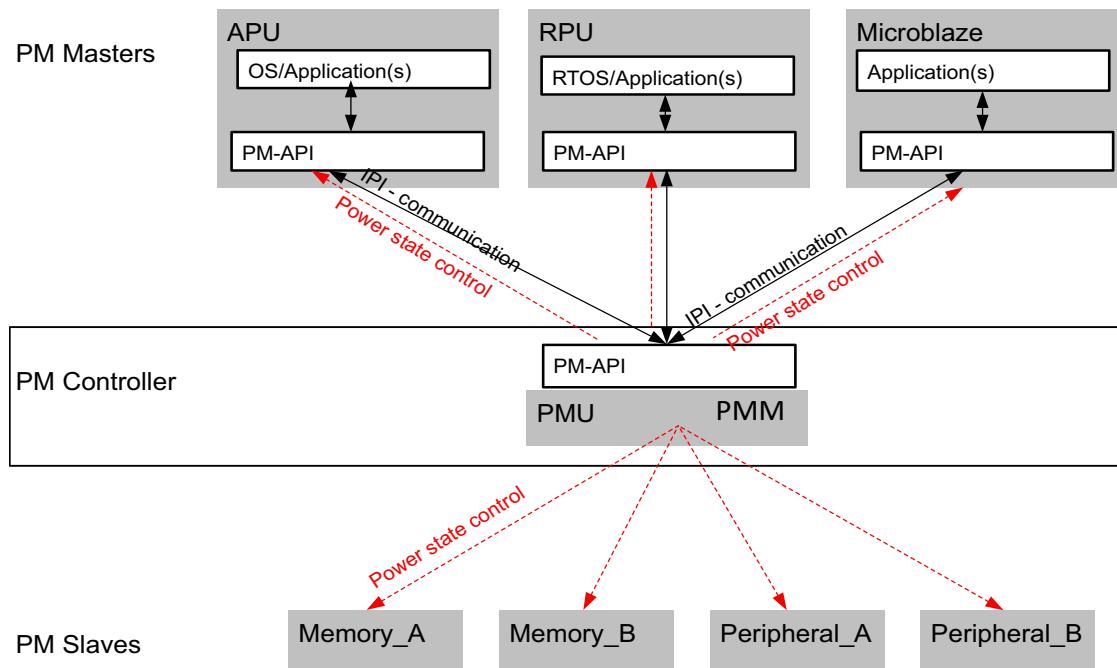


Figure 9-3: **API-Based Power Management Software Architecture -**

Power Management Framework Overview

The Zynq UltraScale+ MPSoC device power management framework (PMF) is based on an implementation of EEMI, see the *Embedded Energy Management API Specification* (UG1200) [Ref 15]. 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 device, 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+ MPSoC Technical Reference Manual* (UG1085) [Ref 10] for more details on IPIs.

Each PU has a dedicated IPI channel with the power management controller, consisting of an interrupt and a payload buffer. The buffer passes the API ID and up to five arguments. The IPI interrupt to the target triggers the processing of the API, as follows:

- When calling an API function, a PU generates an IPI to the power management unit (PMU), prompting the execution of necessary power management action.
- The PMU performs each PM action atomically, meaning that the action cannot be interrupted.
- To support PM callbacks, which are used for notifications from the PMU to a PU, each PU implements handling of these callback IPIs.

Acknowledge Mechanism

The Zynq UltraScale+ MPSoC power management framework (PMF) supports blocking and non-blocking acknowledges. In most API calls that offer an acknowledge argument, the caller can choose one of the following three acknowledge options:

- REQUEST_ACK_NO: No acknowledge requested
- REQUEST_ACK_BLOCKING: Blocking acknowledge requested
- REQUEST_ACK_NON_BLOCKING: Non-blocking acknowledge using callback requested

Multiple power management API calls are serialized because each processor unit (PU) uses a single IPI channel for the API calls. After one request is sent to the power management controller, the next one can be issued only after the power management controller has completed servicing the first one. Therefore, no matter which acknowledge mechanism is used, the caller can be blocked when issuing subsequent requests.

No Acknowledge

If no acknowledge is requested (REQUEST_ACK_NO), the power management controller processes the request without returning an acknowledge to the caller, otherwise an acknowledgment is sent.

Blocking Acknowledge

After initiating a PM request with the `(REQUEST_ACK_BLOCKING)` specified, a caller remains blocked as long as the power management controller does not provide the acknowledgment.

The platform management unit (PMU) writes the acknowledge values into the response portion of the IPI buffer before it clears the IPI interrupt. The caller reads the acknowledge values from the IPI buffer after the IPI observation register shows that the interrupt is cleared, which is when PMU has completed servicing the issued IPI. The IPI for the PU is disabled until the PMU is ready to handle the next request.

Non-Blocking Acknowledge

After initiating a PM request with the `(REQUEST_ACK_NON_BLOCKING)` specified, a caller does not wait for the platform management unit (PMU) to process that request. Moreover, the caller is free to perform some other activities while waiting for the acknowledge from the PMU.

After the PMU completes servicing the request, it writes the acknowledge values into the IPI buffer. Next, the PMU triggers the IPI to the caller PU to interrupt its activities, and to inform it about the sent acknowledge.

Non-blocking acknowledges are implemented using a callback function that is implemented by the calling PU, see `XPm_NotifyCb` Callback.

For more information about `XPm_NotifyCb`, see [Appendix I, XilPM Library v2.1](#).

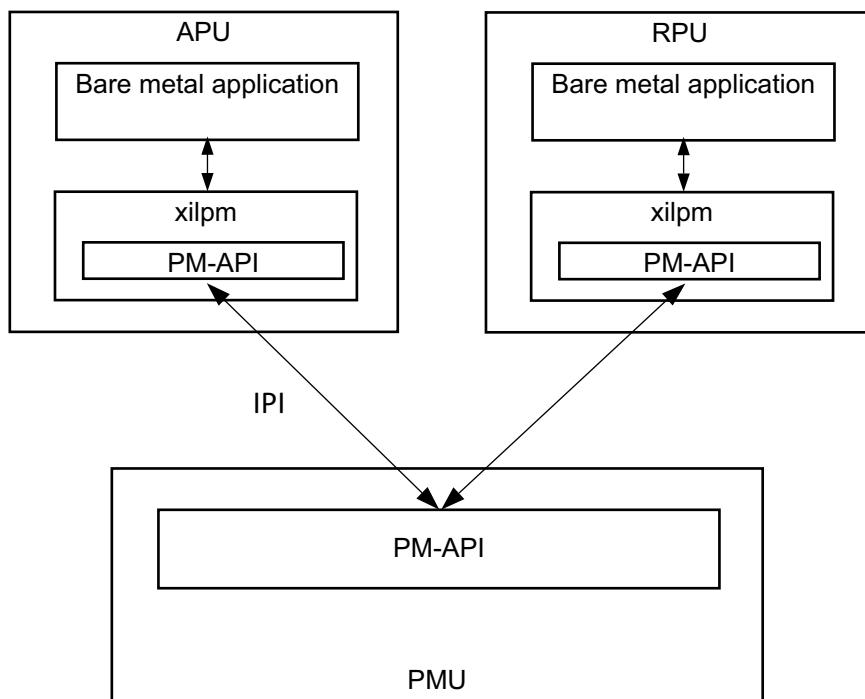
Power Management Framework Layers

There are different API layers in the power management framework (PMF) implementation for Zynq UltraScale+ MPSoC devices, 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.
- **PMUFW:** 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+ MPSoC Technical Reference Manual* (UG1085) [Ref 10]

The following figure shows the interaction between the APU, the RPU, and the PMF APIs.



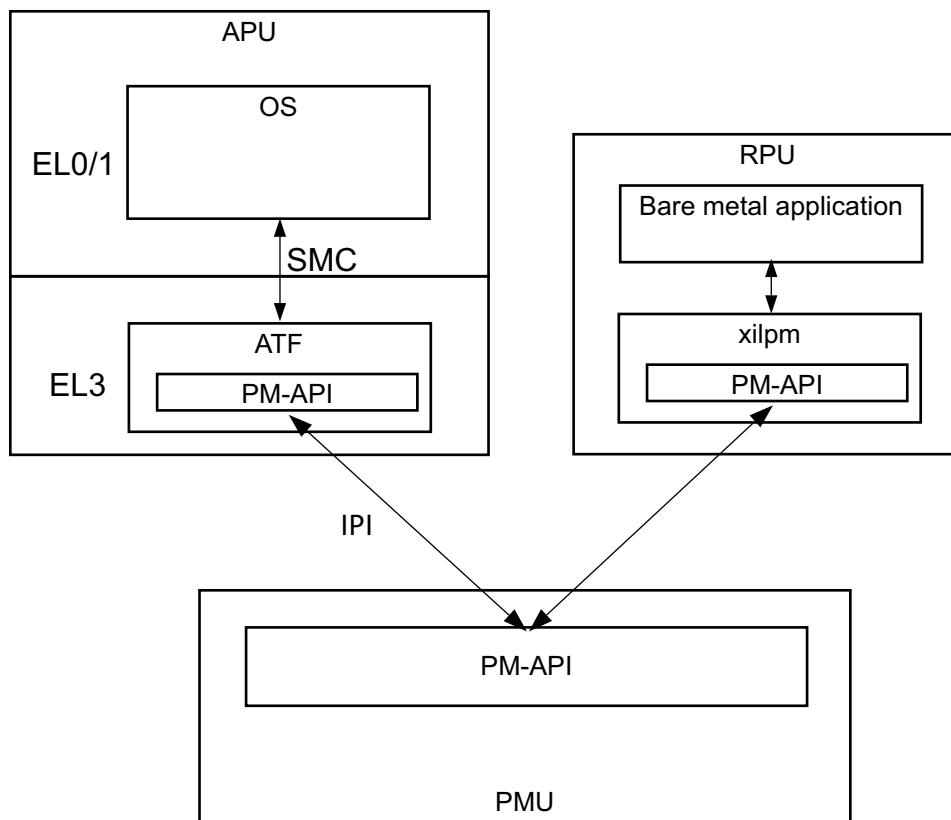
X19094-041717

Figure 9-4: API Layers Used with Bare-Metal Applications Only

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* [Ref 42] for more details on the ARMv8 architecture and its different execution modes.

The following figure illustrates the PMF layers that are involved when running a full software stack on the APU.



X19093-041717

Figure 9-5: PM Framework Layers Involved When Running a Full Software Stack on the APU

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.

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 must be released, allowing the slave node to be powered off.

The following figure shows the call flow for a use-case in which the APU and the RPU are sharing an OCM memory bank, ocm0.

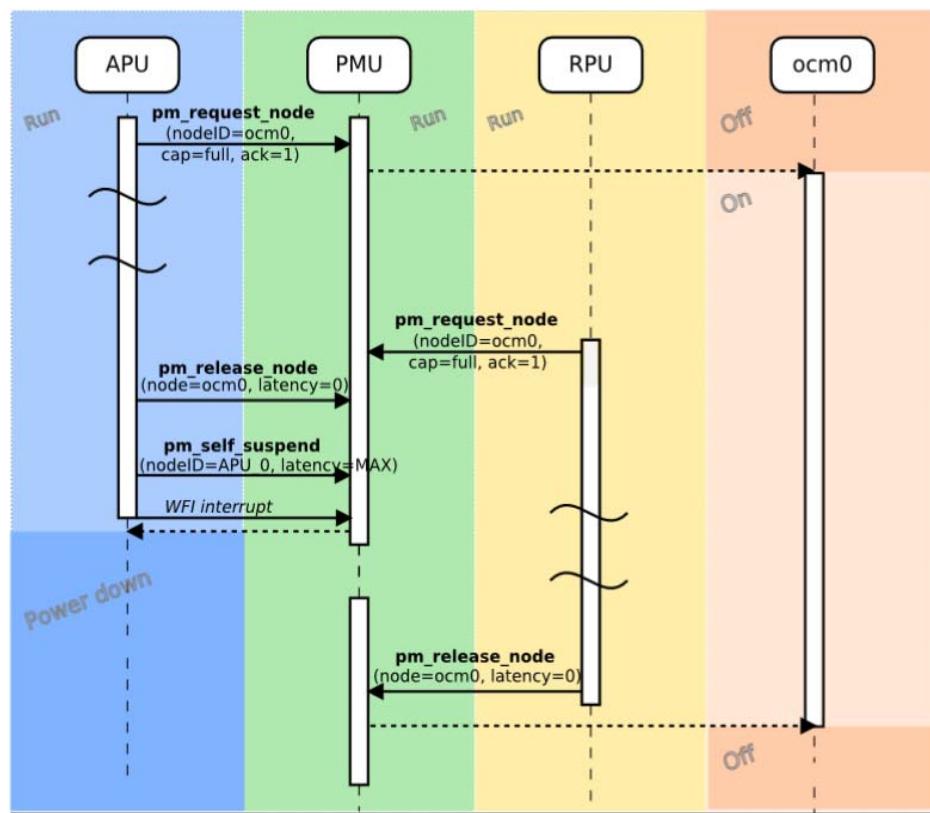


Figure 9-6: PM Framework Call Sequence for APU and RPU Sharing an OCM Memory Bank

Note: The ocm0 memory remains powered on after the APU calls `xStatus = XPM_ReleaseNode`, because the RPU has also requested the same slave node. It is after the power management unit (PMU) also releases the ocm0 node that the PMU powers off the ocm0 memory.

Processor Unit Suspend and Resume

To allow a processor unit (PU) to be powered off, as opposed to just entering an idle state, an external entity is required to take care of the power-down and power-up transitions.

For the Zynq UltraScale+ MPSoC device, the platform management unit (PMU) is the responsible entity for performing all power state changes.

The processor unit (PU) notifies the PMU that a power state transition is being requested. The following figure illustrates the process.

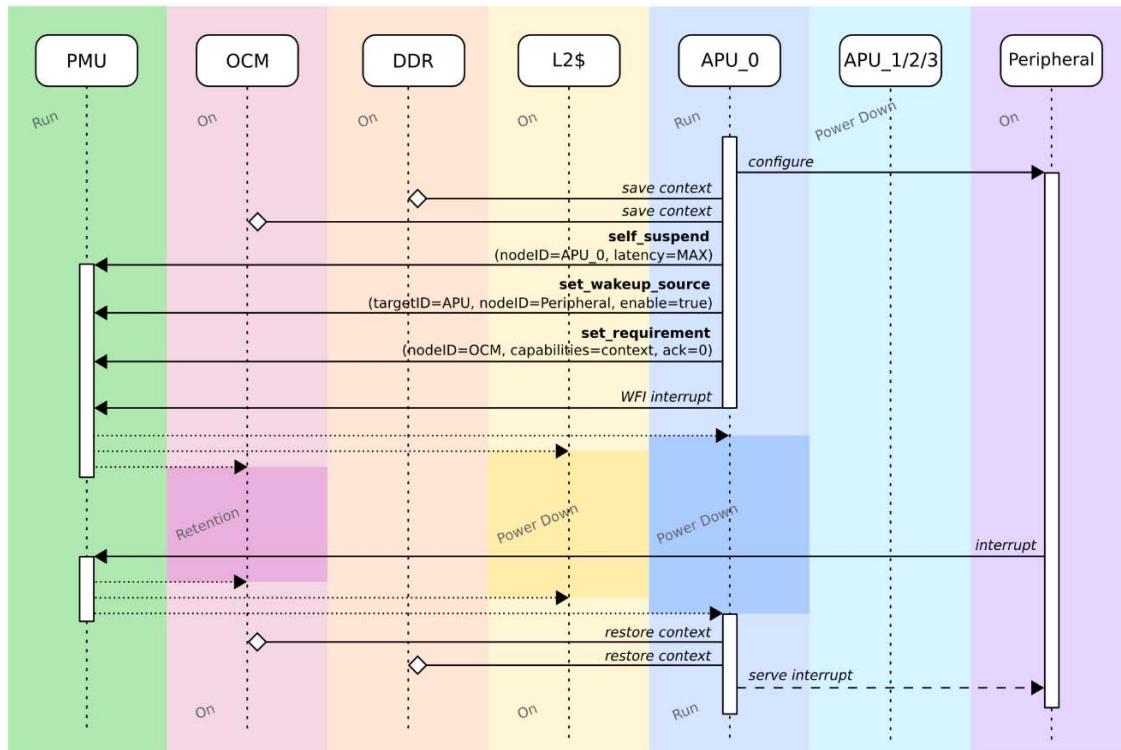


Figure 9-7: APU Suspend and Resume Procedure

The [Self-Suspending a CPU/PU](#) section details the suspend or resume procedure. Each PU depends on a slave node to be able to operate.

Access Rights

To prevent a processing unit (PU) from making a request to a peripheral that it is not supposed to access, the PMF includes a default assignment of peripherals (slave nodes) to each PU, as described in the following sections. See [Node IDs: XPMNODEID](#) for basic descriptions of the nodes.

Default Resources

Each PU depends on a set of slave nodes to be able to operate the PU. Each PU has a number of slave nodes required for the PU to be able to resume operation. The power management unit (PMU) pre-allocates those resources to the respective PU prior to waking the PU. For example, the memory nodes from which a CPU starts fetching instructions after being reset must be accessible before the CPU starts execution. The default requirements are, as follows:

- APU default requirements: L2 Cache, DDR
 - RPU default requirements: All TCM memory banks
-

Using the API for Power Management

Introduction

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 interface that the software executing on a PU can use to initiate the power management API calls. To make the API calls, the software executing on a processor unit (PU) needs to use the Xilpm library.

See the *Xilinx Software Developer Kit Help* (UG782) [\[Ref 21\]](#) 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+ MPSoC Technical Reference Manual* (UG1085) [\[Ref 10\]](#). for more information regarding IPIs.

For more information about `XPm_InitXilpm`, see [Appendix I, XilPM Library v2.1](#).

Working with Slave Devices

The Zynq® UltraScale+™ MPSoC device 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 ensures that at all times 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 access to a slave device and assert its requirements on that device. Provided the PU is allowed to access the slave (see [Access Rights](#)), the power management controller ensures the requested device is powered on and placed into its active state. For devices that can only be serving a single PU, any other PU is then blocked from requesting this device.

The power management controller assigns access permissions to one or multiple PUs for each PM slave. When a PU requests a PM slave, the power management controller checks the privilege configuration to determine if the PU is allowed to use the requested PM slave. You must reconfigure the privilege assignment for the power management controller in source code.

After a device is no longer used, the PU typically calls the `XPm_RequestNode` function to allow the PM controller to re-evaluate the power state of that device, and potentially place it into a low-power state. It also then allows other PUs to request that device.

For more information about `XPm_RequestNode`, see [Appendix I, XilPM Library v2.1](#).

Changing Requirements

During use, a PU can change the requirements it asserts on the capability of a PM slave by using the `XPM_SetRequirement` API. Typically, a request for node use occurs when a device is no longer actively used; however, the appropriate wake-interrupts or preservation of the context of the node must be issued.

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);
```

At some point, the PU could have no requirements, and then it has the option to release the PM slave or to set its requirements to zero.



IMPORTANT: *Setting requirements of a node to zero is not equivalent to releasing the PM slave, because by releasing the PM slave, a PU releases its access rights, potentially allowing other PUs to use this device*

If the requirements are set to zero, the power management controller can place the device into a lower-power state while still keeping the access rights of the PU in place.

When multiple PUs share a PM slave (this applies mostly to memories), the power management controller configures a power state of the PM slave that satisfies all requirements of the requesting PUs.

For more information about `XPM_SetRequirement`, see [Appendix I, XilPM Library v2.1](#).

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.

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 ensure such behavior in the case of the APU and RPU, after the `XPM_SelfSuspend()` call has completed, all of the interrupts to a CPU which redirects the interrupts 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 can 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 redirect to the power management controller (PMC) as a GIC wake interrupt assigned to that particular CPU. Because the interrupts are redirected, the CPU can only be woken up using the power management controller.
- Suspending a PU requires suspending all of its CPUs individually.

For more information about `XPM_SelfSuspend` and `XPM_SuspendFinalize`, see [Appendix I, XilPM Library v2.1](#).

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_RequestSuspend` call.

For more information about `XPM_RequestWakeup` and `XPM_RequestSuspend`, see [Appendix I, XilPM Library v2.1](#).

Setting up a Wake-up Source

The PM controller can power down the entire FPD if none of the FPD devices are in use and existing latency requirements allow this action. If the FPD is powered off and the APU is to be woken up by an interrupt triggered by a device in the LPD, the GIC Proxy must be configured to allow propagation of FPD wake events. The APU can ensure this by calling `XPM_SetWakeUpSource` for all devices that might need to issue wake interrupts.

Hence, prior to suspending, the APU must call `XPM_SetWakeUpSource(NODE_APU, node, 1)` to add the required slaves as a wake-up source. The APU can then set the requirements to zero for all slaves it is using. After the APU finalizes its suspend procedure, and provided that no other PU is using any resource in the FPD, the PM controller powers off the entire FPD and configures the GIC proxy to enable propagation of the wake event of the LPD slaves.

For more information about `XPM_SetWakeUpSource`, see [Appendix I, XilPM Library v2.1](#).

Aborting a Suspend Procedure

If a PU decides to abort the suspend procedure after calling the `XPM_SetSelfSuspend` function, it must inform the power management controller about the aborted suspend by calling the `XPM_AbortSuspend` function.

For more information about `XPM_SetSelfSuspend` and `XPM_AbortSuspend`, see [Appendix I, XilPM Library v2.1](#).

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 to 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; such as after calling `XPM_SetSelfSuspend`, the setting is deferred until the CPU finishes the suspend. This

deference ensures that devices that are needed for completing the suspend procedure can enter a low power state after the calling CPU finishes suspend.

A common example is instruction memory, which a CPU can access until the end of a suspend. After the CPU suspends a memory, that memory can be placed into retention. All deferred requirements reverse automatically before the respective CPU is woken up.

When an entire PU suspends, the last awake CPU within the PU must manage the changes to the devices.

For more information about XPM_SelfSuspend, see [Appendix I, XilPM Library v2.1](#).

Example Code for Suspending an APU/RPU

There the following is an example of source code for suspending the APU or RPU:

```
/* Base address of vector table (reset-vector) */
extern void *_vector_table;
/* Inform PM controller that APU_0 intends to suspend */
XPm_SelfSuspend(NODE_APU_0, MAX_LATENCY, 0,
(u64)&_vector_table);
/**
 * Set requirements for OCM banks to preserve their context.
 * The PM controller will defer putting OCMs into retention
until the suspend is finalized
 */
XPm_SetRequirement(NODE_OCM_BANK_0, PM_CAP_CONTEXT, 0,
REQUEST_ACK_NO);
XPm_SetRequirement(NODE_OCM_BANK_1, PM_CAP_CONTEXT, 0,
REQUEST_ACK_NO);
XPm_SetRequirement(NODE_OCM_BANK_2, PM_CAP_CONTEXT, 0,
REQUEST_ACK_NO);
XPm_SetRequirement(NODE_OCM_BANK_3, PM_CAP_CONTEXT, 0,
REQUEST_ACK_NO);

/* Flush data cache */
Xil_DCacheFlush();
/* Inform PM controller that suspend procedure is completed */
XPm_SuspendFinalize();
```

Suspending the Entire FPD Domain

To power-down the entire full power domain, the power management controller must suspend the APU at a time when none of the FPD devices is in use. After this condition is met, the power management controller can power-down the FPD automatically. The power management controller powers down the FPD if no latency requirements constrain this action, otherwise the FPD remains powered on.

Forcefully Powering Down the FPD

There is the option to force the FPD to power-down by calling the function `XPM_ForcePowerdown`. This requires that the requesting PU has proper privileges configured in the power management controller. The power management controller releases all PM Slaves used by the APU automatically.

Note: This force method is typically not recommended, especially when running complex operating systems on the APU because it could result in loss of data or system corruption, due to the OS not suspending itself gracefully.



IMPORTANT: Use the `XPM_RequestSuspend` API.

For more information about `XPM_ForcePowerdown`, see [Appendix I, XilPM Library v2.1](#).

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_CB_STANDARD, MAX_LATENCY, 0);
```

The following diagram shows the general sequence triggered by a call to the `XPM_RequestSuspend`.

For more information about `XPM_RequestSuspend`, `XPM_InitSuspendCb`, and `XPM_AbortSuspend`, see [Appendix I, XilPM Library v2.1](#).

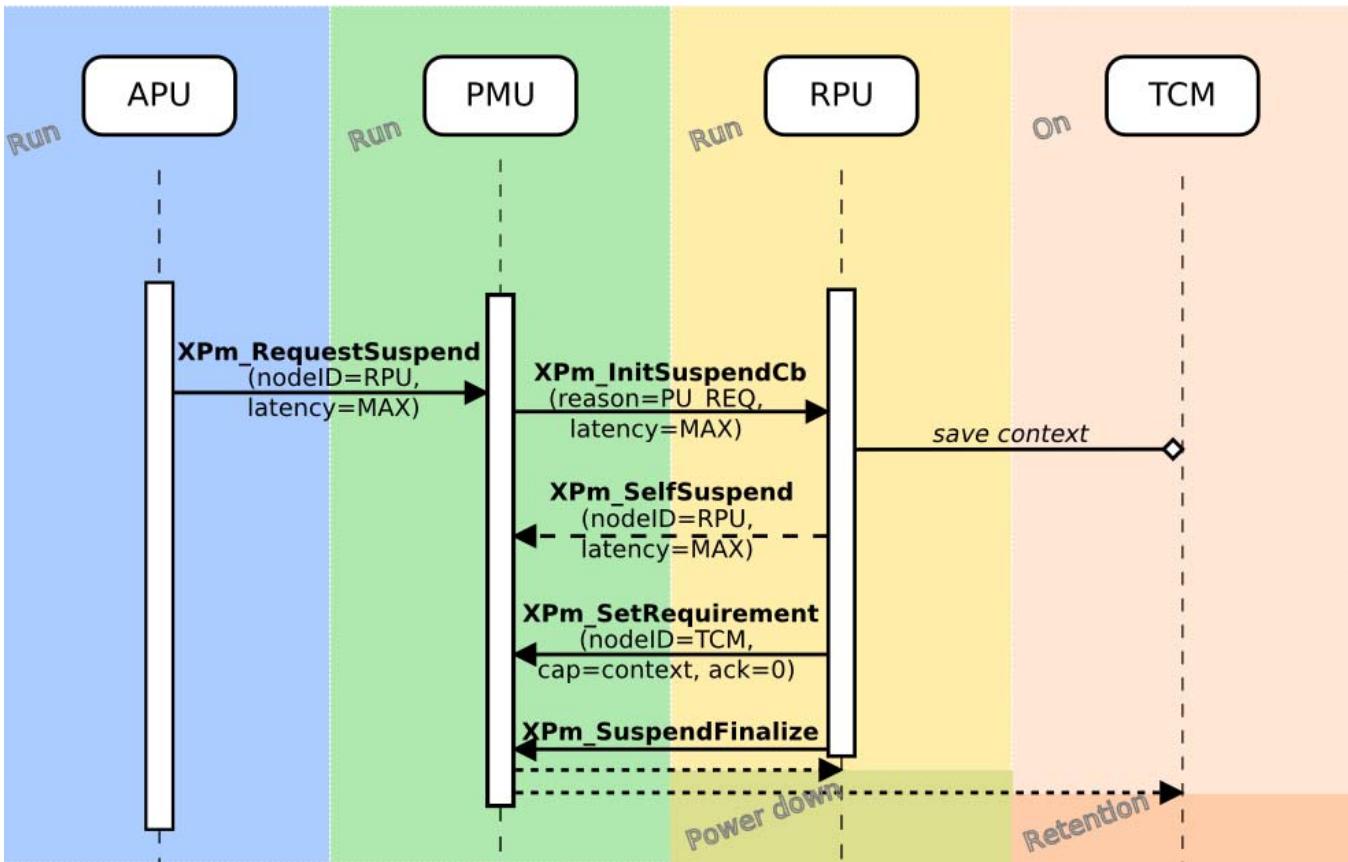


Figure 9-8: APU initiating suspend for the RPU by calling **XPm_RequestSuspend**

Waking a PU

Additionally, a PU can request the wake-up of one of its CPUs or of another PU by calling **XPm_RequestWakeup**.

- When processing the call, the power management controller causes a target CPU or PU to be awakened.
- If a PU is the target, only one of its CPUs is woken-up by this request.
- The CPU chosen by the power management controller is considered the primary CPU within the PU.

The following is an example of a wake-up request:

```
XPm_RequestWakeup(NODE_APU_1, REQUEST_ACK_NO);
```

For more information about **XPm_RequestWakeup**, see [Appendix I, XilPM Library v2.1](#).

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. For more information about the acknowledge see section 1.3.2.2.2.

Payload mapping for API calls to PMU

Each EEMI API call is uniquely identified by the following data:

- EEMI API identifier (ID)
- EEMI API arguments

Please see Appendix A for a list of all API identifiers as well as API argument values.

Prior to initiating an IPI to the PMU, the PU shall write the information about the call into the IPI request buffer. Each data written into the IPI buffer is a 32-bit word. Total size of the payload is six 32-bit words - one word is reserved for the EEMI API identifier, while the remaining words are used for the arguments. Writing to the IPI buffer starts from offset zero. The information is mapped as follows:

- Word [0]EEMI API ID
- Word [1:5]EEMI API arguments

The IPI response buffer is used to return the status of the operation as well as up to 3 values.

- Word [0]success or error code
- Word [1:3]value 1..3

Payload mapping for API callbacks from the PMU

The EEMI API includes callback functions, invoked by the PM controller, sent to a PU.

- Word [0]EEMI API Callback ID
- Word [1:5]EEMI API arguments

Refer to [Appendix I, XilPM Library v2.1](#) 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.

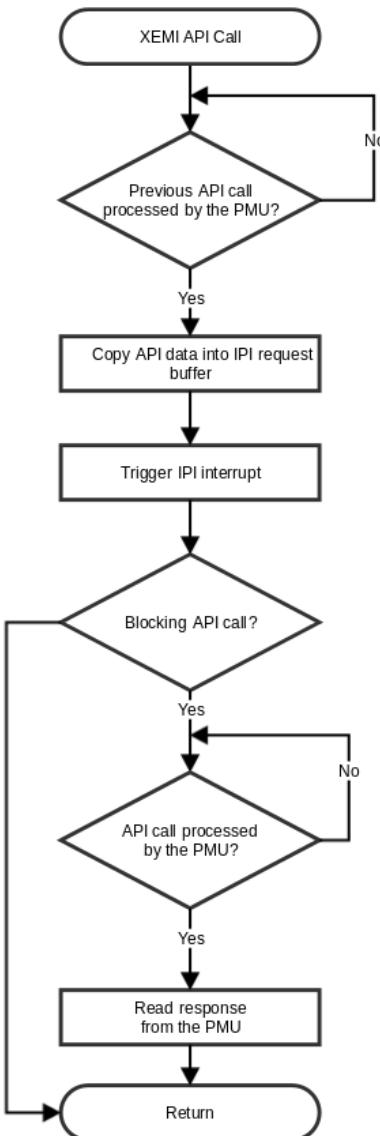


Figure 9-9: Example Flow of Issuing API Call to the PMU

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 accessed 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

The user 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

The user 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.

```
echo release_node 69 > /sys/kernel/debug/zynqmp_pm/power
```

Use this command to power up the PL again:

```
echo request_node 69 > /sys/kernel/debug/zynqmp_pm/power
```

For information about how to shut down the PL sub-system, see the *Zynq UltraScale+ MPSoC OpenAMP: Getting Started Guide* (UG1186) [Ref 13].

Reboot

The user can use the `reboot` command to reset the APU, the PS or the System. By default, the `reboot` command resets the System.

The user can change the scope of the `reboot` command to APU or PS if required.

To change the reboot scope to APU:

```
echo system_shutdown 2 0 > /sys/kernel/debug/zynqmp_pm/power
```

To change the reboot scope to PS:

```
echo system_shutdown 2 1 > /sys/kernel/debug/zynqmp_pm/power
```

To change the reboot scope to System:

```
echo system_shutdown 2 2 > /sys/kernel/debug/zynqmp_pm/power
```

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 Zynq MPSoC driver support

Note that 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/ttys0/power/wakeup
```

- RTC

If enabled as a wake-up source, the kernel will resume from the suspend mode when the RTC timer expires.

Kernel configurations required:

- Same as [Suspend](#).

For example, up the 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.

Kernel configurations required:

- 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) -> see note below
- 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 Freq

If enabled, the CPU cores may switch between different operation clock frequencies.

Kernel configurations required:

- 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
```

Power Management for the Devices

Clock Gating

Stop device clocks when they are not being used (also called Common Clock Framework.)

Kernel configurations required:

- 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.

Kernel configurations required:

- Power management options
 - [*] Suspend to RAM and standby
- Device Drivers
 - SOC (System On Chip) specific Drivers
 - [*] Xilinx Zynq MPSoC driver support

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.

Kernel configurations required (in this order):

- Kernel hacking
 - Compile-time checks and compiler options
 - [*] Debug Filesystem
- Device Drivers
 - SOC (System On Chip) specific Drivers
 - [*] Enable Xilinx Zynq MPSoC Power Management API debugfs functionality
 - [*] Xilinx Zynq MPSoC driver support

The user may invoke any EEMI API except for: [TODO: Need to confirm]

- 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 4 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 "REQUEST_NODE 22 1 100 1" > /sys/kernel/debug/zynqmp_pm/power
```

Command List

Get API Version

Get the API version.

```
$ echo get_api_version > /sys/kernel/debug/zynqmp_pm/power
```

Request Suspend

Request another PU to suspend itself.

```
$ echo request_suspend <node> > /sys/kernel/debug/zynqmp_pm/power
```

Self Suspend

Notify PMU that this PU is about to suspend itself.

```
$ echo self_suspend <node> > /sys/kernel/debug/zynqmp_pm/power
```

Force Power Down

Force another PU to power down.

```
$ echo force_powerdown <node> > /sys/kernel/debug/zynqmp_pm/power
```

Abort Suspend

Notify PMU that the attempt to suspend has been aborted.

```
$ echo abort_suspend > /sys/kernel/debug/zynqmp_pm/power
```

Request Wake-up

Request another PU to wake up from suspend state.

```
$ echo request_wakeup <node> <set_address> <address> >  
/sys/kernel/debug/zynqmp_pm/power
```

Set Wake-up Source

Set up a node as the wake-up source.

```
$ echo set_wakeup_source <target> <wkup_node> <enable> >  
/sys/kernel/debug/zynqmp_pm/power
```

Request Node

Request to use a node.

```
$ echo request_node <node> > /sys/kernel/debug/zynqmp_pm/power
```

Release Node

Free a node that is no longer being used.

```
$ echo release_node <node> > /sys/kernel/debug/zynqmp_pm/power
```

Set Requirement

Set the power requirement on the node.

```
$ echo set_requirement <node> <capabilities> > /sys/kernel/debug/zynqmp_pm/power
```

Set Max Latency

Set the maximum wake-up latency requirement for a node.

```
$ echo set_max_latency <node> <latency> > /sys/kernel/debug/zynqmp_pm/power
```

Get Node Status

Get status information of a node. (Any PU can check the status of any node, regardless of the node assignment.)

```
$ echo get_node_status <node> > /sys/kernel/debug/zynqmp_pm/power
```

Get Operating Characteristic

Get operating characteristic information of a node.

```
$ echo get_operating_characteristic <node> > /sys/kernel/debug/zynqmp_pm/power
```

Reset Assert

Assert/de-assert on specific reset lines.

```
$ echo reset_assert <reset> <action> > /sys/kernel/debug/zynqmp_pm/power
```

Reset Get Status

Get the status of the reset line.

```
$ echo reset_get_status <reset> > /sys/kernel/debug/zynqmp_pm/power
```

MMIO Read

Read from a memory-mapped I/O address.

```
$ echo mmio_read <address> > /sys/kernel/debug/zynqmp_pm/power
```

Note: Address is in hex format (e.g. 0xFFFF0000).

MMIO Write

Write to a memory-mapped I/O address.

```
$ echo mmio_write <mask> <address> <value> > /sys/kernel/debug/zynqmp_pm/power
```

Mask, address and value are in hex format (e.g. 0xFFFF0000).

PM Platform Driver

The Zynq UltraScale+ MPSoC power management for Linux is encapsulated in a PM platform driver. The system-level API functions (see API layers in EEMI API Specification, UG1200) are exported and as such, can be called by other Linux modules with GPL compatible license. The function declarations are located in `include/linux/soc/xilinx/zynqmp/pm.h`. The function implementations are in `drivers/soc/xilinx/zynqmp/pm.c`.

For proper driver initialization, the correct node must be provided in the Linux device tree. The PM API 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:

1. compatible: Must contain "xlnx,zynqmp-pm"
2. method: The method of calling the PM framework firmware. Should be "smc".

Note: Additional information is available in the Linux Documentation, see file:
 Documentation/devicetree/bindings/soc/xilinx/zynq_mpsoc.txt

example:

```
zynqmp-firmware {
    compatible = "xlnx,zynqmp-pm";
    method = "smc";
    interrupt-parent = <&gic>;
    interrupts = <0 35 4>;
};
```

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 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.

How to check EEMI API version

The EEMI version implemented in the ATF is defined in the local EEMI_API_VERSION flag. The local function `XPm_GetApiVersion()` is available for checking the EEMI API versions between the ATF and the PMU firmware. If the versions are different, this call will report an error.

Note: This EEMI API call is version independent; every EEMI version implements it.

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 9-1: 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
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

Table 9-1: PSCI v0.2 Functions Supported by the ATF (Cont'd)

Functions	Description	Supported
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 API 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 API 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.)

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 IPIO 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 acknowledgement (if one is requested.) An EEMI request often usually 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):

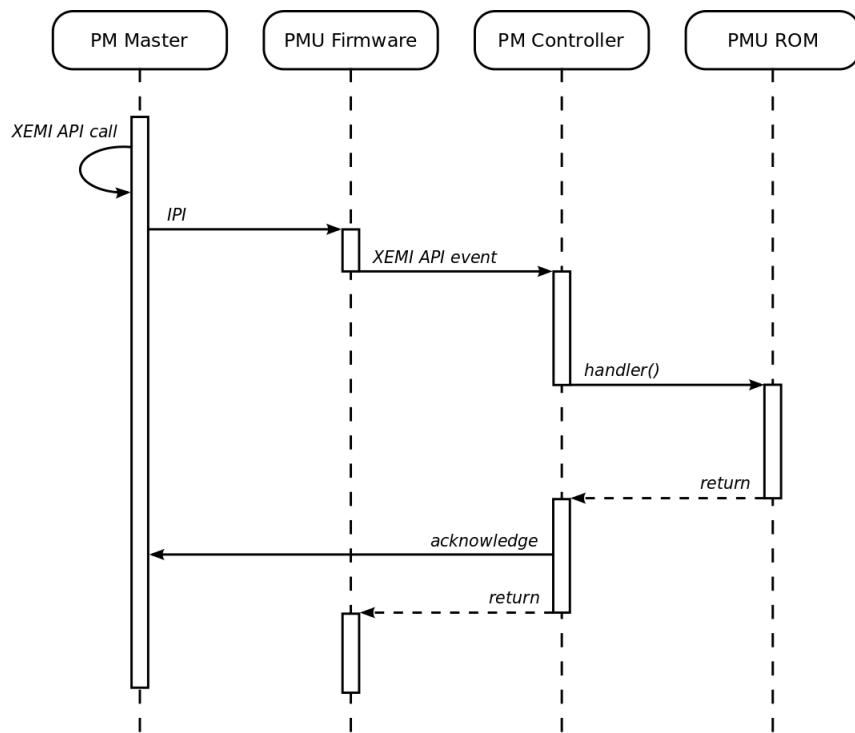


Figure 9-10: EEMI API Call Sequence Diagram

The diagram above shows 4 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.

Platform Management

Introduction

Zynq® UltraScale+™ MPSoC devices 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 device 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 (PMUFW) functionality, see the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) and the *Zynq UltraScale+ MPSoC Power Management Framework User Guide* (UG1199) [\[Ref 14\]](#).

For more information on dynamically changing the PS clocks, see “[Chapter 13, 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 device 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-R5 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.

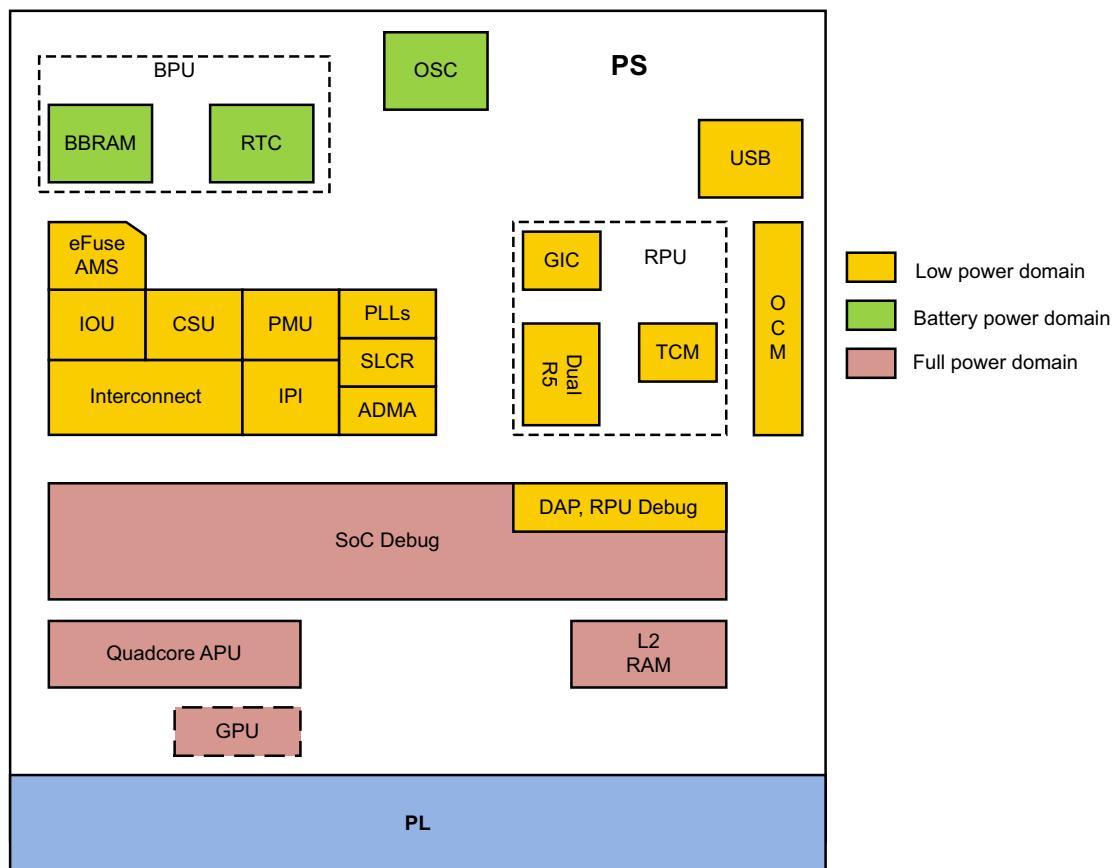

X19226-050317

Figure 10-1: Platform Management at the PS Level

From the power perspective, Zynq UltraScale+ MPSoC devices 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 [Figure 10-1](#)), 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+ MPSoC Technical Reference Manual* (UG1085) [\[Ref 10\]](#).

Low-Power Operation Mode

In the low-power operation mode, a subset of the PS (shown as low-power domain in the [Figure 10-1](#)) 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-R5, USBs, and the TCM and OCM block RAMs offer power gating.

You can control power gating to different blocks through software by configuring the LPD_SLCR registers. See the *SLCR Registers* link [\[Ref 11\]](#) for more information on LPD_SLCR register.

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

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-R5 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 *Zynq UltraScale+ MPSoC Power Management Framework* (UG1199) [\[Ref 9\]](#) and the *Embedded Energy Management Interface Specification* (UG1200) [\[Ref 10\]](#) describe the APIs to invoke shutdown.

For more details, see this [link](#) to the “Platform Management Unit Programming Model” section in the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [\[Ref 8\]](#).

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 (shown in the [Figure 10-1](#)):

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

The Zynq UltraScale+ MPSoC device 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.

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+ MPSoC Technical Reference Manual* (UG1085) [\[Ref 10\]](#).

Platform Management for Memory

The Zynq UltraScale+ MPSoC devices 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+ MPSoC Technical Reference Manual* (UG1085) [\[Ref 10\]](#).

Using Custom PMU Firmware

Besides 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 can execute a custom program that implements advanced system monitoring and power management algorithms, which is referred to as *custom PMU firmware* (PMUFW).

For more details, see this [link](#) to the “Platform Management Unit Programming Model” section in “Chapter 6” of the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [\[Ref 10\]](#).

You can use the Xilinx® SDK to create custom PMU firmware. It provides the source code for the PMUFW template and the necessary library support. For details on how to create an SDK project, see [Chapter 5, Software Development Flow](#).

Power Management Framework

The *Zynq UltraScale+ MPSoC Power Management Framework* (UG1199) [\[Ref 9\]](#) and the *Embedded Energy Management Interface Specification* (UG1200) [\[Ref 10\]](#) describe how to use the power API functions.

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

Reset

Introduction

The Zynq® UltraScale+™ MPSoC device reset block is responsible for handling both internal and external reset inputs to the system, and to ensure that the reset requirements are met 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+ MPSoC devices let you reset individual blocks such as the APU, RPU or even individual power domains like 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+ MPSoC Technical Reference Manual* (UG1085) [Ref 10].

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+ MPSoC Technical Reference Manual* (UG1085) [\[Ref 10\]](#) describes the PS-only reset sequence.

APU 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+ MPSoC Technical Reference Manual* (UG1085) [\[Ref 10\]](#) describes the APU reset sequence.

RPU Reset

Each CortexTM-R5 core can be independently reset. In lockstep mode, only the Cortex-R5_0 needs to be reset to reset both Cortex-R5 cores. It can be triggered by errors or a software register write. The Cortex-R5 reset can be triggered due to a lockstep error to be able to reset and restart RPU.

It needs to gracefully terminate Cortex-R5 ingress and egress transactions before initiating reset of corresponding Cortex-R5. RPU reset sequence is as follows.

FPD Reset

The FPD-reset resets all of 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+ MPSoC devices has general-purpose output pins from the PMU block that can be used to reset the blocks in PL. Additionally, GPIO using the EMIO interface can also be used to reset PL logic blocks. For a detailed description of the reset flow, see the [this link](#) to the "Reset System" chapter in the *Zynq UltraScale+ MPSoC Technical Reference Manual* [Ref 10].

For more information on the software APIs for reset, see the [Using Custom PMU Firmware](#) in [Chapter 10, Platform Management](#)

High-Speed Bus Interfaces

Introduction

The Zynq® UltraScale+™ MPSoC device has a serial input/output unit (SIOU) for a high-speed serial interface. It supports protocols such as PCIe™, USB 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 device 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.

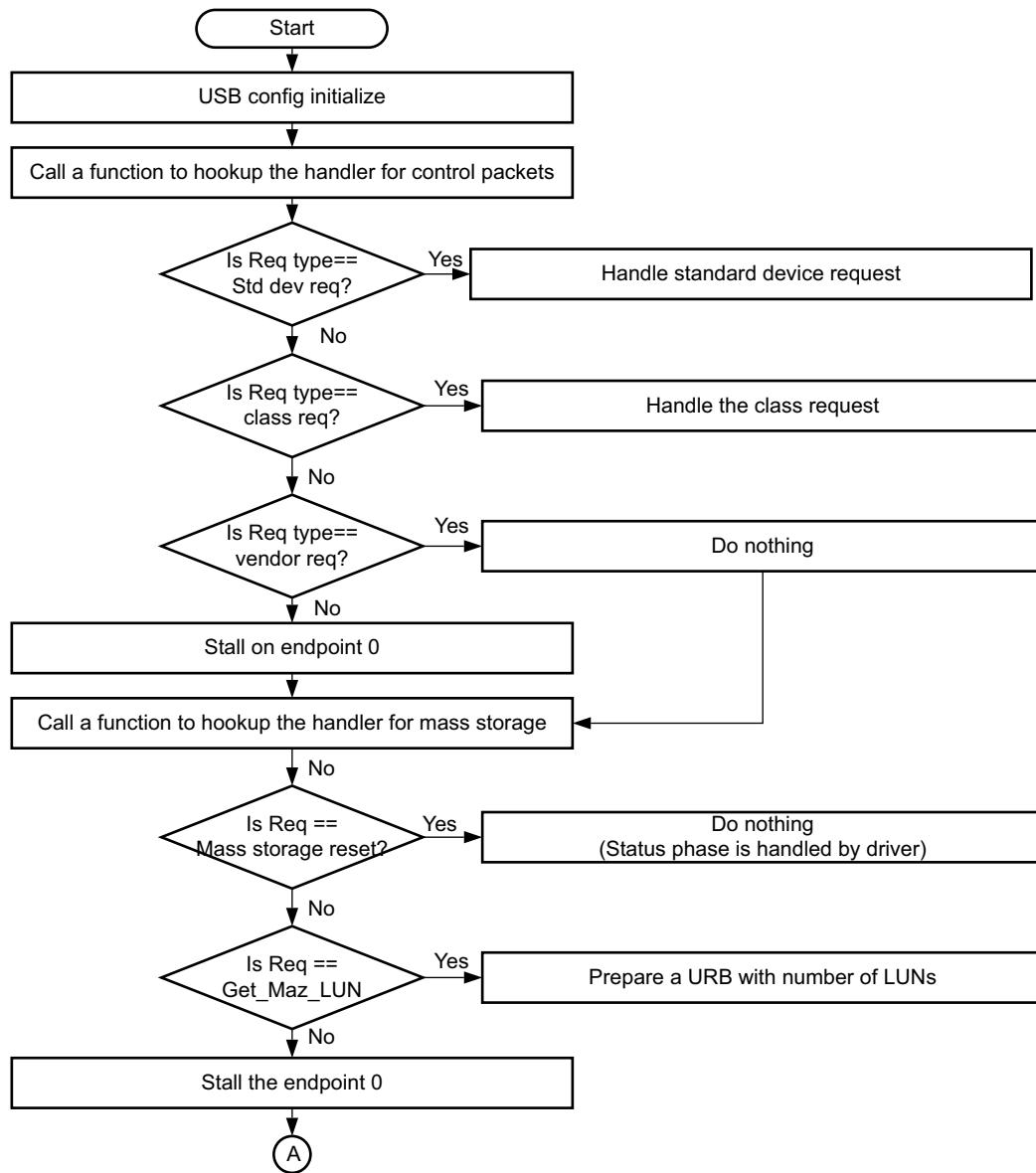
See this [link](#) to the "High-Speed PS-GTR Transceiver Interface" of the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [Ref 10] for more information.

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.

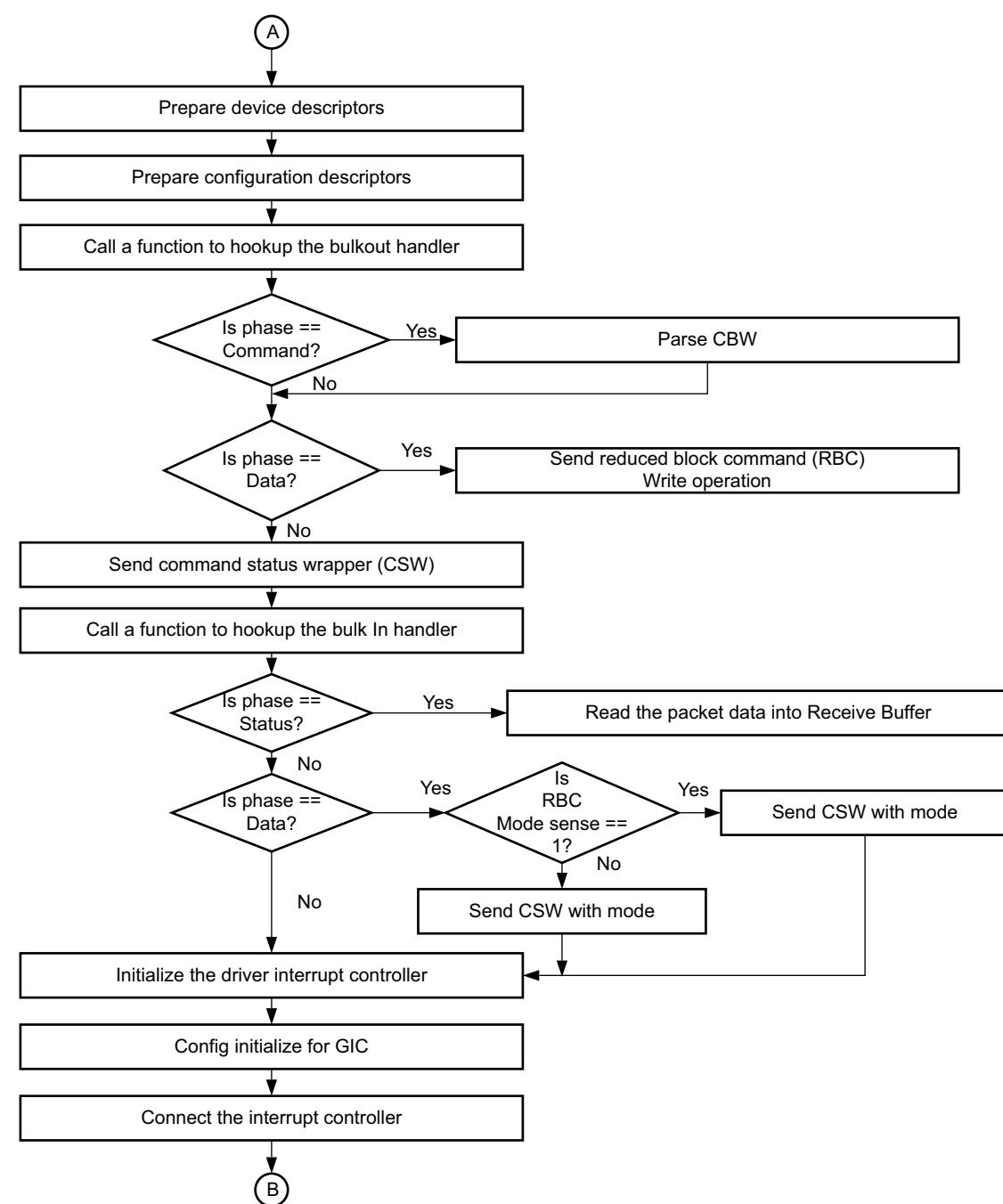
- An internal DMA engine is present in the controller and it uses the AXI master interface to transfer data.
- The three dual-port RAM configurations implement the RX data FIFO, TX data FIFO, and the descriptor/register cache.

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



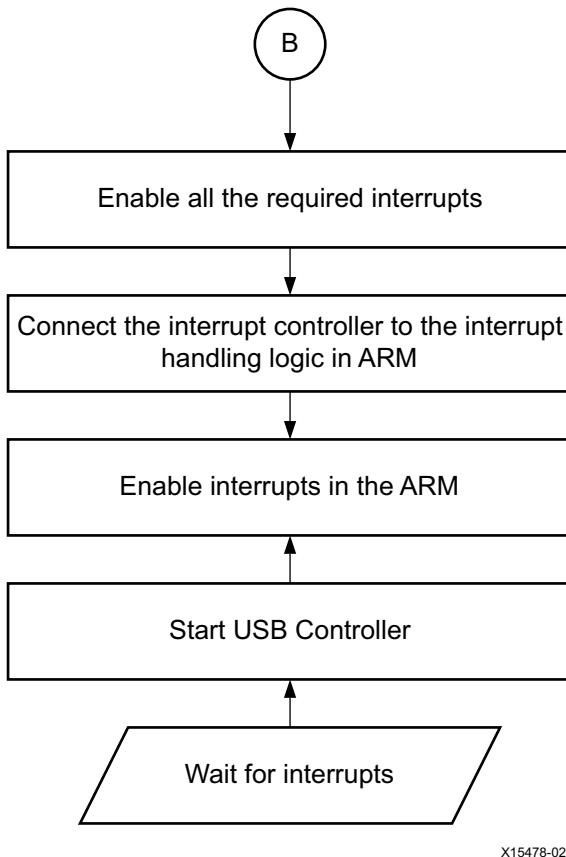
X15463-021317

Figure 12-1: USB Example Flow (USB Initialization)



X15477-021317

Figure 12-2: Example USB Flow (Hookup Bulk in and Bulk out Handlers and Initialize Interrupt Controller)



X15478-021317

Figure 12-3: Enable Interrupts and Start the USB Controller

For more information on USB controller, see this [link](#) to the “USB 2.0/3.0 Host, Device, and Controller,” chapter of the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [[Ref 10](#)].

Gigabit Ethernet Controller

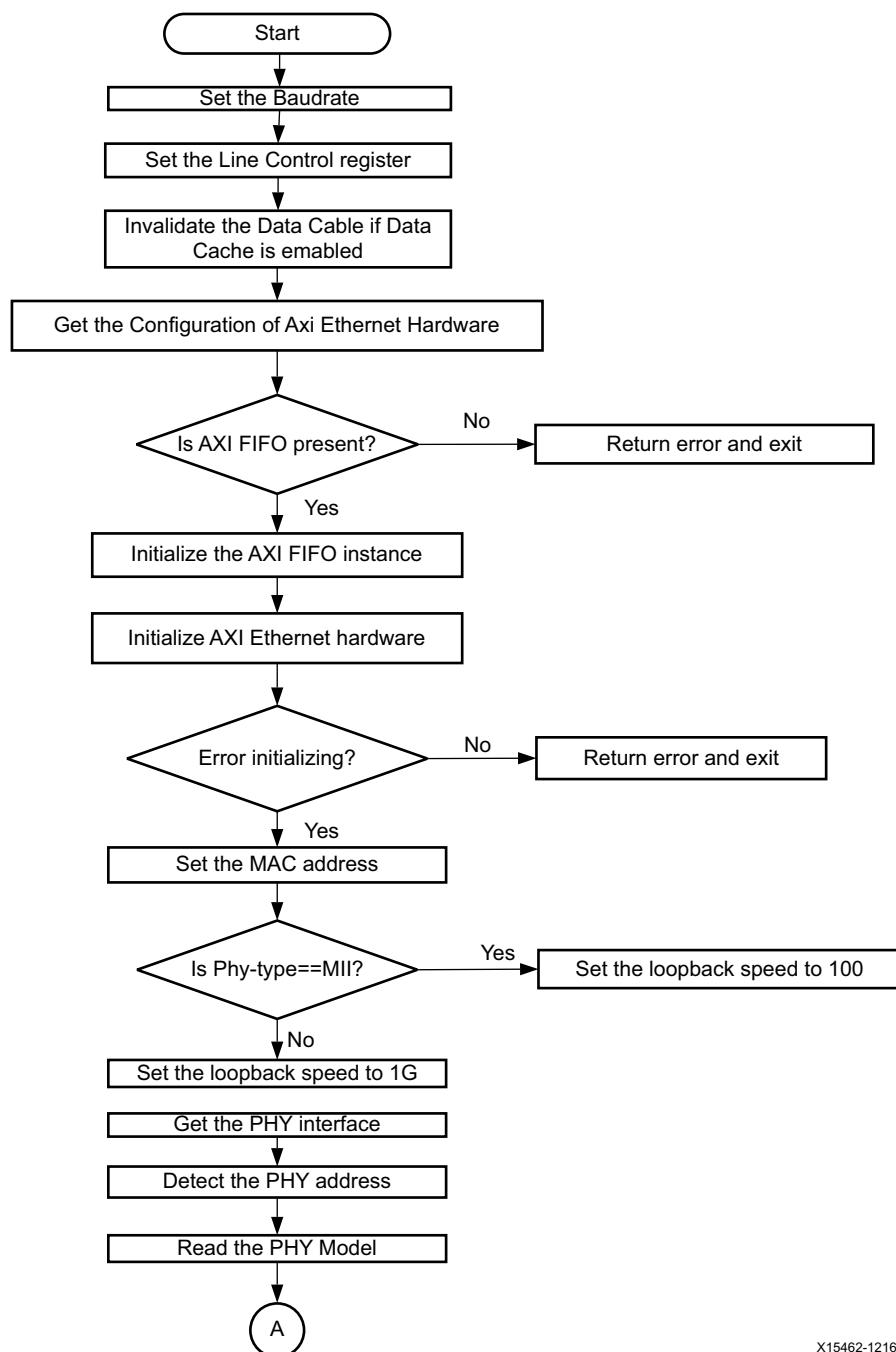
The gigabit Ethernet controller (GEM) implements a 10/100/1000 Mb/s Ethernet MAC compatible with *IEEE Standard for Ethernet* (IEEE Std 802.3-2008) and is capable of operating in either half or full-duplex mode in 10/100 mode and full-duplex in 1000 mode.

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.

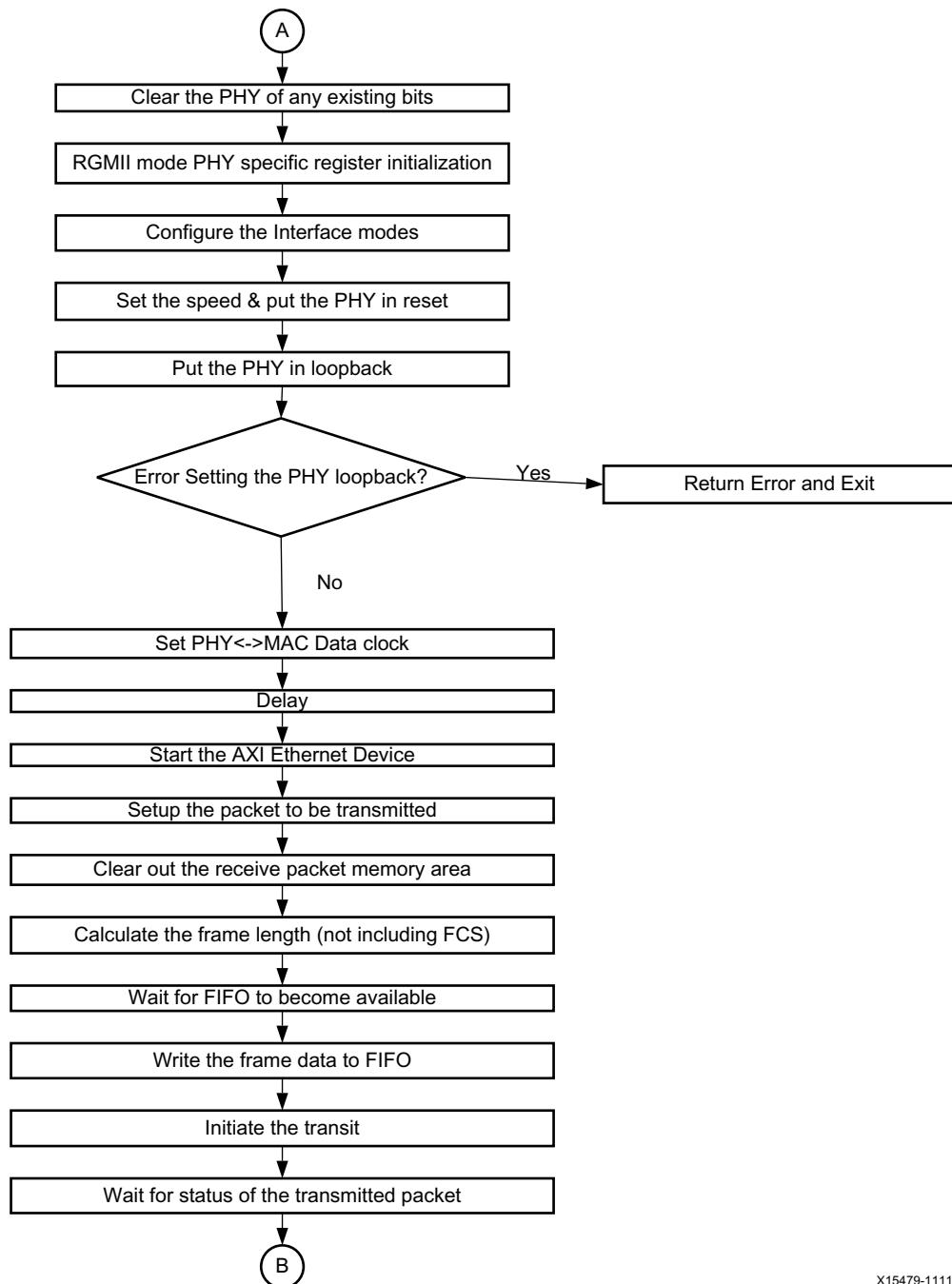
The DMA controller connects to memory through the advanced eXtensible interface (AXI). It is attached to the FIFO interface of the controller of the MAC to provide a scatter-gather type capability for packet data storage in an embedded processing system.

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



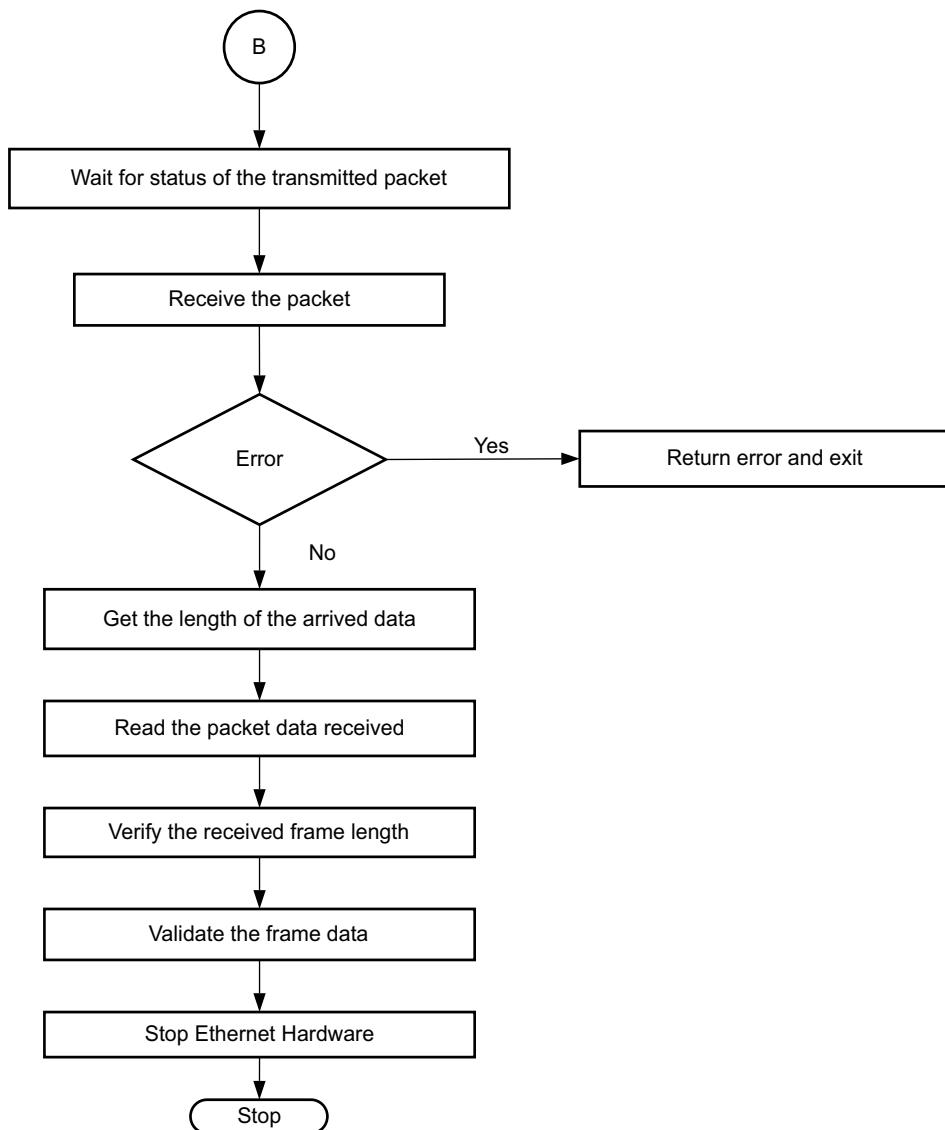
X15462-121616

Figure 12-4: Example Ethernet Flow (Initialize Ethernet Controller)



X15479-111115

Figure 12-5: Example Ethernet Flow (Configure the Ethernet Parameters and Initiate the Transmit)



X15480-111115

Figure 12-6: Example Ethernet Flow (Receive and Validate the Data)

For more information on Ethernet Controller, see this [link](#) to the "Gigabit Ethernet Controller" chapter in the *Zynq UltraScale+ MPSoC Technical Reference Manual (UG1085)* [Ref 10].

PCI Express

The Zynq UltraScale+ MPSoC device 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.

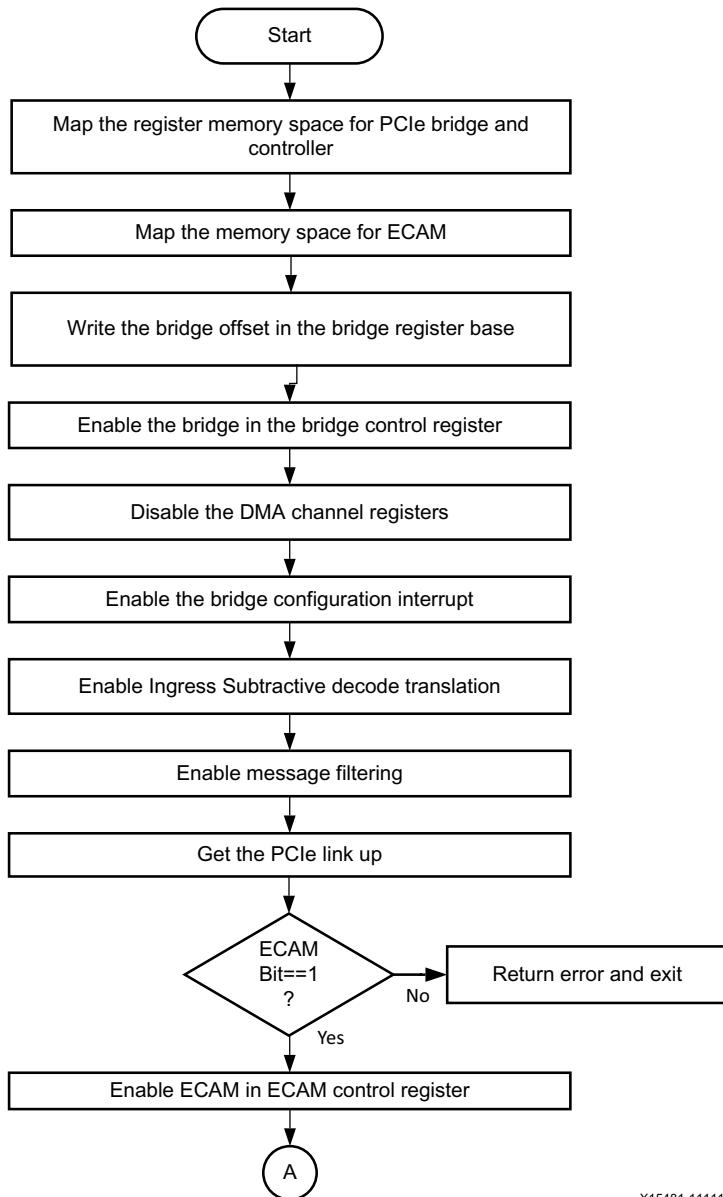
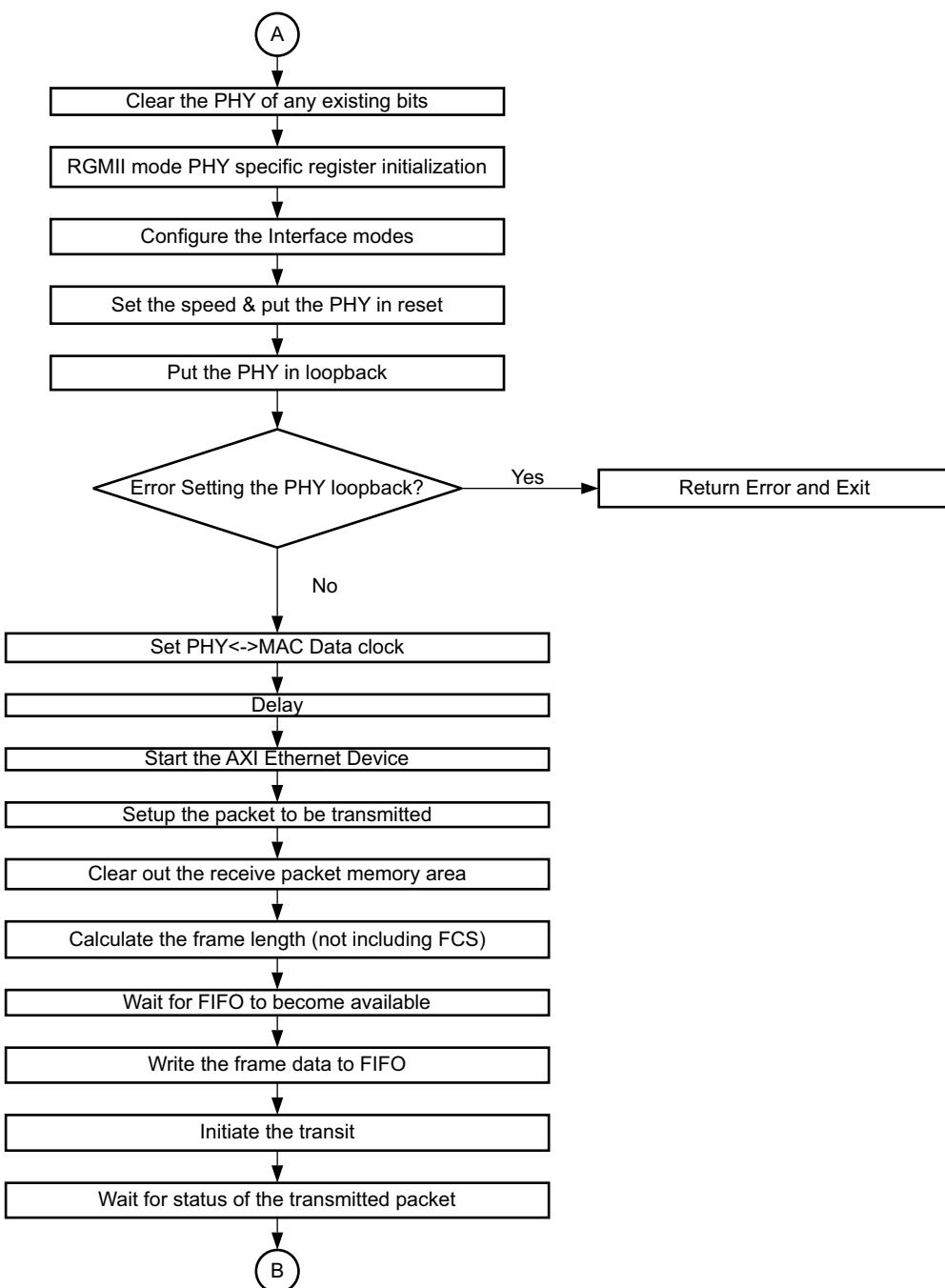
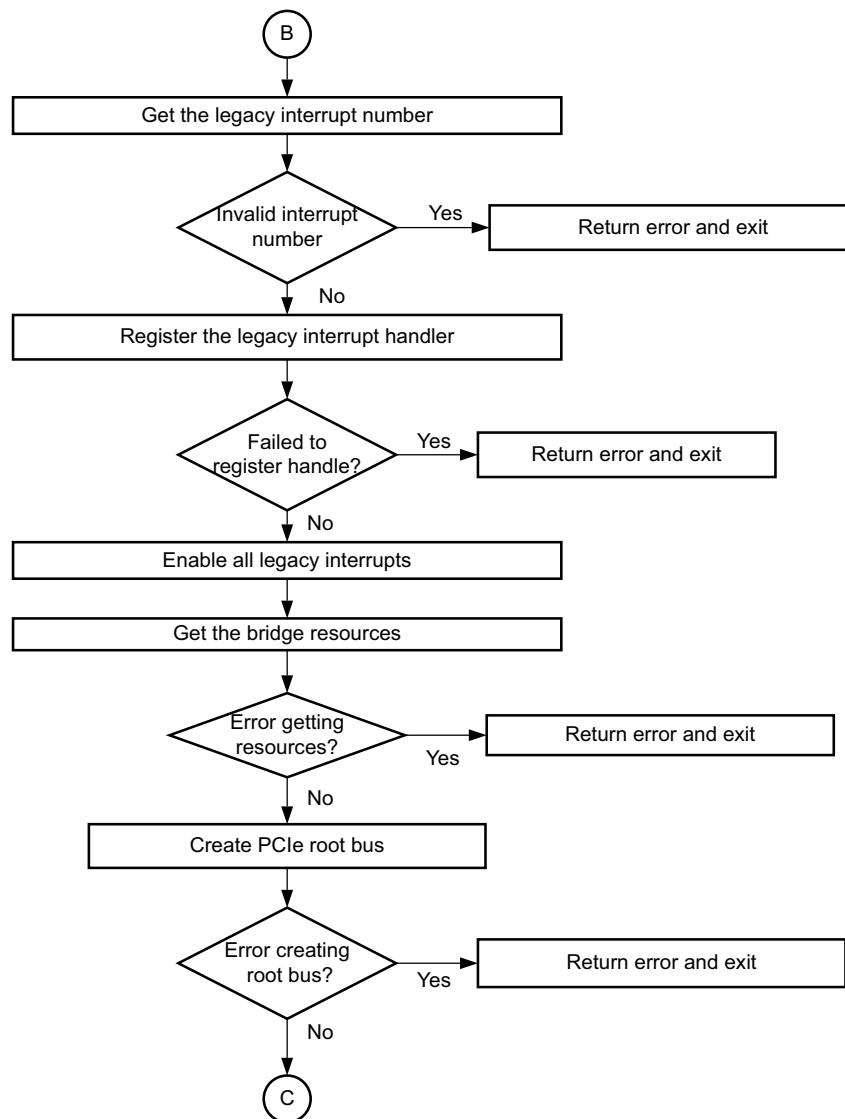

X15481-111115

Figure 12-7: Example PCIe Flow (Enable the Legacy Interrupts and Create PCIe Root Bus)



X15479-041917

Figure 12-8: Example PCIe Flow (Configure the PCIe Parameters and Initialize the Transmit)



X15483-041917

Figure 12-9: Example PCIe Flow (Enable the Legacy Interrupts and Create PCIe Root Bus)

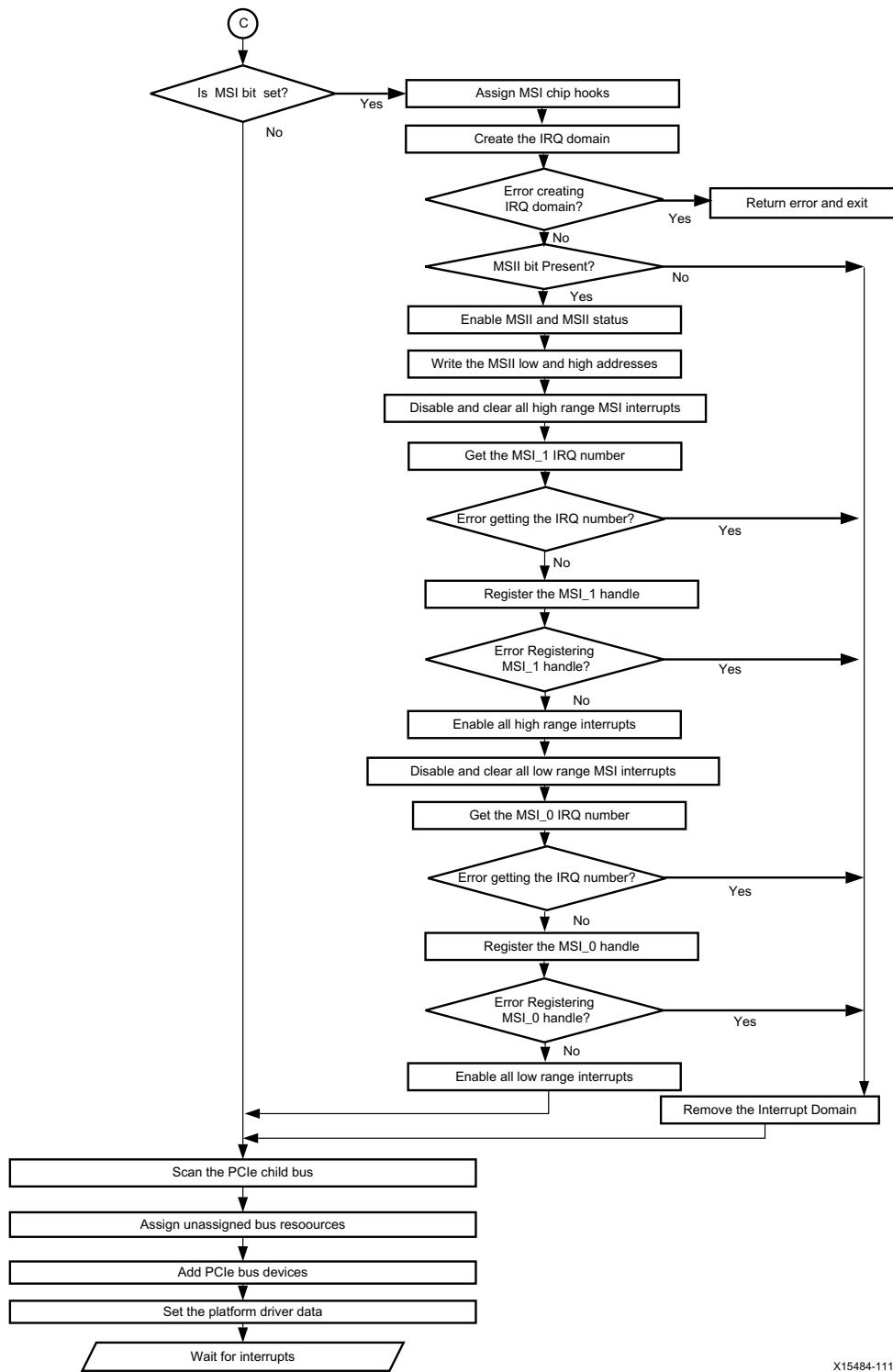


Figure 12-10: Example PCIe Flow (Enable MSI Interrupts and Wait for Interrupts)

Note: For endpoint operation, refer to this [link](#) to “Controller for PCI Express” in the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [Ref 10].

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+ MPSoC Technical Reference Manual* (UG1085) [\[Ref 10\]](#).

Clock and Frequency Management

Introduction

The Zynq® UltraScale+™ MPSoC device 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 to the different PS peripherals.

Unlike the USB and Ethernet peripherals, some peripherals like 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.

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 peripherals.

Table 13-1: Standalone APIs

APIs	Description
XSDPS_change_clkfreq	Change the clock frequency of SD.
XSPIPPS_setclkprescaler XSPIPPS_getclkprescaler	Pre-scale the SPI frequency.
XRtcPSU_calculatecalibration	Change the oscillator frequency.
XQSPIPPSU_setclkprescaler	Change the clock frequency of QSPI.

In case of a Linux application, frequency for 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@0xFF0F0000 {  
    #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).
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 *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [\[Ref 10\]](#).

Target Development Platforms

Introduction

This chapter describes the different development platforms available for the Zynq® UltraScale+™ MPSoC device, 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 that must be passed to it using the command line. See the *Xilinx QEMU User Guide* (UG1169) [\[Ref 8\]](#), 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 [\[Ref 33\]](#).

See the *Zynq UltraScale+ MPSoC Products Page* [\[Ref 7\]](#) to know the different Zynq UltraScale+ MPSoC devices.

Bootgen Image Creation

Introduction

Bootgen is a standalone tool for creating a bootable image suitable for the Zynq® UltraScale+™ MPSoC device. The program assembles the boot image by prefixing a header block to a list of partitions. Optionally, you can encrypt the bitstream, and each partition, and authenticate it with Bootgen. The output is a single file that you can program directly into the boot flash memory of the system. The Bootgen utility can generate other peripheral files to support authentication and encryption as well.

The tool is integrated into the Xilinx® Software Developer Kit (SDK) for automatic image generation. You can also use Bootgen on the command-line or in a script you initiate on the command line.

BIF File Parameters

Table 15-1: BIF File Parameters

Parameters	Descriptions and Options
[aeskeyfile]	AES key file for encryption. <.nky key file>
[init]	Register initialization file. Supports a maximum of 256 initialization pairs.
[udf_bh]	Imports a file containing up to 40 bytes of data to be copied to the User Defined Field of the Boot Header.
[headersignature]	Imports header signature into header authentication.
[ppkfile]	Primary public key file used to authenticate partition. <.pub .pem key files>
[pskfile]	Primary secret key file used to authenticate partition. <.pub .pem key files>
[spkfile]	Secondary public key file used to authenticate partition. <.pub .pem key files>

Table 15-1: BIF File Parameters (Cont'd)

Parameters	Descriptions and Options	
[sskfile]	Secondary secret key file used to authenticate partition. <.pub .pem key files>	
[spksignature]	Imports SPK signature into authentication certificate.	
	These parameters configures the FSBL image.	
[fsbl_config]	r5_single a53_x32 a53_x64 r5_dual	Specifies the core on which the FSBL is run.
	bh_auth_enable	Boot Header Authentication Enable - RSA authentication of the boot image will be done excluding the verification of PK hash and SPK ID. For integration and test only.
	auth_sha2_enable	While doing RSA authentication, use SHA2. (Default is SHA3).
	bi_integrity_sha3	Boot image Integrity check with SHA3. Note: bi_integrity_sha3 will be deprecated in future releases. Use checksum=sha3 option.
	auth_only	Boot image is only RSA signed. FSBL should not be decrypted.
	opt_key	Optional key is used for Block-0 decryption. Secure Header has the optional key.
	shutter	Shutter Open Time. (SOPEN * 16) – 16 is the number of APB_CLK clock cycles for PUF generation.
	pufhd_bh	When PUF helper data is not actually programmed on eFUSE and placed in boot header.
	puf4kmode	To tell Bootgen that helper data provided for PUF boot header encryption is 4K.
	Example: [fsbl_config] a53_x64 opt_key,	
[auth_params]	Authentication parameters. ppk_select = <0/1>, spk_id = <4-byte value> Example: [auth_params] ppk_select=0; spk_id=0x12345678	

Table 15-1: BIF File Parameters (Cont'd)

Parameters	Descriptions and Options
[keysrc_encryption]	Key source for encryption. efuse_red_key bbram_red_key bh_blk_key efuse_blk_key
[pmufw_image]	PMU Firmware Image <PMU ELF file>. This PMU partition will be loaded along with FSBL by BootROM.
[bootloader]	Partition that is FSBL boot loader.
authentication = <value>	Option to enable authentication for partitions: none rsa
checksum = <value>	Options to enable checksum for partitions: none sha3
encryption = <value>	Options to enable encryption for partitions: none aes
load = <value>	Sets the load address for the partition.
offset = <value>	Sets the absolute offset in the image.
reserve = <value>	Reserves the memory and padded after the partition.
startup = <value>	Sets the entry address for the partition, after it is loaded. This is ignored for partitions that do not execute.
udf_data = <filename>	Imports a file containing up to 56 bytes of data to be copied to the User Defined Field of the partitions authentication certificate.
presign = <filename>	Imports partition signature into partition authenticate certificate.
blocks = <block sizes>	Block sizes for encryption key rolling. blocksize(# blocks); blocksize(# blocks); blocksize(#blocks) Example: blocks=4096(2);1024;2048;1024(2) This partition will have 6 blocks with: 4096 bytes 1. 4096 bytes 2. 1024 bytes 3. 2048 bytes 4. 1024 bytes 5. 1024 bytes 6. 1024 bytes 7. Remaining bytes of the partition.

Table 15-1: BIF File Parameters (Cont'd)

Parameters	Descriptions and Options
destination_cpu = <value>	<p>Specifies which core will execute the partition.</p> <p>a5x-0 a5x-1 a5x-2 a5x-3 r5-0 r5-1 r5-lockstep pmu</p> <p>Note: When PMU is selected as the destination CPU, the partition will be loaded and handed off by FSBL to PMU.</p>
destination_device = <value>	<p>Specifies destination of the partition.</p> <p>ps pl pmufw xip</p> <p>Note: This option will be deprecated in a future release. Use cases for 'destination_device=pmufw' will be supported through 'destination_cpu=pmu'. Use cases for 'destination_device=xip' will be supported through 'xip_mode' attribute.</p>
exception_level = <value>	<p>Exception level for which the core should be configured.</p> <p>el-0 el-1 el-2 el-3</p>
trustzone	Configures the core to be Trust Zone secure.
partition_owner = <value>	<p>Owner of the partition which is responsible to load the partition.</p> <p>fsbl uboot</p>
bh_key_iv	Initialization vector used when decrypting the obfuscated key
bh_keyfile	Black key generated from Red key and PUF key should be captured in .txt and should be provided as input for PUF boot header encryption.
puf_file	Helper data generated on same silicon should be captured in .txt format and should be provided input to Bootgen for PUF bootheader encryption

Bootgen Command Line Options

Table 15-2: Bootgen Command Line Options

ARGUMENT	OPTIONS
-arch <type>	Select architecture: ◦ zynq ◦ zynqmp ◦ fpga
-bif_help	Prints out the BIF help summary.
-efuseppkbits <filename>	Specifies the name of the eFuse file to be written to contain the PK hash. This option generates a direct 32-byte/48-byte hash without any padding, depending upon the SHA2/SHA3 selection for authentication.
-fill <hex_byte>	Specifies the byte to use for fill.
-generate_hashes	Specifies to outputting SHA2/SHA3 hash files with padding in PKCS#1v1.5 format.
-generate_keys auth <option>	Generate the authentication keys in PEM or RSA format: ◦ pem ◦ rsa Example: bootgen -arch zynqmp -image test.bif - generate_keys auth pem Note: test.bif should have the paths and keys that need to be generated. image: { [ppkfile] C:\path\to\ppk.txt [pskfile] C:\path\to\psk.txt [spkfile] C:\path\to\spk.txt [sskfile] C:\path\to\ssk.txt } The table continues with more rows, but they are mostly blank or have very short descriptions.
-h, -help	Prints out help summary.
-image <filename.bif>	Boot image format (BIF) file name.
-log <option>	Generates a log file at the current working directory with following message types: ◦ error ◦ warning ◦ info ◦ debug ◦ trace
-o <filename>	Specifies the output file: ◦ .bin ◦ .mcs

Table 15-2: Bootgen Command Line Options (Cont'd)

ARGUMENT	OPTIONS
-p <partname>	Specify the Xilinx part name while generating the encryption key (.nky). This name is copied verbatim to the key file in the "Device" line.
-process_bitstream	Specifies that the bitstream is processed and output as either: .bin .mcs file. For example, if encryption is selected for bitstream in BIF file, the output is an encrypted bitstream.
-split <option>	Splits the boot image into partitions as different output files: <ul style="list-style-type: none">◦ bin◦ mcs
-w <option>	Specifies whether to overwrite the output files: <ul style="list-style-type: none">◦ on◦ off Note: The -w without an option is interpreted as -w on.

Bootgen Command Example

The bootgen command is as follows:

```
bootgen -arch zynqmp -image bootimage.bif -w -o BOOT.bin
```

Boot Image Format

The boot image consists of a boot header and partitions for different images along with a partition header. The following table shows the boot image.

Table 15-3: Boot Image

Image Type	Offset	Bytes
Boot Header (BH)	0x0	0xB8
Register Initialization	0xB8	0x800
Image Header Table (IHT)	Word Offset will be at 0x98 of BH	0x40
Image Headers (IH1-IHn)	First Header Word Offset will be at 0xC of IHT	n*(0x40)
Partition Header Tables (BPHT1-PHTm)	First Header Word Offset will be at 0x9C of BH	m*(0x40)
Header Authentication (optional)	Word Offset will be at 0x10 of IHT	0xEC0

Table 15-3: Boot Image (Cont'd)

Image Type	Offset	Bytes
Partition – 1 (FSBL) (Encryption-optional)	Word Offset will be at 0x20 of PHT1	Word Length will be at 0x08 of PHT1
Partition-1 RSA Authentication certificate (optional)	Word Offset will be at 0x34 of PHT1	0xEC0
Partition – 2 (Bit Stream) (Encryption-optional)	Word Offset will be at 0x20 of PHT2	Word Length will be at 0x08 of PHT2
Partition-2 RSAS Authentication certificate (optional)	Word Offset will be at 0x34 of PHT2	0xEC0
Partition – 3 (U-boot) (Encryption-optional)	Word Offset will be at 0x20 of PHT3	Word Length will be at 0x08 of PHT3
Partition-3 RSA Authentication certificate (optional)	Word Offset will be at 0x34 of PHT3	0xEC0
Partition – 4 (Linux Image) (Encryption-optional)	Word Offset will be at 0x20 of PHT4	Word Length will be at 0x08 of PHT4
Partition-4 RSA Authentication certificate (optional)	Word Offset will be at 0x34 of PHT4	0xEC0

Boot Header Table

See the previous section, [Boot Image Format](#), for information on the boot header.

Register Initialization Table

Table 15-4: Register Initializations (Offsets, Parameters, and Descriptions)

Address Offsets	Parameter	Description
0xB8 to 0x8B4	Register Init Pairs <code><address></code> <code><value></code>	This word region contains a simple system to initialize PS registers for the MIO multiplexer and flash clocks. This allows the MIO multiplexer to be fully configured before the FSBL image is copied into OCM or executed from flash with XIP, and allows for flash device clocks to be set to maximum bandwidth speeds. Total 256 pairs. Address = 0xFFFFFFFF means skip that register and ignore the value All the unused register fields must be set to Address=0xFFFFFFFF and value = 0x0

Image Header Table

The following tables explain various fields in the generated BOOT.BIN image.

For a high level overview of the `BOOT.BIN` file, see this [link](#) to the *Boot Image Format* section in the “Boot and Configure” chapter of the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [\[Ref 10\]](#).

Table 15-5: Image Header Table

Address Offset	Parameter	Description
0x00	Version	0x01020000
0x04	Count of Image Headers	Total number of partitions in the image.
0x08	1st Partition Header Word Offset	Pointer to first partition header.
0x0C	1st Image Header Word Offset	Pointer to first image header.
0x10	Header Authentication Certificate Word Offset	Pointer to the header authentication certificate.
0x14	Boot Device	0 -Same boot device 1 - QSPI 2 - NAND 3 - SD 4 - MMC 5 - USB 6 - ETHERNET 7 - PCIE 8 – SATA
0x18 – 0x38	Reserved	Reserved (0x0)
0x3C	Checksum	A sum of all the previous words in the image header.

Partition Header Tables

Table 15-6: Partition Headers (Offsets, Parameters, and Description)

Address Offset	Parameter	Description
0x00	Partition Data Word Length (x4)	Encrypted partition length.
0x04	Extracted Data Word Length (x4)	Unencrypted partition data length.
0x08	Total Partition Word Length (Includes Authentication Certificate) (x4)	The total encrypted + padding + expansion +authentication length.
0x0C	Next Partition Header Offset	Location of next partition header.
0x10	Destination Execution Address LO	The executable address of this partition after loading.
0x14	Destination Execution Address HI	The executable address of this partition after loading.
0x18	Destination Load Address LO	The RAM address into which this partition is to be loaded.
0x1C	Destination Load Address HI	The RAM address into which this partition is to be loaded.
0x20	Actual Partition Word Offset	The position of the partition data relative to the start of the boot image.
0x24	Attributes	See Table 15-7 .
0x28	Reserved	For internal use.
0x2C	Checksum Word Offset (x4)	The location of the checksum word in the boot image.
0x30	Reserved	For internal use.
0x34	Authentication Certificate Word Offset (x4)	The location of the Authentication Certification in the boot image.
0x38	unused	0x00000000
0x3C	Header Checksum	A sum of the previous words in the Partition Header.

Table 15-7: Partition Header Attributes

Bits	Field Name	Description
31:24	Reserved	.
23	Vector location	1 - HIVEC 0 - LOVEC
22:20	Bitstream block size	000 - For non-authenticated bitstream and all other partitions. 010 - For 8MB blocks, when authentication is enabled.
19	Early handoff	

Table 15-7: Partition Header Attributes (Cont'd)

Bits	Field Name	Description
18	Endianness	0 - Little Endian 1 - Big Endian
17:16	Partition Owner	0 - FSBL 1 - UBOOT 2 and 3 - Reserved
15	RSA Signature Present	0 - No RSA auth. certificate 1 - RSA auth. certificate
14:12	Checksum Type	0 - None 1 - MD5 2 through 7 - Reserved
11:8	Destination CPU	0 - None 1 - A53-0 2 - A53-1 3 - A53-2 4 - A53-3 5 - R5-0 6 - R5-1 7 - R5-lockstep 8 through 15 - Reserved
7	Encryption Present	0 - Not encrypted 1 - Encrypted
6:4	Destination Device	0 - None 1 - PS 2 - PL 3 - PMU FW 4 - XIP 5 through 15 - Reserved
3	A5X Exec State	0 - AARCH64 (default) 1 - AARCH32
2:1	Exception Level	0 - EL0 1 - EL1 2 - EL2 3 - EL3
0	TrustZone	0 - Non-Secure 1 - Secure

Authentication Certificate

Table 15-8: Authentication Certificate

Address Offset	Parameter	Description
0x00	Authentication Certificate Header	See Table 15-9 .
0x04	SPK ID	32-bit SPK ID
0x08	User Defined Field (UDF)	56 bytes
0x40	Primary Modulous	512 bytes
0x240		512 bytes
0x440		4 bytes - recommended 0x00010001
0x444		60 bytes - 0x0000000000
0x480	Secondary Modulous	512 bytes
0x680		512 bytes
0x880		4 bytes - recommended 0x00010001
0x884		60 bytes - 0x0000000000
0x8c0	SPK Signature	512 bytes
0xAC0	Boot Header Signature	512 bytes
0xCC0	Partition signature	512 bytes

Authentication Certificate Header

Table 15-9: Bits, Field Names, and Description

Bits	Field Name	Description
31:18	Reserved	0
17:16	PPK Key Select	0: PPK0 1: PPK1
15:14	Authentication Certificate Format	00: PKCS #1 v1.5
13:12	Authentication Certificate Version	00: Current AC
11	PPK Key Type	0: Hash Key
10:9	PPK Key Source	0: eFUSE
8	SPK Enable	1: SPK Enabled
7:4	Public Strength	0: Reserved 1 : 4096 2:3 : Reserved

Table 15-9: Bits, Field Names, and Description (Cont'd)

Bits	Field Name	Description
3:2	Hash Algorithm	
1:0	Public Algorithm	0: Reserved 1: RSA 2: Reserved

Note: Keys and authentication certificate generation are done using Linux open tools, and are not provided with any package.

Power Management Framework Appendix

XilPM Argument Value Definitions

Introduction

The following are value definitions for the various arguments used in the power management APIs, as defined in the file `pm_defs.h`.

Node IDs: XPMNODEID

The following table lists the defined Node IDs in the Zynq® UltraScale™+ MPSoC device.

```
enum XPMNodeID {
```

Table A-1: XPMNodeIds

Node IDs for Zynq UltraScale+ MPSoC Devices		
NODE_UNKNOWN		0U
NODE_APU	APU Controller	1U
NODE_APU_0	APU Controller 0	2U
NODE_APU_1	APU Controller 1	3U
NODE_APU_2	APU Controller 2	4U
NODE_APU_3	APU Controller 3	5U
NODE_RPU	RPU Controller	6U
NODE_RPU_0	RPU Controller 0	7U
NODE_RPU_1	RPU Controller 1	8U
NODE_PL	PL Controller	9U
NODE_FPD	FPD Controller	10U
NODE_OCM_BANK_0	OCM Memory Tile 0	11U
NODE_OCM_BANK_1	OCM Memory Tile 1	12U
NODE_OCM_BANK_2	OCM Memory Tile 2	13U
NODE_OCM_BANK_3	OCM Memory Tile 3	14U

Table A-1: XPMNodeIds (Cont'd)

Node IDs for Zynq UltraScale+ MPSoC Devices		
NODE_TCM_0_A	Tightly coupled memory (0A)	15U
NODE_TCM_0_B	Tightly coupled memory (0B)	16U
NODE_TCM_1_A	Tightly coupled memory (1A)	17U
NODE_TCM_1_B	Tightly coupled memory (1B)	18U
NODE_L2	L2 Cache system	19U
NODE_GPU_PP_0	Graphics Processing Unit 0	20U
NODE_GPU_PP_1	Graphics Processing Unit 1	21U
NODE_USB_0	USB Controller 0	22U
NODE_USB_1	USB Controller 1	23U
NODE_TTC_0	Triple-timer Counter 0	24U
NODE_TTC_1	Triple-timer Counter 1	25U
NODE_TTC_2	Triple-timer Counter 2	26U
NODE_TTC_3	Triple-timer Counter 3	27U
NODE_SATA	SATA Controller	28U
NODE_ETH_0	Gigabit Ethernet Controller 0	29U
NODE_ETH_1	Gigabit Ethernet Controller 1	30U
NODE_ETH_2	Gigabit Ethernet Controller 2	31U
NODE_ETH_3	Gigabit Ethernet Controller 3	32U
NODE_UART_0	UART Controller 0	33U
NODE_UART_1	UART Controller 1	34U
NODE_SPI_0	SPI Controller 0	35U
NODE_SPI_1	SPI Controller 1	36U
NODE_I2C_0	SPI Controller 2	37U
NODE_I2C_1	SPI Controller 3	38U
NODE_SD_0	SD/SDIO Controller 0	39U
NODE_SD_1	SD/SDIO Controller 1	40U
NODE_DP	DisplayPort Controller	41U
NODE_GDMA	FPD DMA Controller	42U
NODE_ADMA	APU DMA	43U
NODE_NAND	NAND Controller	44U
NODE_QSPI	QSPI Controller	45U
NODE_GPIO	GPIO Controller	46U
NODE_CAN_0	CAN Controller 0	47U
NODE_CAN_1	CAN Controller 1	48U
NODE_AFI	AFI Block	49U

Table A-1: XPmNodeIds (Cont'd)

Node IDs for Zynq UltraScale+ MPSoC Devices		
NODE_APLL	APU PLL	50U
NODE_VPLL	Video PLL	51U
NODE_DPLL	DDR Controller PLL	52U
NODE_RPLL	RPU PLL	53U
NODE_IOPLL	Peripheral I/O PLL	54U
NODE_DDR	DDR Controller	55U
NODE_IPI_APU	IPI APU Controller	56U
NODE_IPI_RPU_0	IPI APU Controller 0	57U
NODE_GPU	Graphics Processing Unit Controller	58U
NODE_PCIE	PCIE Controller	59U

Acknowledge Request Types: XPmRequestAck

```
enum XPmRequestAck {
    REQUEST_ACK_NO = 1,
    REQUEST_ACK_BLOCKING,
    REQUEST_ACK_NON_BLOCKING,
};
```

Abort Reasons: XPmAbortReason

```
enum XPmAbortReason {
    ABORT_REASON_WKUP_EVENT = 100,
    ABORT_REASON_PU_BUSY,
    ABORT_REASON_NO_PWRDN,
    ABORT_REASON_UNKNOWN,
};
```

Suspend Reasons

```
enum XPmSuspendReason {
    SUSPEND_REASON_PU_REQ = 201,
    SUSPEND_REASON_ALERT,
    SUSPEND_REASON_SYS_SHUTDOWN,
};
```

Operating Characteristic Types: XPmOpCharType

```
enum XPmOpCharType {
    PM_OPCHAR_TYPE_POWER = 1,
    PM_OPCHAR_TYPE_ENERGY,
    PM_OPCHAR_TYPE_TEMP,
};
```

Notify Event Types: XPmNotifyEvent

```
enum XPmNotifyEvent {
    EVENT_STATE_CHANGE = 1,
    EVENT_ZERO_USERS = 2,
    EVENT_ERROR_CONDITION = 4,
};
```

Reset Line IDs

```
enum XPmReset {
    XILPM_RESET_PCIE_CFG = 1000,
    XILPM_RESET_PCIE_BRIDGE,
    XILPM_RESET_PCIE_CTRL,
    XILPM_RESET_DP,
    XILPM_RESET_SWDT_CRF,
    XILPM_RESET_AFI_FM5,
    XILPM_RESET_AFI_FM4,
    XILPM_RESET_AFI_FM3,
    XILPM_RESET_AFI_FM2,
    XILPM_RESET_AFI_FM1,
    XILPM_RESET_AFI_FM0,
    XILPM_RESET_GDMA,
    XILPM_RESET_GPU_PP1,
    XILPM_RESET_GPU_PP0,
    XILPM_RESET_GPU,
    XILPM_RESET_GT,
    XILPM_RESET_SATA,
    XILPM_RESET_ACPU3_PWRON,
    XILPM_RESET_ACPU2_PWRON,
    XILPM_RESET_ACPU1_PWRON,
    XILPM_RESET_ACPU0_PWRON,
    XILPM_RESET_APU_L2,
    XILPM_RESET_ACPU3,
    XILPM_RESET_ACPU2,
    XILPM_RESET_ACPU1,
    XILPM_RESET_ACPU0,
    XILPM_RESET_DDR,
    XILPM_RESET_APM_FPD,
    XILPM_RESET_SOFT,
    XILPM_RESET_GEM0,
    XILPM_RESET_GEM1,
    XILPM_RESET_GEM2,
    XILPM_RESET_GEM3,
```

```
XILPM_RESET_QSPI,  
XILPM_RESET_UART0,  
XILPM_RESET_UART1,  
XILPM_RESET_SPI0,  
XILPM_RESET_SPI1,  
XILPM_RESET_SDIO0,  
XILPM_RESET_SDIO1,  
XILPM_RESET_CAN0,  
XILPM_RESET_CAN1,  
XILPM_RESET_I2C0,  
XILPM_RESET_I2C1,  
XILPM_RESET_TTC0  
XILPM_RESET_TTC1,  
XILPM_RESET_TTC2,  
XILPM_RESET_TTC3,  
XILPM_RESET_SWDT_CRL,  
XILPM_RESET_NAND,  
XILPM_RESET_ADMA,  
XILPM_RESET_GPIO,  
XILPM_RESET_IOU_CC,  
XILPM_RESET_TIMESTAMP,  
XILPM_RESET_RPU_R50,  
XILPM_RESET_RPU_R51,  
XILPM_RESET_RPU_AMBA,  
XILPM_RESET_OCM,  
XILPM_RESET_RPU_PGE,  
XILPM_RESET_USB0_COREREST,  
XILPM_RESET_USB1_COREREST,  
XILPM_RESET_USB0_HIBERREST,  
XILPM_RESET_USB1_HIBERREST,  
XILPM_RESET_USB0_APB,  
XILPM_RESET_USB1_APB,  
XILPM_RESET_IPI,  
XILPM_RESET_APM_LPD,  
XILPM_RESET_RTC,  
XILPM_RESET_SYSMON,  
XILPM_RESET_AFI_FM6,  
XILPM_RESET_LPD_SWDT,  
XILPM_RESET_FPD,  
XILPM_RESET_RPU_DBG1,  
XILPM_RESET_RPU_DBG0,  
XILPM_RESET_DBG_LPD,
```

```
XILPM_RESET_DBG_FPD,  
XILPM_RESET_APLL,  
XILPM_RESET_DPLL,  
XILPM_RESET_VPLL,  
XILPM_RESET_IOPLL,  
XILPM_RESET_RPLL,  
XILPM_RESET_GPO3_PL_0,  
XILPM_RESET_GPO3_PL_1,  
XILPM_RESET_GPO3_PL_2,  
XILPM_RESET_GPO3_PL_3,  
XILPM_RESET_GPO3_PL_4,  
XILPM_RESET_GPO3_PL_5,  
XILPM_RESET_GPO3_PL_6,  
XILPM_RESET_GPO3_PL_7,  
XILPM_RESET_GPO3_PL_8,  
XILPM_RESET_GPO3_PL_9,  
XILPM_RESET_GPO3_PL_10,  
XILPM_RESET_GPO3_PL_11,  
XILPM_RESET_GPO3_PL_12,  
XILPM_RESET_GPO3_PL_13,  
XILPM_RESET_GPO3_PL_14,  
XILPM_RESET_GPO3_PL_15,  
XILPM_RESET_GPO3_PL_16,  
XILPM_RESET_GPO3_PL_17,  
XILPM_RESET_GPO3_PL_18,  
XILPM_RESET_GPO3_PL_19,  
XILPM_RESET_GPO3_PL_20,  
XILPM_RESET_GPO3_PL_21,  
XILPM_RESET_GPO3_PL_22,  
XILPM_RESET_GPO3_PL_23,  
XILPM_RESET_GPO3_PL_24,  
XILPM_RESET_GPO3_PL_25,  
XILPM_RESET_GPO3_PL_26,  
XILPM_RESET_GPO3_PL_27,  
XILPM_RESET_GPO3_PL_28,  
XILPM_RESET_GPO3_PL_29,  
XILPM_RESET_GPO3_PL_30,  
XILPM_RESET_GPO3_PL_31,  
};
```

XPm_Notifier struct

The XPm_Notifier struct is the structure to be passed in XPm_RegisterNotifier.

```
typedef struct XPm_Notifier {
    void (*const callback)(XPm_Notifier* const notifier);
    enum XPmNodeId node;
    enum XPmNotifyEvent event;
    u32 flags;
    volatile u32 appoint;
    volatile u32 received;
    XPm_Notifier* next;
} XPm_Notifier;
```

Struct Members

Table A-2: Struct Members

Name	Description
callback	Custom callback handler to be called when the notification is received. The custom handler executes from the interrupt context; hence, it shall return quickly and must not block! (enables event-driven notifications),
node	Node argument (the node to receive notifications about).
event	Event argument (the event type to receive notifications about).
flags	Flags: Currently the flags only contain the wake option in bit0. <ul style="list-style-type: none"> • flags = 1: wake up on event • flags = 0: do not wake up (only notify if awake), no buffering or queueing will take place
appoint	Operating point of node in question. Contains the value updated when the last event notification is received. User shall not modify this value while the notifier is registered.
received	How many times the notification has been received - to be used by application (enables polling). User shall not modify this value while the notifier is registered.
next	Pointer to next notifier in linked list. Must not be modified while the notifier is registered. User shall not ever modify this value.

XPm_NodeStatus struct

The XPm_NodeStatus struct is used to pass node status information.

```
typedef struct XPm_NodeStatus {
    u32 status;
    u32 requirements;
    u32 usage;
} XPm_NodeStatus;
```

Struct Members

Table A-3: Struct Members

Name	Description
status	Node power state.
requirements	Current requirements asserted on the node (slaves only).
usage	Usage information (which master is currently using the slave). This information is used for slave nodes only. It is encoded based on the IPI bits for the masters. If the respective bit is set, the corresponding master is currently using the node.

XilPM Error Codes

Introduction

The following is a list of possible error codes returned by the PM API.

Table A-4: Error Codes and Explanations

Error Code	Explanation
XST_FAILURE	Power management controller has failed to comply with the request, because of a hardware/PMU-ROM failure or because the API cannot be processed in the given circumstances.
XST_INVALID_PARAM	An argument is either out-of-range or its value is not admissible in the respective API call.
XST_NO_FEATURE	The requested feature is not available for the selected PM slave.
XST_PM_CONFLICT	Conflicting requirements have been asserted when more than one PU is using the same PM slave.
XST_PM_DOUBLE_REQ	XPm_RequestNode: A PU has already been assigned access to a PM slave and has issued a duplicate request for that PM slave.
XST_PM_INTERNAL	Unexpected error in the PMU state machine. Should be reported as a bug.
XST_PM_INVALID_NODE	The API function does not apply to the node passed as argument.
XST_PM_NO_ACCESS	The PU does not have access to the requested node or operation.
XST_PM_ABORT_SUSPEND	The target PU has aborted suspend.

Appendix B:

Xilinx Standard C Libraries

Xilinx Standard C Libraries

Overview

The Xilinx® Software Development Kit (SDK) libraries and device drivers provide standard C library functions, as well as functions to access peripherals. The SDK libraries are automatically configured based on the Microprocessor Software Specification (MSS) file. These libraries and include files are saved in the current project lib and include directories, respectively. The -I and -L options of mb-gcc are used to add these directories to its library search paths.

Standard C Library (libc.a)

The standard C library, `libc.a`, contains the standard C functions compiled for the MicroBlaze™ processor or the Cortex A9 processor. You can find the header files corresponding to these C standard functions in the `<XILINX_SDK>/gnu/<processor>/<platform>/<processor-lib>/include` folder, where:

- `<XILINX_SDK>` is the Xilinx SDK installation path
- `<processor>` is ARM or MicroBlaze
- `<platform>` is Solaris (sol), Windows (nt), or Linux (lin)
- `<processor-lib>` is `arm-xilinx-eabi` or `microblaze-xilinx-elf`

The `lib.c` directories and functions are:

<code>_ansi.h</code>	<code>fastmath.h</code>	<code>machine/</code>	<code>reent.h</code>	<code>stdlib.h</code>	<code>utime.h</code>	<code>_syslist.h</code>	<code>fcntl.h</code>	<code>malloc.h</code>
<code>regdef.h</code>	<code>string.h</code>	<code>utmp.h</code>	<code>ar.h</code>	<code>float.h</code>	<code>math.h</code>	<code>setjmp.h</code>	<code>sys/</code>	<code>assert.h</code>
<code>grp.h</code>	<code>paths.h</code>	<code>signal.h</code>	<code>termios.h</code>	<code>ctype.h</code>	<code>ieeefp.h</code>	<code>process.h</code>	<code>stdarg.h</code>	<code>time.h</code>
<code>dirent.h</code>	<code>imits.h</code>	<code>pthread.h</code>	<code>stddef.h</code>	<code>nctrl.h</code>	<code>errno.h</code>	<code>locale.h</code>	<code>pwd.h</code>	<code>stdio.h</code>
<code>unistd.h</code>								

Programs accessing standard C library functions must be compiled as follows:

- For MicroBlaze processors:

```
mb-gcc <C files>
```

- For Cortex A9 processors:

```
arm-xilinx-eabi-gcc <C files>
```

The `libc` library is included automatically. For programs that access `libm` math functions, specify the `-lm` option. For more information on the C runtime library, see *MicroBlaze Processor Reference Guide* (UG081).

Xilinx C Library (libxil.a)

The Xilinx C library, libxil.a, contains the following object files for the MicroBlaze processor embedded processor:

- _exception_handler.o
- _interrupt_handler.o
- _program_clean.o
- _program_init.o

Default exception and interrupt handlers are provided. The libxil.a library is included automatically. Programs accessing Xilinx C library functions must be compiled as follows:

```
mb-gcc <C files>
```

Memory Management Functions

The MicroBlaze processor and Cortex A9 processor C libraries support the standard memory management functions such as malloc(), calloc(), and free(). Dynamic memory allocation provides memory from the program heap. The heap pointer starts at low memory and grows toward high memory. The size of the heap cannot be increased at runtime. Therefore an appropriate value must be provided for the heap size at compile time. The malloc() function requires the heap to be at least 128 bytes in size to be able to allocate memory dynamically (even if the dynamic requirement is less than 128 bytes).

Note

The return value of malloc must always be checked to ensure that it could actually allocate the memory requested.

Arithmetic Operations

Software implementations of integer and floating point arithmetic is available as library routines in libgcc.a for both processors. The compiler for both the processors inserts calls to these routines in the code produced, in case the hardware does not support the arithmetic primitive with an instruction.

MicroBlaze Processor

Details of the software implementations of integer and floating point arithmetic for MicroBlaze processors are listed below:

Integer Arithmetic

By default, integer multiplication is done in software using the library function `__mulsi3`. Integer multiplication is done in hardware if the `-mno-xl-soft-mul mb-gcc` option is specified.

Integer divide and mod operations are done in software using the library functions `__divsi3` and `__modsi3`. The MicroBlaze processor can also be customized to use a hard divider, in which case the `div` instruction is used in place of the `__divsi3` library routine.

Double precision multiplication, division and mod functions are carried out by the library functions `__muldi3`, `__divdi3`, and `__moddi3` respectively.

The unsigned version of these operations correspond to the signed versions described above, but are prefixed with an `__u` instead of `__`.

Floating Point Arithmetic

All floating point addition, subtraction, multiplication, division, and conversions are implemented using software functions in the C library.

Thread Safety

The standard C library provided with SDK is not built for a multi-threaded environment. STDIO functions like `printf()`, `scanf()` and memory management functions like `malloc()` and `free()` are common examples of functions that are not thread-safe. When using the C library in a multi-threaded environment, proper mutual exclusion techniques must be used to protect thread unsafe functions.

Modules

- Input/Output Functions
-

Input/Output Functions

Overview

The SDK libraries contains standard C functions for I/O, such as `printf` and `scanf`. These functions are large and might not be suitable for embedded processors. The prototypes for these functions are available in the `stdio.h` file.

Note

The C standard I/O routines such as `printf`, `scanf`, `vfprintf` are, by default, line buffered. To change the buffering scheme to no buffering, you must call `setvbuf` appropriately. For example:

```
setvbuf (stdout, NULL, _IONBF, 0);
```

These Input/Output routines require that a newline is terminated with both a CR and LF. Ensure that your terminal CR/LF behavior corresponds to this requirement.

For more information on setting the standard input and standard output devices for a system, see *Embedded System Tools Reference Manual* (UG1043). In addition to the standard C functions, the SDK processors library provides the following smaller I/O functions:

Functions

- void `print` (char *)
- void `putnum` (int)
- void `xil_printf` (const *char ctrl1,...)

Function Documentation

void print (char *)

This function prints a string to the peripheral designated as standard output in the Microprocessor Software Specification (MSS) file. This function outputs the passed string as is and there is no interpretation of the string passed. For example, a \n passed is interpreted as a new line character and not as a carriage return and a new line as is the case with ANSI C printf function.

void putnum (int)

This function converts an integer to a hexadecimal string and prints it to the peripheral designated as standard output in the MSS file.

void xil_printf (const *char ctrl1, ...)

`xil_printf()` is a light-weight implementation of `printf`. It is much smaller in size (only 1 Kb). It does not have support for floating point numbers. `xil_printf()` also does not support printing of long (such as 64-bit) numbers.

About format string support:

The format string is composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character %, and ends with a conversion specifier.

In between there can be (in order) zero or more flags, an optional minimum field width and an optional precision. Supported flag characters are:

The character % is followed by zero or more of the following flags:

- 0 The value should be zero padded. For d, x conversions, the converted value is padded on the left with zeros rather than blanks. If the 0 and - flags both appear, the 0 flag is ignored.
- - The converted value is to be left adjusted on the field boundary. (The default is right justification.) Except for n conversions, the converted value is padded on the right with blanks, rather than on the left with blanks or zeros. A - overrides a 0 if both are given.

About supported field widths

Field widths are represented with an optional decimal digit string (with a nonzero in the first digit) specifying a minimum field width. If the converted value has fewer characters than the field width, it is padded with spaces on the left (or right, if the left-adjustment flag has been given). The supported conversion specifiers are:

- d The int argument is converted to signed decimal notation.

- l The int argument is converted to a signed long notation.
- x The unsigned int argument is converted to unsigned hexadecimal notation. The letters abcdef are used for x conversions.
- c The int argument is converted to an unsigned char, and the resulting character is written.
- s The const char* argument is expected to be a pointer to an array of character type (pointer to a string).

Characters from the array are written up to (but not including) a terminating NULL character; if a precision is specified, no more than the number specified are written. If a precision s given, no null character need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating NULL character.

Appendix C:

Standalone Library Reference v6.2

Xilinx Hardware Abstraction Layer API

Overview

This section describes the Xilinx® Hardware Abstraction Layer API. These APIs are applicable for all processors supported by Xilinx.

Modules

- Assert APIs
- IO interfacing APIs
- Definitions for available xilinx platforms
- Data types for Xilinx Software IP Cores
- Customized APIs for memory operations
- Xilinx software status codes
- Test utilities for memory and caches

Assert APIs

Overview

The `xil_assert.h` file contains the assert related functions.

Macros

- `#define Xil_AssertVoid(Expression)`
- `#define Xil_AssertNonvoid(Expression)`
- `#define Xil_AssertVoidAlways()`
- `#define Xil_AssertNonvoidAlways()`

Typedefs

- `typedef void(* Xil_AssertCallback) (const char8 *File, s32 Line)`

Functions

- void `Xil_Assert` (const `char8` *File, s32 Line)
- void `XNullHandler` (void *NullParameter)
- void `Xil_AssertSetCallback` (`Xil_AssertCallback` Routine)

Variables

- u32 `Xil_AssertStatus`
- s32 `Xil_AssertWait`

Macro Definition Documentation

#define Xil_AssertVoid(*Expression*)

This assert macro is to be used for void functions. This in conjunction with the `Xil_AssertWait` boolean can be used to accomodate tests so that asserts which fail allow execution to continue.

Parameters

<code>Expression</code>	expression to be evaluated. If it evaluates to false, the assert occurs.
-------------------------	--

Returns

Returns void unless the `Xil_AssertWait` variable is true, in which case no return is made and an infinite loop is entered.

#define Xil_AssertNonvoid(*Expression*)

This assert macro is to be used for functions that do return a value. This in conjunction with the `Xil_AssertWait` boolean can be used to accomodate tests so that asserts which fail allow execution to continue.

Parameters

<code>Expression</code>	expression to be evaluated. If it evaluates to false, the assert occurs.
-------------------------	--

Returns

Returns 0 unless the `Xil_AssertWait` variable is true, in which case no return is made and an infinite loop is entered.

#define Xil_AssertVoidAlways()

Always assert. This assert macro is to be used for void functions. Use for instances where an assert should always occur.

Returns

Returns void unless the Xil_AssertWait variable is true, in which case no return is made and an infinite loop is entered.

#define Xil_AssertNonvoidAlways()

Always assert. This assert macro is to be used for functions that do return a value. Use for instances where an assert should always occur.

Returns

Returns void unless the Xil_AssertWait variable is true, in which case no return is made and an infinite loop is entered.

TypeDef Documentation

typedef void(* Xil_AssertCallback) (const char8 *File, s32 Line)

This data type defines a callback to be invoked when an assert occurs. The callback is invoked only when asserts are enabled

Function Documentation

void Xil_Assert (const char8 * File, s32 Line)

Implement assert. Currently, it calls a user-defined callback function if one has been set. Then, it potentially enters an infinite loop depending on the value of the Xil_AssertWait variable.

Parameters

<i>file</i>	filename of the source
<i>line</i>	linenumber within File

Returns

None.

Note

None.

void XNullHandler (void * *NullParameter*)

Null handler function. This follows the XIInterruptHandler signature for interrupt handlers. It can be used to assign a null handler (a stub) to an interrupt controller vector table.

Parameters

<i>NullParameter</i>	arbitrary void pointer and not used.
----------------------	--------------------------------------

Returns

None.

Note

None.

void Xil_AssertSetCallback (Xil_AssertCallback *Routine*)

Set up a callback function to be invoked when an assert occurs. If a callback is already installed, then it will be replaced.

Parameters

<i>routine</i>	callback to be invoked when an assert is taken
----------------	--

Returns

None.

Note

This function has no effect if NDEBUG is set

Variable Documentation

u32 Xil_AssertStatus

This variable allows testing to be done easier with asserts. An assert sets this variable such that a driver can evaluate this variable to determine if an assert occurred.

s32 Xil AssertWait

This variable allows the assert functionality to be changed for testing such that it does not wait infinitely. Use the debugger to disable the waiting during testing of asserts.

IO interfacing APIs

Overview

The `xil_io.h` file contains the interface for the general IO component, which encapsulates the Input/Output functions for processors that do not require any special I/O handling.

Functions

- u16 `Xil_EndianSwap16` (u16 Data)
- u32 `Xil_EndianSwap32` (u32 Data)
- static INLINE u8 `Xil_In8` (UINTPTR Addr)
- static INLINE u16 `Xil_In16` (UINTPTR Addr)
- static INLINE u32 `Xil_In32` (UINTPTR Addr)
- static INLINE u64 `Xil_In64` (UINTPTR Addr)
- static INLINE void `Xil_Out8` (UINTPTR Addr, u8 Value)
- static INLINE void `Xil_Out16` (UINTPTR Addr, u16 Value)
- static INLINE void `Xil_Out32` (UINTPTR Addr, u32 Value)
- static INLINE void `Xil_Out64` (UINTPTR Addr, u64 Value)
- static INLINE u16 `Xil_In16LE` (UINTPTR Addr)
- static INLINE u32 `Xil_In32LE` (UINTPTR Addr)
- static INLINE void `Xil_Out16LE` (UINTPTR Addr, u16 Value)
- static INLINE void `Xil_Out32LE` (UINTPTR Addr, u32 Value)
- static INLINE u16 `Xil_In16BE` (UINTPTR Addr)
- static INLINE u32 `Xil_In32BE` (UINTPTR Addr)
- static INLINE void `Xil_Out16BE` (UINTPTR Addr, u16 Value)
- static INLINE void `Xil_Out32BE` (UINTPTR Addr, u32 Value)

Function Documentation

u16 Xil_EndianSwap16 (u16 Data)

Perform a 16-bit endian converion.

Parameters

<i>Data</i>	16 bit value to be converted
-------------	------------------------------

Returns

converted value.

u32 Xil_EndianSwap32 (u32 Data)

Perform a 32-bit endian converion.

Parameters

<i>Data</i>	32 bit value to be converted
-------------	------------------------------

Returns

converted value.

static INLINE u8 Xil_In8 (UINTPTR Addr) [static]

Performs an input operation for an 8-bit memory location by reading from the specified address and returning the Value read from that address.

Parameters

<i>Addr</i>	contains the address to perform the input operation at.
-------------	---

Returns

The Value read from the specified input address.

Note

None.

static INLINE u16 Xil_In16 (**UINTPTR Addr**) [static]

Performs an input operation for a 16-bit memory location by reading from the specified address and returning the Value read from that address.

Parameters

<i>Addr</i>	contains the address to perform the input operation at.
-------------	---

Returns

The Value read from the specified input address.

Note

None.

static INLINE u32 Xil_In32 (**UINTPTR Addr**) [static]

Performs an input operation for a 32-bit memory location by reading from the specified address and returning the Value read from that address.

Parameters

<i>Addr</i>	contains the address to perform the input operation at.
-------------	---

Returns

The Value read from the specified input address.

Note

None.

static INLINE u64 Xil_In64 (**UINTPTR Addr**) [static]

Performs an input operation for a 64-bit memory location by reading the specified Value to the the specified address.

Parameters

<i>Addr</i>	contains the address to perform the output operation at.
<i>Value</i>	contains the Value to be output at the specified address.

Returns

None.

Note

None.

static INLINE void Xil_Out8 (**UINTPTR Addr, u8 Value) [static]**

Performs an output operation for an 8-bit memory location by writing the specified Value to the the specified address.

Parameters

<i>Addr</i>	contains the address to perform the output operation at.
<i>Value</i>	contains the Value to be output at the specified address.

Returns

None.

Note

None.

static INLINE void Xil_Out16 (**UINTPTR Addr, u16 Value) [static]**

Performs an output operation for a 16-bit memory location by writing the specified Value to the the specified address.

Parameters

<i>Addr</i>	contains the address to perform the output operation at.
<i>Value</i>	contains the Value to be output at the specified address.

Returns

None.

Note

None.

static INLINE void Xil_Out32 (**UINTPTR Addr, u32 Value) [static]**

Performs an output operation for a 32-bit memory location by writing the specified Value to the the specified address.

Parameters

<i>Addr</i>	contains the address to perform the output operation at.
<i>Value</i>	contains the Value to be output at the specified address.

Returns

None.

Note

None.

static INLINE void Xil_Out64 (**UINTPTR Addr, u64 Value) [static]**

Performs an output operation for a 64-bit memory location by writing the specified Value to the the specified address.

Parameters

<i>Addr</i>	contains the address to perform the output operation at.
<i>Value</i>	contains the Value to be output at the specified address.

Returns

None.

Note

None.

static INLINE u16 Xil_In16LE (**UINTPTR Addr) [static]**

Perform a little-endian input operation for a 16-bit memory location by reading from the specified address and returning the value read from that address.

Parameters

<i>Addr</i>	contains the address at which to perform the input operation.
-------------	---

Returns

The value read from the specified input address with the proper endianness. The return value has the same endianness as that of the processor. For example, if the processor is big-endian, the return value is the byte-swapped value read from the address.

static INLINE u32 Xil_In32LE (**UINTPTR Addr**) [static]

Perform a little-endian input operation for a 32-bit memory location by reading from the specified address and returning the value read from that address.

Parameters

Addr	contains the address at which to perform the input operation.
-------------	---

Returns

The value read from the specified input address with the proper endianness. The return value has the same endianness as that of the processor. For example, if the processor is big-endian, the return value is the byte-swapped value read from the address.

static INLINE void Xil_Out16LE (**UINTPTR Addr**, **u16 Value**) [static]

Perform a little-endian output operation for a 16-bit memory location by writing the specified value to the specified address.

Parameters

Addr	contains the address at which to perform the output operation.
Value	contains the value to be output at the specified address. The value has the same endianness as that of the processor. For example, if the processor is big-endian, the byteswapped value is written to the address.

static INLINE void Xil_Out32LE (**UINTPTR Addr**, **u32 Value**) [static]

Perform a little-endian output operation for a 32-bit memory location by writing the specified value to the specified address.

Parameters

Addr	contains the address at which to perform the output operation.
Value	contains the value to be output at the specified address. The value has the same endianness as that of the processor. For example, if the processor is big-endian, the byteswapped value is written to the address.

static INLINE u16 Xil_In16BE (UINTPTR Addr) [static]

Perform a big-endian input operation for a 16-bit memory location by reading from the specified address and returning the value read from that address.

Parameters

Addr	contains the address at which to perform the input operation.
------	---

Returns

The value read from the specified input address with the proper endianness. The return value has the same endianness as that of the processor. For example, if the processor is little-endian, the return value is the byte-swapped value read from the address.

static INLINE u32 Xil_In32BE (UINTPTR Addr) [static]

Perform a big-endian input operation for a 32-bit memory location by reading from the specified address and returning the value read from that address.

Parameters

Addr	contains the address at which to perform the input operation.
------	---

Returns

The value read from the specified input address with the proper endianness. The return value has the same endianness as that of the processor. For example, if the processor is little-endian, the return value is the byte-swapped value read from the address.

static INLINE void Xil_Out16BE (UINTPTR Addr, u16 Value) [static]

Perform a big-endian output operation for a 16-bit memory location by writing the specified value to the specified address.

Parameters

Addr	contains the address at which to perform the output operation.
Value	contains the value to be output at the specified address. The value has the same endianness as that of the processor. For example, if the processor is little-endian, the byteswapped value is written to the address.

static INLINE void Xil_Out32BE (**UINTPTR Addr, u32 Value**) [static]

Perform a big-endian output operation for a 32-bit memory location by writing the specified value to the specified address.

Parameters

<i>Addr</i>	contains the address at which to perform the output operation.
<i>Value</i>	contains the value to be output at the specified address. The value has the same endianness as that of the processor. For example, if the processor is little-endian, the byteswapped value is written to the address.

Definitions for available xilinx platforms

Overview

The xplatform_info.h file contains definitions for various available Xilinx® platforms.

Functions

- u32 [XGetPlatform_Info \(\)](#)
- u32 [XGetPSVersion_Info \(\)](#)
- u32 [XGet_Zynq_UltraMp_Platform_info \(\)](#)

Function Documentation

u32 XGetPlatform_Info ()

This API is used to provide information about platform.

Parameters

<i>None.</i>	
--------------	--

Returns

The information about platform defined in xplatform_info.h

u32 XGetPSVersion_Info()

This API is used to provide information about PS Silicon version.

Parameters

None.	
-------	--

Returns

The information about PS Silicon version.

u32 XGet_Zynq_UltraMp_Platform_info()

This API is used to provide information about zynq ultrascale MP platform.

Parameters

None.	
-------	--

Returns

The information about zynq ultrascale MP platform defined in xplatform_info.h

Data types for Xilinx Software IP Cores

Overview

The `xil_types.h` file contains basic types for Xilinx® software IP cores. These data types are applicable for all processors supported by Xilinx.

Macros

- `#define XIL_COMPONENT_IS_READY`
- `#define XIL_COMPONENT_IS_STARTED`

New types

New simple types.

- `typedef uint8_t u8`
- `typedef uint16_t u16`
- `typedef uint32_t u32`
- `typedef char char8`
- `typedef int8_t s8`

- `typedef int16_t s16`
- `typedef int32_t s32`
- `typedef int64_t s64`
- `typedef uint64_t u64`
- `typedef int sint32`
- `typedef intptr_t INTPTR`
- `typedef uintptr_t UINTPTR`
- `typedef ptrdiff_t PTRDIFF`
- `typedef long LONG`
- `typedef unsigned long ULONG`
- `typedef void(* XInterruptHandler) (void *InstancePtr)`
- `typedef void(* XExceptionHandler) (void *InstancePtr)`
- `#define __XUINT64__`
- `#define XUINT64_MSW(x)`
- `#define XUINT64_LSW(x)`
- `#define ULONG64_HI_MASK`
- `#define ULONG64_LO_MASK`
- `#define UPPER_32_BITS(n)`
- `#define LOWER_32_BITS(n)`

Macro Definition Documentation

#define XIL_COMPONENT_IS_READY

component has been initialized

#define XIL_COMPONENT_IS_STARTED

component has been started

#define XUINT64_MSW(x)

Return the most significant half of the 64 bit data type.

Parameters

<code>x</code>	is the 64 bit word.
----------------	---------------------

Returns

The upper 32 bits of the 64 bit word.

#define XUINT64_LSW(x)

Return the least significant half of the 64 bit data type.

Parameters

x	is the 64 bit word.
---	---------------------

Returns

The lower 32 bits of the 64 bit word.

#define UPPER_32_BITS(n)

return bits 32-63 of a number

Parameters

n	: the number we're accessing
---	------------------------------

Returns

bits 32-63 of number

Note

A basic shift-right of a 64- or 32-bit quantity. Use this to suppress the "right shift count >= width of type" warning when that quantity is 32-bits.

#define LOWER_32_BITS(n)

return bits 0-31 of a number

Parameters

n	: the number we're accessing
---	------------------------------

Returns

bits 0-31 of number

Typedef Documentation

typedef uint8_t u8

guarded against xbasic_types.h.

typedef char char8

xbasic_types.h does not typedef s* or u64

typedef void(* XInterruptHandler) (void *InstancePtr)

This data type defines an interrupt handler for a device. The argument points to the instance of the component

typedef void(* XExceptionHandler) (void *InstancePtr)

This data type defines an exception handler for a processor. The argument points to the instance of the component

Customized APIs for memory operations

Overview

The xil_mem.h file contains prototypes for function related to memory operations. These APIs are applicable for all processors supported by Xilinx®.

Functions

- void [Xil_MemCpy](#) (void *dst, const void *src, u32 cnt)

Function Documentation

void Xil_MemCpy (void * dst, const void * src, u32 cnt)

This function copies memory from once location to other.

Parameters

<i>dst</i>	pointer pointing to destination memory
<i>src</i>	pointer pointing to source memory
<i>cnt</i>	32 bit length of bytes to be copied

Xilinx software status codes

Overview

The xstatus.h file contains Xilinx® software status codes. Status codes have their own data type called int. These codes are used throughout the Xilinx device drivers.

Test utilities for memory and caches

Overview

The xil_testcache.h, xil_testio.h and the xil_testmem.h files contain utility functions to test cache and memory. Details of supported tests and subtests are listed below.

- **Cache test** : xil_testcache.h contains utility functions to test cache.
- **I/O test** : The Xil_testio.h file contains endian related memory IO functions. A subset of the memory tests can be selected or all of the tests can be run in order. If there is an error detected by a subtest, the test stops and the failure code is returned. Further tests are not run even if all of the tests are selected.
- **Memory test** : The xil_testmem.h file contains utility functions to test memory. A subset of the memory tests can be selected or all of the tests can be run in order. If there is an error detected by a subtest, the test stops and the failure code is returned. Further tests are not run even if all of the tests are selected.

Following are descriptions of Memory test subtests:

- XIL_TESTMEM_ALLMEMTESTS: Runs all of the subtests.
- XIL_TESTMEM_INCREMENT: Incrementing Value Test. This test starts at XIL_TESTMEM_INIT_VALUE and uses the incrementing value as the test value for memory.
- XIL_TESTMEM_WALKONES: Walking Ones Test. This test uses a walking 1 as the test value for memory.

```
location 1 = 0x00000001  
location 2 = 0x00000002  
...
```

- XIL_TESTMEM_WALKZEROS: Walking Zero's Test. This test uses the inverse value of the walking ones test as the test value for memory.

```
location 1 = 0xFFFFFFF  
location 2 = 0xFFFFFFFF  
...
```

- XIL_TESTMEM_INVERSEADDR: Inverse Address Test. This test uses the inverse of the address of the location under test as the test value for memory.
- XIL_TESTMEM_FIXEDPATTERN: Fixed Pattern Test. This test uses the provided patters as the test value for memory. If zero is provided as the pattern the test uses 0xDEADBEEF.

WARNING: The tests are **DESTRUCTIVE**. Run before any initialized memory spaces have been set up.



The address provided to the memory tests is not checked for validity except for the NULL case. It is possible to provide a code-space pointer for this test to start with and ultimately destroy executable code causing random failures.

Note

Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2 ** width, the patterns used in XIL_TESTMEM_WALKONES and XIL_TESTMEM_WALKZEROS will repeat on a boundary of a power of two making it more difficult to detect addressing errors. The XIL_TESTMEM_INCREMENT and XIL_TESTMEM_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

Functions

- s32 [Xil_TestIO8](#) (u8 *Addr, s32 Length, u8 Value)
- s32 [Xil_TestIO16](#) (u16 *Addr, s32 Length, u16 Value, s32 Kind, s32 Swap)
- s32 [Xil_TestIO32](#) (u32 *Addr, s32 Length, u32 Value, s32 Kind, s32 Swap)
- s32 [Xil_TestMem32](#) (u32 *Addr, u32 Words, u32 Pattern, u8 Subtest)
- s32 [Xil_TestMem16](#) (u16 *Addr, u32 Words, u16 Pattern, u8 Subtest)
- s32 [Xil_TestMem8](#) (u8 *Addr, u32 Words, u8 Pattern, u8 Subtest)

Memory subtests

- #define [XIL_TESTMEM_ALLMEMTESTS](#)
- #define [XIL_TESTMEM_INCREMENT](#)
- #define [XIL_TESTMEM_WALKONES](#)
- #define [XIL_TESTMEM_WALKZEROS](#)
- #define [XIL_TESTMEM_INVERSEADDR](#)
- #define [XIL_TESTMEM_FIXEDPATTERN](#)
- #define [XIL_TESTMEM_MAXTEST](#)

Macro Definition Documentation

#define XIL_TESTMEM_ALLMEMTESTS

See the detailed description of the subtests in the file description.

Function Documentation

s32 Xil_TestIO8 (*u8 * Addr, s32 Length, u8 Value*)

Perform a destructive 8-bit wide register IO test where the register is accessed using Xil_Out8 and Xil_In8, and comparing the written values by reading them back.

Parameters

<i>Addr</i>	a pointer to the region of memory to be tested.
<i>Length</i>	Length of the block.
<i>Value</i>	constant used for writting the memory.

Returns

- -1 is returned for a failure
- 0 is returned for a pass

s32 Xil_TestIO16 (*u16 * Addr, s32 Length, u16 Value, s32 Kind, s32 Swap*)

Perform a destructive 16-bit wide register IO test. Each location is tested by sequentially writing a 16-bit wide register, reading the register, and comparing value. This function tests three kinds of register IO functions, normal register IO, little-endian register IO, and big-endian register IO. When testing little/big-endian IO, the function performs the following sequence, Xil_Out16LE/Xil_Out16BE, Xil_In16, Compare In-Out values, Xil_Out16, Xil_In16LE/Xil_In16BE, Compare In-Out values. Whether to swap the read-in value before comparing is controlled by the 5th argument.

Parameters

<i>Addr</i>	a pointer to the region of memory to be tested.
<i>Length</i>	Length of the block.
<i>Value</i>	constant used for writting the memory.
<i>Kind</i>	Type of test. Acceptable values are: XIL_TESTIO_DEFAULT, XIL_TESTIO_LE, XIL_TESTIO_BE.
<i>Swap</i>	indicates whether to byte swap the read-in value.

Returns

- -1 is returned for a failure
- 0 is returned for a pass

s32 Xil_TestIO32 (**u32 * Addr, s32 Length, u32 Value, s32 Kind, s32 Swap**)

Perform a destructive 32-bit wide register IO test. Each location is tested by sequentially writing a 32-bit wide register, reading the register, and comparing value. This function tests three kinds of register IO functions, normal register IO, little-endian register IO, and big-endian register IO. When testing little/big-endian IO, the function performs the following sequence, Xil_Out32LE/ Xil_Out32BE, Xil_In32, Compare, Xil_Out32, Xil_In32LE/Xil_In32BE, Compare. Whether to swap the read-in value before comparing is controlled by the 5th argument.

Parameters

<i>Addr</i>	a pointer to the region of memory to be tested.
<i>Length</i>	Length of the block.
<i>Value</i>	constant used for writing the memory.
<i>Kind</i>	type of test. Acceptable values are: XIL_TESTIO_DEFAULT, XIL_TESTIO_LE, XIL_TESTIO_BE.
<i>Swap</i>	indicates whether to byte swap the read-in value.

Returns

- -1 is returned for a failure
- 0 is returned for a pass

s32 Xil_TestMem32 (**u32 * Addr, u32 Words, u32 Pattern, u8 Subtest**)

Perform a destructive 32-bit wide memory test.

Parameters

<i>Addr</i>	pointer to the region of memory to be tested.
<i>Words</i>	length of the block.
<i>Pattern</i>	constant used for the constant pattern test, if 0, 0xDEADBEEF is used.
<i>Subtest</i>	test type selected. See xil_testmem.h for possible values.

Returns

- 0 is returned for a pass
- 1 is returned for a failure

Note

Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2 ** Width, the patterns used in XIL_TESTMEM_WALKONES and XIL_TESTMEM_WALKZEROS will repeat on a boundary of a power of two making it more difficult to detect addressing errors. The XIL_TESTMEM_INCREMENT and XIL_TESTMEM_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

s32 Xil_TestMem16 (u16 * Addr, u32 Words, u16 Pattern, u8 Subtest)

Perform a destructive 16-bit wide memory test.

Parameters

<i>Addr</i>	pointer to the region of memory to be tested.
<i>Words</i>	length of the block.
<i>Pattern</i>	constant used for the constant Pattern test, if 0, 0xDEADBEEF is used.
<i>Subtest</i>	type of test selected. See <i>xil_testmem.h</i> for possible values.

Returns

- -1 is returned for a failure
- 0 is returned for a pass

Note

Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2 ** Width, the patterns used in XIL_TESTMEM_WALKONES and XIL_TESTMEM_WALKZEROS will repeat on a boundary of a power of two making it more difficult to detect addressing errors. The XIL_TESTMEM_INCREMENT and XIL_TESTMEM_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

s32 Xil_TestMem8 (u8 * Addr, u32 Words, u8 Pattern, u8 Subtest)

Perform a destructive 8-bit wide memory test.

Parameters

<i>Addr</i>	pointer to the region of memory to be tested.
<i>Words</i>	length of the block.
<i>Pattern</i>	constant used for the constant pattern test, if 0, 0xDEADBEEF is used.
<i>Subtest</i>	type of test selected. See <i>xil_testmem.h</i> for possible values.

Returns

- -1 is returned for a failure
- 0 is returned for a pass

Note

Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than $2^{*\ast} \text{Width}$, the patterns used in XIL_TESTMEM_WALKONES and XIL_TESTMEM_WALKZEROS will repeat on a boundary of a power of two making it more difficult to detect addressing errors. The XIL_TESTMEM_INCREMENT and XIL_TESTMEM_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

MicroBlaze Processor API

Overview

This section provides a linked summary and detailed descriptions of the MicroBlaze Processor APIs.

Modules

- MicroBlaze Pseudo-asm Macros and Interrupt handling APIs
- MicroBlaze exception APIs
- MicroBlaze Processor Cache APIs
- MicroBlaze Processor FSL Macros
- MicroBlaze PVR access routines and macros
- Sleep Routines for MicroBlaze

MicroBlaze Pseudo-asm Macros and Interrupt handling APIs

Overview

Standalone includes macros to provide convenient access to various registers in the MicroBlaze processor. Some of these macros are very useful within exception handlers for retrieving information about the exception. Also, the interrupt handling functions help manage interrupt handling on MicroBlaze processor devices. To use these functions, include the header file mb_interface.h in your source code

Functions

- void `microblaze_register_handler` (`XlInterruptHandler` Handler, `void *DataPtr`)
- void `microblaze_register_exception_handler` (`u32 ExceptionId`, `Xil_ExceptionHandler` Handler, `void *DataPtr`)

MicroBlaze pseudo-asm macros

The following is a summary of the MicroBlaze processor pseudo-asm macros.

- #define `mfgpr(rn)`
- #define `mfmsr()`
- #define `mfear()`
- #define `mfear()`
- #define `mfesr()`
- #define `mfssr()`

Macro Definition Documentation

#define `mfgpr(rn)`

Return value from the general purpose register (GPR) `rn`.

Parameters

<code>rn</code>	General purpose register to be read.
-----------------	--------------------------------------

#define `mfmsr()`

Return the current value of the MSR.

Parameters

<code>None</code>	
-------------------	--

#define `mfear()`

Return the current value of the Exception Address Register (EAR).

Parameters

<code>None</code>	
-------------------	--

#define `mfesr()`

Return the current value of the Exception Status Register (ESR).

Parameters

<code>None</code>	
-------------------	--

#define mffsr()

Return the current value of the Floating Point Status (FPS).

Parameters

None	
------	--

Function Documentation

void microblaze_register_handler (XlInterruptHandler Handler, void * DataPtr)

Registers a top-level interrupt handler for the MicroBlaze. The argument provided in this call as the DataPtr is used as the argument for the handler when it is called.

Parameters

<i>Handler</i>	Top level handler.
<i>DataPtr</i>	a reference to data that will be passed to the handler when it gets called.

Returns

None.

void microblaze_register_exception_handler (u32 ExceptionId, Xil_ExceptionHandler Handler, void * DataPtr)

Registers an exception handler for the MicroBlaze. The argument provided in this call as the DataPtr is used as the argument for the handler when it is called.

Parameters

<i>ExceptionId</i>	is the id of the exception to register this handler for.
<i>Top</i>	level handler.
<i>DataPtr</i>	is a reference to data that will be passed to the handler when it gets called.

Returns

None.

Note

None.

MicroBlaze exception APIs

Overview

The `xil_exception.h` file, available in the `<install-directory>/src/MicroBlaze` folder, contains MicroBlaze specific exception related APIs and macros. Application programs can use these APIs for various exception related operations. For example, enable exception, disable exception, register exception handler.

Note

To use exception related functions, `xil_exception.h` must be added in source code

Data Structures

- struct [MB_ExceptionVectorTableEntry](#)

Typedefs

- typedef void(* [Xil_ExceptionHandler](#)) (void *Data)
- typedef void(* [XlInterruptHandler](#)) (void *InstancePtr)

Functions

- void [Xil_ExceptionInit](#) (void)
- void [Xil_ExceptionEnable](#) (void)
- void [Xil_ExceptionDisable](#) (void)
- void [Xil_ExceptionRegisterHandler](#) (u32 Id, [Xil_ExceptionHandler](#) Handler, void *Data)
- void [Xil_ExceptionRemoveHandler](#) (u32 Id)

Data Structure Documentation

struct MB_ExceptionVectorTableEntry

Currently HAL is an augmented part of standalone BSP, so the old definition of [MB_ExceptionVectorTableEntry](#) is used here.

Typedef Documentation

typedef void(* Xil_ExceptionHandler) (void *Data)

This typedef is the exception handler function.

typedef void(* XlInterruptHandler) (void *InstancePtr)

This data type defines an interrupt handler for a device. The argument points to the instance of the component

Function Documentation

void Xil_ExceptionInit (void)

Initialize exception handling for the processor. The exception vector table is setup with the stub handler for all exceptions.

Parameters

None.	
-------	--

Returns

None.

void Xil_ExceptionEnable (void)

Enable Exceptions.

Returns

None.

void Xil_ExceptionDisable (void)

Disable Exceptions.

Parameters

None.	
-------	--

Returns

None.

void Xil_ExceptionRegisterHandler (u32 *Id*, Xil_ExceptionHandler *Handler*, void * *Data*)

Makes the connection between the Id of the exception source and the associated handler that is to run when the exception is recognized. The argument provided in this call as the DataPtr is used as the argument for the handler when it is called.

Parameters

<i>Id</i>	contains the 32 bit ID of the exception source and should be XIL_EXCEPTION_INT or be in the range of 0 to XIL_EXCEPTION_LAST. See xil_mach_exception.h for further information.
<i>Handler</i>	handler function to be registered for exception
<i>Data</i>	a reference to data that will be passed to the handler when it gets called.

void Xil_ExceptionRemoveHandler (u32 *Id*)

Removes the handler for a specific exception Id. The stub handler is then registered for this exception Id.

Parameters

<i>Id</i>	contains the 32 bit ID of the exception source and should be XIL_EXCEPTION_INT or in the range of 0 to XIL_EXCEPTION_LAST. See xexception_l.h for further information.
-----------	--

MicroBlaze Processor Cache APIs

Overview

Cache functions provide access to cache related operations such as flush and invalidate for instruction and data caches. It gives option to perform the cache operations on a single cacheline, a range of memory and an entire cache.

Note

Macros

- void [Xil_L1DCacheInvalidate\(\)](#)
- void [Xil_L2CacheInvalidate\(\)](#)
- void [Xil_L1DCacheInvalidateRange\(Addr, Len\)](#)
- void [Xil_L2CacheInvalidateRange\(Addr, Len\)](#)
- void [Xil_L1DCacheFlushRange\(Addr, Len\)](#)

- void [Xil_L2CacheFlushRange](#)(Addr, Len)
- void [Xil_L1DCacheFlush](#)()
- void [Xil_L2CacheFlush](#)()
- void [Xil_L1ICacheInvalidateRange](#)(Addr, Len)
- void [Xil_L1ICacheInvalidate](#)()
- void [Xil_L1DCacheEnable](#)()
- void [Xil_L1DCacheDisable](#)()
- void [Xil_L1ICacheEnable](#)()
- void [Xil_L1ICacheDisable](#)()
- void [Xil_DCacheEnable](#)()
- void [Xil_ICacheEnable](#)()

Functions

- void [Xil_DCacheDisable](#) (void)
- void [Xil_ICacheDisable](#) (void)

Macro Definition Documentation

void Xil_L1DCacheInvalidate()

Invalidate the entire L1 data cache. If the cacheline is modified (dirty), the modified contents are lost.

Parameters

None.	
-------	--

Returns

None.

Note

Processor must be in real mode.

void Xil_L2CacheInvalidate()

Invalidate the entire L2 data cache. If the cacheline is modified (dirty),the modified contents are lost.

Parameters

None.	
-------	--

Returns

None.

Note

Processor must be in real mode.

void Xil_L1DCacheInvalidateRange(*Addr*, *Len*)

Invalidate the L1 data cache for the given address range. If the bytes specified by the address (*Addr*) are cached by the L1 data cache, the cacheline containing that byte is invalidated.If the cacheline is modified (dirty), the modified contents are lost.

Parameters

<i>Addr</i>	is address of range to be invalidated.
<i>Len</i>	is the length in bytes to be invalidated.

Returns

None.

Note

Processor must be in real mode.

void Xil_L2CacheInvalidateRange(Addr, Len)

Invalidate the L1 data cache for the given address range. If the bytes specified by the address (Addr) are cached by the L1 data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost.

Parameters

<i>Addr</i>	address of range to be invalidated.
<i>Len</i>	length in bytes to be invalidated.

Returns

None.

Note

Processor must be in real mode.

void Xil_L1DCacheFlushRange(Addr, Len)

Flush the L1 data cache for the given address range. If the bytes specified by the address (Addr) are cached by the data cache, and is modified (dirty), the cacheline will be written to system memory. The cacheline will also be invalidated.

Parameters

<i>Addr</i>	the starting address of the range to be flushed.
<i>Len</i>	length in byte to be flushed.

Returns

None.

void Xil_L2CacheFlushRange(Addr, Len)

Flush the L2 data cache for the given address range. If the bytes specified by the address (Addr) are cached by the data cache, and is modified (dirty), the cacheline will be written to system memory. The cacheline will also be invalidated.

Parameters

<i>Addr</i>	the starting address of the range to be flushed.
<i>Len</i>	length in byte to be flushed.

Returns

None.

void Xil_L1DCacheFlush()

Flush the entire L1 data cache. If any cacheline is dirty, the cacheline will be written to system memory. The entire data cache will be invalidated.

Returns

None.

void Xil_L2CacheFlush()

Flush the entire L2 data cache. If any cacheline is dirty, the cacheline will be written to system memory. The entire data cache will be invalidated.

Returns

None.

void Xil_L1ICacheInvalidateRange(Addr, Len)

Invalidate the instruction cache for the given address range.

Parameters

<i>Addr</i>	is address of range to be invalidated.
<i>Len</i>	is the length in bytes to be invalidated.

Returns

None.

void Xil_L1ICacheInvalidate()

Invalidate the entire instruction cache.

Parameters

<i>None</i>	
-------------	--

Returns

None.

void Xil_L1DCacheEnable()

Enable the L1 data cache.

Returns

None.

void Xil_L1DCacheDisable()

Disable the L1 data cache.

Returns

None.

Note

This is processor specific.

void Xil_L1ICacheEnable()

Enable the instruction cache.

Returns

None.

Note

This is processor specific.

void Xil_L1ICacheDisable()

Disable the L1 Instruction cache.

Returns

None.

Note

This is processor specific.

void Xil_DCacheEnable()

Enable the data cache.

Parameters

<i>None</i>	
-------------	--

Returns

None.

void Xil_ICacheEnable()

Enable the instruction cache.

Parameters

<i>None</i>	
-------------	--

Returns

None.

Note

Function Documentation

void Xil_DCacheDisable (void)

Disable the data cache.

Parameters

<i>None</i>	
-------------	--

Returns

None.

void Xil_ICacheDisable(void)

Disable the instruction cache.

Parameters

None	
------	--

Returns

None.

MicroBlaze Processor FSL Macros

Overview

Standalone includes macros to provide convenient access to accelerators connected to the MicroBlaze Fast Simplex Link (FSL) Interfaces. To use these functions, include the header file fsl.h in your source code

Macros

- #define `getfslx`(val, id, flags)
- #define `putfslx`(val, id, flags)
- #define `tgetfslx`(val, id, flags)
- #define `tputfslx`(id, flags)
- #define `getdfslx`(val, var, flags)
- #define `putdfslx`(val, var, flags)
- #define `tgetdfslx`(val, var, flags)
- #define `tputdfslx`(var, flags)

Macro Definition Documentation

#define getfslx(val, id, flags)

Performs a get function on an input FSL of the MicroBlaze processor

Parameters

<i>val</i>	literal in the range of 0 to 7 (0 to 15 for MicroBlaze v7.00.a and later)
<i>id</i>	FSL identifier
<i>flags</i>	valid FSL macro flags

#define putfsIx(*val*, *id*, *flags*)

Performs a put function on an input FSL of the MicroBlaze processor

Parameters

<i>val</i>	literal in the range of 0 to 7 (0 to 15 for MicroBlaze v7.00.a and later)
<i>id</i>	FSL identifier
<i>flags</i>	valid FSL macro flags

#define tgetfsIx(*val*, *id*, *flags*)

Performs a test get function on an input FSL of the MicroBlaze processor

Parameters

<i>val</i>	literal in the range of 0 to 7 (0 to 15 for MicroBlaze v7.00.a and later)
<i>id</i>	FSL identifier
<i>flags</i>	valid FSL macro flags

#define tputfsIx(*id*, *flags*)

Performs a put function on an input FSL of the MicroBlaze processor

Parameters

<i>id</i>	FSL identifier
<i>flags</i>	valid FSL macro flags

#define getdfsIx(*val*, *var*, *flags*)

Performs a get function on an input FSL of the MicroBlaze processor

Parameters

<i>val</i>	literal in the range of 0 to 7 (0 to 15 for MicroBlaze v7.00.a and later)
<i>var</i>	FSL identifier
<i>flags</i>	valid FSL macro flags

#define putdfsIx(val, var, flags)

Performs a put function on an input FSL of the MicroBlaze processor

Parameters

val	literal in the range of 0 to 7 (0 to 15 for MicroBlaze v7.00.a and later)
var	FSL identifier
flags	valid FSL macro flags

#define tgetdfsIx(val, var, flags)

Performs a test get function on an input FSL of the MicroBlaze processor;

Parameters

val	literal in the range of 0 to 7 (0 to 15 for MicroBlaze v7.00.a and later)
var	FSL identifier
flags	valid FSL macro flags

#define tputdfsIx(var, flags)

Performs a put function on an input FSL of the MicroBlaze processor

Parameters

var	FSL identifier
flags	valid FSL macro flags

MicroBlaze PVR access routines and macros

Overview

MicroBlaze processor v5.00.a and later versions have configurable Processor Version Registers (PVRs). The contents of the PVR are captured using the pvr_t data structure, which is defined as an array of 32-bit words, with each word corresponding to a PVR register on hardware. The number of PVR words is determined by the number of PVRs configured in the hardware. You should not attempt to access PVR registers that are not present in hardware, as the pvr_t data structure is resized to hold only as many PVRs as are present in hardware. To access information in the PVR:

1. Use the [microblaze_get_pvr\(\)](#) function to populate the PVR data into a pvr_t data structure.
2. In subsequent steps, you can use any one of the PVR access macros list to get individual data stored in the PVR.
3. pvr.h header file must be included to source to use PVR macros.

Macros

- #define MICROBLAZE_PVR_IS_FULL(_pvr)
- #define MICROBLAZE_PVR_USE_BARREL(_pvr)
- #define MICROBLAZE_PVR_USE_DIV(_pvr)
- #define MICROBLAZE_PVR_USE_HW_MUL(_pvr)
- #define MICROBLAZE_PVR_USE_FPU(_pvr)
- #define MICROBLAZE_PVR_USE_ICACHE(_pvr)
- #define MICROBLAZE_PVR_USE_DCACHE(_pvr)
- #define MICROBLAZE_PVR_MICROBLAZE_VERSION(_pvr)
- #define MICROBLAZE_PVR_USER1(_pvr)
- #define MICROBLAZE_PVR_USER2(_pvr)
- #define MICROBLAZE_PVR_D_LMB(_pvr)
- #define MICROBLAZE_PVR_D_PLB(_pvr)
- #define MICROBLAZE_PVR_I_LMB(_pvr)
- #define MICROBLAZE_PVR_I_PLB(_pvr)
- #define MICROBLAZE_PVR_INTERRUPT_IS_EDGE(_pvr)
- #define MICROBLAZE_PVR_EDGE_IS_POSITIVE(_pvr)
- #define MICROBLAZE_PVR_INTERCONNECT(_pvr)
- #define MICROBLAZE_PVR_USE_MUL64(_pvr)
- #define MICROBLAZE_PVR_OPCODE_0x0_ILLEGAL(_pvr)
- #define MICROBLAZE_PVR_UNALIGNED_EXCEPTION(_pvr)
- #define MICROBLAZE_PVR_ILL_OPCODE_EXCEPTION(_pvr)
- #define MICROBLAZE_PVR_IPLB_BUS_EXCEPTION(_pvr)
- #define MICROBLAZE_PVR_DPLB_BUS_EXCEPTION(_pvr)
- #define MICROBLAZE_PVR_DIV_ZERO_EXCEPTION(_pvr)
- #define MICROBLAZE_PVR_FPU_EXCEPTION(_pvr)
- #define MICROBLAZE_PVR_FSL_EXCEPTION(_pvr)
- #define MICROBLAZE_PVR_DEBUG_ENABLED(_pvr)
- #define MICROBLAZE_PVR_NUMBER_OF_PC_BRK(_pvr)
- #define MICROBLAZE_PVR_NUMBER_OF_RD_ADDR_BRK(_pvr)
- #define MICROBLAZE_PVR_NUMBER_OF_WR_ADDR_BRK(_pvr)
- #define MICROBLAZE_PVR_FSL_LINKS(_pvr)
- #define MICROBLAZE_PVR_ICACHE_ADDR_TAG_BITS(_pvr)
- #define MICROBLAZE_PVR_ICACHE_ALLOW_WR(_pvr)
- #define MICROBLAZE_PVR_ICACHE_LINE_LEN(_pvr)
- #define MICROBLAZE_PVR_ICACHE_BYTE_SIZE(_pvr)
- #define MICROBLAZE_PVR_DCACHE_ADDR_TAG_BITS(_pvr)
- #define MICROBLAZE_PVR_DCACHE_ALLOW_WR(_pvr)
- #define MICROBLAZE_PVR_DCACHE_LINE_LEN(_pvr)
- #define MICROBLAZE_PVR_DCACHE_BYTE_SIZE(_pvr)
- #define MICROBLAZE_PVR_ICACHE_BASEADDR(_pvr)
- #define MICROBLAZE_PVR_ICACHE_HIGHADDR(_pvr)
- #define MICROBLAZE_PVR_DCACHE_BASEADDR(_pvr)

- #define MICROBLAZE_PVR_DCACHE_HIGHADDR(_pvr)
- #define MICROBLAZE_PVR_TARGET_FAMILY(_pvr)
- #define MICROBLAZE_PVR_MSR_RESET_VALUE(_pvr)
- #define MICROBLAZE_PVR_MMU_TYPE(_pvr)

Functions

- int `microblaze_get_pvr` (pvr_t *pvr)

Macro Definition Documentation

#define MICROBLAZE_PVR_IS_FULL(_pvr)

Return non-zero integer if PVR is of type FULL, 0 if basic

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_USE_BARREL(_pvr)

Return non-zero integer if hardware barrel shifter present.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_USE_DIV(_pvr)

Return non-zero integer if hardware divider present.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_USE_HW_MUL(_pvr)

Return non-zero integer if hardware multiplier present.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_USE_FPU(_pvr)

Return non-zero integer if hardware floating point unit (FPU) present.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_USE_ICACHE(_pvr)

Return non-zero integer if I-cache present.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_USE_DCACHE(_pvr)

Return non-zero integer if D-cache present.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_MICROBLAZE_VERSION(_pvr)

Return MicroBlaze processor version encoding. Refer to the MicroBlaze Processor Reference Guide (UG081) for mappings from encodings to actual hardware versions.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_USER1(_pvr)

Return the USER1 field stored in the PVR.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_USER2(*_pvr*)

Return the USER2 field stored in the PVR.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

#define MICROBLAZE_PVR_D_LMB(*_pvr*)

Return non-zero integer if Data Side PLB interface is present.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

#define MICROBLAZE_PVR_D_PLB(*_pvr*)

Return non-zero integer if Data Side PLB interface is present.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

#define MICROBLAZE_PVR_I_LMB(*_pvr*)

Return non-zero integer if Instruction Side Local Memory Bus (LMB) interface present.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

#define MICROBLAZE_PVR_I_PLB(*_pvr*)

Return non-zero integer if Instruction Side PLB interface present.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

#define MICROBLAZE_PVR_INTERRUPT_IS_EDGE(_pvr)

Return non-zero integer if interrupts are configured as edge-triggered.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_EDGE_IS_POSITIVE(_pvr)

Return non-zero integer if interrupts are configured as positive edge triggered.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_INTERCONNECT(_pvr)

Return non-zero if MicroBlaze processor has PLB interconnect; otherwise return zero.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_USE_MUL64(_pvr)

Return non-zero integer if MicroBlaze processor supports 64-bit products for multiplies.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_OPCODE_0x0_ILLEGAL(_pvr)

Return non-zero integer if opcode 0x0 is treated as an illegal opcode. multiplies.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_UNALIGNED_EXCEPTION(*_pvr*)

Return non-zero integer if unaligned exceptions are supported.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

#define MICROBLAZE_PVR_ILL_OPCODE_EXCEPTION(*_pvr*)

Return non-zero integer if illegal opcode exceptions are supported.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

#define MICROBLAZE_PVR_IPLB_BUS_EXCEPTION(*_pvr*)

Return non-zero integer if I-PLB exceptions are supported.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

#define MICROBLAZE_PVR_DPLB_BUS_EXCEPTION(*_pvr*)

Return non-zero integer if I-PLB exceptions are supported.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

#define MICROBLAZE_PVR_DIV_ZERO_EXCEPTION(*_pvr*)

Return non-zero integer if divide by zero exceptions are supported.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

#define MICROBLAZE_PVR_FPU_EXCEPTION(_pvr)

Return non-zero integer if FPU exceptions are supported.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_FSL_EXCEPTION(_pvr)

Return non-zero integer if FSL exceptions are present.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_DEBUG_ENABLED(_pvr)

Return non-zero integer if debug is enabled.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_NUMBER_OF_PC_BRK(_pvr)

Return the number of hardware PC breakpoints available.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_NUMBER_OF_RD_ADDR_BRK(_pvr)

Return the number of read address hardware watchpoints supported.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_NUMBER_OF_WR_ADDR_BRK(*_pvr*)

Return the number of write address hardware watchpoints supported.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

#define MICROBLAZE_PVR_FSL_LINKS(*_pvr*)

Return the number of FSL links present.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

#define MICROBLAZE_PVR_ICACHE_ADDR_TAG_BITS(*_pvr*)

Return the number of address tag bits for the I-cache.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

#define MICROBLAZE_PVR_ICACHE_ALLOW_WR(*_pvr*)

Return non-zero if writes to I-caches are allowed.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

#define MICROBLAZE_PVR_ICACHE_LINE_LEN(*_pvr*)

Return the length of each I-cache line in bytes.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

#define MICROBLAZE_PVR_ICACHE_BYTE_SIZE(_pvr)

Return the size of the D-cache in bytes.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_DCACHE_ADDR_TAG_BITS(_pvr)

Return the number of address tag bits for the D-cache.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_DCACHE_ALLOW_WR(_pvr)

Return non-zero if writes to D-cache are allowed.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_DCACHE_LINE_LEN(_pvr)

Return the length of each line in the D-cache in bytes.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_DCACHE_BYTE_SIZE(_pvr)

Return the size of the D-cache in bytes.

Parameters

_pvr	pvr data structure
------	--------------------

#define MICROBLAZE_PVR_ICACHE_BASEADDR(*_pvr*)

Return the base address of the I-cache.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

#define MICROBLAZE_PVR_ICACHE_HIGHADDR(*_pvr*)

Return the high address of the I-cache.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

#define MICROBLAZE_PVR_DCACHE_BASEADDR(*_pvr*)

Return the base address of the D-cache.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

#define MICROBLAZE_PVR_DCACHE_HIGHADDR(*_pvr*)

Return the high address of the D-cache.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

#define MICROBLAZE_PVR_TARGET_FAMILY(*_pvr*)

Return the encoded target family identifier.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

#define MICROBLAZE_PVR_MSR_RESET_VALUE(*_pvr*)

Refer to the MicroBlaze Processor Reference Guide (UG081) for mappings from encodings to target family name strings.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

#define MICROBLAZE_PVR_MMU_TYPE(*_pvr*)

Returns the value of C_USE_MMU. Refer to the MicroBlaze Processor Reference Guide (UG081) for mappings from MMU type values to MMU function.

Parameters

<i>_pvr</i>	pvr data structure
-------------	--------------------

Function Documentation

int microblaze_get_pvr(pvr_t * *pvr*)

Populate the PVR data structure to which *pvr* points with the values of the hardware PVR registers.

Parameters

<i>pvr</i>	address of PVR data structure to be populated
------------	---

Returns

0 - SUCCESS -1 - FAILURE

Sleep Routines for MicroBlaze

Overview

`microblaze_sleep.h` contains microblaze sleep APIs. These APIs provides delay for requested duration.

Note

`microblaze_sleep.h` may contain architecture-dependent items.

Functions

- void `MB_Sleep` (u32 MilliSeconds) __attribute__((__deprecated__))

Function Documentation

void MB_Sleep (u32 *MilliSeconds*)

Provides delay for requested duration..

Parameters

<i>MilliSeconds-</i>	Delay time in milliseconds.
----------------------	-----------------------------

Returns

None.

Note

Instruction cache should be enabled for this to work.

Cortex R5 Processor API

Overview

Standalone BSP contains boot code, cache, exception handling, file and memory management, configuration, time and processor-specific include functions. It supports gcc compiler. This section provides a linked summary and detailed descriptions of the Cortex R5 processor APIs.

Modules

- [Cortex R5 Processor Boot Code](#)
- [Cortex R5 Processor MPU specific APIs](#)
- [Cortex R5 Processor Cache Functions](#)
- [Cortex R5 Time Functions](#)
- [Cortex R5 Event Counters Functions](#)
- [Cortex R5 Processor Specific Include Files](#)

Cortex R5 Processor Boot Code

Overview

The boot .S file contains a minimal set of code for transferring control from the processor's reset location to the start of the application. The boot code performs minimum configuration which is required for an application to run starting from processor's reset state. Below is a sequence illustrating what all configuration is performed before control reaches to main function.

1. Program vector table base for exception handling
2. Program stack pointer for various modes (IRQ, FIQ, supervisor, undefined, abort, system)
3. Disable instruction cache, data cache and MPU
4. Invalidate instruction and data cache
5. Configure MPU with short descriptor translation table format and program base address of translation table

6. Enable data cache, instruction cache and MPU
7. Enable Floating point unit
8. Transfer control to _start which clears BSS sections and jumping to main application

Cortex R5 Processor MPU specific APIs

Overview

MPU functions provides access to MPU operations such as enable MPU, disable MPU and set attribute for section of memory. Boot code invokes Init_MPU function to configure the MPU. A total of 10 MPU regions are allocated with another 6 being free for users. Overview of the memory attributes for different MPU regions is as given below,

	Memory Range	Attributes of MPURegion	Note
DDR	0x00000000 - 0x7FFFFFFF	Normal write-back Cacheable	For a system where DDR is less than 2GB, region after DDR and before PL is marked as undefined in translation table
PL	0x80000000 - 0xBFFFFFFF	Strongly Ordered	
QSPI	0xC0000000 - 0xDFFFFFFF	Device Memory	
PCIe	0xE0000000 - 0xFFFFFFFF	Device Memory	
STM_CORESIGHT	0xF8000000 - 0xF8FFFFFF	Device Memory	

	Memory Range	Attributes of MPURegion	Note
RPU_R5_GIC	0xF9000000 - 0xF90FFFFF	Device Memory	
FPS	0xFD000000 - 0xFDFFFFFF	Device Memory	
LPS	0xFE000000 - 0xFFFFFFFF	Device Memory	0xFE000000 - 0xFFFFFFFF upper LPS slaves, 0xFF000000 - 0xFFFFFFFF lower LPS slaves
OCM	0xFFFFC0000 - 0xFFFFFFFF	Normal write-back Cacheable	

Functions

- void [Xil_SetTlbAttributes](#) (INTPTR Addr, u32 attrib)
- void [Xil_EnableMPU](#) (void)
- void [Xil_DisableMPU](#) (void)
- void [Xil_SetMPURegion](#) (INTPTR addr, u64 size, u32 attrib)

Function Documentation

void Xil_SetTlbAttributes (INTPTR *addr*, u32 *attrib*)

This function sets the memory attributes for a section covering 1MB, of memory in the translation table.

Parameters

<i>Addr</i>	32-bit address for which memory attributes need to be set.
<i>attrib</i>	Attribute for the given memory region.

Returns

None.

void Xil_EnableMPU (void)

Enable MPU for Cortex R5 processor. This function invalidates I cache and flush the D Caches, and then enables the MPU.

Parameters

None.	
-------	--

Returns

None.

void Xil_DisableMPU (void)

Disable MPU for Cortex R5 processors. This function invalidates I cache and flush the D Caches, and then disables the MPU.

Parameters

None.	
-------	--

Returns

None.

void Xil_SetMPURegion (INTPTR addr, u64 size, u32 attrib)

Set the memory attributes for a section of memory in the translation table.

Parameters

<i>Addr</i>	32-bit address for which memory attributes need to be set..
<i>size</i>	size is the size of the region.
<i>attrib</i>	Attribute for the given memory region.

Returns

None.

Cortex R5 Processor Cache Functions

Overview

Cache functions provide access to cache related operations such as flush and invalidate for instruction and data caches. It gives option to perform the cache operations on a single cacheline, a range of memory and an entire cache.

Functions

- void [Xil_DCacheEnable](#) (void)
- void [Xil_DCacheDisable](#) (void)
- void [Xil_DCacheInvalidate](#) (void)
- void [Xil_DCacheInvalidateRange](#) (INTPTR adr, u32 len)
- void [Xil_DCacheFlush](#) (void)
- void [Xil_DCacheFlushRange](#) (INTPTR adr, u32 len)
- void [Xil_DCacheInvalidateLine](#) (INTPTR adr)
- void [Xil_DCacheFlushLine](#) (INTPTR adr)
- void [Xil_DCacheStoreLine](#) (INTPTR adr)
- void [Xil_ICacheEnable](#) (void)
- void [Xil_ICacheDisable](#) (void)
- void [Xil_ICacheInvalidate](#) (void)
- void [Xil_ICacheInvalidateRange](#) (INTPTR adr, u32 len)
- void [Xil_ICacheInvalidateLine](#) (INTPTR adr)

Function Documentation

void Xil_DCacheEnable (void)

Enable the Data cache.

Parameters

None.	
-------	--

Returns

None.

Note

None.

void Xil_DCacheDisable (void)

Disable the Data cache.

Parameters

None.	
-------	--

Returns

None.

Note

None.

void Xil_DCachelnvalidate (void)

Invalidate the entire Data cache.

Parameters

None.	
-------	--

Returns

None.

void Xil_DCachelnvalidateRange (INTPTR adr, u32 len)

Invalidate the Data cache for the given address range. If the bytes specified by the address (adr) are cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are NOT written to system memory before the line is invalidated.

Parameters

adr	32bit start address of the range to be invalidated.
len	Length of range to be invalidated in bytes.

Returns

None.

void Xil_DCacheflush (void)

Flush the entire Data cache.

Parameters

None.	
-------	--

Returns

None.

void Xil_DCacheFlushRange (INTPTR adr, u32 len)

Flush the Data cache for the given address range. If the bytes specified by the address (adr) are cached by the Data cache, the cacheline containing those bytes is invalidated. If the cacheline is modified (dirty), the written to system memory before the lines are invalidated.

Parameters

<i>adr</i>	32bit start address of the range to be flushed.
<i>len</i>	Length of the range to be flushed in bytes

Returns

None.

void Xil_DCachelnvalidateLine (INTPTR adr)

Invalidate a Data cache line. If the byte specified by the address (adr) is cached by the data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are NOT written to system memory before the line is invalidated.

Parameters

<i>adr</i>	32bit address of the data to be flushed.
------------	--

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

void Xil_DCacheFlushLine (INTPTR adr)

Flush a Data cache line. If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

Parameters

<i>adr</i>	32bit address of the data to be flushed.
------------	--

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

void Xil_DCacheStoreLine (INTPTR adr)

Store a Data cache line. If the byte specified by the address (adr) is cached by the Data cache and the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory. After the store completes, the cacheline is marked as unmodified (not dirty).

Parameters

<i>adr</i>	32bit address of the data to be stored
------------	--

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

void Xil_ICacheEnable (void)

Enable the instruction cache.

Parameters

<i>None.</i>	
--------------	--

Returns

None.

void Xil_ICacheDisable (void)

Disable the instruction cache.

Parameters

<i>None.</i>	
--------------	--

Returns

None.

void Xil_ICacheInvalidate (void)

Invalidate the entire instruction cache.

Parameters

None.	
-------	--

Returns

None.

void Xil_ICacheInvalidateRange (INTPTR adr, u32 len)

Invalidate the instruction cache for the given address range. If the bytes specified by the address (adr) are cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are NOT written to system memory before the line is invalidated.

Parameters

adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

void Xil_ICacheInvalidateLine (INTPTR adr)

Invalidate an instruction cache line. If the instruction specified by the address is cached by the instruction cache, the cacheline containing that instruction is invalidated.

Parameters

adr	32bit address of the instruction to be invalidated.
-----	---

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

Cortex R5 Time Functions

Overview

The `xtime_l.c` file and corresponding `xtime_l.h` include file provide access to the 32-bit counter in TTC. The `sleep.c`, `usleep.c` file and the corresponding `sleep.h` include file implement sleep functions. Sleep functions are implemented as busy loops.

Functions

- void `XTime_StartTimer` (void)
- void `XTime_SetTime` (`XTime Xtime_Global`)
- void `XTime_GetTime` (`XTime *Xtime_Global`)

Function Documentation

`void XTime_StartTimer (void)`

Starts the TTC timer 3 counter 0 if present and if it is not already running with desired parameters for sleep functionalities.

Parameters

<code>None.</code>	
--------------------	--

Returns

None.

Note

When this function is called by any one processor in a multi- processor environment, reference time will reset/lost for all processors.

`void XTime_SetTime (XTime Xtime_Global)`

TTC Timer runs continuously and the time can not be set as desired. This API doesn't contain anything. It is defined to have uniformity across platforms.

Parameters

<code>Xtime_Global</code>	32 bit value to be written to the timer counter register.
---------------------------	---

Returns

None.

Note

In multiprocessor environment reference time will reset/lost for all processors, when this function called by any one processor.

void XTime_GetTime (XTime * Xtime_Global)

Get the time from the timer counter register.

Parameters

<i>Xtime_Global</i>	Pointer to the 32 bit location to be updated with the time current value of timer counter register.
---------------------	---

Returns

None.

Cortex R5 Event Counters Functions

Overview

Cortex R5 event counter functions can be utilized to configure and control the Cortex-R5 performance monitor events. Cortex-R5 Performance Monitor has 6 event counters which can be used to count a variety of events described in Coretx-R5 TRM. *xpm_counter.h* defines configurations *XPM_CNTRCFGx* which can be used to program the event counters to count a set of events.

Note

It doesn't handle the Cortex-R5 cycle counter, as the cycle counter is being used for time keeping.

Functions

- void [Xpm_SetEvents](#) (s32 PmcrCfg)
- void [Xpm_GetEventCounters](#) (u32 *PmCtrValue)

Function Documentation

void Xpm_SetEvents (s32 PmcrCfg)

This function configures the Cortex R5 event counters controller, with the event codes, in a configuration selected by the user and enables the counters.

Parameters

<i>PmcrCfg</i>	Configuration value based on which the event counters are configured. XPM_CNTRCFG* values defined in xpm_counter.h can be utilized for setting configuration
----------------	--

Returns

None.

void Xpm_GetEventCounters (u32 * PmCtrValue)

This function disables the event counters and returns the counter values.

Parameters

<i>PmCtrValue</i>	Pointer to an array of type u32 PmCtrValue[6]. It is an output parameter which is used to return the PM counter values.
-------------------	---

Returns

None.

Cortex R5 Processor Specific Include Files

Overview

The xpseudo_asm.h file includes xreg_cortexr5.h and xpseudo_asm_gcc.h.

The xreg_cortexr5.h include file contains the register numbers and the register bits for the ARM Cortex-R5 processor.

The xpseudo_asm_gcc.h file contains the definitions for the most often used inline assembler instructions, available as macros. These can be very useful for tasks such as setting or getting special purpose registers, synchronization, or cache manipulation. These inline assembler instructions can be used from drivers and user applications written in C.

ARM Processor Common API

Overview

This section provides a linked summary and detailed descriptions of the ARM Processor Common APIs.

Modules

- ARM Processor Exception Handling

ARM Processor Exception Handling

Overview

ARM processors specific exception related APIs for cortex A53,A9 and R5 can utilized for enabling/disabling IRQ, registering/removing handler for exceptions or initializing exception vector table with null handler.

Macros

- #define `Xil_ExceptionEnableMask`(Mask)
- #define `Xil_ExceptionEnable`()
- #define `Xil_ExceptionDisableMask`(Mask)
- #define `Xil_ExceptionDisable`()
- #define `Xil_EnableNestedInterrupts`()
- #define `Xil_DisableNestedInterrupts`()

Typedefs

- typedef void(* `Xil_ExceptionHandler`) (void *data)

Functions

- void `Xil_ExceptionRegisterHandler` (u32 Exception_id, `Xil_ExceptionHandler` Handler, void *Data)

- void [Xil_ExceptionRemoveHandler](#) (u32 Exception_id)
- void [Xil_ExceptionInit](#) (void)
- void [Xil_DataAbortHandler](#) (void *CallBackRef)
- void [Xil_PrefetchAbortHandler](#) (void *CallBackRef)
- void [Xil_UndefinedExceptionHandler](#) (void *CallBackRef)

Macro Definition Documentation

#define Xil_ExceptionEnableMask(Mask)

Enable Exceptions.

Parameters

<i>Mask</i>	for exceptions to be enabled.
-------------	-------------------------------

Returns

None.

Note

If bit is 0, exception is enabled. C-Style signature: void [Xil_ExceptionEnableMask\(Mask\)](#)

#define Xil_ExceptionEnable()

Enable the IRQ exception.

Returns

None.

Note

None.

#define Xil_ExceptionDisableMask(Mask)

Disable Exceptions.

Parameters

<i>Mask</i>	for exceptions to be enabled.
-------------	-------------------------------

Returns

None.

Note

If bit is 1, exception is disabled. C-Style signature: [Xil_ExceptionDisableMask\(Mask\)](#)

#define Xil_ExceptionDisable()

Disable the IRQ exception.

Returns

None.

Note

None.

#define Xil_EnableNestedInterrupts()

Enable nested interrupts by clearing the I and F bits in CPSR. This API is defined for cortex-a9 and cortex-r5.

Returns

None.

Note

This macro is supposed to be used from interrupt handlers. In the interrupt handler the interrupts are disabled by default (I and F are 1). To allow nesting of interrupts, this macro should be used. It clears the I and F bits by changing the ARM mode to system mode. Once these bits are cleared and provided the preemption of interrupt conditions are met in the GIC, nesting of interrupts will start happening. Caution: This macro must be used with caution. Before calling this macro, the user must ensure that the source of the current IRQ is appropriately cleared. Otherwise, as soon as we clear the I and F bits, there can be an infinite loop of interrupts with an eventual crash (all the stack space getting consumed).

#define Xil_DisableNestedInterrupts()

Disable the nested interrupts by setting the I and F bits. This API is defined for cortex-a9 and cortex-r5.

Returns

None.

Note

This macro is meant to be called in the interrupt service routines. This macro cannot be used independently. It can only be used when nesting of interrupts have been enabled by using the macro [Xil_EnableNestedInterrupts\(\)](#). In a typical flow, the user first calls the `Xil_EnableNestedInterrupts` in the ISR at the appropriate point. The user then must call this macro before exiting the interrupt service routine. This macro puts the ARM back in IRQ/FIQ mode and hence sets back the I and F bits.

TypeDef Documentation

typedef void(* Xil_ExceptionHandler) (void *data)

This typedef is the exception handler function.

Function Documentation

void Xil_ExceptionRegisterHandler (u32 *Exception_id*, Xil_ExceptionHandler *Handler*, void * *Data*)

Register a handler for a specific exception. This handler is being called when the processor encounters the specified exception.

Parameters

<i>exception_id</i>	contains the ID of the exception source and should be in the range of 0 to XIL_EXCEPTION_ID_LAST. See xil_exception.h for further information.
<i>Handler</i>	to the Handler for that exception.
<i>Data</i>	is a reference to Data that will be passed to the Handler when it gets called.

Returns

None.

Note

None.

void Xil_ExceptionRemoveHandler (u32 *Exception_id*)

Removes the Handler for a specific exception Id. The stub Handler is then registered for this exception Id.

Parameters

<i>exception_id</i>	contains the ID of the exception source and should be in the range of 0 to XIL_EXCEPTION_ID_LAST. See xil_exception.h for further information.
---------------------	--

Returns

None.

Note

None.

void Xil_ExceptionInit (void)

The function is a common API used to initialize exception handlers across all supported arm processors. For ARM Cortex-A53, Cortex-R5, and Cortex-A9, the exception handlers are being initialized statically and this function does not do anything. However, it is still present to take care of backward compatibility issues (in earlier versions of BSPs, this API was being used to initialize exception handlers).

Parameters

None.	
-------	--

Returns

None.

Note

None.

void Xil_DataAbortHandler (void * *CallBackRef*)

Default Data abort handler which prints data fault status register through which information about data fault can be acquired

Parameters

None	
------	--

Returns

None.

Note

None.

void Xil_PrefetchAbortHandler (void * *CallBackRef*)

Default Prefetch abort handler which prints prefetch fault status register through which information about instruction prefetch fault can be acquired

Parameters

None	
------	--

Returns

None.

Note

None.

void Xil_UndefinedExceptionHandler (void * *CallBackRef*)

Default undefined exception handler which prints address of the undefined instruction if debug prints are enabled

Parameters

None	
------	--

Returns

None.

Note

None.

Cortex A9 Processor API

Overview

Standalone BSP contains boot code, cache, exception handling, file and memory management, configuration, time and processor-specific include functions. It supports gcc compilers.

Modules

- Cortex A9 Processor Boot Code
- Cortex A9 Processor Cache Functions
- Cortex A9 Processor MMU Functions
- Cortex A9 Time Functions
- Cortex A9 Event Counter Function
- PL310 L2 Event Counters Functions
- Cortex A9 Processor and pl310 Errata Support
- Cortex A9 Processor Specific Include Files

Cortex A9 Processor Boot Code

Overview

The boot .S file contains a minimal set of code for transferring control from the processor reset location to the start of the application. The boot code performs minimum configuration which is required for an application to run starting from processor's reset state. Below is a sequence illustrating what all configuration is performed before control reaches to main function.

1. Program vector table base for exception handling
2. Invalidate instruction cache, data cache and TLBs
3. Program stack pointer for various modes (IRQ, FIQ, supervisor, undefine, abort, system)
4. Configure MMU with short descriptor translation table format and program base address of translation table
5. Enable data cache, instruction cache and MMU

6. Enable Floating point unit
7. Transfer control to `_start` which clears BSS sections, initializes global timer and runs global constructor before jumping to main application

The `translation_table.S` file contains a static page table required by MMU for cortex-A9. This translation table is flat mapped (input address = output address) with default memory attributes defined for zynq architecture. It utilizes short descriptor translation table format with each section defining 1MB of memory. The overview of translation table memory attributes is described below.

	Memory Range	Definition in Translation Table	Note
DDR	0x00000000 - 0x3FFFFFFF	Normal write-back Cacheable	For a system where DDR is less than 1GB, region after DDR and before PL is marked as undefined/reserved in translation table
PL	0x40000000 - 0xBFFFFFFF	Strongly Ordered	
Reserved	0xC0000000 - 0xDFFFFFFF	Unassigned	
Memory mapped devices	0xE0000000 - 0xE02FFFFF	Device Memory	
Reserved	0xE0300000 - 0xE0FFFFFF	Unassigned	
NAND, NOR	0xE1000000 - 0xE3FFFFFF	Device memory	
SRAM	0xE4000000 - 0xE5FFFFFF	Normal write-back Cacheable	
Reserved	0xE6000000 - 0xF7FFFFFF	Unassigned	
AMBA APB Peripherals	0xF8000000 - 0xF8FFFFFF	Device Memory	0xF8000C00 - 0xF8000FFF, 0xF8010000 -0xF88FFFFFF and 0xF8F03000 to 0xF8FFFFFF are reserved but due to granular size of 1MB, it is not possible to define separate regions for them

	Memory Range	Definition in Translation Table	Note
Reserved	0xF9000000 - 0xFBFFFFFF	Unassigned	
Linear QSPI - XIP	0xFC000000 - 0xFDFFFFFF	Normal write-through cacheable	
Reserved	0xFE000000 - 0xFFFFFFF	Unassigned	
OCM	0xFFF00000 - 0xFFFFFFFF	Normal inner write-back cacheable	0xFFF00000 to 0xFFFFB0000 is reserved but due to 1MB granual size, it is not possible to define separate region for it

Cortex A9 Processor Cache Functions

Overview

Cache functions provide access to cache related operations such as flush and invalidate for instruction and data caches. It gives option to perform the cache operations on a single cacheline, a range of memory and an entire cache.

Functions

- void `Xil_DCacheEnable` (void)
- void `Xil_DCacheDisable` (void)
- void `Xil_DCachelnvalidate` (void)
- void `Xil_DCachelnvalidateRange` (INTPTR adr, u32 len)
- void `Xil_DCacheFlush` (void)
- void `Xil_DCacheFlushRange` (INTPTR adr, u32 len)
- void `Xil_ICacheEnable` (void)
- void `Xil_ICacheDisable` (void)
- void `Xil_ICachelnvalidate` (void)
- void `Xil_ICachelnvalidateRange` (INTPTR adr, u32 len)
- void `Xil_DCachelnvalidateLine` (u32 adr)
- void `Xil_DCacheFlushLine` (u32 adr)
- void `Xil_DCacheStoreLine` (u32 adr)
- void `Xil_ICachelnvalidateLine` (u32 adr)
- void `Xil_L1DCacheEnable` (void)
- void `Xil_L1DCacheDisable` (void)

- void [Xil_L1DCacheInvalidate](#) (void)
- void [Xil_L1DCacheInvalidateLine](#) (u32 adr)
- void [Xil_L1DCacheInvalidateRange](#) (u32 adr, u32 len)
- void [Xil_L1DCacheFlush](#) (void)
- void [Xil_L1DCacheFlushLine](#) (u32 adr)
- void [Xil_L1DCacheFlushRange](#) (u32 adr, u32 len)
- void [Xil_L1DCacheStoreLine](#) (u32 adr)
- void [Xil_L1ICacheEnable](#) (void)
- void [Xil_L1ICacheDisable](#) (void)
- void [Xil_L1ICacheInvalidate](#) (void)
- void [Xil_L1ICacheInvalidateLine](#) (u32 adr)
- void [Xil_L1ICacheInvalidateRange](#) (u32 adr, u32 len)
- void [Xil_L2CacheEnable](#) (void)
- void [Xil_L2CacheDisable](#) (void)
- void [Xil_L2CacheInvalidate](#) (void)
- void [Xil_L2CacheInvalidateLine](#) (u32 adr)
- void [Xil_L2CacheInvalidateRange](#) (u32 adr, u32 len)
- void [Xil_L2CacheFlush](#) (void)
- void [Xil_L2CacheFlushLine](#) (u32 adr)
- void [Xil_L2CacheFlushRange](#) (u32 adr, u32 len)
- void [Xil_L2CacheStoreLine](#) (u32 adr)

Function Documentation

void Xil_DCacheEnable (void)

Enable the Data cache.

Parameters

None.	
-------	--

Returns

None.

Note

None.

void Xil_DCACHEDisable(void)

Disable the Data cache.

Parameters

None.	
-------	--

Returns

None.

Note

None.

void Xil_DCACHEInvalidate(void)

Invalidate the entire Data cache.

Parameters

None.	
-------	--

Returns

None.

Note

None.

void Xil_DCACHEInvalidateRange(INTPTR adr, u32 len)

Invalidate the Data cache for the given address range. If the bytes specified by the address range are cached by the Data cache, the cachelines containing those bytes are invalidated. If the cachelines are modified (dirty), the modified contents are lost and NOT written to the system memory before the lines are invalidated.

In this function, if start address or end address is not aligned to cache-line, particular cache-line containing unaligned start or end address is flush first and then invalidated the others as invalidating the same unaligned cache line may result into loss of data. This issue raises few possibilities.

If the address to be invalidated is not cache-line aligned, the following choices are available:

1. Invalidate the cache line when required and do not bother much for the side effects. Though it sounds good, it can result in hard-to-debug issues. The problem is, if some other variable are allocated in the same cache line and had been recently updated (in cache), the invalidation would result in loss of data.
2. Flush the cache line first. This will ensure that if any other variable present in the same cache line and updated recently are flushed out to memory. Then it can safely be invalidated. Again it sounds good, but this can result in issues. For example, when the invalidation happens in a typical ISR (after a DMA transfer has updated the memory), then flushing the cache line means, loosing data that were updated recently before the ISR got invoked.

Linux prefers the second one. To have uniform implementation (across standalone and Linux), the second option is implemented. This being the case, following needs to be taken care of:

1. Whenever possible, the addresses must be cache line aligned. Please note that, not just start address, even the end address must be cache line aligned. If that is taken care of, this will always work.
2. Avoid situations where invalidation has to be done after the data is updated by peripheral/DMA directly into the memory. It is not tough to achieve (may be a bit risky). The common use case to do invalidation is when a DMA happens. Generally for such use cases, buffers can be allocated first and then start the DMA. The practice that needs to be followed here is, immediately after buffer allocation and before starting the DMA, do the invalidation. With this approach, invalidation need not to be done after the DMA transfer is over.

This is going to always work if done carefully. However, the concern is, there is no guarantee that invalidate has not needed to be done after DMA is complete. For example, because of some reasons if the first cache line or last cache line (assuming the buffer in question comprises of multiple cache lines) are brought into cache (between the time it is invalidated and DMA completes) because of some speculative prefetching or reading data for a variable present in the same cache line, then we will have to invalidate the cache after DMA is complete.

Parameters

<i>adr</i>	32bit start address of the range to be invalidated.
<i>len</i>	Length of the range to be invalidated in bytes.

Returns

None.

Note

None.

void Xil_DCacheFlush(void)

Flush the entire Data cache.

Parameters

<i>None.</i>	
--------------	--

Returns

None.

Note

None.

void Xil_DCacheFlushRange (INTPTR adr, u32 len)

Flush the Data cache for the given address range. If the bytes specified by the address range are cached by the data cache, the cachelines containing those bytes are invalidated. If the cachelines are modified (dirty), they are written to the system memory before the lines are invalidated.

Parameters

<i>adr</i>	32bit start address of the range to be flushed.
<i>len</i>	Length of the range to be flushed in bytes.

Returns

None.

Note

None.

void Xil_ICacheEnable (void)

Enable the instruction cache.

Parameters

<i>None.</i>	
--------------	--

Returns

None.

Note

None.

void Xil_ICacheDisable (void)

Disable the instruction cache.

Parameters

<i>None.</i>	
--------------	--

Returns

None.

Note

None.

void Xil_ICacheInvalidate (void)

Invalidate the entire instruction cache.

Parameters

None.	
-------	--

Returns

None.

Note

None.

void Xil_ICacheInvalidateRange (INTPTR adr, u32 len)

Invalidate the instruction cache for the given address range. If the instructions specified by the address range are cached by the instruction cache, the cachelines containing those instructions are invalidated.

Parameters

<i>adr</i>	32bit start address of the range to be invalidated.
<i>len</i>	Length of the range to be invalidated in bytes.

Returns

None.

Note

None.

void Xil_DCachelineInvalidateLine (u32 adr)

Invalidate a Data cache line. If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are NOT written to the system memory before the line is invalidated.

Parameters

<i>adr</i>	32bit address of the data to be flushed.
------------	--

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

void Xil_DCacheFlushLine (u32 adr)

Flush a Data cache line. If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

Parameters

<i>adr</i>	32bit address of the data to be flushed.
------------	--

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

void Xil_DCacheStoreLine (u32 adr)

Store a Data cache line. If the byte specified by the address (adr) is cached by the Data cache and the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory. After the store completes, the cacheline is marked as unmodified (not dirty).

Parameters

<i>adr</i>	32bit address of the data to be stored.
------------	---

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

void Xil_ICacheInvalidateLine (u32 adr)

Invalidate an instruction cache line. If the instruction specified by the address is cached by the instruction cache, the cacheline containing that instruction is invalidated.

Parameters

<i>adr</i>	32bit address of the instruction to be invalidated.
------------	---

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

void Xil_L1DCacheEnable (void)

Enable the level 1 Data cache.

Parameters

<i>None.</i>	
--------------	--

Returns

None.

Note

None.

void Xil_L1DCacheDisable (void)

Disable the level 1 Data cache.

Parameters

<i>None.</i>	
--------------	--

Returns

None.

Note

None.

void Xil_L1DCacheInvalidate (void)

Invalidate the level 1 Data cache.

Parameters

None.	
-------	--

Returns

None.

Note

In Cortex A9, there is no cp instruction for invalidating the whole D-cache. This function invalidates each line by set/way.

void Xil_L1DCacheInvalidateLine (u32 adr)

Invalidate a level 1 Data cache line. If the byte specified by the address (Addr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are NOT written to system memory before the line is invalidated.

Parameters

adr	32bit address of the data to be invalidated.
-----	--

Returns

None.

Note

The bottom 5 bits are set to 0, forced by architecture.

void Xil_L1DCacheInvalidateRange (u32 adr, u32 len)

Invalidate the level 1 Data cache for the given address range. If the bytes specified by the address range are cached by the Data cache, the cachelines containing those bytes are invalidated. If the cachelines are modified (dirty), the modified contents are lost and NOT written to the system memory before the lines are invalidated.

Parameters

adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Note

None.

void Xil_L1DCacheFlush (void)

Flush the level 1 Data cache.

Parameters

None.	
-------	--

Returns

None.

Note

In Cortex A9, there is no cp instruction for flushing the whole D-cache. Need to flush each line.

void Xil_L1DCacheFlushLine (u32 adr)

Flush a level 1 Data cache line. If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

Parameters

adr	32bit address of the data to be flushed.
-----	--

Returns

None.

Note

The bottom 5 bits are set to 0, forced by architecture.

void Xil_L1DCacheFlushRange (u32 adr, u32 len)

Flush the level 1 Data cache for the given address range. If the bytes specified by the address range are cached by the Data cache, the cacheline containing those bytes are invalidated. If the cachelines are modified (dirty), they are written to system memory before the lines are invalidated.

Parameters

<i>adr</i>	32bit start address of the range to be flushed.
<i>len</i>	Length of the range to be flushed in bytes.

Returns

None.

Note

None.

void Xil_L1DCacheStoreLine (u32 adr)

Store a level 1 Data cache line. If the byte specified by the address (adr) is cached by the Data cache and the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory. After the store completes, the cacheline is marked as unmodified (not dirty).

Parameters

<i>Address</i>	to be stored.
----------------	---------------

Returns

None.

Note

The bottom 5 bits are set to 0, forced by architecture.

void Xil_L1ICacheEnable (void)

Enable the level 1 instruction cache.

Parameters

<i>None.</i>	
--------------	--

Returns

None.

Note

None.

void Xil_L1ICacheDisable (void)

Disable level 1 the instruction cache.

Parameters

None.	
-------	--

Returns

None.

Note

None.

void Xil_L1ICacheInvalidate (void)

Invalidate the entire level 1 instruction cache.

Parameters

None.	
-------	--

Returns

None.

Note

None.

void Xil_L1ICacheInvalidateLine (u32 adr)

Invalidate a level 1 instruction cache line. If the instruction specified by the address is cached by the instruction cache, the cacheline containing that instruction is invalidated.

Parameters

adr	32bit address of the instruction to be invalidated.
-----	---

Returns

None.

Note

The bottom 5 bits are set to 0, forced by architecture.

void Xil_L1ICacheInvalidateRange (u32 adr, u32 len)

Invalidate the level 1 instruction cache for the given address range. If the instructions specified by the address range are cached by the instruction cache, the cacheline containing those bytes are invalidated.

Parameters

<i>adr</i>	32bit start address of the range to be invalidated.
<i>len</i>	Length of the range to be invalidated in bytes.

Returns

None.

Note

None.

void Xil_L2CacheEnable (void)

Enable the L2 cache.

Parameters

<i>None.</i>	
--------------	--

Returns

None.

Note

None.

void Xil_L2CacheDisable (void)

Disable the L2 cache.

Parameters

<i>None.</i>	
--------------	--

Returns

None.

Note

None.

void Xil_L2CacheInvalidate (void)

Invalidate the entire level 2 cache.

Parameters

None.	
-------	--

Returns

None.

Note

None.

void Xil_L2CacheInvalidateLine (u32 adr)

Invalidate a level 2 cache line. If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the modified contents are lost and are NOT written to system memory before the line is invalidated.

Parameters

adr	32bit address of the data/instruction to be invalidated.
-----	--

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

void Xil_L2CacheInvalidateRange (u32 adr, u32 len)

Invalidate the level 2 cache for the given address range. If the bytes specified by the address range are cached by the L2 cache, the cacheline containing those bytes are invalidated. If the cachelines are modified (dirty), the modified contents are lost and are NOT written to system memory before the lines are invalidated.

Parameters

<i>adr</i>	32bit start address of the range to be invalidated.
<i>len</i>	Length of the range to be invalidated in bytes.

Returns

None.

Note

None.

void Xil_L2CacheFlush (void)

Flush the entire level 2 cache.

Parameters

<i>None.</i>	
--------------	--

Returns

None.

Note

None.

void Xil_L2CacheFlushLine (u32 adr)

Flush a level 2 cache line. If the byte specified by the address (adr) is cached by the L2 cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

Parameters

<i>adr</i>	32bit address of the data/instruction to be flushed.
------------	--

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

void Xil_L2CacheFlushRange (u32 adr, u32 len)

Flush the level 2 cache for the given address range. If the bytes specified by the address range are cached by the L2 cache, the cacheline containing those bytes are invalidated. If the cachelines are modified (dirty), they are written to the system memory before the lines are invalidated.

Parameters

<i>adr</i>	32bit start address of the range to be flushed.
<i>len</i>	Length of the range to be flushed in bytes.

Returns

None.

Note

None.

void Xil_L2CacheStoreLine (u32 adr)

Store a level 2 cache line. If the byte specified by the address (adr) is cached by the L2 cache and the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory. After the store completes, the cacheline is marked as unmodified (not dirty).

Parameters

<i>adr</i>	32bit address of the data/instruction to be stored.
------------	---

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

Cortex A9 Processor MMU Functions

Overview

MMU functions equip users to enable MMU, disable MMU and modify default memory attributes of MMU table as per the need.

Functions

- void [Xil_SetTlbAttributes](#) (INTPTR Addr, u32 attrib)
- void [Xil_EnableMMU](#) (void)
- void [Xil_DisableMMU](#) (void)

Function Documentation

void Xil_SetTlbAttributes (INTPTR Addr, u32 attrib)

This function sets the memory attributes for a section covering 1MB of memory in the translation table.

Parameters

<i>Addr</i>	32-bit address for which memory attributes need to be set.
<i>attrib</i>	Attribute for the given memory region. <i>xil_mmu.h</i> contains definitions of commonly used memory attributes which can be utilized for this function.

Returns

None.

Note

The MMU or D-cache does not need to be disabled before changing a translation table entry.

void Xil_EnableMMU (void)

Enable MMU for cortex A9 processor. This function invalidates the instruction and data caches, and then enables MMU.

Parameters

<i>None.</i>	
--------------	--

Returns

None.

void Xil_DisableMMU (void)

Disable MMU for Cortex A9 processors. This function invalidates the TLBs, Branch Predictor Array and flushed the D Caches before disabling the MMU.

Parameters

None.	
-------	--

Returns

None.

Note

When the MMU is disabled, all the memory accesses are treated as strongly ordered.

Cortex A9 Time Functions

Overview

xtime_l.h provides access to the 64-bit Global Counter in the PMU. This counter increases by one at every two processor cycles. These functions can be used to get/set time in the global timer.

Functions

- void [XTime_SetTime](#) (XTime Xtime_Global)
- void [XTime_GetTime](#) (XTIME *Xtime_Global)

Function Documentation

void XTime_SetTime (XTime Xtime_Global)

Set the time in the Global Timer Counter Register.

Parameters

Xtime_Global	64-bit Value to be written to the Global Timer Counter Register.
--------------	--

Returns

None.

Note

When this function is called by any one processor in a multi- processor environment, reference time will reset/lost for all processors.

void XTime_GetTime (XTime * Xtime_Global)

Get the time from the Global Timer Counter Register.

Parameters

<i>Xtime_Global</i>	Pointer to the 64-bit location which will be updated with the current timer value.
---------------------	--

Returns

None.

Note

None.

Cortex A9 Event Counter Function

Overview

Cortex A9 event counter functions can be utilized to configure and control the Cortex-A9 performance monitor events.

Cortex-A9 performance monitor has six event counters which can be used to count a variety of events described in Coretx-A9 TRM. xpm_counter.h defines configurations XPM_CNTRCFGx which can be used to program the event counters to count a set of events.

Note

It doesn't handle the Cortex-A9 cycle counter, as the cycle counter is being used for time keeping.

Functions

- void [Xpm_SetEvents](#) (s32 PmcrCfg)
- void [Xpm_GetEventCounters](#) (u32 *PmCtrValue)

Function Documentation

void Xpm_SetEvents (s32 PmcrCfg)

This function configures the Cortex A9 event counters controller, with the event codes, in a configuration selected by the user and enables the counters.

Parameters

<i>PmcrCfg</i>	Configuration value based on which the event counters are configured. XPM_CNTRCFG* values defined in xpm_counter.h can be utilized for setting configuration.
----------------	---

Returns

None.

Note

None.

void Xpm_GetEventCounters (u32 * *PmCtrValue*)

This function disables the event counters and returns the counter values.

Parameters

<i>PmCtrValue</i>	Pointer to an array of type u32 PmCtrValue[6]. It is an output parameter which is used to return the PM counter values.
-------------------	---

Returns

None.

Note

None.

PL310 L2 Event Counters Functions

Overview

xl2cc_counter.h contains APIs for configuring and controlling the event counters in PL310 L2 cache controller. PL310 has two event counters which can be used to count variety of events like DRHIT, DRREQ, DWHIT, DWREQ, etc. xl2cc_counter.h contains definitions for different configurations which can be used for the event counters to count a set of events.

Functions

- void [XL2cc_EventCtrlInit](#) (s32 Event0, s32 Event1)
- void [XL2cc_EventCtrlStart](#) (void)
- void [XL2cc_EventCtrlStop](#) (u32 *EveCtr0, u32 *EveCtr1)

Function Documentation

void XL2cc_EventCtrlInit (s32 Event0, s32 Event1)

This function initializes the event counters in L2 Cache controller with a set of event codes specified by the user.

Parameters

<i>Event0</i>	Event code for counter 0.
<i>Event1</i>	Event code for counter 1.

Returns

None.

Note

The definitions for event codes XL2CC_* can be found in xl2cc_counter.h.

void XL2cc_EventCtrlStart (void)

This function starts the event counters in L2 Cache controller.

Parameters

<i>None.</i>	
--------------	--

Returns

None.

Note

None.

void XL2cc_EventCtrlStop (u32 * EveCtr0, u32 * EveCtr1)

This function disables the event counters in L2 Cache controller, saves the counter values and resets the counters.

Parameters

<i>EveCtr0</i>	Output parameter which is used to return the value in event counter 0. EveCtr1:
	Output parameter which is used to return the value in event counter 1.

Returns

None.

Note

None.

Cortex A9 Processor and pl310 Errata Support

Overview

Various ARM errata are handled in the standalone BSP. The implementation for errata handling follows ARM guidelines and is based on the open source Linux support for these errata.

Note

The errata handling is enabled by default. To disable handling of all the errata globally, un-define the macro ENABLE_ARM_ERRATA in xil_errata.h. To disable errata on a per-erratum basis, un-define relevant macros in xil_errata.h.

errata_definitions

The errata conditions handled in the standalone BSP are listed below

- #define **ENABLE_ARM_ERRATA**
- #define **CONFIG_ARM_ERRATA_742230**
- #define **CONFIG_ARM_ERRATA_743622**
- #define **CONFIG_ARM_ERRATA_775420**
- #define **CONFIG_ARM_ERRATA_794073**
- #define **CONFIG_PL310_ERRATA_588369**
- #define **CONFIG_PL310_ERRATA_727915**
- #define **CONFIG_PL310_ERRATA_753970**

Macro Definition Documentation

#define CONFIG_ARM_ERRATA_742230

Errata No: 742230 Description: DMB operation may be faulty

#define CONFIG_ARM_ERRATA_743622

Errata No: 743622 Description: Faulty hazard checking in the Store Buffer may lead to data corruption.

#define CONFIG_ARM_ERRATA_775420

Errata No: 775420 Description: A data cache maintenance operation which aborts, might lead to deadlock

#define CONFIG_ARM_ERRATA_794073

Errata No: 794073 Description: Speculative instruction fetches with MMU disabled might not comply with architectural requirements

#define CONFIG_PL310_ERRATA_588369

PL310 L2 Cache Errata Errata No: 588369 Description: Clean & Invalidate maintenance operations do not invalidate clean lines

#define CONFIG_PL310_ERRATA_727915

Errata No: 727915 Description: Background Clean and Invalidate by Way operation can cause data corruption

#define CONFIG_PL310_ERRATA_753970

Errata No: 753970 Description: Cache sync operation may be faulty

Cortex A9 Processor Specific Include Files

The xpseudo_asm.h includes xreg_cortexa9.h and xpseudo_asm_gcc.h.

The xreg_cortexa9.h file contains definitions for inline assembler code. It provides inline definitions for Cortex A9 GPRs, SPRs, MPE registers, co-processor registers and Debug registers.

The xpseudo_asm_gcc.h contains the definitions for the most often used inline assembler instructions, available as macros. These can be very useful for tasks such as setting or getting special purpose registers, synchronization, or cache manipulation etc. These inline assembler instructions can be used from drivers and user applications written in C.

Cortex A53 32-bit Processor API

Overview

Cortex-A53 standalone BSP contains two separate BSPs for 32-bit mode and 64-bit mode. The 32-bit mode of cortex-A53 is compatible with ARMv7-A architecture.

Modules

- Cortex A53 32-bit Processor Boot Code
- Cortex A53 32-bit Processor Cache Functions
- Cortex A53 32-bit Processor MMU Handling
- Cortex A53 32-bit Mode Time Functions
- Cortex A53 32-bit Processor Specific Include Files

Cortex A53 32-bit Processor Boot Code

Overview

The boot .S file contains a minimal set of code for transferring control from the processor reset location to the start of the application. The boot code performs minimum configuration which is required for an application to run starting from processor's reset state. Below is a sequence illustrating what all configuration is performed before control reaches to main function.

1. Program vector table base for exception handling
2. Invalidate instruction cache, data cache and TLBs
3. Program stack pointer for various modes (IRQ, FIQ, supervisor, undefined, abort, system)
4. Program counter frequency
5. Configure MMU with short descriptor translation table format and program base address of translation table
6. Enable data cache, instruction cache and MMU

7. Transfer control to `_start` which clears BSS sections and runs global constructor before jumping to main application

The `translation_table.S` file contains a static page table required by MMU for cortex-A53. This translation table is flat mapped (input address = output address) with default memory attributes defined for zynq ultrascale+ architecture. It utilizes short descriptor translation table format with each section defining 1MB of memory. The overview of translation table memory attributes is described below.

	Memory Range	Definition in Translation Table	Note
DDR	0x00000000 - 0x7FFFFFFF	Normal write-back Cacheable	For a system where DDR is less than 2GB, region after DDR and before PL is marked as undefined/reserved in translation table
PL	0x80000000 - 0xBFFFFFFF	Strongly Ordered	
QSPI, lower PCIe	0xC0000000 - 0xFFFFFFFF	Device Memory	
Reserved	0xF0000000 - 0xF7FFFFFF	Unassigned	
STM Coresight	0xF8000000 - 0xF8FFFFFF	Device Memory	
GIC	0xF9000000 - 0xF90FFFFFF	Device memory	
Reserved	0xF9100000 - 0xFCFFFFFF	Unassigned	
FPS, LPS slaves	0xFD000000 - 0xFFBFFFFFF	Device memory	
CSU, PMU	0xFFC00000 - 0xFFDFFFFFF	Device Memory	This region contains CSU and PMU memory which are marked as Device since it is less than 1MB and falls in a region with device memory
TCM, OCM	0FFE00000 - 0xFFFFFFFF	Normal write-back cacheable	

Cortex A53 32-bit Processor Cache Functions

Overview

Cache functions provide access to cache related operations such as flush and invalidate for instruction and data caches. It gives option to perform the cache operations on a single cacheline, a range of memory and an entire cache.

Functions

- void `Xil_DCacheEnable` (void)
- void `Xil_DCacheDisable` (void)
- void `Xil_DCachel Invalidate` (void)
- void `Xil_DCachel InvalidateRange` (INTPTR adr, u32 len)
- void `Xil_DCachel InvalidateLine` (u32 adr)
- void `Xil_DCacheFlush` (void)
- void `Xil_DCacheFlushLine` (u32 adr)
- void `Xil_ICacheEnable` (void)
- void `Xil_ICacheDisable` (void)
- void `Xil_ICachel Invalidate` (void)
- void `Xil_ICachel InvalidateRange` (INTPTR adr, u32 len)
- void `Xil_ICachel InvalidateLine` (u32 adr)

Function Documentation

`void Xil_DCacheEnable (void)`

Enable the Data cache.

Parameters

None.	
-------	--

Returns

None.

Note

None.

void Xil_DCACHEDisable(void)

Disable the Data cache.

Parameters

<i>None.</i>	
--------------	--

Returns

None.

Note

None.

void Xil_DCACHEInvalidate(void)

Invalidate the Data cache. The contents present in the data cache are cleaned and invalidated.

Parameters

<i>None.</i>	
--------------	--

Returns

None.

Note

In Cortex-A53, functionality to simply invalidate the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate to avoid such corruption.

void Xil_DCACHEInvalidateRange(INTPTR adr, u32 len)

Invalidate the Data cache for the given address range. The cachelines present in the address range are cleaned and invalidated.

Parameters

<i>adr</i>	32bit start address of the range to be invalidated.
<i>len</i>	Length of the range to be invalidated in bytes.

Returns

None.

Note

In Cortex-A53, functionality to simply invalide the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate to avoid such corruption.

void Xil_DCachelnvalidateLine (u32 adr)

Invalidate a Data cache line. The cacheline is cleaned and invalidated.

Parameters

adr	32 bit address of the data to be invalidated.
-----	---

Returns

None.

Note

In Cortex-A53, functionality to simply invalide the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate to avoid such corruption.

void Xil_DCacheFlush (void)

Flush the Data cache.

Parameters

None.	
-------	--

Returns

None.

Note

None.

void Xil_DCacheFlushLine (u32 adr)

Flush a Data cache line. If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

Parameters

adr	32bit address of the data to be flushed.
-----	--

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

void Xil_ICacheEnable (void)

Enable the instruction cache.

Parameters

None.	
-------	--

Returns

None.

Note

None.

void Xil_ICacheDisable (void)

Disable the instruction cache.

Parameters

None.	
-------	--

Returns

None.

Note

None.

void Xil_ICacheInvalidate (void)

Invalidate the entire instruction cache.

Parameters

None.	
-------	--

Returns

None.

Note

None.

void Xil_ICacheInvalidateRange (INTPTR adr, u32 len)

Invalidate the instruction cache for the given address range. If the instructions specified by the address range are cached by the instruction cache, the cachelines containing those instructions are invalidated.

Parameters

adr	32bit start address of the range to be invalidated.
len	Length of the range to be invalidated in bytes.

Returns

None.

Note

None.

void Xil_ICacheInvalidateLine (u32 adr)

Invalidate an instruction cache line. If the instruction specified by the address is cached by the instruction cache, the cache line containing that instruction is invalidated.

Parameters

adr	32bit address of the instruction to be invalidated..
-----	--

Returns

None.

Note

The bottom 4 bits are set to 0, forced by architecture.

Cortex A53 32-bit Processor MMU Handling

Overview

MMU functions equip users to enable MMU, disable MMU and modify default memory attributes of MMU table as per the need.

Functions

- void [Xil_SetTlbAttributes](#) (INTPTR Addr, u32 attrib)
- void [Xil_EnableMMU](#) (void)
- void [Xil_DisableMMU](#) (void)

Function Documentation

void Xil_SetTlbAttributes (INTPTR Addr, u32 attrib)

This function sets the memory attributes for a section covering 1MB of memory in the translation table.

Parameters

<i>Addr</i>	32-bit address for which the attributes need to be set.
<i>attrib</i>	Attributes for the specified memory region. <i>xil_mmu.h</i> contains commonly used memory attributes definitions which can be utilized for this function.

Returns

None.

Note

The MMU or D-cache does not need to be disabled before changing a translation table entry.

void Xil_EnableMMU (void)

Enable MMU for Cortex-A53 processor in 32bit mode. This function invalidates the instruction and data caches before enabling MMU.

Parameters

<i>None.</i>	
--------------	--

Returns

None.

void Xil_DisableMMU (void)

Disable MMU for Cortex A53 processors in 32bit mode. This function invalidates the TLBs, Branch Predictor Array and flushed the data cache before disabling the MMU.

Parameters

None.	
-------	--

Returns

None.

Note

When the MMU is disabled, all the memory accesses are treated as strongly ordered.

Cortex A53 32-bit Mode Time Functions

Overview

The `xtime_l.c` file and corresponding `xtime_l.h` include file provide access to the 64-bit generic counter in Cortex-A53. The `sleep.c`, `usleep.c` file and the corresponding `sleep.h` include file implement sleep functions. Sleep functions are implemented as busy loops.

Functions

- void `XTime_StartTimer` (void)
- void `XTime_SetTime` (XTime Xtime_Global)
- void `XTime_GetTime` (XTime *Xtime_Global)

Function Documentation

void XTime_StartTimer (void)

Start the 64-bit physical timer counter.

Parameters

None.	
-------	--

Returns

None.

Note

The timer is initialized only if it is disabled. If the timer is already running this function does not perform any operation.

void XTime_SetTime (XTime Xtime_Global)

Timer of A53 runs continuously and the time can not be set as desired. This API doesn't contain anything. It is defined to have uniformity across platforms.

Parameters

<i>Xtime_Global</i>	64bit Value to be written to the Global Timer Counter Register.
---------------------	---

Returns

None.

Note

None.

void XTime_GetTime (XTime * Xtime_Global)

Get the time from the physical timer counter register.

Parameters

<i>Xtime_Global</i>	Pointer to the 64-bit location to be updated with the current value in physical timer counter.
---------------------	--

Returns

None.

Note

None.

Cortex A53 32-bit Processor Specific Include Files

The `xreg_cortexa53.h` file contains definitions for inline assembler code. It provides inline definitions for Cortex A53 GPRs, SPRs and floating point registers.

The `xpseudo_asm_gcc.h` contains the definitions for the most often used inline assembler instructions, available as macros. These can be very useful for tasks such as setting or getting special purpose registers, synchronization, or cache manipulation. These inline assembler instructions can be used from drivers and user applications written in C.

Cortex A53 64-bit Processor API

Overview

Cortex-A53 standalone BSP contains two separate BSPs for 32-bit mode and 64-bit mode. The 64-bit mode of cortex-A53 contains ARMv8-A architecture. This section provides a linked summary and detailed descriptions of the Cortex A53 64-bit Processor APIs.

Modules

- [Cortex A53 64-bit Processor Boot Code](#)
- [Cortex A53 64-bit Processor Cache Functions](#)
- [Cortex A53 64-bit Processor MMU Handling](#)
- [Cortex A53 64-bit Mode Time Functions](#)
- [Cortex A53 64-bit Processor Specific Include Files](#)

Cortex A53 64-bit Processor Boot Code

Overview

The boot .S file contains a minimal set of code for transferring control from the processor reset location to the start of the application. The boot code performs minimum configuration which is required for an application to run starting from processor's reset state. Cortex-A53 starts execution from EL3 and currently application is also run from EL3. Below is a sequence illustrating what all configuration is performed before control reaches to main function.

1. Program vector table base for exception handling
2. Set reset vector table base address
3. Program stack pointer for EL3
4. Routing of interrupts to EL3
5. Enable ECC protection
6. Program generic counter frequency

7. Invalidate instruction cache, data cache and TLBs
8. Configure MMU registers and program base address of translation table
9. Transfer control to `_start` which clears BSS sections and runs global constructor before jumping to main application

Cortex A53 64-bit Processor Cache Functions

Overview

Cache functions provide access to cache related operations such as flush and invalidate for instruction and data caches. It gives option to perform the cache operations on a single cacheline, a range of memory and an entire cache.

Functions

- void `Xil_DCacheEnable` (void)
- void `Xil_DCacheDisable` (void)
- void `Xil_DCachelnvalidate` (void)
- void `Xil_DCachelnvalidateRange` (INTPTR adr, INTPTR len)
- void `Xil_DCachelnvalidateLine` (INTPTR adr)
- void `Xil_DCacheFlush` (void)
- void `Xil_DCacheFlushLine` (INTPTR adr)
- void `Xil_ICacheEnable` (void)
- void `Xil_ICacheDisable` (void)
- void `Xil_ICachelnvalidate` (void)
- void `Xil_ICachelnvalidateRange` (INTPTR adr, INTPTR len)
- void `Xil_ICachelnvalidateLine` (INTPTR adr)

Function Documentation

void Xil_DCacheEnable (void)

Enable the Data cache.

Parameters

<code>None.</code>	
--------------------	--

Returns

None.

Note

None.

void Xil_DCacheDisable (void)

Disable the Data cache.

Parameters

None.	
-------	--

Returns

None.

Note

None.

void Xil_DCachelnvalidate (void)

Invalidate the Data cache. The contents present in the cache are cleaned and invalidated.

Parameters

None.	
-------	--

Returns

None.

Note

In Cortex-A53, functionality to simply invalidate the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate which avoids such corruption.

void Xil_DCachelnvalidateRange (INTPTR adr, INTPTR len)

Invalidate the Data cache for the given address range. The cachelines present in the adderss range are cleaned and invalidated.

Parameters

<i>adr</i>	64bit start address of the range to be invalidated.
<i>len</i>	Length of the range to be invalidated in bytes.

Returns

None.

Note

In Cortex-A53, functionality to simply invalide the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate which avoids such corruption.

void Xil_DCachelnvalidateLine (INTPTR adr)

Invalidate a Data cache line. The cacheline is cleaned and invalidated.

Parameters

<i>adr</i>	64bit address of the data to be flushed.
------------	--

Returns

None.

Note

In Cortex-A53, functionality to simply invalide the cachelines is not present. Such operations are a problem for an environment that supports virtualisation. It would allow one OS to invalidate a line belonging to another OS. This could lead to the other OS crashing because of the loss of essential data. Hence, such operations are promoted to clean and invalidate which avoids such corruption.

void Xil_DCacheFlush (void)

Flush the Data cache.

Parameters

None.	
-------	--

Returns

None.

Note

None.

void Xil_DCacheFlushLine (INTPTR adr)

Flush a Data cache line. If the byte specified by the address (adr) is cached by the Data cache, the cacheline containing that byte is invalidated. If the cacheline is modified (dirty), the entire contents of the cacheline are written to system memory before the line is invalidated.

Parameters

adr	64bit address of the data to be flushed.
-----	--

Returns

None.

Note

The bottom 6 bits are set to 0, forced by architecture.

void Xil_ICacheEnable (void)

Enable the instruction cache.

Parameters

None.	
-------	--

Returns

None.

Note

None.

void Xil_ICacheDisable (void)

Disable the instruction cache.

Parameters

None.	
-------	--

Returns

None.

Note

None.

void Xil_ICacheInvalidate (void)

Invalidate the entire instruction cache.

Parameters

None.	
-------	--

Returns

None.

Note

None.

void Xil_ICacheInvalidateRange (INTPTR adr, INTPTR len)

Invalidate the instruction cache for the given address range. If the instructions specified by the address range are cached by the instruction cache, the cachelines containing those instructions are invalidated.

Parameters

<i>adr</i>	64bit start address of the range to be invalidated.
<i>len</i>	Length of the range to be invalidated in bytes.

Returns

None.

Note

None.

void Xil_ICacheInvalidateLine (INTPTR adr)

Invalidate an instruction cache line. If the instruction specified by the parameter adr is cached by the instruction cache, the cacheline containing that instruction is invalidated.

Parameters

adr	64bit address of the instruction to be invalidated.
-----	---

Returns

None.

Note

The bottom 6 bits are set to 0, forced by architecture.

Cortex A53 64-bit Processor MMU Handling

Overview

MMU function equip users to modify default memory attributes of MMU table as per the need.

Functions

- void [Xil_SetTlbAttributes](#) (INTPTR Addr, u64 attrib)

Function Documentation

void Xil_SetTlbAttributes (INTPTR Addr, u64 attrib)

brief It sets the memory attributes for a section, in the translation table. If the address (defined by Addr) is less than 4GB, the memory attribute(attrib) is set for a section of 2MB memory. If the address (defined by Addr) is greater than 4GB, the memory attribute (attrib) is set for a section of 1GB memory.

Parameters

Addr	64-bit address for which attributes are to be set.
attrib	Attribute for the specified memory region. xil_mmu.h contains commonly used memory attributes definitions which can be utilized for this function.

Returns

None.

Note

The MMU and D-cache need not be disabled before changing an translation table attribute.

Cortex A53 64-bit Mode Time Functions

Overview

The `xtime_l.c` file and corresponding `xtime_l.h` include file provide access to the 64-bit generic counter in Cortex-A53. The `sleep.c`, `usleep.c` file and the corresponding `sleep.h` include file implement sleep functions. Sleep functions are implemented as busy loops.

Functions

- void `XTime_StartTimer` (void)
- void `XTime_SetTime` (XTime Xtime_Global)
- void `XTime_GetTime` (XTime *Xtime_Global)

Function Documentation

`void XTime_StartTimer(void)`

Start the 64-bit physical timer counter.

Parameters

None.	
-------	--

Returns

None.

Note

The timer is initialized only if it is disabled. If the timer is already running this function does not perform any operation.

void XTime_SetTime (*XTime Xtime_Global*)

Timer of A53 runs continuously and the time can not be set as desired. This API doesn't contain anything. It is defined to have uniformity across platforms.

Parameters

<i>Xtime_Global</i>	64bit value to be written to the physical timer counter register.
---------------------	---

Returns

None.

Note

None.

void XTime_GetTime (*XTime * Xtime_Global*)

Get the time from the physical timer counter register.

Parameters

<i>Xtime_Global</i>	Pointer to the 64-bit location to be updated with the current value of physical timer counter register.
---------------------	---

Returns

None.

Note

None.

Cortex A53 64-bit Processor Specific Include Files

The xreg_cortexa53.h file contains definitions for inline assembler code. It provides inline definitions for Cortex A53 GPRs, SPRs and floating point registers.

The xpseudo_asm_gcc.h contains the definitions for the most often used inline assembler instructions, available as macros. These can be very useful for tasks such as setting or getting special purpose registers, synchronization, or cache manipulation. These inline assembler instructions can be used from drivers and user applications written in C.

Appendix D:

XilFlash Library v4.3

Overview

The XilFlash library provides read/write/erase/lock/unlock features to access a parallel flash device. This library implements the functionality for flash memory devices that conform to the "Common Flash Interface" (CFI) standard. CFI allows a single flash library to be used for an entire family of parts and helps us determine the algorithm to utilize during runtime.

Note

All the calls in the library are blocking in nature in that the control is returned back to user only after the current operation is completed successfully or an error is reported.

Library Initialization

The `XFlash_Initialize()` function should be called by the application before any other function in the library. The initialization function checks for the device family and initializes the XFlash instance with the family specific data. The VT table (contains the function pointers to family specific APIs) is setup and family specific initialization routine is called.

Device Geometry

The device geometry varies for different flash device families. Following sections describes the geometry of different flash device families:

Intel Flash Device Geometry

Flash memory space is segmented into areas called blocks. The size of each block is based on a power of 2. A region is defined as a contiguous set of blocks of the same size. Some parts have several regions while others have one. The arrangement of blocks and regions is referred to by this module as the part's geometry. Some Intel flash supports multiple banks on the same device. This library supports single and multiple bank flash devices.

AMD Flash Device Geometry

Flash memory space is segmented into areas called banks and further in to regions and blocks. The size of each block is based on a power of 2. A region is defined as a contiguous set of blocks of the same size. Some parts have several regions while others have one. A bank is defined as a contiguous set of blocks. The bank

may contain blocks of different size. The arrangement of blocks, regions and banks is referred to by this module as the part's geometry.

The cells within the part can be programmed from a logic 1 to a logic 0 and not the other way around. To change a cell back to a logic 1, the entire block containing that cell must be erased. When a block is erased all bytes contain the value 0xFF. The number of times a block can be erased is finite. Eventually the block will wear out and will no longer be capable of erasure. As of this writing, the typical flash block can be erased 100,000 or more times.

Write Operation

The write call can be used to write a minimum of zero bytes and a maximum entire flash. If the Offset Address specified to write is out of flash or if the number of bytes specified from the Offset address exceed flash boundaries an error is reported back to the user. The write is blocking in nature in that the control is returned back to user only after the write operation is completed successfully or an error is reported.

Read Operation

The read call can be used to read a minimum of zero bytes and maximum of entire flash. If the Offset Address specified to write is out of flash boundary an error is reported back to the user. The read function reads memory locations beyond Flash boundary. Care should be taken by the user to make sure that the Number of Bytes + Offset address is within the Flash address boundaries. The write is blocking in nature in that the control is returned back to user only after the read operation is completed successfully or an error is reported.

Erase Operation

The erase operations are provided to erase a Block in the Flash memory. The erase call is blocking in nature in that the control is returned back to user only after the erase operation is completed successfully or an error is reported.

Sector Protection

The Flash Device is divided into Blocks. Each Block can be protected individually from unwarranted writing/erasing. The Block locking can be achieved using [XFlash_Lock\(\)](#) lock. All the memory locations from the Offset address specified will be locked. The block can be unlocked using [XFlash_UnLock\(\)](#) call. All the Blocks which are previously locked will be unlocked. The Lock and Unlock calls are blocking in nature in that the control is returned back to user only after the operation is completed successfully or an error is reported. The AMD flash device requires high voltage on Reset pin to perform lock and unlock operation. User must provide this high voltage (As defined in datasheet) to reset pin before calling lock and unlock API for AMD flash devices. Lock and Unlock features are not tested for AMD flash device.

Device Control

Functionalities specific to a Flash Device Family are implemented as Device Control.

The following are the Intel specific device control:

- Retrieve the last error data.
- Get Device geometry.
- Get Device properties.
- Set RYBY pin mode.
- Set the Configuration register (Platform Flash only).

The following are the AMD specific device control:

- Get Device geometry.
- Get Device properties.
- Erase Resume.
- Erase Suspend.
- Enter Extended Mode.
- Exit Extended Mode.
- Get Protection Status of Block Group.
- Erase Chip.

Note

This library needs to know the type of EMC core (AXI or XPS) used to access the cfi flash, to map the correct APIs. This library should be used with the emc driver, v3_01_a and above, so that this information can be automatically obtained from the emc driver.

This library is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads, mutual exclusion, virtual memory, cache control, or HW write protection management must be satisfied by the layer above this library.

All writes to flash occur in units of bus-width bytes. If more than one part exists on the data bus, then the parts are written in parallel. Reads from flash are performed in any width up to the width of the data bus. It is assumed that the flash bus controller or local bus supports these types of accesses.

XilFlash Library API

Overview

This chapter provides a linked summary and detailed descriptions of the LibXil Flash library APIs.

Functions

- int `XFlash_Initialize` (XFlash *InstancePtr, u32 BaseAddress, u8 BusWidth, int IsPlatformFlash)
- int `XFlash_Reset` (XFlash *InstancePtr)
- int `XFlash_DeviceControl` (XFlash *InstancePtr, u32 Command, DeviceCtrlParam *Parameters)
- int `XFlash_Read` (XFlash *InstancePtr, u32 Offset, u32 Bytes, void *DestPtr)
- int `XFlash_Write` (XFlash *InstancePtr, u32 Offset, u32 Bytes, void *SrcPtr)
- int `XFlash_Erase` (XFlash *InstancePtr, u32 Offset, u32 Bytes)
- int `XFlash_Lock` (XFlash *InstancePtr, u32 Offset, u32 Bytes)
- int `XFlash_Unlock` (XFlash *InstancePtr, u32 Offset, u32 Bytes)
- int `XFlash_IsReady` (XFlash *InstancePtr)

Function Documentation

int XFlash_Initialize (*XFlash * InstancePtr*, *u32 BaseAddress*, *u8 BusWidth*, *int IsPlatformFlash*)

This function initializes a specific XFlash instance.

The initialization entails:

- Check the Device family type.
- Issuing the CFI query command.
- Get and translate relevant CFI query information.
- Set default options for the instance.
- Setup the VTable.
- Call the family initialize function of the instance.

Initialize the Xilinx Platform Flash XL to Async mode if the user selects to use the Platform Flash XL in the MLD. The Platform Flash XL is an Intel CFI complaint device.

Parameters

<i>InstancePtr</i>	Pointer to the XFlash instance.
<i>BaseAddress</i>	Base address of the flash memory.
<i>BusWidth</i>	Total width of the flash memory, in bytes.
<i>IsPlatformFlash</i>	Used to specify if the flash is a platform flash.

Returns

- XST_SUCCESS if successful.
- XFLASH_PART_NOT_SUPPORTED if the command set algorithm or Layout is not supported by any flash family compiled into the system.
- XFLASH_CFI_QUERY_ERROR if the device would not enter CFI query mode. Either the device(s) do not support CFI, the wrong BaseAddress param was used, an unsupported part layout exists, or a hardware problem exists with the part.

Note

BusWidth is not the width of an individual part. Its the total operating width. For example, if there are two 16-bit parts, with one tied to data lines D0-D15 and other tied to D15-D31, BusWidth would be $(32 / 8) = 4$. If a single 16-bit flash is in 8-bit mode, then BusWidth should be $(8 / 8) = 1$.

int XFlash_Reset (XFlash * *InstancePtr*)

This function resets the flash device and places it in read mode.

Parameters

<i>InstancePtr</i>	Pointer to the XFlash instance.
--------------------	---------------------------------

Returns

- XST_SUCCESS if successful.
- XFLASH_BUSY if the flash devices were in the middle of an operation and could not be reset.
- XFLASH_ERROR if the device(s) have experienced an internal error during the operation. [XFlash_DeviceControl\(\)](#) must be used to access the cause of the device specific error condition.

Note

None.

int XFlash_DeviceControl (XFlash * InstancePtr, u32 Command, DeviceCtrlParam * Parameters)

This function is used to execute device specific commands.

For a list of device specific commands, see the xilflash.h.

Parameters

<i>InstancePtr</i>	Pointer to the XFlash instance.
<i>Command</i>	Device specific command to issue.
<i>Parameters</i>	Specifies the arguments passed to the device control function.

Returns

- XST_SUCCESS if successful.
- XFLASH_NOT_SUPPORTED if the command is not recognized/supported by the device(s).

Note

None.

int XFlash_Read (XFlash * InstancePtr, u32 Offset, u32 Bytes, void * DestPtr)

This function reads the data from the Flash device and copies it into the specified user buffer. The source and destination addresses can be on any alignment supported by the processor. The device is polled until an error or the operation completes successfully.

Parameters

<i>InstancePtr</i>	Pointer to the XFlash instance.
<i>Offset</i>	Offset into the device(s) address space from which to read.
<i>Bytes</i>	Number of bytes to copy.
<i>DestPtr</i>	Destination address to copy data to.

Returns

- XST_SUCCESS if successful.
- XFLASH_ADDRESS_ERROR if the source address does not start within the addressable areas of the device(s).

Note

This function allows the transfer of data past the end of the device's address space. If this occurs, then results are undefined.

int XFlash_Write (XFlash * InstancePtr, u32 Offset, u32 Bytes, void * SrcPtr)

This function programs the flash device(s) with data specified in the user buffer.

The source and destination address must be aligned to the width of the flash's data bus.

The device is polled until an error or the operation completes successfully.

Parameters

<i>InstancePtr</i>	Pointer to the XFlash instance.
<i>Offset</i>	Offset into the device(s) address space from which to begin programming. Must be aligned to the width of the flash's data bus.
<i>Bytes</i>	Number of bytes to program.
<i>SrcPtr</i>	Source address containing data to be programmed. Must be aligned to the width of the flash's data bus. The SrcPtr doesn't have to be aligned to the flash width if the processor supports unaligned access. But, since this library is generic, and some processors(eg. Microblaze) do not support unaligned access; this API requires the SrcPtr to be aligned.

Returns

- XST_SUCCESS if successful.
- XFLASH_ERROR if a write error occurred. This error is usually device specific. Use [XFlash_DeviceControl\(\)](#) to retrieve specific error conditions. When this error is returned, it is possible that the target address range was only partially programmed.

Note

None.

int XFlash_Erase (XFlash * InstancePtr, u32 Offset, u32 Bytes)

This function erases the specified address range in the flash device.

The number of bytes to erase can be any number as long as it is within the bounds of the device(s).

The device is polled until an error or the operation completes successfully.

Parameters

<i>InstancePtr</i>	Pointer to the XFlash instance.
<i>Offset</i>	Offset into the device(s) address space from which to begin erasure.
<i>Bytes</i>	Number of bytes to erase.

Returns

- XST_SUCCESS if successful.
- XFLASH_ADDRESS_ERROR if the destination address range is not completely within the addressable areas of the device(s).

Note

Due to flash memory design, the range actually erased may be larger than what was specified by the Offset & Bytes parameters. This will occur if the parameters do not align to block boundaries.

int XFlash_Lock (XFlash * InstancePtr, u32 Offset, u32 Bytes)

This function Locks the blocks in the specified range of the flash device(s).

The device is polled until an error or the operation completes successfully.

Parameters

<i>InstancePtr</i>	Pointer to the XFlash instance.
<i>Offset</i>	Offset into the device(s) address space from which to begin block locking. The first three bytes of every block is reserved for special purpose. The offset should be atleast three bytes from start of the block.
<i>Bytes</i>	Number of bytes to Lock in the Block starting from Offset.

Returns

- XST_SUCCESS if successful.
- XFLASH_ADDRESS_ERROR if the destination address range is not completely within the addressable areas of the device(s).

Note

Due to flash memory design, the range actually locked may be larger than what was specified by the Offset & Bytes parameters. This will occur if the parameters do not align to block boundaries.

int XFlash_Unlock (XFlash * InstancePtr, u32 Offset, u32 Bytes)

This function Unlocks the blocks in the specified range of the flash device(s).

The device is polled until an error or the operation completes successfully.

Parameters

<i>InstancePtr</i>	Pointer to the XFlash instance.
<i>Offset</i>	Offset into the device(s) address space from which to begin block UnLocking. The first three bytes of every block is reserved for special purpose. The offset should be atleast three bytes from start of the block.
<i>Bytes</i>	Number of bytes to UnLock in the Block starting from Offset.

Returns

- XST_SUCCESS if successful.
- XFLASH_ADDRESS_ERROR if the destination address range is not completely within the addressable areas of the device(s).

Note

None.

int XFlash_IsReady (XFlash * *InstancePtr*)

This function checks the readiness of the device, which means it has been successfully initialized.

Parameters

<i>InstancePtr</i>	Pointer to the XFlash instance.
--------------------	---------------------------------

Returns

TRUE if the device has been initialized (but not necessarily started), and FALSE otherwise.

Note

None.

Library Parameters in MSS File

XilFlash Library can be integrated with a system using the following snippet in the Microprocessor Software Specification (MSS) file:

```
BEGIN LIBRARY
PARAMETER LIBRARY_NAME = xilflash
PARAMETER LIBRARY_VER = 4.3
PARAMETER PROC_INSTANCE = microblaze_0
PARAMETER ENABLE_INTEL = true
PARAMETER ENABLE_AMD = false
END
```

The table below describes the libgen customization parameters.

Parameter	Default Value	Description
LIBRARY_NAME	xilflash	Specifies the library name.
LIBRARY_VER	4.3	Specifies the library version.
PROC_INSTANCE	microblaze_0	Specifies the processor name.
ENABLE_INTEL	true/false	Enables or disables the Intel flash device family.
ENABLE_AMD	true/false	Enables or disables the AMD flash device family.

Appendix E:

Xillsf Library v5.8

Overview

The LibXil Isf library:

- Allows you to Write, Read, and Erase the Serial Flash.
- Allows protection of the data stored in the Serial Flash from unwarranted modification by enabling the Sector Protection feature.
- Supports multiple instances of Serial Flash at a time, provided they are of the same device family (Atmel, Intel, STM, Winbond, SST, or Spansion) as the device family is selected at compile time.
- Allows the user application to perform Control operations on Intel, STM, Winbond, SST, and Spansion Serial Flash.
- Requires the underlying hardware platform to contain the axi_quad_spi, ps7_spi, ps7_qspi, psu_qspi or psu_spi device for accessing the Serial Flash.
- Uses the Xilinx® SPI interface drivers in interrupt-driven mode or polled mode for communicating with the Serial Flash. In interrupt mode, the user application must acknowledge any associated interrupts from the Interrupt Controller.

Additional information:

- In interrupt mode, the application is required to register a callback to the library and the library registers an internal status handler to the selected interface driver.
- When the user application requests a library operation, it is initiated and control is given back to the application. The library tracks the status of the interface transfers, and notifies the user application upon completion of the selected library operation.
- Added support in the library for SPI PS and QSPI PS. You must select one of the interfaces at compile time.
- Added support for QSPIPSU and SPIPS flash interface on Zynq® UltraScale+™ MPSoC.
- When the user application requests selection of QSPIPS interface during compilation, the QSPI PS or QSPI PSU interface, based on the hardware platform, are selected. Similarly, if the SPIPS interface is selected during compilation, SPI PS or SPI PSU interface are selected.

Supported Devices

The table below lists the supported Xilinx in-system and external serial flash memories.

Device Series	Manufacturer
AT45DB011D AT45DB021D AT45DB041D AT45DB081D AT45DB161D AT45DB321D AT45DB642D	Atmel
W25Q16 W25Q32 W25Q64 W25Q80 W25Q128 W25X10 W25X20 W25X40 W25X80 W25X16 W25X32 W25X64	Winbond
S25FL004 S25FL008 S25FL016 S25FL032 S25FL064 S25FL128 S25FL129 S25FL256 S25FL512 S70FL01G	Spansion
SST25WF080	SST

Device Series	Manufacturer
N25Q032 N25Q064 N25Q128 N25Q256 N25Q512 N25Q00AA MT25Q01 MT25Q02	Micron

Note

Intel, STM, and Numonyx serial flash devices are now a part of Serial Flash devices provided by Micron.

References

- Spartan-3AN FPGA In-System Flash User Guide (UG333):
http://www.xilinx.com/support/documentation/user_guides/ug333.pdf
- Atmel Serial Flash Memory website (AT45XXD):
http://www.atmel.com/dyn/products/devices.asp?family_id=616#1802
- Intel (Numonyx) S33 Serial Flash Memory website (S33):
http://www.numonyx.com/Documents/Datasheets/314822_S33_Discrete_DS.pdf
- STM (Numonyx) M25PXX Serial Flash Memory website (M25PXX):
<http://www.numonyx.com/en-US/MemoryProducts/NORserial/Pages/M25PTechnicalDocuments.aspx>
- Winbond Serial Flash Page:
<http://www.winbond-usa.com/hq/enu/ProductAndSales/ProductLines/FlashMemory/SerialFlash/>
- Spansion website:
<http://www.spansion.com/Support/Pages/DatasheetsIndex.aspx>
- SST SST25WF080:
<http://www.sst.com/dotAsset/40369.pdf>
- Micron N25Q flash family:
<http://www.micron.com/products/nor-flash/serial-norflash/n25q#/>

Xllsf Library API

Overview

This chapter provides a linked summary and detailed descriptions of the Xllsf library APIs.

Functions

- int [Xlsf_Initialize](#) (Xlsf *InstancePtr, Xlsf_Iface *SpiInstPtr, u8 SlaveSelect, u8 *WritePtr)
- int [Xlsf_GetStatus](#) (Xlsf *InstancePtr, u8 *ReadPtr)
- int [Xlsf_GetStatusReg2](#) (Xlsf *InstancePtr, u8 *ReadPtr)
- int [Xlsf_GetDeviceInfo](#) (Xlsf *InstancePtr, u8 *ReadPtr)
- int [Xlsf_Write](#) (Xlsf *InstancePtr, Xlsf_WriteOperation Operation, void *OpParamPtr)
- int [Xlsf_Read](#) (Xlsf *InstancePtr, Xlsf_ReadOperation Operation, void *OpParamPtr)
- int [Xlsf_Erase](#) (Xlsf *InstancePtr, Xlsf_EraseOperation Operation, u32 Address)
- int [Xlsf_MicronFlashEnter4BAddMode](#) (Xlsf *InstancePtr)
- int [Xlsf_MicronFlashExit4BAddMode](#) (Xlsf *InstancePtr)
- int [Xlsf_SectorProtect](#) (Xlsf *InstancePtr, Xlsf_SpOperation Operation, u8 *BufferPtr)
- int [Xlsf_Ioctl](#) (Xlsf *InstancePtr, Xlsf_IoctlOperation Operation)
- int [Xlsf_WriteEnable](#) (Xlsf *InstancePtr, u8 WriteEnable)
- void [Xlsf_RegisterInterface](#) (Xlsf *InstancePtr)
- int [Xlsf_SetSpiConfiguration](#) (Xlsf *InstancePtr, Xlsf_Iface *SpiInstPtr, u32 Options, u8 PreScaler)
- void [Xlsf_SetStatusHandler](#) (Xlsf *InstancePtr, Xlsf_Iface *XlifaceInstancePtr, Xlsf_StatusHandler Xllsf_Handler)
- void [Xlsf_IfaceHandler](#) (void *CallBackRef, u32 StatusEvent, unsigned int ByteCount)

Function Documentation

int [Xlsf_Initialize](#) (*Xlsf * InstancePtr, Xlsf_Iface * SpiInstPtr, u8 SlaveSelect, u8 * WritePtr*)

This API when called initializes the SPI interface with default settings.

With custom settings, user should call [XIsf_SetSpiConfiguration\(\)](#) and then call this API. The geometry of the underlying Serial Flash is determined by reading the Joint Electron Device Engineering Council (JEDEC) Device Information and the Status Register of the Serial Flash.

Parameters

<i>InstancePtr</i>	Pointer to the XIsf instance.
<i>SpiInstPtr</i>	Pointer to XIsf_Iface instance to be worked on.
<i>SlaveSelect</i>	It is a 32-bit mask with a 1 in the bit position of slave being selected. Only one slave can be selected at a time.
<i>WritePtr</i>	<p>Pointer to the buffer allocated by the user to be used by the In-system and Serial Flash Library to perform any read/write operations on the Serial Flash device. User applications must pass the address of this buffer for the Library to work.</p> <ul style="list-style-type: none"> • Write operations : <ul style="list-style-type: none"> ◦ The size of this buffer should be equal to the Number of bytes to be written to the Serial Flash + XISF_CMD_MAX_EXTRA_BYTES. ◦ The size of this buffer should be large enough for usage across all the applications that use a common instance of the Serial Flash. ◦ A minimum of one byte and a maximum of ISF_PAGE_SIZE bytes can be written to the Serial Flash, through a single Write operation. • Read operations : <ul style="list-style-type: none"> ◦ The size of this buffer should be equal to XISF_CMD_MAX_EXTRA_BYTES, if the application only reads from the Serial Flash (no write operations).

Returns

- XST_SUCCESS if successful.
- XST_DEVICE_IS_STOPPED if the device must be started before transferring data.
- XST_FAILURE, otherwise.

Note

- The [XIsf_Initialize\(\)](#) API is a blocking call (for both polled and interrupt modes of the Spi driver). It reads the JEDEC information of the device and waits till the transfer is complete before checking if the information is valid.
- This library can support multiple instances of Serial Flash at a time, provided they are of the same device family (either Atmel, Intel or STM, Winbond or Spansion) as the device family is selected at compile time.

int XIsf_GetStatus (*XIsf * InstancePtr, u8 * ReadPtr*)

This API reads the Serial Flash Status Register.

Parameters

<i>InstancePtr</i>	Pointer to the XIsf instance.
<i>ReadPtr</i>	Pointer to the memory where the Status Register content is copied.

Returns

XST_SUCCESS if successful else XST_FAILURE.

Note

The contents of the Status Register is stored at second byte pointed by the ReadPtr.

int XIsf_GetStatusReg2 (*XIsf * InstancePtr, u8 * ReadPtr*)

This API reads the Serial Flash Status Register 2.

Parameters

<i>InstancePtr</i>	Pointer to the XIsf instance.
<i>ReadPtr</i>	Pointer to the memory where the Status Register content is copied.

Returns

XST_SUCCESS if successful else XST_FAILURE.

Note

The contents of the Status Register 2 is stored at the second byte pointed by the ReadPtr. This operation is available only in Winbond Serial Flash.

int XIsf_GetDeviceInfo (*XIsf * InstancePtr, u8 * ReadPtr*)

This API reads the Joint Electron Device Engineering Council (JEDEC) information of the Serial Flash.

Parameters

<i>InstancePtr</i>	Pointer to the XIsf instance.
<i>ReadPtr</i>	Pointer to the buffer where the Device information is copied.

Returns

XST_SUCCESS if successful else XST_FAILURE.

Note

The Device information is stored at the second byte pointed by the ReadPtr.

int XIsf_Write (*XIsf * InstancePtr, XIsf_WriteOperation Operation, void * OpParamPtr*)

This API writes the data to the Serial Flash.

Parameters

<i>InstancePtr</i>	Pointer to the XIsf instance.
<i>Operation</i>	<p>Type of write operation to be performed on the Serial Flash. The different operations are</p> <ul style="list-style-type: none"> • XISF_WRITE: Normal Write • XISF_DUAL_IP_PAGE_WRITE: Dual Input Fast Program • XISF_DUAL_IP_EXT_PAGE_WRITE: Dual Input Extended Fast Program • XISF_QUAD_IP_PAGE_WRITE: Quad Input Fast Program • XISF_QUAD_IP_EXT_PAGE_WRITE: Quad Input Extended Fast Program • XISF_AUTO_PAGE_WRITE: Auto Page Write • XISF_BUFFER_WRITE: Buffer Write • XISF_BUF_TO_PAGE_WRITE_WITH_ERASE: Buffer to Page Transfer with Erase • XISF_BUF_TO_PAGE_WRITE_WITHOUT_ERASE: Buffer to Page Transfer without Erase • XISF_WRITE_STATUS_REG: Status Register Write • XISF_WRITE_STATUS_REG2: 2 byte Status Register Write • XISF OTP_WRITE: OTP Write.
<i>OpParamPtr</i>	Pointer to a structure variable which contains operational parameters of the specified operation. This parameter type is dependant on value of first argument(Operation). For more details, refer Operations .

Operations

- Normal Write(XISF_WRITE), Dual Input Fast Program (XISF_DUAL_IP_PAGE_WRITE), Dual Input Extended Fast Program(XISF_DUAL_IP_EXT_PAGE_WRITE), Quad Input Fast Program(XISF_QUAD_IP_PAGE_WRITE), Quad Input Extended Fast Program (XISF_QUAD_IP_EXT_PAGE_WRITE):
 - The OpParamPtr must be of type struct XIsf_WriteParam.
 - OpParamPtr->Address is the start address in the Serial Flash.
 - OpParamPtr->WritePtr is a pointer to the data to be written to the Serial Flash.
 - OpParamPtr->NumBytes is the number of bytes to be written to Serial Flash.
 - This operation is supported for Atmel, Intel, STM, Winbond and Spansion Serial Flash.
- Auto Page Write (XISF_AUTO_PAGE_WRITE):
 - The OpParamPtr must be of 32 bit unsigned integer variable.
 - This is the address of page number in the Serial Flash which is to be refreshed.
 - This operation is only supported for Atmel Serial Flash.
- Buffer Write (XISF_BUFFER_WRITE):
 - The OpParamPtr must be of type struct XIsf_BufferToFlashWriteParam.
 - OpParamPtr->BufferNum specifies the internal SRAM Buffer of the Serial Flash. The valid values are XISF_PAGE_BUFFER1 or XISF_PAGE_BUFFER2. XISF_PAGE_BUFFER2 is not valid in case of AT45DB011D Flash as it contains a single buffer.
 - OpParamPtr->WritePtr is a pointer to the data to be written to the Serial Flash SRAM Buffer.
 - OpParamPtr->ByteOffset is byte offset in the buffer from where the data is to be written.
 - OpParamPtr->NumBytes is number of bytes to be written to the Buffer. This operation is supported only for Atmel Serial Flash.
- Buffer To Memory Write With Erase (XISF_BUF_TO_PAGE_WRITE_WITH_ERASE)/ Buffer To Memory Write Without Erase (XISF_BUF_TO_PAGE_WRITE_WITHOUT_ERASE):
 - The OpParamPtr must be of type struct XIsf_BufferToFlashWriteParam.
 - OpParamPtr->BufferNum specifies the internal SRAM Buffer of the Serial Flash. The valid values are XISF_PAGE_BUFFER1 or XISF_PAGE_BUFFER2. XISF_PAGE_BUFFER2 is not valid in case of AT45DB011D Flash as it contains a single buffer.
 - OpParamPtr->Address is starting address in the Serial Flash memory from where the data is to be written. These operations are only supported for Atmel Serial Flash.
- Write Status Register (XISF_WRITE_STATUS_REG):
 - The OpParamPtr must be of type of 8 bit unsigned integer variable. This is the value to be written to the Status Register.
 - This operation is only supported for Intel, STM Winbond and Spansion Serial Flash.
- Write Status Register2 (XISF_WRITE_STATUS_REG2):

- The OpParamPtr must be of type (u8 *) and should point to two 8 bit unsigned integer values. This is the value to be written to the 16 bit Status Register. This operation is only supported in Winbond (W25Q) Serial Flash.
- One Time Programmable Area Write(XISF OTP_WRITE):
 - The OpParamPtr must be of type struct Xlsf_WriteParam.
 - OpParamPtr->Address is the address in the SRAM Buffer of the Serial Flash to which the data is to be written.
 - OpParamPtr->WritePtr is a pointer to the data to be written to the Serial Flash.
 - OpParamPtr->NumBytes should be set to 1 when performing OTPWrite operation. This operation is only supported for Intel Serial Flash.

Returns

XST_SUCCESS if successful else XST_FAILURE.

Note

- Application must fill the structure elements of the third argument and pass its pointer by type casting it with void pointer.
- For Intel, STM, Winbond and Spansion Serial Flash, the user application must call the [Xlsf_WriteEnable\(\)](#) API by passing XISF_WRITE_ENABLE as an argument, before calling the [Xlsf_Write\(\)](#) API.

int XIsf_Read (XIsf * InstancePtr, XIsf_ReadOperation Operation, void * OpParamPtr)

This API reads the data from the Serial Flash.

Parameters

<i>InstancePtr</i>	Pointer to the XIsf instance.
<i>Operation</i>	<p>Type of the read operation to be performed on the Serial Flash. The different operations are</p> <ul style="list-style-type: none"> • XISF_READ: Normal Read • XISF_FAST_READ: Fast Read • XISF_PAGE_TO_BUF_TRANS: Page to Buffer Transfer • XISF_BUFFER_READ: Buffer Read • XISF_FAST_BUFFER_READ: Fast Buffer Read • XISF OTP_READ: One Time Programmable Area (OTP) Read • XISF_DUAL_OP_FAST_READ: Dual Output Fast Read • XISF_DUAL_IO_FAST_READ: Dual Input/Output Fast Read • XISF_QUAD_OP_FAST_READ: Quad Output Fast Read • XISF_QUAD_IO_FAST_READ: Quad Input/Output Fast Read
<i>OpParamPtr</i>	Pointer to structure variable which contains operational parameter of specified Operation. This parameter type is dependant on the type of Operation to be performed. For more details, refer Operations .

Operations

- Normal Read (XISF_READ), Fast Read (XISF_FAST_READ), One Time Programmable Area Read(XISF OTP_READ), Dual Output Fast Read (XISF_CMD_DUAL_OP_FAST_READ), Dual Input/Output Fast Read (XISF_CMD_DUAL_IO_FAST_READ), Quad Output Fast Read (XISF_CMD_QUAD_OP_FAST_READ) and Quad Input/Output Fast Read (XISF_CMD_QUAD_IO_FAST_READ):
 - The OpParamPtr must be of type struct XIsf_ReadParam.
 - OpParamPtr->Address is start address in the Serial Flash.
 - OpParamPtr->ReadPtr is a pointer to the memory where the data read from the Serial Flash is stored.
 - OpParamPtr->NumBytes is number of bytes to read.
 - OpParamPtr->NumDummyBytes is the number of dummy bytes to be transmitted for the Read command. This parameter is only used in case of Dual and Quad reads.

- Normal Read and Fast Read operations are supported for Atmel, Intel, STM, Winbond and Spansion Serial Flash.
 - Dual and quad reads are supported for Winbond (W25QXX), Numonyx(N25QXX) and Spansion (S25FL129) quad flash.
 - OTP Read operation is only supported in Intel Serial Flash.
- Page To Buffer Transfer (XISF_PAGE_TO_BUF_TRANS):
 - The OpParamPtr must be of type struct XIsf_FlashToBufTransferParam .
 - OpParamPtr->BufferNum specifies the internal SRAM Buffer of the Serial Flash. The valid values are XISF_PAGE_BUFFER1 or XISF_PAGE_BUFFER2. XISF_PAGE_BUFFER2 is not valid in case of AT45DB011D Flash as it contains a single buffer.
 - OpParamPtr->Address is start address in the Serial Flash. This operation is only supported in Atmel Serial Flash.
 - Buffer Read (XISF_BUFFER_READ) and Fast Buffer Read(XISF_FAST_BUFFER_READ):
 - The OpParamPtr must be of type struct XIsf_BufferReadParam.
 - OpParamPtr->BufferNum specifies the internal SRAM Buffer of the Serial Flash. The valid values are XISF_PAGE_BUFFER1 or XISF_PAGE_BUFFER2. XISF_PAGE_BUFFER2 is not valid in case of AT45DB011D Flash as it contains a single buffer.
 - OpParamPtr->ReadPtr is pointer to the memory where data read from the SRAM buffer is to be stored.
 - OpParamPtr->ByteOffset is byte offset in the SRAM buffer from where the first byte is read.
 - OpParamPtr->NumBytes is the number of bytes to be read from the Buffer. These operations are supported only in Atmel Serial Flash.

Returns

XST_SUCCESS if successful else XST_FAILURE.

Note

- Application must fill the structure elements of the third argument and pass its pointer by type casting it with void pointer.
- The valid data is available from the fourth location pointed to by the ReadPtr for Normal Read and Buffer Read operations.
- The valid data is available from fifth location pointed to by the ReadPtr for Fast Read, Fast Buffer Read and OTP Read operations.
- The valid data is available from the (4 + NumDummyBytes)th location pointed to by ReadPtr for Dual/Quad Read operations.

int XIsf_Erase (*XIsf * InstancePtr, XIsf_EraseOperation Operation, u32 Address*)

This API erases the contents of the specified memory in the Serial Flash.

Parameters

<i>InstancePtr</i>	Pointer to the XIsf instance.
<i>Operation</i>	Type of Erase operation to be performed on the Serial Flash. The different operations are <ul style="list-style-type: none"> • XISF_PAGE_ERASE: Page Erase • XISF_BLOCK_ERASE: Block Erase • XISF_SECTOR_ERASE: Sector Erase • XISF_BULK_ERASE: Bulk Erase
<i>Address</i>	Address of the Page/Block/Sector to be erased. The address can be either Page address, Block address or Sector address based on the Erase operation to be performed.

Returns

XST_SUCCESS if successful else XST_FAILURE.

Note

- The erased bytes will read as 0xFF.
- For Intel, STM, Winbond or Spansion Serial Flash the user application must call [XIsf_WriteEnable\(\)](#) API by passing XISF_WRITE_ENABLE as an argument before calling [XIsf_Erase\(\)](#) API.
- Atmel Serial Flash support Page/Block/Sector Erase
- operations.
- Intel, Winbond, Numonyx (N25QXX) and Spansion Serial Flash support Sector/Block/Bulk Erase operations.
- STM (M25PXX) Serial Flash support Sector/Bulk Erase operations.

int XIsf_MicronFlashEnter4BAddMode (*XIsf * InstancePtr*)

This API enters the Micron flash device into 4 bytes addressing mode.

As per the Micron spec, before issuing the command to enter into 4 byte addr mode, a write enable command is issued.

Parameters

<i>InstancePtr</i>	Pointer to the Xlsf instance.
--------------------	-------------------------------

Returns

- XST_SUCCESS if successful.
- XST_FAILURE if it fails.

Note

Applicable only for Micron flash devices

int Xlsf_MicronFlashExit4BAddMode (*Xlsf * InstancePtr*)

This API exits the Micron flash device from 4 bytes addressing mode.

As per the Micron spec, before issuing this command a write enable command is first issued.

Parameters

<i>InstancePtr</i>	Pointer to the Xlsf instance.
--------------------	-------------------------------

Returns

- XST_SUCCESS if successful.
- XST_FAILURE if it fails.

Note

Applicable only for Micron flash devices

int Xlsf_SectorProtect (*Xlsf * InstancePtr, Xlsf_SpOperation Operation, u8 * BufferPtr*)

This API is used for performing Sector Protect related operations.

Parameters

<i>InstancePtr</i>	Pointer to the Xlsf instance.
<i>Operation</i>	Type of Sector Protect operation to be performed on the Serial Flash. The different operations are <ul style="list-style-type: none"> • XISF_SPR_READ: Read Sector Protection Register • XISF_SPR_WRITE: Write Sector Protection Register • XISF_SPR_ERASE: Erase Sector Protection Register • XISF_SP_ENABLE: Enable Sector Protection • XISF_SP_DISABLE: Disable Sector Protection
<i>BufferPtr</i>	Pointer to the memory where the SPR content is read to/written from. This argument can be NULL if the Operation is SprErase, SpEnable and SpDisable.

Returns

XST_SUCCESS if successful else XST_FAILURE.

Note

- The SPR content is stored at the fourth location pointed by the BufferPtr when performing XISF_SPR_READ operation.
- For Intel, STM, Winbond and Spansion Serial Flash, the user application must call the [Xlsf_WriteEnable\(\)](#) API by passing XISF_WRITE_ENABLE as an argument, before calling the [Xlsf_SectorProtect\(\)](#) API, for Sector Protect Register Write (XISF_SPR_WRITE) operation.
- Atmel Flash supports all these Sector Protect operations.
- Intel, STM, Winbond and Spansion Flash support only Sector Protect Read and Sector Protect Write operations.

int Xlsf_loctl (Xlsf * *InstancePtr*, Xlsf_loctl*Operation*)

This API configures and controls the Intel, STM, Winbond and Spansion Serial Flash.

Parameters

<i>InstancePtr</i>	Pointer to the Xlsf instance.
<i>Operation</i>	<p>Type of Control operation to be performed on the Serial Flash. The different control operations are</p> <ul style="list-style-type: none"> • XISF_RELEASE_DPD: Release from Deep Power Down (DPD) Mode • XISF_ENTER_DPD: Enter DPD Mode • XISF_CLEAR_SR_FAIL_FLAGS: Clear Status Register Fail Flags

Returns

XST_SUCCESS if successful else XST_FAILURE.

Note

- Atmel Serial Flash does not support any of these operations.
- Intel Serial Flash support Enter/Release from DPD Mode and Clear Status Register Fail Flags.
- STM, Winbond and Spansion Serial Flash support Enter/Release from DPD Mode.
- Winbond (W25QXX) Serial Flash support Enable High Performance mode.

int Xlsf_WriteEnable (Xlsf * *InstancePtr*, u8 *WriteEnable*)

This API Enables/Disables writes to the Intel, STM, Winbond and Spansion Serial Flash.

Parameters

<i>InstancePtr</i>	Pointer to the Xlsf instance.
<i>WriteEnable</i>	Specifies whether to Enable (XISF_CMD_ENABLE_WRITE) or Disable (XISF_CMD_DISABLE_WRITE) the writes to the Serial Flash.

Returns

XST_SUCCESS if successful else XST_FAILURE.

Note

This API works only for Intel, STM, Winbond and Spansion Serial Flash. If this API is called for Atmel Flash, XST_FAILURE is returned.

void XIsf_RegisterInterface (*XIsf * InstancePtr*)

This API registers the interface SPI/SPI PS/QSPI PS.

Parameters

<i>InstancePtr</i>	Pointer to the XIsf instance.
--------------------	-------------------------------

Returns

None

int XIsf_SetSpiConfiguration (*XIsf * InstancePtr, XIsf_Iface * SpilInstPtr, u32 Options, u8 PreScaler*)

This API sets the configuration of SPI.

This will set the options and clock prescaler (if applicable).

Parameters

<i>InstancePtr</i>	Pointer to the XIsf instance.
<i>SpilInstPtr</i>	Pointer to XIsf_Iface instance to be worked on.
<i>Options</i>	Specified options to be set.
<i>PreScaler</i>	Value of the clock prescaler to set.

Returns

XST_SUCCESS if successful else XST_FAILURE.

Note

This API can be called before calling [XIsf_Initialize\(\)](#) to initialize the SPI interface in other than default options mode. PreScaler is only applicable to PS SPI/QSPI.

void XIsf_SetStatusHandler (*XIsf * InstancePtr, XIsf_Iface * XIfaceInstancePtr, XIsf_StatusHandler XIsf_Handler*)

This API is to set the Status Handler when an interrupt is registered.

Parameters

<i>InstancePtr</i>	Pointer to the XIsf Instance.
<i>QspilInstancePtr</i>	Pointer to the XIsf_Iface instance to be worked on.
<i>XIsf_Handler</i>	Status handler for the application.

Returns

None

Note

None.

`void XIsf_IfaceHandler(void *CallBackRef, u32 StatusEvent, unsigned int ByteCount)`

This API is the handler which performs processing for the QSPI driver.

It is called from an interrupt context such that the amount of processing performed should be minimized. It is called when a transfer of QSPI data completes or an error occurs.

This handler provides an example of how to handle QSPI interrupts but is application specific.

Parameters

<i>CallBackRef</i>	Reference passed to the handler.
<i>StatusEvent</i>	Status of the QSPI .
<i>ByteCount</i>	Number of bytes transferred.

Returns

None

Note

None.

Library Parameters in MSS File

Xillsf Library can be integrated with a system using the following snippet in the Microprocessor Software Specification (MSS) file:

```
BEGIN LIBRARY
PARAMETER LIBRARY_NAME = xilisf
PARAMETER LIBRARY_VER = 5.8
PARAMETER serial_flash_family = 1
PARAMETER serial_flash_interface = 1
END
```

The table below describes the libgen customization parameters.

Parameter	Default Value	Description
LIBRARY_NAME	xilisf	Specifies the library name.
LIBRARY_VER	5.8	Specifies the library version.
serial_flash_family	1	Specifies the serial flash family. Supported numerical values are: 1 = Xilinx In-system Flash or Atmel Serial Flash 2 = Intel (Numonyx) S33 Serial Flash 3 = STM (Numonyx) M25PXX/N25QXX Serial Flash 4 = Winbond Serial Flash 5 = Spansion Serial Flash/Micron Serial Flash 6 = SST Serial Flash
Serial_flash_interface	1	Specifies the serial flash interface. Supported numerical values are: 1 = AXI QSPI Interface 2 = SPI PS Interface 3 = QSPI PS Interface or QSPI PSU Interface

Note

Intel, STM, and Numonyx serial flash devices are now a part of Serial Flash devices provided by Micron.

Appendix F:

XilFFS Library Reference 3.6

Overview

The LibXil fat file system (FFS) library consists of a file system and a glue layer. This FAT file system can be used with an interface supported in the glue layer. The file system code is open source and is used as it is. Glue layer implementation supports SD/eMMC interface presently.

Application should make use of APIs provided in ff.h. These file system APIs access the driver functions through the glue layer.

File System Files

The table below describes the file system files.

File	Description
ff.c	Implements all the file system APIs
ff.h	File system header
ffconf.h	File system configuration header – File system configurations such as READ_ONLY, MINIMAL. can be set here. This library uses _FS_MINIMIZE and _FS_TINY and Read/Write (NOT read only)
integer.h	Contains type definitions used by file system

Glue Layer Files

The table below describes the glue layer files.

File	Description
diskio.c	Glue layer – implements the function used by file system to call the driver APIs
diskio.h	Glue layer header

Selecting a File System with an SD Interface

To select a file system with an SD interface:

1. Launch Xilinx SDK and create a new bsp
2. Select the xilffs library.
3. In xilffs options, set fs_interface = 1 to select SD/eMMC. This is the default value. When this option is set, ensure that an SD/eMMC interface is available.
4. Build the bsp and application to use the file system with SD/eMMC.
5. SD or eMMC will be recognized by the low level driver.

Library Parameters in MSS File

XilFFS Library can be integrated with a system using the following snippet in the Microprocessor Software Specification (MSS) file:

```
parameter LIBRARY_NAME = xilffs
parameter LIBRARY_VER = 3.6
parameter fs_interface = 1
parameter read_only = false
parameter use_lfn = false
parameter enable_multi_partition = false
parameter num_logical_vol = 2
parameter use_mkfs = true
```

The table below describes the libgen customization parameters.

Parameter	Description
LIBRARY_NAME	Specifies the library name.
LIBRARY_VER	Specifies the library version.
fs_interface	File system interface. Currently SD/eMMC is the only interface supported. Default is 1.

Parameter	Description
read_only	Enables the file system in Read Only mode, if true. Default is false. Zynq® UltraScale+™ MPSoC fsbl sets this option as true.
use_lfn	Enables the long file name (LFN) support, if true. Default is false.
enable_multi_partition	Enables the multi partition support, if true. Default is false.
num_logical_vol	Number of volumes (logical drives, from 1 to 10) to be used. Default is 2
use_mkfs	Enables the mkfs support, if true. Default is true. For Zynq UltraScale+ MPSOC fsbl, set this option as false.

File System

The file system supports FAT16 and FAT32. The APIs are standard file system APIs. A detailed description can be found at http://elm-chan.org/fsw/ff/00index_e.html. Revision R0.10b is used in the library.

Appendix G:

XilRSA Library v1.3

Overview

The XilRSA library provides APIs to use RSA encryption and decryption algorithms and SHA algorithms for Zynq®-7000 All Programmable SoC devices.

Note

The RSA-2048 bit is used for RSA and the SHA-256 bit is used for hash.

For an example on usage of this library, refer to the RSA Authentication application and its documentation.

Source Files

The following is a list of source files shipped as a part of the XilRSA library:

- `librsa.a`: Pre-compiled file which contains the implementation.
 - `xilrsa.h`: This file contains the APIs for SHA2 and RSA-2048..
-

Usage of SHA-256 Functions

When all the data is available on which sha2 must be calculated, the `sha_256()` function can be used with appropriate parameters, as described. When all the data is not available on which sha2 must be calculated, use the sha2 functions in the following order:

1. `sha2_update()` can be called multiple times till input data is completed.
2. `sha2_context` is updated by the library only; do not change the values of the context.

SHA2 API Example Usage

```
sha2_context ctx;
sha2_starts(&ctx);
sha2_update(&ctx, (unsigned char *)in, size);
sha2_finish(&ctx, out);
```

Following is the source code of the `sha2_context` class.

```
typedef struct
{
    unsigned int state[8];
    unsigned char buffer[SHA_BLKBYTES];
    unsigned long long bytes;
} sha2_context;
```

XilRSA APIs

Overview

This section provides detailed descriptions of the XilRSA library APIs.

Functions

- void `rsa2048_exp` (const unsigned char *base, const unsigned char *modular, const unsigned char *modular_ext, const unsigned char *exponent, unsigned char *result)
- void `rsa2048_pubexp` (RSA_NUMBER a, RSA_NUMBER x, unsigned long e, RSA_NUMBER m, RSA_NUMBER rrm)
- void `sha_256` (const unsigned char *in, const unsigned int size, unsigned char *out)
- void `sha2_starts` (sha2_context *ctx)
- void `sha2_update` (sha2_context *ctx, unsigned char *input, unsigned int ilen)
- void `sha2_finish` (sha2_context *ctx, unsigned char *output)

Function Documentation

`void rsa2048_exp (const unsigned char * base, const unsigned char * modular, const unsigned char * modular_ext, const unsigned char * exponent, unsigned char * result)`

This function is used to encrypt the data using 2048 bit private key.

Parameters

<code>modular</code>	A char pointer which contains the key modulus
<code>modular_ext</code>	A char pointer which contains the key modulus extension
<code>exponent</code>	A char pointer which contains the private key exponent
<code>result</code>	A char pointer which contains the encrypted data

Returns

None

void rsa2048_pubexp (RSA_NUMBER a, RSA_NUMBER x, unsigned long e, RSA_NUMBER m, RSA_NUMBER rrm)

This function is used to decrypt the data using 2048 bit public key.

Parameters

a	RSA_NUMBER containing the decrypted data.
x	RSA_NUMBER containing the input data
e	Unsigned number containing the public key exponent
m	RSA_NUMBER containing the public key modulus
rrm	RSA_NUMBER containing the public key modulus extension.

Returns

None

void sha_256 (const unsigned char * in, const unsigned int size, unsigned char * out)

This function calculates the hash for the input data using SHA-256 algorithm. This function internally calls the sha2_init, updates and finishes functions and updates the result.

Parameters

In	Char pointer which contains the input data.
Size	Length of the input data
Out	Pointer to location where resulting hash will be written.

Returns

None

void sha2_starts (sha2_context * ctx)

This function initializes the SHA2 context.

Parameters

<i>ctx</i>	Pointer to sha2_context structure that stores status and buffer.
------------	--

Returns

None

void sha2_update (sha2_context * *ctx*, unsigned char * *input*, unsigned int *ilen*)

This function adds the input data to SHA256 calculation.

Parameters

<i>ctx</i>	Pointer to sha2_context structure that stores status and buffer.
<i>input</i>	Pointer to the data to add.
<i>Out</i>	Length of the input data.

Returns

None

void sha2_finish (sha2_context * *ctx*, unsigned char * *output*)

This function finishes the SHA calculation.

Parameters

<i>ctx</i>	Pointer to sha2_context structure that stores status and buffer.
<i>output</i>	Pointer to the calculated hash data.

Returns

None

Appendix H:

XilSKey Library v6.2

Overview

The XilSKey library provides APIs for programming and reading eFUSE bits and for programming the battery-backed RAM (BBRAM) of Zynq®-7000 All Programmable SoC, UltraScale™ and the Zynq® UltraScale+™ MPSoC devices.

- In Zynq-7000 devices:
 - PS eFUSE holds the RSA primary key hash bits and user feature bits, which can enable or disable some Zynq-7000 processor features.
 - PL eFUSE holds the AES key, the user key and some of the feature bits.
 - PL BBRAM holds the AES key.
- In UltraScale:
 - PL eFuse holds the AES key, 32 bit and 128 bit user key, RSA hash and some of the feature bits.
 - PL BBRAM holds AES key with or without DPA protection enable or obfuscated key programming.
- In Zynq UltraScale+ MPSoC:
 - PS eFUSE holds the AES key, the user fuses, PPK0 and PPK1 hash, SPK ID and some user feature bits which can be used to enable or disable some Zynq UltraScale+ MPSoC features.
 - BBRAM holds the AES key.

Hardware Setup

This section describes the hardware setup required for programming PL BBRAM or PL eFUSE.

Hardware setup for Zynq PL

This chapter describes the hardware setup required for programming BBRAM or eFUSE of Zynq PL devices. PL eFUSE or PL BBRAM is accessed through PS via MIO pins which are used for communication PL eFUSE or PL BBRAM through JTAG signals, these can be changed depending on the hardware setup.

A hardware setup which dedicates four MIO pins for JTAG signals should be used and the MIO pins should be mentioned in application header file (xilskey_input.h). There should be a method to download this example and have the MIO pins connected to JTAG before running this application. You can change the listed pins at your discretion.

MUX Usage Requirements

To write the PL eFUSE or PL BBRAM using a driver you must:

- Use four MIO lines (TCK,TMS,TDI,TDO)
- Connect the MIO lines to a JTAG port

If you want to switch between the external JTAG and JTAG operation driven by the MIOs, you must:

- Include a MUX between the external JTAG and the JTAG operation driven by the MIOs
- Assign a MUX selection PIN

To rephrase, to select JTAG for PL EFUSE or PL BBRAM writing, you must define the following:

- The MIOs used for JTAG operations (TCK,TMS,TDI,TDO).
- The MIO used for the MUX Select Line.
- The Value on the MUX Select line, to select JTAG for PL eFUSE or PL BBRAM writing.

The following graphic illustrates the correct MUX usage.

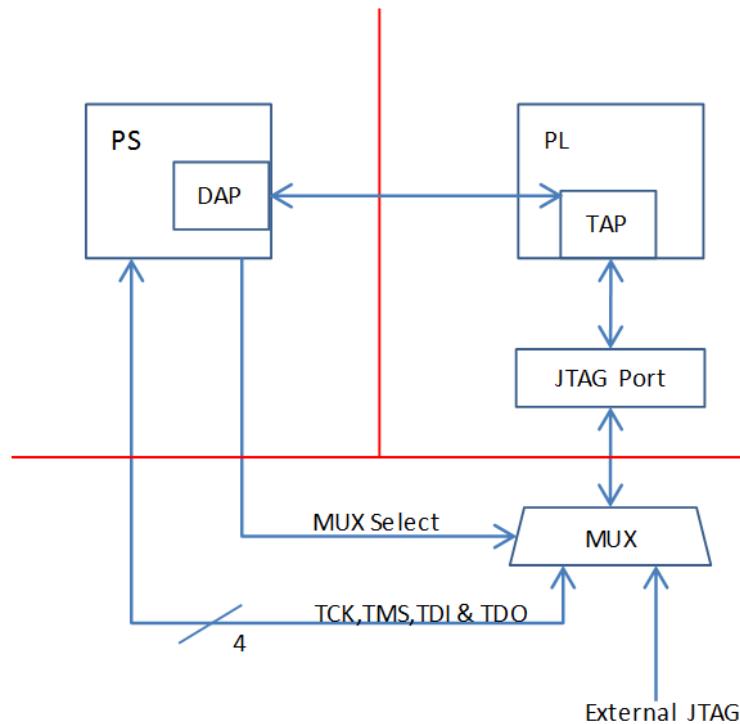


Figure 19.1: MUX Usage

Note

If you use the Vivado® Device Programmer tool to burn PL eFUSES, there is no need for MUX circuitry or MIO pins.

Hardware setup for UltraScale

This chapter describes the hardware setup required for programming BBRAM or eFUSE of UltraScale devices. Accessing UltraScale MicroBlaze eFuse is done by using block RAM initialization. UltraScale eFUSE programming is done through MASTER JTAG. Crucial Programming sequence will be taken care by Hardware module. It is mandatory to add Hardware module in the design. Use hardware module's vhd code and instructions provided to add Hardware module in the design.

- You need to add the Master JTAG primitive to design, that is, the `MASTER_JTAG_inst` instantiation has to be performed and AXI GPIO pins have to be connected to TDO, TDI, TMS and TCK signals of the `MASTER_JTAG` primitive.
- For programming eFUSE, along with master JTAG, hardware module(HWM) has to be added in design and it's signals `XSK_EFUSEPL_AXI_GPIO_HWM_READY`, `XSK_EFUSEPL_AXI_GPIO_HWM_END` and `XSK_EFUSEPL_AXI_GPIO_HWM_START`, needs to be connected to AXI GPIO pins to communicate with HWM. Hardware module is not mandatory for programming BBRAM. If your design has a HWM, it is not harmful for accessing BBRAM.
- All inputs (Master JTAG's TDO and HWM's HWM_READY, HWM_END) and all outputs (Master JTAG TDI, TMS, TCK and HWM's HWM_START) can be connected in one channel (or) inputs in one channel and outputs in other channel.
- Some of the outputs of GPIO in one channel and some others in different channels are not supported.
- The design should contain AXI BRAM control memory mapped (1MB).

Note

`MASTER_JTAG` will disable all other JTAGs.

For providing inputs of `MASTER_JTAG` signals and `HWM` signals connected to the `GPIO` pins and `GPIO` channels, refer `GPIO Pins Used for PL Master JTAG Signal` and `GPIO Channels` sections of the `UltraScale User-Configurable PL eFUSE Parameters` and `UltraScale User-Configurable PL BBRAM Parameters`.

The procedure for programming BBRAM of eFUSE of UltraScale can be referred at `UltraScale BBRAM Access Procedure` and `UltraScale eFUSE Access Procedure`.

Source Files

The following is a list of eFUSE and BBRAM application project files, folders and macros.

- `xilskey_efuse_example.c`: This file contains the main application code. The file helps in the PS/PL structure initialization and writes/reads the PS/PL eFUSE based on the user settings provided in the `xilskey_input.h` file.
- `xilskey_input.h`: This file contains all the actions that are supported by the eFUSE library. Using the preprocessor directives given in the file, you can read/write the bits in the PS/PL eFUSE. More explanation of each directive is provided in the following sections. Burning or reading the PS/PL eFUSE bits is based on the values set in the `xilskey_input.h` file. Also contains `GPIO` pins and channels connected to `MASTER JTAG` primitive and hardware module to access UltraScale eFUSE.
In this file:

- specify the 256 bit key to be programmed into BBRAM.
- specify the AES(256 bit) key, User (32 bit and 128 bit) keys and RSA key hash(384 bit) key to be programmed into UltraScale eFUSE.
- XSK_EFUSEPS_DRIVER: Define to enable the writing and reading of PS eFUSE.
- XSK_EFUSEPL_DRIVER: Define to enable the writing of PL eFUSE.
- `xilskey_bbram_example.c`: This file contains the example to program a key into BBRAM and verify the key.

Note

This algorithm only works when programming and verifying key are both executed in the recommended order.

- `xilskey_efuseps_zynqmp_example.c`: This file contains the example code to program the PS eFUSE and read back of eFUSE bits from the cache.
- `xilskey_efuseps_zynqmp_input.h`: This file contains all the inputs supported for eFUSE PS of Zynq UltraScale+ MPSoC. eFUSE bits are programmed based on the inputs from the `xilskey_efuseps_zynqmp_input.h` file.
- `xilskey_bbramps_zynqmp_example.c`: This file contains the example code to program and verify BBRAM key. Default is zero. You can modify this key on top of the file.
- `xilskey_bbram_ultrascale_example.c`: This file contains example code to program and verify BBRAM key of UltraScale.

Note

Programming and verification of BBRAM key cannot be done separately.

- `xilskey_bbram_ultrascale_input.h`: This file contains all the preprocessor directives you need to provide. In this file, specify BBRAM AES key or Obfuscated AES key to be programmed, DPA protection enable and, GPIO pins and channels connected to MASTER JTAG primitive.
- `xilskey_puf_registration.c`: This file contains all the PUF related code. This example illustrates PUF registration and generating black key and programming eFUSE with PUF helper data, CHash and Auxiliary data along with the Black key.
- `xilskey_puf_registration.h`: This file contains all the preprocessor directives based on which read/write the eFUSE bits and Syndrome data generation. More explanation of each directive is provided in the following sections.



WARNING: Ensure that you enter the correct information before writing or 'burning' eFUSE bits. Once burned, they cannot be changed. The BBRAM key can be programmed any number of times.

Note

POR reset is required for the eFUSE values to be recognized.

BBRAM PL API

Overview

This chapter provides a linked summary and detailed descriptions of the battery-backed RAM (BBRAM) APIs of Zynq® PL and UltraScale™ devices.

Example Usage

- Zynq BBRAM PL example usage:
 - The Zynq BBRAM PL example application should contain the `xilskey_bbram_example.c` and `xilskey_input.h` files.
 - You should provide user configurable parameters in the `xilskey_input.h` file. For more information, refer [Zynq User-Configurable PL BBRAM Parameters](#).
- UltraScale BBRAM example usage:
 - The UltraScale BBRAM example application should contain the `xilskey_bbram_ultrascale_input.h` and `xilskey_bbram_ultrascale_example.c` files.
 - You should provide user configurable parameters in the `xilskey_bbram_ultrascale_input.h` file. For more information, refer [UltraScale User-Configurable BBRAM PL Parameters](#).

Note

It is assumed that you have set up your hardware prior to working on the example application. For more information, refer [Hardware Setup](#).

Functions

- int `XilSKey_Bbram_Program` (`XilSKey_Bbram *InstancePtr`)
-

Function Documentation

int XilSKey_Bbram_Program (`XilSKey_Bbram * InstancePtr`)

This function implements the BBRAM algorithm for programming and verifying key.
The program and verify will only work together in and in that order.

Parameters

<code>InstancePtr</code>	Pointer to <code>XilSKey_Bbram</code>
--------------------------	---------------------------------------

Returns

- `XST_FAILURE` - In case of failure
- `XST_SUCCESS` - In case of Success

Note

This function will program BBRAM of Ultrascale and Zynq as well.

Zynq UltraScale+ MPSoC BBRAM PS API

Overview

This chapter provides a linked summary and detailed descriptions of the battery-backed RAM (BBRAM) APIs for Zynq® UltraScale+™ MPSoC devices.

Example Usage

- The Zynq UltraScale+ MPSoc example application should contain the `xilskey_bbramps_zynqmp_example.c` file.
- User configurable key can be modified in the same file (`xilskey_bbramps_zynqmp_example.c`), at the `XSK_ZYNQMP_BBRAMPS_AES_KEY` macro.

Functions

- `u32 XilSKey_ZynqMp_Bbram_Program (u32 *AesKey)`
- `void XilSKey_ZynqMp_Bbram_Zeroise ()`

Function Documentation

u32 XilSKey_ZynqMp_Bbram_Program (u32 * AesKey)

This function implements the BBRAM programming and verifying the key written.
Program and verification of AES will work only together. CRC of the provided key will be calculated internally and verified.

Parameters

<code>AesKey</code>	Pointer to the key which has to be programmed.
---------------------	--

Returns

- Error code from XskZynqMp_Ps_Bbram_ErrorCodes enum if it fails
- XST_SUCCESS if programming is done.

void XilSKey_ZynqMp_Bbram_Zeroise()

This function zeroize's Bbram Key.

Parameters

<i>None.</i>	
--------------	--

Returns

None.

Note

BBRAM key will be zeroized.

Zynq eFUSE PS API

Overview

This chapter provides a linked summary and detailed descriptions of the Zynq eFUSE PS APIs.

Example Usage

- The Zynq eFUSE PS example application should contain the `xilskey_efuse_example.c` and the `xilskey_input.h` files.
- There is no need of any hardware setup. By default, both the eFUSE PS and PL are enabled in the application. You can comment '`XSK_EFUSEPL_DRIVER`' to execute only the PS. For more details, refer [Zynq User-Configurable PS eFUSE Parameters](#).

Functions

- `u32 XilSKey_EfusePs_Write (XilSKey_EPs *PsInstancePtr)`
- `u32 XilSKey_EfusePs_Read (XilSKey_EPs *PsInstancePtr)`
- `u32 XilSKey_EfusePs_ReadStatus (XilSKey_EPs *InstancePtr, u32 *StatusBits)`

Function Documentation

u32 XilSKey_EfusePs_Write (*XilSKey_EPs * InstancePtr*)

PS eFUSE interface functions.

PS eFUSE interface functions.

Parameters

<i>InstancePtr</i>	Pointer to the PsEfuseHandle which describes which PS eFUSE bit should be burned.
--------------------	---

Returns

- XST_SUCCESS.
- In case of error, value is as defined in xilskey_utils.h Error value is a combination of Upper 8 bit value and Lower 8 bit value. For example, 0x8A03 should be checked in error.h as 0x8A00 and 0x03. Upper 8 bit value signifies the major error and lower 8 bit values tells more precisely.

Note

When called, this API initializes the timer, XADC subsystems. Unlocks the PS eFUSE controller. Configures the PS eFUSE controller. Writes the hash and control bits if requested. Programs the PS eFUSE to enable the RSA authentication if requested. Locks the PS eFUSE controller. Returns an error if: The reference clock frequency is not in between 20 and 60 MHz. The system not in a position to write the requested PS eFUSE bits (because the bits are already written or not allowed to write) The temperature and voltage are not within range

u32 XilSKey_EfusePs_Read (XilSKey_EP * InstancePtr)

This function is used to read the PS eFUSE.

Parameters

<i>InstancePtr</i>	Pointer to the PsEfuseHandle which describes which PS eFUSE should be burned.
--------------------	---

Returns

- XST_SUCCESS no errors occurred.
- In case of error, value is as defined in xilskey_utils.h. Error value is a combination of Upper 8 bit value and Lower 8 bit value. For example, 0x8A03 should be checked in error.h as 0x8A00 and 0x03. Upper 8 bit value signifies the major error and lower 8 bit values tells more precisely.

Note

When called: This API initializes the timer, XADC subsystems. Unlocks the PS eFUSE Controller. Configures the PS eFUSE Controller and enables read-only mode. Reads the PS eFUSE (Hash Value), and enables read-only mode. Locks the PS eFUSE Controller. Returns error if: The reference clock frequency is not in between 20 and 60MHz. Unable to unlock PS eFUSE controller or requested address corresponds to restricted bits. Temperature and voltage are not within range

u32 XilSKey_EfusePs_ReadStatus (XilSKey_EP * InstancePtr, u32 * StatusBits)

This function is used to read the PS efuse status register.

Parameters

<i>InstancePtr</i>	Pointer to the PS eFUSE instance.
<i>StatusBits</i>	Buffer to store the status register read.

Returns

- XST_SUCCESS.
- XST_FAILURE

Note

This API unlocks the controller and reads the Zynq PS eFUSE status register.

Zynq UltraScale+ MPSoC eFUSE PS API

Overview

This chapter provides a linked summary and detailed descriptions of the Zynq MPSoC UltraScale+ eFUSE PS APIs.

Example Usage

- For programming eFUSE other than PUF, the Zynq UltraScale+ MPSoC example application should contain the `xilskey_efuseps_zynqmp_example.c` and the `xilskey_efuseps_zynqmp_input.h` files.
- For PUF registration and programming PUF, the the Zynq UltraScale+ MPSoC example application should contain the `xilskey_puf_registration.c` and the `xilskey_puf_registration.h` files.
- For more details on the user configurable parameters, refer [Zynq UltraScale+ MPSoC User-Configurable PS eFUSE Parameters](#) and [Zynq UltraScale+ MPSoC User-Configurable PS PUF Parameters](#).

Functions

- `u32 XilSKey_ZynqMp_EfusePs_CheckAesKeyCrc (u32 CrcValue)`
- `u32 XilSKey_ZynqMp_EfusePs_ReadUserFuse (u32 *UseFusePtr, u8 UserFuse_Num, u8 ReadOption)`
- `u32 XilSKey_ZynqMp_EfusePs_ReadPpk0Hash (u32 *Ppk0Hash, u8 ReadOption)`
- `u32 XilSKey_ZynqMp_EfusePs_ReadPpk1Hash (u32 *Ppk1Hash, u8 ReadOption)`
- `u32 XilSKey_ZynqMp_EfusePs_ReadSpkId (u32 *SpkId, u8 ReadOption)`
- `void XilSKey_ZynqMp_EfusePs_ReadDna (u32 *DnaRead)`
- `u32 XilSKey_ZynqMp_EfusePs_ReadSecCtrlBits (XilSKey_SecCtrlBits *ReadBackSecCtrlBits, u8 ReadOption)`
- `u32 XilSKey_ZynqMp_EfusePs_Write (XilSKey_ZynqMpEPs *InstancePtr)`
- `u32 XilSKey_ZynqMp_EfusePs_WritePufHelperData (XilSKey_Puf *InstancePtr)`
- `u32 XilSKey_ZynqMp_EfusePs_ReadPufHelperData (u32 *Address)`
- `u32 XilSKey_ZynqMp_EfusePs_WritePufChash (XilSKey_Puf *InstancePtr)`
- `u32 XilSKey_ZynqMp_EfusePs_ReadPufChash (u32 *Address, u8 ReadOption)`
- `u32 XilSKey_ZynqMp_EfusePs_WritePufAux (XilSKey_Puf *InstancePtr)`

- u32 [XilSKey_ZynqMp_EfusePs_ReadPufAux](#) (u32 *Address, u8 ReadOption)
- u32 [XilSKey_Write_Puf_EfusePs_SecureBits](#) (XilSKey_Puf_Secure *WriteSecureBits)
- u32 [XilSKey_Read_Puf_EfusePs_SecureBits](#) (XilSKey_Puf_Secure *SecureBitsRead, u8 ReadOption)
- u32 [XilSKey_Puf_Debug2](#) (XilSKey_Puf *InstancePtr)
- u32 [XilSKey_Puf_Registration](#) (XilSKey_Puf *InstancePtr)

Function Documentation

u32 XilSKey_ZynqMp_EfusePs_CheckAesKeyCrc (u32 CrcValue)

This function performs CRC check of AES key.

Parameters

<i>CrcValue</i>	32 bit CRC of expected AES key.
-----------------	---------------------------------

Returns

XST_SUCCESS - On success ErrorCode - on Failure

Note

For Calculating CRC of AES key use [XilSKey_CrcCalculation\(\)](#) API or [XilSkey_CrcCalculation_AesKey\(\)](#) API.

u32 XilSKey_ZynqMp_EfusePs_ReadUserFuse (u32 *UseFusePtr, u8 UserFuse_Num, u8 ReadOption)

This function is used to read user fuse from efuse or cache based on user's read option.

Parameters

<i>UseFusePtr</i>	Pointer to an array which holds the readback user fuse in.
<i>UserFuse_Num</i>	A variable which holds the user fuse number. Range is (User fuses: 0 to 7)
<i>ReadOption</i>	<p>A variable which has to be provided by user based on this input reading is happened from cache or from efuse array.</p> <ul style="list-style-type: none"> • 0 Reads from cache • 1 Reads from efuse array

Returns

XST_SUCCESS - On success ErrorCode - on Failure

u32 XilSKey_ZynqMp_EfusePs_ReadPpk0Hash (u32 * Ppk0Hash, u8 ReadOption)

This function is used to read PPK0 hash from efuse or cache based on user's read option.

Parameters

<i>Ppk0Hash</i>	A pointer to an array which holds the readback PPK0 hash in.
<i>ReadOption</i>	<p>A variable which has to be provided by user based on this input reading is happened from cache or from efuse array.</p> <ul style="list-style-type: none"> • 0 Reads from cache • 1 Reads from efuse array

Returns

XST_SUCCESS - On success ErrorCode - on Failure

Note

None.

u32 XilSKey_ZynqMp_EfusePs_ReadPpk1Hash (u32 * Ppk1Hash, u8 ReadOption)

This function is used to read PPK1 hash from efuse or cache based on user's read option.

Parameters

<i>Ppk1Hash</i>	Pointer to an array which holds the readback PPK1 hash in.
<i>ReadOption</i>	<p>A variable which has to be provided by user based on this input reading is happened from cache or from efuse array.</p> <ul style="list-style-type: none"> • 0 Reads from cache • 1 Reads from efuse array

Returns

XST_SUCCESS - On success ErrorCode - on Failure

u32 XilSKey_ZynqMp_EfusePs_ReadSpkId (u32 * SpkId, u8 ReadOption)

This function is used to read SPKID from efuse or cache based on user's read option.

Parameters

<i>SpkId</i>	Pointer to a 32 bit variable which holds SPK ID.
<i>ReadOption</i>	<p>A variable which has to be provided by user based on this input reading is happened from cache or from efuse array.</p> <ul style="list-style-type: none"> • 0 Reads from cache • 1 Reads from efuse array

Returns

XST_SUCCESS - On success ErrorCode - on Failure

void XilSKey_ZynqMp_EfusePs_ReadDna (u32 * DnaRead)

This function is used to read DNA from efuse.

Parameters

<i>DnaRead</i>	Pointer to 32 bit variable which holds the readback DNA in.
----------------	---

Returns

None.

u32 XilSKey_ZynqMp_EfusePs_ReadSecCtrlBits (XilSKey_SecCtrlBits * ReadBackSecCtrlBits, u8 ReadOption)

This function is used to read the PS efuse secure control bits from cache or eFUSE based on user input provided.

Parameters

<i>ReadBackSecCtrlBits</i>	Pointer to the XilSKey_SecCtrlBits which holds the read secure control bits.
<i>ReadOption</i>	Variable which has to be provided by user based on this input reading is happened from cache or from efuse array. <ul style="list-style-type: none"> • 0 Reads from cache • 1 Reads from efuse array

Returns

- XST_SUCCESS if reads successfully
- XST_FAILURE if reading is failed

Note

Cache reload is required for obtaining updated values for ReadOption 0.

u32 XilSKey_ZynqMp_EfusePs_Write (XilSKey_ZynqMpEPs * InstancePtr)

This function is used to program the PS efuse of ZynqMP, based on user inputs.

Parameters

<i>InstancePtr</i>	Pointer to the XilSKey_ZynqMpEPs.
--------------------	-----------------------------------

Returns

- XST_SUCCESS if programs successfully.
- Errorcode on failure

Note

Cache reload is required for obtaining updated values through cache, to reload cache use XilSKey_ZynqMp_EfusePs_CacheLoad() API.

u32 XilSKey_ZynqMp_EfusePs_WritePufHelprData (XilSKey_Puf * InstancePtr)

This function programs the PS efuse's with puf helper data of ZynqMp.

Parameters

<i>InstancePtr</i>	Pointer to the XilSKey_Puf instance.
--------------------	--------------------------------------

Returns

- XST_SUCCESS if programs successfully.
- Errorcode on failure

Note

To generate PufSyndromeData please use XilSKey_Puf_Registration API

u32 XilSKey_ZynqMp_EfusePs_ReadPufHelperData (u32 * Address)

This function reads the puf helper data from eFUSE.

Parameters

<i>Address</i>	Pointer to data array which holds the Puf helper data read from ZynqMp efuse.
----------------	---

Returns

- XST_SUCCESS if reads successfully.
- Errorcode on failure.

Note

This function only reads from eFUSE non-volatile memory. There is no option to read from Cache.

u32 XilSKey_ZynqMp_EfusePs_WritePufChash (XilSKey_Puf * InstancePtr)

This API programs eFUSE with CHash value.

Parameters

<i>InstancePtr</i>	Pointer to the XilSKey_Puf instance.
--------------------	--------------------------------------

Returns

- XST_SUCCESS if hash is programmed successfully.
- Errorcode on failure

Note

To generate CHash value please use XilSKey_Puf_Registration API

u32 XilSKey_ZynqMp_EfusePs_ReadPufChash (u32 *Address, u8 ReadOption)

This API reads efuse puf CHash Data from efuse array or cache based on the user read option.

Parameters

<i>Address</i>	Pointer which holds the read back value of chash
<i>ReadOption</i>	<p>A u8 variable which has to be provided by user based on this input reading is happened from cache or from efuse array.</p> <ul style="list-style-type: none"> • 0(XSK_EFUSEPS_READ_FROM_CACHE)Reads from cache • 1(XSK_EFUSEPS_READ_FROM_EFUSE)Reads from efuse array

Returns

- XST_SUCCESS if programs successfully.
- Errorcode on failure

Note

Cache reload is required for obtaining updated values for ReadOption 0.

u32 XilSKey_ZynqMp_EfusePs_WritePufAux (XilSKey_Puf *InstancePtr)

This API programs efuse puf Auxilary Data.

Parameters

<i>InstancePtr</i>	Pointer to the XilSKey_Puf instance.
--------------------	--------------------------------------

Returns

- XST_SUCCESS if programs successfully.
- Errorcode on failure

Note

To generate Auxilary data please use the below API u32 [XilSKey_Puf_Registration\(XilSKey_Puf *InstancePtr\)](#)

u32 XilSKey_ZynqMp_EfusePs_ReadPufAux (u32 * Address, u8 ReadOption)

This API reads efuse puf Auxiliary Data from efuse array or cache based on user read option.

Parameters

<i>Address</i>	Pointer which holds the read back value of Auxiliary
<i>ReadOption</i>	<p>A u8 variable which has to be provided by user based on this input reading is happened from cache or from efuse array.</p> <ul style="list-style-type: none"> • 0(XSK_EFUSEPS_READ_FROM_CACHE)Reads from cache • 1(XSK_EFUSEPS_READ_FROM_EFUSE)Reads from efuse array

Returns

- XST_SUCCESS if programs successfully.
- Errorcode on failure

Note

Cache reload is required for obtaining updated values for ReadOption 0.

u32 XilSKey_Write_Puf_EfusePs_SecureBits (XilSKey_Puf_Secure * WriteSecureBits)

This function programs the eFUSE PUF secure bits.

Parameters

<i>WriteSecureBits</i>	Pointer to the XilSKey_Puf_Secure structure
------------------------	---

Returns

XST_SUCCESS - On success ErrorCode - on Failure

u32 XilSKey_Read_Puf_EfusePs_SecureBits (XilSKey_Puf_Secure * SecureBitsRead, u8 ReadOption)

This function is used to read the PS efuse PUF secure bits from cache or from eFUSE array based on user selection.

Parameters

<i>SecureBits</i>	Pointer to the XilSKey_Puf_Secure which holds the read eFUSE secure bits of PUF.
<i>ReadOption</i>	A u8 variable which has to be provided by user based on this input reading is happened from cache or from efuse array. <ul style="list-style-type: none"> • 0(XSK_EFUSEPS_READ_FROM_CACHE) Reads from cache • 1(XSK_EFUSEPS_READ_FROM_EFUSE) Reads from efuse array

Returns

XST_SUCCESS - On success ErrorCode - on Failure

u32 XilSKey_Puf_Debug2 (XilSKey_Puf * *InstancePtr*)

PUF Debug 2 operation.

Parameters

<i>InstancePtr</i>	Pointer to the XilSKey_Puf instance.
--------------------	--------------------------------------

Returns

- XST_SUCCESS if debug 2 mode was successful.
- ERROR if registration was unsuccessful.

Note

Updates the Debug 2 mode result @ InstancePtr->Debug2Data

u32 XilSKey_Puf_Registration (XilSKey_Puf * *InstancePtr*)

PUF Registration/Re-registration.

Parameters

<i>InstancePtr</i>	Pointer to the XilSKey_Puf instance.
--------------------	--------------------------------------

Returns

- XST_SUCCESS if registration/re-registration was successful.
- ERROR if registration was unsuccessful

Note

Updates the syndrome data @ InstancePtr->SyndromeData

eFUSE PL API

Overview

This chapter provides a linked summary and detailed descriptions of the eFUSE APIs of Zynq eFUSE PL and UltraScale eFUSE.

Example Usage

- The Zynq eFUSE PL and UltraScale example application should contain the `xilskey_efuse_example.c` and the `xilskey_input.h` files.
- By default, both the eFUSE PS and PL are enabled in the application. You can comment '`XSK_EFUSEPL_DRIVER`' to execute only the PS.
- For UltraScale, it is mandatory to comment '`XSK_EFUSEPS_DRIVER`' else the example will generate an error.
- For more details on the user configurable parameters, refer [Zynq User-Configurable PL eFUSE Parameters](#) and [UltraScale User-Configurable PL eFUSE Parameters](#).
- Requires hardware setup to program PL eFUSE of Zynq or UltraScale.

Functions

- `u32 XilSKey_EfusePI_Program (XilSKey_EPI *PInstancePtr)`
- `u32 XilSKey_EfusePI_ReadStatus (XilSKey_EPI *InstancePtr, u32 *StatusBits)`
- `u32 XilSKey_EfusePI_ReadKey (XilSKey_EPI *InstancePtr)`
- `u32 XilSKey_CrcCalculation (u8 *Key)`

Function Documentation

`u32 XilSKey_EfusePI_Program (XilSKey_EPI * InstancePtr)`

Programs PL eFUSE with input data given through `InstancePtr`.

When called, this API: Initializes the timer, XADC/xsysmon and JTAG server subsystems. Writes the AES & User Keys if requested. In UltraScale, if requested it also programs the RSA Hash. Writes the Control Bits if requested. Returns an error if: In Zynq the reference clock frequency is not in between 20 and 60 MHz. The PL DAP ID is not identified. The system is not in a position to write the requested PL eFUSE bits (because the bits are already written or not allowed to write) Temperature and voltage are not within range.

Parameters

<i>InstancePtr</i>	Pointer to PL eFUSE instance which holds the input data to be written to PL eFUSE.
--------------------	--

Returns

- XST_FAILURE - In case of failure
- XST_SUCCESS - In case of Success

Note

In Zynq Updates the global variable ErrorCode with error code(if any).

u32 XilSKey_EfusePI_ReadStatus (XilSKey_EPI * *InstancePtr*, u32 * *StatusBits*)

Reads the PL efuse status bits and gets all secure and control bits.

Parameters

<i>InstancePtr</i>	Pointer to PL eFUSE instance.
<i>StatusBits</i>	Buffer to store the status bits read.

Returns

- XST_FAILURE - In case of failure
- XST_SUCCESS - In case of Success

Note

In Zynq Updates the global variable ErrorCode with error code(if any).

u32 XilSKey_EfusePI_ReadKey (*XilSKey_EPI * InstancePtr*)

Reads the PL efuse keys and stores them in the corresponding arrays in instance structure, it initializes the timer, XADC and JTAG server subsystems, if not already done so.

In Zynq - Reads AES key and User keys In Ultrascale - 32 bit and 128 bit User keys and RSA hash But in Ultrascale AES key cannot be read directly it can be verified with CRC check, for that we need to update the instance with 32 bit CRC value, API updates whether provided CRC value is matched with actuals or not. To calculate the CRC of expected AES key one can use any of the following APIs [XilSKey_CrcCalculation\(\)](#) or [XilSKey_CrcCalculation_AesKey\(\)](#)

Parameters

<i>InstancePtr</i>	Pointer to PL eFUSE instance.
--------------------	-------------------------------

Returns

- XST_FAILURE - In case of failure
- XST_SUCCESS - In case of Success

Note

In Zynq updates the global variable ErrorCode with error code(if any).

u32 XilSKey_CrcCalculation (*u8 * Key*)

Calculates CRC value of provided key, this API expects key in string format.

Parameters

<i>Key</i>	is the string contains AES key in hexa decimal of length less than or equal to 64.
------------	--

Returns

On Success returns the Crc of AES key value. On failure returns the error code

- when string length is greater than 64

Note

This API calculates CRC of AES key for Ultrascale Microblaze's PL eFuse and ZynqMp UltraScale PS eFuse. If length of the string provided is lesser than 64, API appends the string with zeros.

CRC Calculation API

Overview

This chapter provides a linked summary and detailed descriptions of the CRC calculation APIs. For UltraScale and Zynq UltraScale+ MPSoC devices, programmed AES cannot be read back. The programmed AES key can only be confirmed by doing CRC check of AES key.

Functions

- u32 [XilSKey_CrcCalculation \(u8 *Key\)](#)
- u32 [XilSkey_CrcCalculation_AesKey \(u8 *Key\)](#)

Function Documentation

u32 XilSKey_CrcCalculation (u8 * Key)

Calculates CRC value of provided key, this API expects key in string format.

Parameters

Key	is the string contains AES key in hexa decimal of length less than or equal to 64.
-----	--

Returns

On Success returns the Crc of AES key value. On failure returns the error code

- when string length is greater than 64

Note

This API calculates CRC of AES key for Ultrascale Microblaze's PL eFuse and ZynqMp UltraScale PS eFuse. If length of the string provided is lesser than 64, API appends the string with zeros.

u32 XilSkey_CrcCalculation_AesKey (u8 * Key)

Calculates CRC value of the provided key.

Key should be provided in hexa buffer.

Parameters

Key	Pointer to an array of size 32 which contains AES key in hexa decimal.
-----	--

Returns

Crc of provided AES key value.

Note

This API calculates CRC of AES key for Ultrascale Microblaze's PL eFuse and ZynqMp Ultrascale's PS eFuse. This API calculates CRC on AES key provided in hexa format. To calculate CRC on the AES key in string format please use XilSKey_CrcCalculation. To call this API one can directly pass array of AES key which exists in an instance. Example for storing key into Buffer: If Key is "123456" buffer should be {0x12 0x34 0x56}

User-Configurable Parameters

Overview

This chapter provides detailed descriptions of the various user configurable parameters.

Modules

- Zynq User-Configurable PS eFUSE Parameters
- Zynq User-Configurable PL eFUSE Parameters
- Zynq User-Configurable PL BBRAM Parameters
- UltraScale User-Configurable BBRAM PL Parameters
- UltraScale User-Configurable PL eFUSE Parameters
- Zynq UltraScale+ MPSoC User-Configurable PS eFUSE Parameters
- Zynq UltraScale+ MPSoC User-Configurable PS BBRAM Parameters
- Zynq UltraScale+ MPSoC User-Configurable PS PUF Parameters

Zynq User-Configurable PS eFUSE Parameters

Define the XSK_EFUSEPS_DRIVER macro to use the PS eFUSE.

After defining the macro, provide the inputs defined with XSK_EFUSEPS_DRIVER to burn the bits in PS eFUSE. If the bit is to be burned, define the macro as TRUE; otherwise define the macro as FALSE. For details, refer the following table.

Macro Name	Description
XSK_EFUSEPS_ENABLE_WRITE_PROTECT	<p>Default = FALSE.</p> <p>TRUE to burn the write-protect bits in eFUSE array. Write protect has two bits. When either of the bits is burned, it is considered write-protected. So, while burning the write-protected bits, even if one bit is blown, write API returns success. As previously mentioned, POR reset is required after burning for write protection of the eFUSE bits to go into effect. It is recommended to do the POR reset after write protection. Also note that, after write-protect bits are burned, no more eFUSE writes are possible. If the write-protect macro is TRUE with other macros, write protect is burned in the last iteration, after burning all the defined values, so that for any error while burning other macros will not effect the total eFUSE array.</p> <p>FALSE does not modify the write-protect bits.</p>
XSK_EFUSEPS_ENABLE_RSA_AUTH	<p>Default = FALSE.</p> <p>Use TRUE to burn the RSA enable bit in the PS eFUSE array. After enabling the bit, every successive boot must be RSA-enabled apart from JTAG. Before burning (blowing) this bit, make sure that eFUSE array has the valid PPK hash. If the PPK hash burning is enabled, only after writing the hash successfully, RSA enable bit will be blown. For the RSA enable bit to take effect, POR reset is required.</p> <p>FALSE does not modify the RSA enable bit.</p>
XSK_EFUSEPS_ENABLE_ROM_128K_CRC	<p>Default = FALSE.</p> <p>TRUE burns the ROM 128K CRC bit. In every successive boot, BootROM calculates 128k CRC.</p> <p>FALSE does not modify the ROM CRC 128K bit.</p>
XSK_EFUSEPS_ENABLE_RSA_KEY_HASH	<p>Default = FALSE.</p> <p>TRUE burns (blows) the eFUSE hash, that is given in XSK_EFUSEPS_RSA_KEY_HASH_VALUE when write API is used. TRUE reads the eFUSE hash when the read API is used and is read into structure.</p> <p>FALSE ignores the provided value.</p>

Macro Name	Description
XSK_EFUSEPS_RSA_KEY_HASH_VALUE	Default = 00000000000000000000000000000000 00000000000000000000000000000000 The specified value is converted to a hexadecimal buffer and written into the PS eFUSE array when the write API is used. This value should be the Primary Public Key (PPK) hash provided in string format. The buffer must be 64 characters long: valid characters are 0-9, a-f, and A-F. Any other character is considered an invalid string and will not burn RSA hash. When the Xilskey_EfusePs_Write() API is used, the RSA hash is written, and the XSK_EFUSEPS_ENABLE_RSA_KEY_HASH must have a value of TRUE.
XSK_EFUSEPS_DISABLE_DFT_JTAG	Default = FALSE TRUE disables DFT JTAG permanently. FALSE will not modify the eFuse PS DFT JTAG disable bit.
XSK_EFUSEPS_DISABLE_DFT_MODE	Default = FALSE TRUE disables DFT mode permanently. FALSE will not modify the eFuse PS DFT mode disable bit.

Zynq User-Configurable PL eFUSE Parameters

Overview

Define the XSK_EFUSEPL_DRIVER macro to use the PL eFUSE.

After defining the macro, provide the inputs defined with XSK_EFUSEPL_DRIVER to burn the bits in PL eFUSE bits. If the bit is to be burned, define the macro as TRUE; otherwise define the macro as FALSE. The table below lists the user-configurable PL eFUSE parameters for Zynq® devices.

Macro Name	Description
XSK_EFUSEPL_FORCE_PCYCLE_RECONFIG	Default = FALSE If the value is set to TRUE, then the part has to be power-cycled to be reconfigured. FALSE does not set the eFUSE control bit.
XSK_EFUSEPL_DISABLE_KEY_WRITE	Default = FALSE TRUE disables the eFUSE write to FUSE_AES and FUSE_USER blocks. FALSE does not affect the EFUSE bit.

Macro Name	Description
XSK_EFUSEPL_DISABLE_AES_KEY_READ	Default = FALSE TRUE disables the write to FUSE_AES and FUSE_USER key and disables the read of FUSE_AES. FALSE does not affect the eFUSE bit.
XSK_EFUSEPL_DISABLE_USER_KEY_READ	Default = FALSE. TRUE disables the write to FUSE_AES and FUSE_USER key and disables the read of FUSE_USER. FALSE does not affect the eFUSE bit.
XSK_EFUSEPL_DISABLE_FUSE_CNTRL_WRITE	Default = FALSE. TRUE disables the eFUSE write to FUSE_CTRL block. FALSE does not affect the eFUSE bit.
XSK_EFUSEPL_FORCE_USE_AES_ONLY	Default = FALSE. TRUE forces the use of secure boot with eFUSE AES key only. FALSE does not affect the eFUSE bit.
XSK_EFUSEPL_DISABLE_JTAG_CHAIN	Default = FALSE. TRUE permanently disables the Zynq ARM DAP and PL TAP. FALSE does not affect the eFUSE bit.
XSK_EFUSEPL_BBRAM_KEY_DISABLE	Default = FALSE. TRUE forces the eFUSE key to be used if booting Secure Image. FALSE does not affect the eFUSE bit.

Modules

- MIO Pins for Zynq PL eFUSE JTAG Operations
- MUX Selection Pin for Zynq PL eFUSE JTAG Operations
- MUX Parameter for Zynq PL eFUSE JTAG Operations
- AES and User Key Parameters

MIO Pins for Zynq PL eFUSE JTAG Operations

The table below lists the MIO pins for Zynq PL eFUSE JTAG operations.
You can change the listed pins at your discretion.

Note

The pin numbers listed in the table below are examples. You must assign appropriate pin numbers as per your hardware design.

Pin Name	Pin Number
XSK_EFUSEPL_MIO_JTAG_TDI	(17)
XSK_EFUSEPL_MIO_JTAG_TDO	(18)
XSK_EFUSEPL_MIO_JTAG_TCK	(19)
XSK_EFUSEPL_MIO_JTAG_TMS	(20)

MUX Selection Pin for Zynq PL eFUSE JTAG Operations

The table below lists the MUX selection pin.

Pin Name	Pin Number	Description
XSK_EFUSEPL_MIO_JTAG_MUX_SELECT	(21)	This pin toggles between the external JTAG or MIO driving JTAG operations.

MUX Parameter for Zynq PL eFUSE JTAG Operations

The table below lists the MUX parameter.

Parameter Name	Description
XSK_EFUSEPL_MIO_MUX_SEL_DEFAULT_VAL	Default = LOW. LOW writes zero on the MUX select line before PL_eFUSE writing. HIGH writes one on the MUX select line before PL_eFUSE writing.

AES and User Key Parameters

The table below lists the AES and user key parameters.

Parameter Name	Description
XSK_EFUSEPL_PROGRAM_AES_AND_USER_LOW_KEY	<p>Default = FALSE.</p> <p>TRUE burns the AES and User Low hash key, which are given in the XSK_EFUSEPL_AES_KEY and the XSK_EFUSEPL_USER_LOW_KEY respectively.</p> <p>FALSE ignores the provided values.</p> <p>You cannot write the AES Key and the User Low Key separately.</p>
XSK_EFUSEPL_PROGRAM_USER_HIGH_KEY	<p>Default = FALSE.</p> <p>TRUE burns the User High hash key, given in XSK_EFUSEPL_PROGRAM_USER_HIGH_KEY.</p> <p>FALSE ignores the provided values.</p>
XSK_EFUSEPL_AES_KEY	<p>Default = 00000000000000000000000000000000 00000000000000000000000000000000</p> <p>This value converted to hex buffer and written into the PL eFUSE array when write API is used. This value should be the AES Key, given in string format. It must be 64 characters long. Valid characters are 0-9, a-f, A-F. Any other character is considered an invalid string and will not burn AES Key.</p> <p>To write AES Key, XSK_EFUSEPL_PROGRAM_AES_AND_USER_LOW_KEY must have a value of TRUE.</p>
XSK_EFUSEPL_USER_LOW_KEY	<p>Default = 00</p> <p>This value is converted to a hexadecimal buffer and written into the PL eFUSE array when the write API is used. This value is the User Low Key given in string format. It must be two characters long; valid characters are 0-9,a-f, and A-F. Any other character is considered as an invalid string and will not burn the User Low Key.</p> <p>To write the User Low Key, XSK_EFUSEPL_PROGRAM_AES_AND_USER_LOW_KEY must have a value of TRUE.</p>

Parameter Name	Description
XSK_EFUSEPL_USER_HIGH_KEY	<p>Default = 000000</p> <p>The default value is converted to a hexadecimal buffer and written into the PL eFUSE array when the write API is used. This value is the User High Key given in string format. The buffer must be six characters long: valid characters are 0-9, a-f, A-F. Any other character is considered to be an invalid string and does not burn User High Key.</p> <p>To write the User High Key, the XSK_EFUSEPL_PROGRAM_USER_HIGH_KEY must have a value of TRUE.</p>

Zynq User-Configurable PL BBRAM Parameters

Overview

The table below lists the MIO pins for Zynq PL BBRAM JTAG operations.

Note

The pin numbers listed in the table below are examples. You must assign appropriate pin numbers as per your hardware design.

Pin Name	Pin Number
XSK_BBRAM_MIO_JTAG_TDI	(17)
XSK_BBRAM_MIO_JTAG_TDO	(21)
XSK_BBRAM_MIO_JTAG_TCK	(19)
XSK_BBRAM_MIO_JTAG_TMS	(20)

The table below lists the MUX selection pin for Zynq BBRAM PL JTAG operations.

Pin Name	Pin Number
XSK_BBRAM_MIO_JTAG_MUX_SELECT	(11)

Modules

- MUX Parameter for Zynq BBRAM PL JTAG Operations
- AES and User Key Parameters

MUX Parameter for Zynq BBRAM PL JTAG Operations

The table below lists the MUX parameter for Zynq BBRAM PL JTAG operations.

Parameter Name	Description
XSK_BBRAM_MIO_MUX_SEL_DEFAULT_VAL	Default = LOW. LOW writes zero on the MUX select line before PL_eFUSE writing. HIGH writes one on the MUX select line before PL_eFUSE writing.

AES and User Key Parameters

The table below lists the AES and user key parameters.

Parameter Name	Description
XSK_BBRAM_AES_KEY	Default = XX. AES key (in HEX) that must be programmed into BBRAM.
XSK_BBRAM_AES_KEY_SIZE_IN_BITS	Default = 256. Size of AES key. Must be 256 bits.

UltraScale User-Configurable BBRAM PL Parameters

Overview

Following parameters need to be configured.

Based on your inputs, BBRAM is programmed with the provided AES key.

Modules

- [AES Keys and Related Parameters](#)
- [DPA Protection for BBRAM key](#)
- [GPIO Pins Used for PL Master JTAG and HWM Signals](#)
- [GPIO Channels](#)

AES Keys and Related Parameters

The following table shows AES key related parameters.

Parameter Name	Description
XSK_BBRAM_PGM_OBFUSCATED_KEY	<p>Default = FALSE By default XSK_BBRAM_PGM_OBFUSCATED_KEY is FALSE, BBRAM is programmed with a non-obfuscated key provided in XSK_BBRAM_AES_KEY and DPA protection can be either in enabled/disabled state. TRUE programs the BBRAM with key provided in XSK_BBRAM_OBFUSCATED_KEY and DPA protection cannot be enabled.</p>
XSK_BBRAM_OBFUSCATED_KEY	<p>Default = b1c276899d71fb4cdd4a0a7905ea46c2e11f9574d09c7ea23b70b67de713cccd1 The value mentioned in this will be converted to hex buffer and the key is programmed into BBRAM, when program API is called. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not program BBRAM.</p> <p>Note</p> <p>For writing the OBFUSCATED Key, XSK_BBRAM_PGM_OBFUSCATED_KEY should have TRUE value.</p>
XSK_BBRAM_AES_KEY	<p>Default = 0000000000000000524156a63950bcdedafeadcddeabaadee34216615aaaabbaaa The value mentioned in this will be converted to hex buffer and the key is programmed into BBRAM, when program API is called. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not program BBRAM.</p> <p>Note</p> <p>For programming BBRAM with the key, XSK_BBRAM_PGM_OBFUSCATED_KEY should have a FALSE value.</p>
XSK_BBRAM_AES_KEY_SIZE_IN_BITS	<p>Default= 256 Size of AES key must be 256 bits.</p>

DPA Protection for BBRAM key

The following table shows DPA protection configurable parameter.

Parameter Name	Description
XSK_BBRAM_DPA_PROTECT_ENABLE	<p>Default = FALSE By default, the DPA protection will be in disabled state. TRUE will enable DPA protection with provided DPA count and configuration in XSK_BBRAM_DPA_COUNT and XSK_BBRAM_DPA_MODE respectively. DPA protection cannot be enabled if BBRAM is been programmed with an obfuscated key.</p>
XSK_BBRAM_DPA_COUNT	<p>Default = 0 This input is valid only when DPA protection is enabled. Valid range of values are 1 - 255 when DPA protection is enabled else 0.</p>
XSK_BBRAM_DPA_MODE	<p>Default = XSK_BBRAM_INVALID_CONFIGURATIONS When DPA protection is enabled it can be XSK_BBRAM_INVALID_CONFIGURATIONS or XSK_BBRAM_ALL_CONFIGURATIONS If DPA protection is disabled this input provided over here is ignored.</p>

GPIO Pins Used for PL Master JTAG and HWM Signals

In Ultrascale the following GPIO pins are used for connecting MASTER_JTAG pins to access BBRAM. These can be changed depending on your hardware. The table below shows the GPIO pins used for PL MASTER JTAG signals.

Master JTAG Signal	Default PIN Number
XSK_BBRAM_AXI_GPIO_JTAG_TDO	0
XSK_BBRAM_AXI_GPIO_JTAG_TDI	0
XSK_BBRAM_AXI_GPIO_JTAG_TMS	1
XSK_BBRAM_AXI_GPIO_JTAG_TCK	2

GPIO Channels

The following table shows GPIO channel number.

Parameter	Default Channel Number	Master JTAG Signal Connected
XSK_BBRAM_GPIO_INPUT_CH	2	TDO
XSK_BBRAM_GPIO_OUTPUT_CH	1	TDI, TMS, TCK

Note

All inputs and outputs of GPIO can be configured in single channel. For example, XSK_BBRAM_GPIO_INPUT_CH = XSK_BBRAM_GPIO_OUTPUT_CH = 1 or 2. Among (TDI, TCK, TMS) Outputs of GPIO cannot be connected to different GPIO channels all the 3 signals should be in same channel. TDO can be a other channel of (TDI, TCK, TMS) or the same. DPA protection can be enabled only when programming non-obfuscated key.

UltraScale User-Configurable PL eFUSE Parameters

Overview

The table below lists the user-configurable PL eFUSE parameters for UltraScale™ devices.

Macro Name	Description
XSK_EFUSEPL_DISABLE_AES_KEY_READ	Default = FALSE TRUE will permanently disable the write to FUSE_AES and check CRC for AES key by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_USER_KEY_READ	Default = FALSE TRUE will permanently disable the write to 32 bit FUSE_USER and read of FUSE_USER key by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_SECURE_READ	Default = FALSE TRUE will permanently disable the write to FUSE_Secure block and reading of secure block by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_FUSE_CNTRL_WRITE	Default = FALSE. TRUE will permanently disable the write to FUSE_CNTRL block by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.

Macro Name	Description
XSK_EFUSEPL_DISABLE_RSA_KEY_READ	Default = FALSE. TRUE will permanently disable the write to FUSE_RSA block and reading of FUSE_RSA Hash by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_KEY_WRITE	Default = FALSE. TRUE will permanently disable the write to FUSE_AES block by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_USER_KEY_WRITE	Default = FALSE. TRUE will permanently disable the write to FUSE_USER block by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_SECURE_WRITE	Default = FALSE. TRUE will permanently disable the write to FUSE_SECURE block by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_RSA_HASH_WRITE	Default = FALSE. TRUE will permanently disable the write to FUSE_RSA authentication key by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_DISABLE_128BIT_USER_KEY_WRITE	Default = FALSE. TRUE will permanently disable the write to 128 bit FUSE_USER by programming control bit of FUSE. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPL_ALLOW_ENCRYPTED_ONLY	Default = FALSE. TRUE will permanently allow encrypted bitstream only. FALSE will not modify this Secure bit of eFuse.
XSK_EFUSEPL_FORCE_USE_FUSE_AES_ONLY	Default = FALSE. TRUE then allows only FUSE's AES key as source of encryption FALSE then allows FPGA to configure an unencrypted bitstream or bitstream encrypted using key stored BBRAM or eFuse.
XSK_EFUSEPL_ENABLE_RSA_AUTH	Default = FALSE. TRUE will enable RSA authentication of bitstream FALSE will not modify this secure bit of eFuse.

Macro Name	Description
XSK_EFUSEPL_DISABLE_JTAG_CHAIN	Default = FALSE. TRUE will disable JTAG permanently. FALSE will not modify this secure bit of eFuse.
XSK_EFUSEPL_DISABLE_TEST_ACCESS	Default = FALSE. TRUE will disables Xilinx test access. FALSE will not modify this secure bit of eFuse.
XSK_EFUSEPL_DISABLE_AES_DECRYPTOR	Default = FALSE. TRUE will disables decoder completely. FALSE will not modify this secure bit of eFuse.

Modules

- GPIO Pins Used for PL Master JTAG Signal
- GPIO Channels
- AES Keys and Related Parameters

GPIO Pins Used for PL Master JTAG Signal

In Ultrascale the following GPIO pins are used for connecting MASTER_JTAG pins to access BBRAM. These can be changed depending on your hardware. The table below shows the GPIO pins used for PL MASTER JTAG signals.

Master JTAG Signal	Default PIN Number
XSK_EFUSEPL_AXI_GPIO_JTAG_TDO	0
XSK_EFUSEPL_AXI_GPIO_HWM_READY	0
XSK_EFUSEPL_AXI_GPIO_HWM_END	1
XSK_EFUSEPL_AXI_GPIO_JTAG_TDI	2
XSK_EFUSEPL_AXI_GPIO_JTAG_TMS	1
XSK_EFUSEPL_AXI_GPIO_JTAG_TCK	2
XSK_EFUSEPL_AXI_GPIO_HWM_START	3

GPIO Channels

The following table shows GPIO channel number.

Parameter	Default Channel Number	Master JTAG Signal Connected
XSK_EFUSEPL_GPIO_INPUT_CH	2	TDO
XSK_EFUSEPL_GPIO_OUTPUT_CH	1	TDI, TMS, TCK

Note

All inputs and outputs of GPIO can be configured in single channel. For example, XSK_BBRAM_GPIO_INPUT_CH = XSK_BBRAM_GPIO_OUTPUT_CH = 1 or 2. Among (TDI, TCK, TMS) Outputs of GPIO cannot be connected to different GPIO channels all the 3 signals should be in same channel. TDO can be a other channel of (TDI, TCK, TMS) or the same. DPA protection can be enabled only when programming non-obfuscated key.

AES Keys and Related Parameters

The following table shows AES key related parameters.

Parameter Name	Description
XSK_EFUSEPL_PROGRAM_AES_KEY_ULTRA	Default = FALSE TRUE will burn the AES key given in XSK_EFUSEPL_AES_KEY. FALSE will ignore the values given.
XSK_EFUSEPL_PROGRAM_USER_KEY_ULTRA	Default = FALSE TRUE will burn 32 bit User key given in XSK_EFUSEPL_USER_KEY. FALSE will ignore the values given.
XSK_EFUSEPL_PROGRAM_RSA_HASH_ULTRA	Default = FALSE TRUE will burn RSA hash given in XSK_EFUSEPL_RSA_KEY_HASH_VALUE. FALSE will ignore the values given.
XSK_EFUSEPL_PROGRAM_USER_KEY_128BIT	Default = FALSE TRUE will burn 128 bit User key given in XSK_EFUSEPL_USER_KEY_128BIT_0, XSK_EFUSEPL_USER_KEY_128BIT_1, XSK_EFUSEPL_USER_KEY_128BIT_2, XSK_EFUSEPL_USER_KEY_128BIT_3 FALSE will ignore the values given.

Parameter Name	Description
XSK_EFUSEPL_CHECK_AES_KEY_ULTRA	Default = FALSE TRUE will perform CRC check of FUSE_AES with provided CRC value in macro XSK_EFUSEPL_CRC_OF_EXPECTED_AES_KEY. And result of CRC check will be updated in XilSKey_EPI instance parameter AESKeyMatched with either TRUE or FALSE. FALSE CRC check of FUSE_AES will not be performed.
XSK_EFUSEPL_READ_USER_KEY_ULTRA	Default = FALSE TRUE will read 32 bit FUSE_USER from UltraScale eFUSE and updates in XilSKey_EPI instance parameter UserKeyReadback. FALSE 32-bit FUSE_USER key read will not be performed.
XSK_EFUSEPL_READ_RSA_HASH_ULTRA	Default = FALSE TRUE will read FUSE_USER from UltraScale eFUSE and updates in XilSKey_EPI instance parameter RSAHashReadback. FALSE FUSE_RSA_HASH read will not be performed.
XSK_EFUSEPL_READ_USER_KEY128_BIT	Default = FALSE TRUE will read 128 bit USER key from UltraScale eFUSE and updates in XilSKey_EPI instance parameter User128BitReadBack. FALSE 128 bit USER key read will not be performed.
XSK_EFUSEPL_AES_KEY	Default = 00000000000000000000000000000000 00000000000000000000000000000000 The value mentioned in this will be converted to hex buffer and written into the PL eFUSE array when write API used. This value should be the PPK(Primary Public Key) hash given in string format. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn AES Key. Note that for writing the AES Key, XSK_EFUSEPL_PROGRAM_AES_KEY_ULTRA should have TRUE value.

Parameter Name	Description
XSK_EFUSEPL_USER_KEY	<p>Default = 00000000</p> <p>The value mentioned in this will be converted to hex buffer and written into the PL eFUSE array when write API used. This value should be the User Key given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn User Key. Note that, for writing the User Key, XSK_EFUSEPL_PROGRAM_USER_KEY_ULTRA should have TRUE value.</p>
XSK_EFUSEPL_RSA_KEY_HASH_VALUE	<p>Default = 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000</p> <p>The value mentioned in this will be converted to hex buffer and written into the PL eFUSE array when write API used. This value should be the RSA Key hash given in string format. It should be 96 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn RSA hash value. Note that, for writing the RSA hash, XSK_EFUSEPL_PROGRAM_RSA_HASH_ULTRA should have TRUE value.</p>
XSK_EFUSEPL_CRC_OF_EXPECTED_AES_KEY	<p>Default = 0x621C42AA</p> <p>0x621C42AA is hexadecimal CRC value of FUSE_AES with all Zeros. Expected FUSE_AES key's CRC value has to be updated in place of 0x621C42AA. For Checking CRC of FUSE_AES XSK_EFUSEPL_CHECK_AES_KEY_ULTRA macro should be TRUE otherwise CRC check will not be performed. For calculation of AES key's CRC one can use u32 XilSKey_CrcCalculation(u8 *Key) API or reverse polynomial 0x82F63B78.</p>

Zynq UltraScale+ MPSoC User-Configurable PS eFUSE Parameters

Overview

The table below lists the user-configurable PS eFUSE parameters for Zynq UltraScale+ MPSoC devices.

Macro Name	Description
XSK_EFUSEPS_AES_RD_LOCK	Default = FALSE TRUE will permanently disable the CRC check of FUSE_AES. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_AES_WR_LOCK	Default = FALSE TRUE will permanently disable the writing to FUSE_AES block. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_ENC_ONLY	Default = FALSE TRUE will permanently enable encrypted booting only using the Fuse key. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_BBRAM_DISABLE	Default = FALSE TRUE will permanently disable the BBRAM key. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_ERR_DISABLE	Default = FALSE TRUE will permanently disables the error messages in JTAG status register. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_JTAG_DISABLE	Default = FALSE TRUE will permanently disable JTAG controller. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_DFT_DISABLE	Default = FALSE TRUE will permanently disable DFT boot mode. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_PROG_GATE_DISABLE	Default = FALSE TRUE will permanently disable PROG_GATE feature in PPD. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_SECURE_LOCK	Default = FALSE TRUE will permanently disable reboot into JTAG mode when doing a secure lockdown. FALSE will not modify thi s control bit of eFuse.

Macro Name	Description
XSK_EFUSEPS_RSA_ENABLE	Default = FALSE TRUE will permanently enable RSA authentication during boot. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_PPK0_WR_LOCK	Default = FALSE TRUE will permanently disable writing to PPK0 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_PPK0_INVLD	Default = FALSE TRUE will permanently revoke PPK0. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_PPK1_WR_LOCK	Default = FALSE TRUE will permanently disable writing PPK1 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_PPK1_INVLD	Default = FALSE TRUE will permanently revoke PPK1. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_0	Default = FALSE TRUE will permanently disable writing to USER_0 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_1	Default = FALSE TRUE will permanently disable writing to USER_1 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_2	Default = FALSE TRUE will permanently disable writing to USER_2 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_3	Default = FALSE TRUE will permanently disable writing to USER_3 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_4	Default = FALSE TRUE will permanently disable writing to USER_4 efuses. FALSE will not modify this control bit of eFuse.

Macro Name	Description
XSK_EFUSEPS_USER_WRLK_5	Default = FALSE TRUE will permanently disable writing to USER_5 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_6	Default = FALSE TRUE will permanently disable writing to USER_6 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_USER_WRLK_7	Default = FALSE TRUE will permanently disable writing to USER_7 efuses. FALSE will not modify this control bit of eFuse.
XSK_EFUSEPS_LBIST_EN	Default = FALSE TRUE will permanently enables logic BIST to be run during boot. FALSE will not modify this control bit of eFUSE.
XSK_EFUSEPS_LPD_SC_EN	Default = FALSE TRUE will permanently enables zeroization of registers in Low Power Domain(LPD) during boot. FALSE will not modify this control bit of eFUSE.
XSK_EFUSEPS_FPD_SC_EN	Default = FALSE TRUE will permanently enables zeroization of registers in Full Power Domain(FPD) during boot. FALSE will not modify this control bit of eFUSE.
XSK_EFUSEPS_PBR_BOOT_ERR	Default = FALSE TRUE will permanently enables the boot halt when there is any PMU error. FALSE will not modify this control bit of eFUSE.

Modules

- AES Keys and Related Parameters
- User Keys and Related Parameters
- PPK0 Keys and Related Parameters
- PPK1 Keys and Related Parameters
- SPK ID and Related Parameters

AES Keys and Related Parameters

The following table shows AES key related parameters.

Parameter Name	Description
XSK_EFUSEPS_WRITE_AES_KEY	Default = TRUE TRUE will burn the AES key provided in XSK_EFUSEPS_AES_KEY. FALSE will ignore the key provide XSK_EFUSEPS_AES_KEY.
XSK_EFUSEPS_AES_KEY	Default = 00000000000000000000000000000000 00000000000000000000000000000000 The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn AES Key.

User Keys and Related Parameters

Single bit programming is allowed for all the USER fuses.

If user requests to revert already programmed bit. Library throws an error. If user fuses is non-zero also library will not throw an error for valid requests The following table shows the user keys and related parameters.

Parameter Name	Description
XSK_EFUSEPS_WRITE_USER0_FUSE	Default = TRUE TRUE will burn User0 Fuse provided in XSK_EFUSEPS_USER0_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER0_FUSES
XSK_EFUSEPS_WRITE_USER1_FUSE	Default = TRUE TRUE will burn User1 Fuse provided in XSK_EFUSEPS_USER1_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER1_FUSES
XSK_EFUSEPS_WRITE_USER2_FUSE	Default = TRUE TRUE will burn User2 Fuse provided in XSK_EFUSEPS_USER2_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER2_FUSES

Parameter Name	Description
XSK_EFUSEPS_WRITE_USER3_FUSE	<p>Default = TRUE TRUE will burn User3 Fuse provided in XSK_EFUSEPS_USER3_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER3_FUSES</p>
XSK_EFUSEPS_WRITE_USER4_FUSE	<p>Default = TRUE TRUE will burn User4 Fuse provided in XSK_EFUSEPS_USER4_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER4_FUSES</p>
XSK_EFUSEPS_WRITE_USER5_FUSE	<p>Default = TRUE TRUE will burn User5 Fuse provided in XSK_EFUSEPS_USER5_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER5_FUSES</p>
XSK_EFUSEPS_WRITE_USER6_FUSE	<p>Default = TRUE TRUE will burn User6 Fuse provided in XSK_EFUSEPS_USER6_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER6_FUSES</p>
XSK_EFUSEPS_WRITE_USER7_FUSE	<p>Default = TRUE TRUE will burn User7 Fuse provided in XSK_EFUSEPS_USER7_FUSES. FALSE will ignore the value provided in XSK_EFUSEPS_USER7_FUSES</p>
XSK_EFUSEPS_USER0_FUSES	<p>Default = 00000000 The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID.</p> <p>Note</p> <p>For writing the User0 Fuse, XSK_EFUSEPS_WRITE_USER0_FUSE should have TRUE value</p>

Parameter Name	Description
XSK_EFUSEPS_USER1_FUSES	<p>Default = 00000000</p> <p>The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID.</p> <p>Note</p> <p>For writing the User1 Fuse, XSK_EFUSEPS_WRITE_USER1_FUSE should have TRUE value</p>
XSK_EFUSEPS_USER2_FUSES	<p>Default = 00000000</p> <p>The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID.</p> <p>Note</p> <p>For writing the User2 Fuse, XSK_EFUSEPS_WRITE_USER2_FUSE should have TRUE value</p>
XSK_EFUSEPS_USER3_FUSES	<p>Default = 00000000</p> <p>The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID.</p> <p>Note</p> <p>For writing the User3 Fuse, XSK_EFUSEPS_WRITE_USER3_FUSE should have TRUE value</p>

Parameter Name	Description
XSK_EFUSEPS_USER4_FUSES	<p>Default = 00000000</p> <p>The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID.</p> <p>Note</p> <p>For writing the User4 Fuse, XSK_EFUSEPS_WRITE_USER4_FUSE should have TRUE value</p>
XSK_EFUSEPS_USER5_FUSES	<p>Default = 00000000</p> <p>The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID.</p> <p>Note</p> <p>For writing the User5 Fuse, XSK_EFUSEPS_WRITE_USER5_FUSE should have TRUE value</p>
XSK_EFUSEPS_USER6_FUSES	<p>Default = 00000000</p> <p>The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID.</p> <p>Note</p> <p>For writing the User6 Fuse, XSK_EFUSEPS_WRITE_USER6_FUSE should have TRUE value</p>

Parameter Name	Description
XSK_EFUSEPS_USER7_FUSES	<p>Default = 00000000</p> <p>The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID.</p> <p>Note</p> <p>For writing the User7 Fuse, XSK_EFUSEPS_WRITE_USER7_FUSE should have TRUE value</p>

PPK0 Keys and Related Parameters

The following table shows the PPK0 keys and related parameters.

Parameter Name	Description
XSK_EFUSEPS_WRITE_PPK0_SHA3_HASH	<p>Default = TRUE</p> <p>TRUE will burn PPK0 sha3 hash provided in XSK_EFUSEPS_PPK0_SHA3_HASH. FALSE will ignore the hash provided in XSK_EFUSEPS_PPK0_SHA3_HASH.</p>
XSK_EFUSEPS_PPK0_IS_SHA3	<p>Default = TRUE</p> <p>TRUE XSK_EFUSEPS_PPK0_SHA3_HASH should be of string length 96 it specifies that PPK0 is used to program SHA3 hash. FALSE XSK_EFUSEPS_PPK0_SHA3_HASH should be of string length 64 it specifies that PPK0 is used to program SHA2 hash.</p>

Parameter Name	Description
XSK_EFUSEPS_PPK0_HASH	<p>Default = 00 00 00</p> <p>The value mentioned in this will be converted to hex buffer and into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 96 or 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn PPK0 hash. Note that, for writing the PPK0 hash, XSK_EFUSEPS_WRITE_PPK0_SHA3_HASH should have TRUE value. While writing SHA2 hash, length should be 64 characters long XSK_EFUSEPS_PPK0_IS_SHA3 macro has to be made FALSE. While writing SHA3 hash, length should be 96 characters long and XSK_EFUSEPS_PPK0_IS_SHA3 macro should be made TRUE</p>

PPK1 Keys and Related Parameters

The following table shows the PPK1 keys and related parameters.

Parameter Name	Description
XSK_EFUSEPS_WRITE_PPK1_SHA3_HASH	<p>Default = TRUE TRUE will burn PPK1 sha3 hash provided in XSK_EFUSEPS_PPK1_SHA3_HASH. FALSE will ignore the hash provided in XSK_EFUSEPS_PPK1_SHA3_HASH.</p>
XSK_EFUSEPS_PPK1_IS_SHA3	<p>Default = FALSE TRUE XSK_EFUSEPS_PPK1_SHA3_HASH should be of string length 96 it specifies that PPK1 is used to program SHA3 hash. FALSE XSK_EFUSEPS_PPK1_SHA3_HASH should be of string length 64 it specifies that PPK1 is used to program SHA2 hash.</p>

SPK ID and Related Parameters

The following table shows the SPK ID and related parameters.

Parameter Name	Description
XSK_EFUSEPS_WRITE_SPKID	Default = TRUE TRUE will burn SPKID provided in XSK_EFUSEPS_SPK_ID. FALSE will ignore the hash provided in XSK_EFUSEPS_SPK_ID.

Parameter Name	Description
XSK_EFUSEPS_SPK_ID	<p>Default = 00000000 The value mentioned in this will be converted to hex buffer and written into the Zynq UltraScale+ MPSoC PS eFUSE array when write API used. This value should be given in string format. It should be 8 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn SPK ID.</p> <p>Note</p> <p>For writing the SPK ID, XSK_EFUSEPS_WRITE_SPKID should have TRUE value.</p>

Note

PPK hash should be unmodified hash generated by bootgen. Single bit programming is allowed for User FUSES (0 to 7), if you specify a value that tries to set a bit that was previously programmed to 1 back to 0, you will get an error. you have to provide already programmed bits also along with new requests.

Zynq UltraScale+ MPSoC User-Configurable PS BBRAM Parameters

The table below lists the AES and user key parameters.

Parameter Name	Description
XSK_ZYNQMP_BBRAMPS_AES_KEY	Default = 00000000000000000000000000000000 AES key (in HEX) that must be programmed into BBRAM.
XSK_ZYNQMP_BBRAMPS_AES_KEY_LEN_IN_BYT	Default = 32. Length of AES key in bytes.
XSK_ZYNQMP_BBRAMPS_AES_KEY_LEN_IN_BITS	Default = 256. Length of AES key in bits.
XSK_ZYNQMP_BBRAMPS_AES_KEY_STR_LEN	Default = 64. String length of the AES key.

Zynq UltraScale+ MPSoC User-Configurable PS PUF Parameters

The table below lists the user-configurable PS PUF parameters for Zynq UltraScale+ MPSoC devices.

Macro Name	Description
XSK_PUF_INFO_ON_UART	Default = FALSE TRUE will display syndrome data on UART com port FALSE will display any data on UART com port.
XSK_PUF_PROGRAM_EFUSE	Default = FALSE TRUE will program the generated syndrome data, CHash and Auxilary values, Black key. FALSE will not program data into eFUSE.
XSK_PUF_IF_CONTRACT_MANUFACTURER	Default = FALSE This should be enabled when application is hand over to contract manufacturer. TRUE will allow only authenticated application. FALSE authentication is not mandatory.
XSK_PUF_REG_MODE	Default = XSK_PUF_MODE4K PUF registration is performed in 4K mode. For only understanding it is provided in this file, but user is not supposed to modify this.

Macro Name	Description
XSK_PUF_READ_SECUREBITS	Default = FALSE TRUE will read status of the puf secure bits from eFUSE and will be displayed on UART. FALSE will not read secure bits.
XSK_PUF_PROGRAM_SECUREBITS	Default = FALSE TRUE will program PUF secure bits based on the user input provided at XSK_PUF_SYN_INVALID, XSK_PUF_SYN_WRLK and XSK_PUF_REGISTER_DISABLE. FALSE will not program any PUF secure bits.
XSK_PUF_SYN_INVALID	Default = FALSE TRUE will permanently invalidate the already programmed syndrome data. FALSE will not modify anything
XSK_PUF_SYN_WRLK	Default = FALSE TRUE will permanently disable programming syndrome data into eFUSE. FALSE will not modify anything.
XSK_PUF_REGISTER_DISABLE	Default = FALSE TRUE permanently does not allow PUF syndrome data registration. FALSE will not modify anything.
XSK_PUF_RESERVED	Default = FALSE TRUE programs this reserved eFUSE bit. FALSE will not modify anything.
XSK_PUF_AES_KEY	Default = 00000000000000000000000000000000 00000000000000000000000000000000 The value mentioned in this will be converted to hex buffer and encrypts this with PUF helper data and generates a black key and written into the Zynq UltraScale+ MPSoC PS eFUSE array when XSK_PUF_PROGRAM_EFUSE macro is TRUE. This value should be given in string format. It should be 64 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string and will not burn AES Key. Note Provided here should be red key and application calculates the black key and programs into eFUSE if XSK_PUF_PROGRAM_EFUSE macro is TRUE. To avoid programming eFUSE results can be displayed on UART com port by making XSK_PUF_INFO_ON_UART to TRUE.

Macro Name	Description
XSK_PUF_IV	<p>Default = 000000000000000000000000000000</p> <p>The value mentioned here will be converted to hex buffer. This is Initialization vector(IV) which is used to generated black key with provided AES key and generated PUF key.</p> <p>This value should be given in string format. It should be 24 characters long, valid characters are 0-9,a-f,A-F. Any other character is considered as invalid string.</p>

Error Codes

Overview

The application error code is 32 bits long.

For example, if the error code for PS is 0x8A05:

- 0x8A indicates that a write error has occurred while writing RSA Authentication bit.
- 0x05 indicates that write error is due to the write temperature out of range.

Applications have the following options on how to show error status. Both of these methods of conveying the status are implemented by default. However, UART is required to be present and initialized for status to be displayed through UART.

- Send the error code through UART pins
- Write the error code in the reboot status register

Modules

- PL eFUSE Error Codes
- PS eFUSE Error Codes
- Zynq UltraScale+ MPSoC BBRAM PS Error Codes

PL eFUSE Error Codes

XSK_EFUSEPL_ERROR_NONE 0

No error.

XSK_EFUSEPL_ERROR_ROW_NOT_ZERO 0x10

Row is not zero.

XSK_EFUSEPL_ERROR_READ_ROW_OUT_OF_RANGE 0x11

Read Row is out of range.

XSK_EFUSEPL_ERROR_READ_MARGIN_OUT_OF_RANGE 0x12

Read Margin is out of range.

XSK_EFUSEPL_ERROR_READ_BUFFER_NULL 0x13

No buffer for read.

XSK_EFUSEPL_ERROR_READ_BIT_VALUE_NOT_SET 0x14
 Read bit not set.

XSK_EFUSEPL_ERROR_READ_BIT_OUT_OF_RANGE <0x15 br>Read bit is out of range.

XSK_EFUSEPL_ERROR_READ_TEMPERATURE_OUT_OF_RANGE 0x16
 Temperature obtained from XADC is out of range to read.

XSK_EFUSEPL_ERROR_READ_VCCAUX_VOLTAGE_OUT_OF_RANGE 0x17
 VCCAUX obtained from XADC is out of range to read.

XSK_EFUSEPL_ERROR_READ_VCCINT_VOLTAGE_OUT_OF_RANGE PL
 VCCINT obtained from XADC is out of range to read.

XSK_EFUSEPL_ERROR_WRITE_ROW_OUT_OF_RANGE 0x19
 To write row is out of range.

XSK_EFUSEPL_ERROR_WRITE_BIT_OUT_OF_RANGE 0x1A
 To read bit is out of range.

XSK_EFUSEPL_ERROR_WRITE_TEMPERATURE_OUT_OF_RANGE 0x1B
 To eFUSE write Temperature obtained from XADC is out of range.

XSK_EFUSEPL_ERROR_WRITE_VCCAUX_VOLTAGE_OUT_OF_RANGE 0x1C
 To write eFUSE VCCAUX obtained from XADC is out of range.

XSK_EFUSEPL_ERROR_WRITE_VCCINT_VOLTAGE_OUT_OF_RANGE 0x1D
 To write into eFUSE VCCINT obtained from XADC is out of range.

XSK_EFUSEPL_ERROR_FUSE_CNTRL_WRITE_DISABLED 0x1E
 Fuse control write is disabled.

XSK_EFUSEPL_ERROR_CNTRL_WRITE_BUFFER_NULL 0x1F
 Buffer pointer that is supposed to contain control data is null.

XSK_EFUSEPL_ERROR_NOT_VALID_KEY_LENGTH 0x20
 Key length invalid.

XSK_EFUSEPL_ERROR_ZERO_KEY_LENGTH 0x21
 Key length zero.

XSK_EFUSEPL_ERROR_NOT_VALID_KEY_CHAR 0x22
 Invalid key characters.

XSK_EFUSEPL_ERROR_NULL_KEY 0x23
 Null key.

XSK_EFUSEPL_ERROR_FUSE_SEC_WRITE_DISABLED 0x24
 Secure bits write is disabled.

XSK_EFUSEPL_ERROR_FUSE_SEC_READ_DISABLED 0x25
 Secure bits reading is disabled.

XSK_EFUSEPL_ERROR_SEC_WRITE_BUFFER_NULL 0x26
 Buffer to write into secure block is NULL.

XSK_EFUSEPL_ERROR_READ_PAGE_OUT_OF_RANGE 0x27
 Page is out of range.

XSK_EFUSEPL_ERROR_FUSE_ROW_RANGE 0x28
 Row is out of range.

- XSK_EFUSEPL_ERROR_IN_PROGRAMMING_ROW** 0x29
 Error programming fuse row.
- XSK_EFUSEPL_ERROR_PRGRMG_ROWS_NOT_EMPTY** 0x2A
 Error when tried to program non Zero rows of eFUSE.
- XSK_EFUSEPL_ERROR_HWM_TIMEOUT** 0x80
 Error when hardware module is exceeded the time for programming eFUSE.
- XSK_EFUSEPL_ERROR_USER_FUSE_REVERT** 0x90
 Error occurs when user requests to revert already programmed user eFUSE bit.
- XSK_EFUSEPL_ERROR_KEY_VALIDATION** 0xF000
 Invalid key.
- XSK_EFUSEPL_ERROR_PL_STRUCT_NULL** 0x1000
 Null PL structure.
- XSK_EFUSEPL_ERROR_JTAG_SERVER_INIT** 0x1100
 JTAG server initialization error.
- XSK_EFUSEPL_ERROR_READING_FUSE_CNTRL** 0x1200
 Error reading fuse control.
- XSK_EFUSEPL_ERROR_DATA_PROGRAMMING_NOT_ALLOWED** 0x1300
 Data programming not allowed.
- XSK_EFUSEPL_ERROR_FUSE_CTRL_WRITE_NOT_ALLOWED** 0x1400
 Fuse control write is disabled.
- XSK_EFUSEPL_ERROR_READING_FUSE_AES_ROW** 0x1500
 Error reading fuse AES row.
- XSK_EFUSEPL_ERROR_AES_ROW_NOT_EMPTY** 0x1600
 AES row is not empty.
- XSK_EFUSEPL_ERROR_PROGRAMMING_FUSE_AES_ROW** 0x1700
 Error programming fuse AES row.
- XSK_EFUSEPL_ERROR_READING_FUSE_USER_DATA_ROW** 0x1800
 Error reading fuse user row.
- XSK_EFUSEPL_ERROR_USER_DATA_ROW_NOT_EMPTY** 0x1900
 User row is not empty.
- XSK_EFUSEPL_ERROR_PROGRAMMING_FUSE_DATA_ROW** 0x1A00
 Error programming fuse user row.
- XSK_EFUSEPL_ERROR_PROGRAMMING_FUSE_CNTRL_ROW** 0x1B00
 Error programming fuse control row.
- XSK_EFUSEPL_ERROR_XADC** 0x1C00
 XADC error.
- XSK_EFUSEPL_ERROR_INVALID_REF_CLK** 0x3000
 Invalid reference clock.
- XSK_EFUSEPL_ERROR_FUSE_SEC_WRITE_NOT_ALLOWED** 0x1D00
 Error in programming secure block.
- XSK_EFUSEPL_ERROR_READING_FUSE_STATUS** 0x1E00
 Error in reading FUSE status.

XSK_EFUSEPL_ERROR_FUSE_BUSY 0x1F00
 Fuse busy.

XSK_EFUSEPL_ERROR_READING_FUSE_RSA_ROW 0x2000
 Error in reading FUSE RSA block.

XSK_EFUSEPL_ERROR_TIMER_INITIALISE_ULTRA 0x2200
 Error in initiating Timer.

XSK_EFUSEPL_ERROR_READING_FUSE_SEC 0x2300
 Error in reading FUSE secure bits.

XSK_EFUSEPL_ERROR_PRGRMG_FUSE_SEC_ROW 0x2500
 Error in programming Secure bits of efuse.

XSK_EFUSEPL_ERROR_PRGRMG_USER_KEY 0x4000
 Error in programming 32 bit user key.

XSK_EFUSEPL_ERROR_PRGRMG_128BIT_USER_KEY 0x5000
 Error in programming 128 bit User key.

XSK_EFUSEPL_ERROR_PRGRMG_RSA_HASH 0x8000
 Error in programming RSA hash.

PS eFUSE Error Codes

XSK_EFUSEPS_ERROR_NONE 0
 No error.

XSK_EFUSEPS_ERROR_ADDRESS_XIL_RESTRICTED 0x01
 Address is restricted.

XSK_EFUSEPS_ERROR_READ_TMEPERATURE_OUT_OF_RANGE 0x02
 Temperature obtained from XADC is out of range.

XSK_EFUSEPS_ERROR_READ_VCCPAUX_VOLTAGE_OUT_OF_RANGE 0x03
 VCCAUX obtained from XADC is out of range.

XSK_EFUSEPS_ERROR_READ_VCCPINT_VOLTAGE_OUT_OF_RANGE 0x04
 VCCINT obtained from XADC is out of range.

XSK_EFUSEPS_ERROR_WRITE_TEMPERATURE_OUT_OF_RANGE 0x05
 Temperature obtained from XADC is out of range.

XSK_EFUSEPS_ERROR_WRITE_VCCPAUX_VOLTAGE_OUT_OF_RANGE 0x06
 VCCAUX obtained from XADC is out of range.

XSK_EFUSEPS_ERROR_WRITE_VCCPINT_VOLTAGE_OUT_OF_RANGE 0x07
 VCCINT obtained from XADC is out of range.

XSK_EFUSEPS_ERROR_VERIFICATION 0x08
 Verification error.

XSK_EFUSEPS_ERROR_RSA_HASH_ALREADY_PROGRAMMED 0x09
 RSA hash was already programmed.

XSK_EFUSEPS_ERROR_CONTROLLER_MODE 0x0A
 Controller mode error

XSK_EFUSEPS_ERROR_REF_CLOCK 0x0B
 Reference clock not between 20 to 60 MHz

XSK_EFUSEPS_ERROR_READ_MODE 0x0C
 Not supported read mode

XSK_EFUSEPS_ERROR_XADC_CONFIG 0x0D
 XADC configuration error.

XSK_EFUSEPS_ERROR_XADC_INITIALIZE 0x0E
 XADC initialization error.

XSK_EFUSEPS_ERROR_XADC_SELF_TEST 0x0F
 XADC self-test failed.

XSK_EFUSEPS_ERROR_PARAMETER_NULL Utils Error Codes. 0x10
 Passed parameter null.

XSK_EFUSEPS_ERROR_STRING_INVALID 0x20
 Passed string is invalid.

XSK_EFUSEPS_ERROR_AES_ALREADY_PROGRAMMED 0x12
 AES key is already programmed.

XSK_EFUSEPS_ERROR_SPKID_ALREADY_PROGRAMMED 0x13
 SPK ID is already programmed.

XSK_EFUSEPS_ERROR_PPK0_HASH_ALREADY_PROGRAMMED 0x14
 PPK0 hash is already programmed.

XSK_EFUSEPS_ERROR_PPK1_HASH_ALREADY_PROGRAMMED 0x15
 PPK1 hash is already programmed.

XSK_EFUSEPS_ERROR_PROGRAMMING_TBIT_PATTERN 0x16
 Error in programming TBITS.

XSK_EFUSEPS_ERROR_BEFORE_PROGRAMMING 0x0080
 Error occurred before programming.

XSK_EFUSEPS_ERROR_PROGRAMMING 0x00A0
 Error in programming eFUSE.

XSK_EFUSEPS_ERROR_READ 0x00B0
 Error in reading.

XSK_EFUSEPS_ERROR_PS_STRUCT_NULL XSKEfuse_Write/Read()common error codes. 0x8100
 PS structure pointer is null.

XSK_EFUSEPS_ERROR_XADC_INIT 0x8200
 XADC initialization error.

XSK_EFUSEPS_ERROR_CONTROLLER_LOCK 0x8300
 PS eFUSE controller is locked.

XSK_EFUSEPS_ERROR_EFUSE_WRITE_PROTECTED 0x8400
 PS eFUSE is write protected .

XSK_EFUSEPS_ERROR_CONTROLLER_CONFIG 0x8500
 Controller configuration error.

XSK_EFUSEPS_ERROR_PS_PARAMETER_WRONG 0x8600
 PS eFUSE parameter is not TRUE/FALSE.

XSK_EFUSEPS_ERROR_WRITE_128K_CRC_BIT 0x9100

Error in enabling 128K CRC.

XSK_EFUSEPS_ERROR_WRITE_NONSECURE_INITB_BIT 0x9200

Error in programming NON secure bit.

XSK_EFUSEPS_ERROR_WRITE_UART_STATUS_BIT 0x9300

Error in writing UART status bit.

XSK_EFUSEPS_ERROR_WRITE_RSA_HASH 0x9400

Error in writing RSA key.

XSK_EFUSEPS_ERROR_WRITE_RSA_AUTH_BIT 0x9500

Error in enabling RSA authentication bit.

XSK_EFUSEPS_ERROR_WRITE_WRITE_PROTECT_BIT 0x9600

Error in writing write-protect bit.

XSK_EFUSEPS_ERROR_READ_HASH_BEFORE_PROGRAMMING 0x9700

Check RSA key before trying to program.

XSK_EFUSEPS_ERROR_WRTIE_DFT_JTAG_DIS_BIT 0x9800

Error in programming DFT JTAG disable bit.

XSK_EFUSEPS_ERROR_WRTIE_DFT_MODE_DIS_BIT 0x9900

Error in programming DFT MODE disable bit.

XSK_EFUSEPS_ERROR_WRTIE_AES_CRC_LK_BIT 0x9A00

Error in enabling AES's CRC check lock.

XSK_EFUSEPS_ERROR_WRTIE_AES_WR_LK_BIT 0x9B00

Error in programming AES write lock bit.

XSK_EFUSEPS_ERROR_WRTIE_USE_AESONLY_EN_BIT 0x9C00

Error in programming use AES only bit.

XSK_EFUSEPS_ERROR_WRTIE_BBRAM_DIS_BIT 0x9D00

Error in programming BBRAM disable bit.

XSK_EFUSEPS_ERROR_WRTIE_PMU_ERR_DIS_BIT 0x9E00

Error in programming PMU error disable bit.

XSK_EFUSEPS_ERROR_WRTIE_JTAG_DIS_BIT 0x9F00

Error in programming JTAG disable bit.

XSK_EFUSEPS_ERROR_READ_RSA_HASH 0xA100

Error in reading RSA key.

XSK_EFUSEPS_ERROR_WRONG_TBIT_PATTERN 0xA200

Error in programming TBIT pattern.

XSK_EFUSEPS_ERROR_WRITE_AES_KEY 0xA300

Error in programming AES key.

XSK_EFUSEPS_ERROR_WRITE_SPK_ID 0xA400

Error in programming SPK ID.

XSK_EFUSEPS_ERROR_WRITE_USER_KEY 0xA500

Error in programming USER key.

XSK_EFUSEPS_ERROR_WRITE_PPK0_HASH 0xA600

Error in programming PPK0 hash.

XSK_EFUSEPS_ERROR_WRITE_PPK1_HASH 0xA700
 Error in programming PPK1 hash.

XSK_EFUSEPS_ERROR_CACHE_LOAD 0xB000
 Error in re-loading CACHE.

XSK_EFUSEPS_ERROR_WRITE_USER0_FUSE 0xC000
 Error in programming USER 0 Fuses.

XSK_EFUSEPS_ERROR_WRITE_USER1_FUSE 0xC100
 Error in programming USER 1 Fuses.

XSK_EFUSEPS_ERROR_WRITE_USER2_FUSE 0xC200
 Error in programming USER 2 Fuses.

XSK_EFUSEPS_ERROR_WRITE_USER3_FUSE 0xC300
 Error in programming USER 3 Fuses.

XSK_EFUSEPS_ERROR_WRITE_USER4_FUSE 0xC400
 Error in programming USER 4 Fuses.

XSK_EFUSEPS_ERROR_WRITE_USER5_FUSE 0xC500
 Error in programming USER 5 Fuses.

XSK_EFUSEPS_ERROR_WRITE_USER6_FUSE 0xC600
 Error in programming USER 6 Fuses.

XSK_EFUSEPS_ERROR_WRITE_USER7_FUSE 0xC700
 Error in programming USER 7 Fuses.

XSK_EFUSEPS_ERROR_WRTIE_USER0_LK_BIT 0xC800
 Error in programming USER 0 fuses lock bit.

XSK_EFUSEPS_ERROR_WRTIE_USER1_LK_BIT 0xC900
 Error in programming USER 1 fuses lock bit.

XSK_EFUSEPS_ERROR_WRTIE_USER2_LK_BIT 0xCA00
 Error in programming USER 2 fuses lock bit.

XSK_EFUSEPS_ERROR_WRTIE_USER3_LK_BIT 0xCB00
 Error in programming USER 3 fuses lock bit.

XSK_EFUSEPS_ERROR_WRTIE_USER4_LK_BIT 0xCC00
 Error in programming USER 4 fuses lock bit.

XSK_EFUSEPS_ERROR_WRTIE_USER5_LK_BIT 0xCD00
 Error in programming USER 5 fuses lock bit.

XSK_EFUSEPS_ERROR_WRTIE_USER6_LK_BIT 0xCE00
 Error in programming USER 6 fuses lock bit.

XSK_EFUSEPS_ERROR_WRTIE_USER7_LK_BIT 0xCF00
 Error in programming USER 7 fuses lock bit.

XSK_EFUSEPS_ERROR_WRTIE_PROG_GATE0_DIS_BIT 0xD000
 Error in programming PROG_GATE0 disabling bit.

XSK_EFUSEPS_ERROR_WRTIE_PROG_GATE1_DIS_BIT 0xD100
 Error in programming PROG_GATE1 disabling bit.

XSK_EFUSEPS_ERROR_WRTIE_PROG_GATE2_DIS_BIT 0xD200
 Error in programming PROG_GATE2 disabling bit.

XSK_EFUSEPS_ERROR_WRTIE_SEC_LOCK_BIT 0xD300
 Error in programming SEC_LOCK bit.

XSK_EFUSEPS_ERROR_WRTIE_PPK0_WR_LK_BIT 0xD400
 Error in programming PPK0 write lock bit.

XSK_EFUSEPS_ERROR_WRTIE_PPK0_RVK_BIT 0xD500
 Error in programming PPK0 revoke bit.

XSK_EFUSEPS_ERROR_WRTIE_PPK1_WR_LK_BIT 0xD600
 Error in programming PPK1 write lock bit.

XSK_EFUSEPS_ERROR_WRTIE_PPK1_RVK_BIT 0xD700
 Error in programming PPK0 revoke bit.

XSK_EFUSEPS_ERROR_WRITE_PUF_SYN_INVLD 0xD800
 Error while programming the PUF syndrome invalidate bit.

XSK_EFUSEPS_ERROR_WRITE_PUF_SYN_WRLK 0xD900
 Error while programming Syndrome write lock bit.

XSK_EFUSEPS_ERROR_WRITE_PUF_SYN_REG_DIS 0xDA00
 Error while programming PUF syndrome register disable bit.

XSK_EFUSEPS_ERROR_PUF_INVALID_REG_MODE 0xE000
 Error when PUF registration is requested with invalid registration mode.

XSK_EFUSEPS_ERROR_PUF_REG_WO_AUTH 0xE100
 Error when write not allowed without authentication enabled.

XSK_EFUSEPS_ERROR_PUF_REG_DISABLED 0xE200
 Error when trying to do PUF registration and when PUF registration is disabled.

XSK_EFUSEPS_ERROR_PUF_INVALID_REQUEST 0xE300
 Error when an invalid mode is requested.

XSK_EFUSEPS_ERROR_PUF_DATA_ALREADY_PROGRAMMED 0xE400
 Error when PUF is already programmed in eFUSE.

XSK_EFUSEPS_ERROR_PUF_DATA_OVERFLOW 0xE500
 Error when an over flow occurs.

XSK_EFUSEPS_ERROR_CMPLTD_EFUSE_PRGRM_WITH_ERR 0x10000
 eFUSE programming is completed with temp and vol read errors.

XSK_EFUSEPS_ERROR_FUSE_PROTECTED 0x00080000
 Requested eFUSE is write protected.

XSK_EFUSEPS_ERROR_USER_BIT_CANT_REVERT 0x00800000
 Already programmed user FUSE bit cannot be reverted.

Zynq UltraScale+ MPSoC BBRAM PS Error Codes

XSK_ZYNQMP_BBRAMPS_ERROR_NONE 0
 No error.

XSK_ZYNQMP_BBRAMPS_ERROR_IN_PRGRMG_ENABLE 0x01
 If this error is occurred programming is not possible.

XSK_ZYNQMP_BBRAMPS_ERROR_IN_CRC_CHECK 0xB000

If this error is occurred programming is done but CRC check is failed.

XSK_ZYNQMP_BBRAMPS_ERROR_IN_PRGRMG 0xC000

programming of key is failed.

Status Codes

For Zynq® and UltraScale™, the status in the `xilskey_efuse_example.c` file is conveyed through a UART or reboot status register in the following format: `0xYYYYZZZZ`, where:

- YYYY represents the PS eFUSE Status.
- ZZZZ represents the PL eFUSE Status.

The table below lists the status codes.

Status Code Values	Description
0x0000ZZZZ	Represents PS eFUSE is successful and PL eFUSE process returned with error.
0xYYYY0000	Represents PL eFUSE is successful and PS eFUSE process returned with error.
0xFFFF0000	Represents PS eFUSE is not initiated and PL eFUSE is successful.
0x0000FFFF	Represents PL eFUSE is not initiated and PS eFUSE is successful.
0xFFFFZZZZ	Represents PS eFUSE is not initiated and PL eFUSE is process returned with error.
0xYYYYFFFF	Represents PL eFUSE is not initiated and PS eFUSE is process returned with error.

For Zynq UltraScale+ MPSoC, the status in the `xilskey_bbramps_zynqmp_example.c`, `xilskey_puf_registration.c` and `xilskey_efuseps_zynqmp_example.c` files is conveyed as 32 bit error code. Where Zero represents that no error has occurred and if the value is other than Zero, a 32 bit error code is returned.

Procedures

This chapter provides detailed descriptions of the various procedures.

Zynq eFUSE Writing Procedure Running from DDR as an Application

This sequence is same as the existing flow described below.

1. Provide the required inputs in `xilskey_input.h`, then compile the SDK project.
2. Take the latest FSBL (ELF), stitch the `<output>.elf` generated to it (using the bootgen utility), and generate a bootable image.
3. Write the generated binary image into the flash device (for example: QSPI, NAND).
4. To burn the eFUSE key bits, execute the image.

Zynq eFUSE Driver Compilation Procedure for OCM

The procedure is as follows:

1. Open the linker script (`lscript.ld`) in the SDK project.
2. Map all the chapters to point to `ps7_ram_0_S_AXI_BASEADDR` instead of `ps7_ddr_0_S_AXI_BASEADDR`. For example, Click the Memory Region tab for the `.text` chapter and select `ps7_ram_0_S_AXI_BASEADDR` from the drop-down list.
3. Copy the `ps7_init.c` and `ps7_init.h` files from the `hw_platform` folder into the example folder.
4. In `xilskey_efuse_example.c`, un-comment the code that calls the `ps7_init()` routine.
5. Compile the project.
The `<Project name>.elf` file is generated and is executed out of OCM.

When executed, this example displays the success/failure of the eFUSE application in a display message via UART (if UART is present and initialized) or the reboot status register.

UltraScale eFUSE Access Procedure

The procedure is as follows:

1. After providing the required inputs in `xilskey_input.h`, compile the project.
2. Generate a memory mapped interface file using TCL command `write_mem_info`

```
$Outfilename
```

3. Update memory has to be done using the tcl command `updatemem`.

```
updatemem -meminfo $file.mmi -data $Outfilename.elf -bit $design.bit  
-proc design_1_i/microblaze_0 -out $Final.bit
```

4. Program the board using `$Final.bit` bitstream.

5. Output can be seen in UART terminal.

UltraScale BBRAM Access Procedure

The procedure is as follows:

1. After providing the required inputs in the `xilskey_bbram_ultrascale_input.h` file, compile the project.
2. Generate a memory mapped interface file using TCL command

```
write_mem_info $Outfilename
```

3. Update memory has to be done using the tcl command `updatemem`:

```
updatemem -meminfo $file.mmi -data $Outfilename.elf -bit $design.bit  
-proc design_1_i/microblaze_0 -out $Final.bit
```

4. Program the board using `$Final.bit` bitstream.

5. Output can be seen in UART terminal.

Appendix I:

XilPM Library v2.1

XiPM APIs

Overview

Xilinx Power Management(XiPM) provides Embedded Energy Management Interface (EEMI) APIs for power management on Zynq® UltraScale+™ MPSoC. For more details about power management on Zynq UltraScale+ MPSoC, see the Zynq UltraScale+ MPSoC Power Management User Guide (UG1199). For more details about EEMI, see the Embedded Energy Management Interface (EEMI) API User Guide(UG1200).

Modules

- Error Status

Data Structures

- struct [XPm_Notifier](#)
- struct [XPm_NodeStatus](#)

Functions

- XStatus [XPm_InitXiPm](#) (XiPiPsu *IpilInst)
- void [XPm_SuspendFinalize](#) ()
- enum XPmBootStatus [XPm_GetBootStatus](#) ()
- XStatus [XPm_RequestSuspend](#) (const enum XPmNodeld node, const enum XPmRequestAck ack, const u32 latency, const u8 state)
- XStatus [XPm_SelfSuspend](#) (const enum XPmNodeld node, const u32 latency, const u8 state, const u64 address)
- XStatus [XPm_ForcePowerDown](#) (const enum XPmNodeld node, const enum XPmRequestAck ack)
- XStatus [XPm_AbortSuspend](#) (const enum XPmAbortReason reason)
- XStatus [XPm_RequestWakeUp](#) (const enum XPmNodeld node, const bool setAddress, const u64 address, const enum XPmRequestAck ack)
- XStatus [XPm_SetWakeUpSource](#) (const enum XPmNodeld target, const enum XPmNodeld wkup_node, const u8 enable)
- XStatus [XPm_SystemShutdown](#) (u32 type, u32 subtype)

- XStatus [XPm_SetConfiguration](#) (const u32 address)
- void [XPm_InitSuspendCb](#) (const enum XPmSuspendReason reason, const u32 latency, const u32 state, const u32 timeout)
- void [XPm_AcknowledgeCb](#) (const enum XPmNodeId node, const XStatus status, const u32 oppoint)
- void [XPm_NotifyCb](#) (const enum XPmNodeId node, const u32 event, const u32 oppoint)
- XStatus [XPm_RequestNode](#) (const enum XPmNodeId node, const u32 capabilities, const u32 qos, const enum XPmRequestAck ack)
- XStatus [XPm_ReleaseNode](#) (const enum XPmNodeId node)
- XStatus [XPm_SetRequirement](#) (const enum XPmNodeId node, const u32 capabilities, const u32 qos, const enum XPmRequestAck ack)
- XStatus [XPm_SetMaxLatency](#) (const enum XPmNodeId node, const u32 latency)
- XStatus [XPm_GetApiVersion](#) (u32 *version)
- XStatus [XPm_GetNodeStatus](#) (const enum XPmNodeId node, [XPm_NodeStatus](#) *const nodestatus)
- XStatus [XPm_RegisterNotifier](#) ([XPm_Notifier](#) *const notifier)
- XStatus [XPm_UnregisterNotifier](#) ([XPm_Notifier](#) *const notifier)
- XStatus [XPm_GetOpCharacteristic](#) (const enum XPmNodeId node, const enum XPmOpCharType type, u32 *const result)
- XStatus [XPm_ResetAssert](#) (const enum XPmReset reset, const enum XPmResetAction assert)
- XStatus [XPm_ResetGetStatus](#) (const enum XPmReset reset, u32 *status)
- XStatus [XPm_MmioWrite](#) (const u32 address, const u32 mask, const u32 value)
- XStatus [XPm_MmioRead](#) (const u32 address, u32 *const value)

Data Structure Documentation

struct [XPm_Notifier](#)

Notifier structure registered with a callback by application.

Data Fields

- void(*const [callback](#)) ([XPm_Notifier](#) *const notifier)
- enum XPmNodeId [node](#)
- enum XPmNotifyEvent [event](#)
- u32 [flags](#)
- volatile u32 [oppoint](#)
- volatile u32 [received](#)
- [XPm_Notifier](#) * [next](#)

Field Documentation

void(*const callback) (XPm_Notifier *const notifier) Custom callback handler to be called when the notification is received. The custom handler would execute from interrupt context, it shall return quickly and must not block! (enables event-driven notifications)

enum XPmNodeid node Node argument (the node to receive notifications about)

enum XPmNotifyEvent event Event argument (the event type to receive notifications about)

u32 flags Flags

volatile u32 oppoint Operating point of node in question. Contains the value updated when the last event notification is received. User shall not modify this value while the notifier is registered.

volatile u32 received How many times the notification has been received - to be used by application (enables polling). User shall not modify this value while the notifier is registered.

XPm_Notifier* next Pointer to next notifier in linked list. Must not be modified while the notifier is registered. User shall not ever modify this value.

struct XPm_NodeStatus

Data structure containing node status information.

Data Fields

- u32 **status**
- u32 **requirements**
- u32 **usage**

Field Documentation

u32 status Node power state

u32 requirements Current requirements asserted on the node (slaves only)

u32 usage Usage information (which master is currently using the slave)

Function Documentation

XStatus XPm_InitXilpm (*XipiPsu * IpiInst*)

Initialize xilpm library.

Parameters

<i>IpInst</i>	Pointer to IPI driver instance
---------------	--------------------------------

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

Note

None

void XPm_SuspendFinalize (void)

This Function waits for PMU to finish all previous API requests sent by the PU and performs client specific actions to finish suspend procedure (e.g. execution of wfi instruction on A53 and R5 processors).

Note

This function should not return if the suspend procedure is successful.

enum XPmBootStatus XPm_GetBootStatus (void)

This Function returns information about the boot reason. If the boot is not a system startup but a resume, power down request bitfield for this processor will be cleared.

Returns

Returns processor boot status

- PM_RESUME : If the boot reason is because of system resume.
- PM_INITIAL_BOOT : If this boot is the initial system startup.

Note

None

XStatus XPm_RequestSuspend (const enum XPmNodeID target, const enum XPmRequestAck ack, const u32 latency, const u8 state)

This function is used by a PU to request suspend of another PU. This call triggers the power management controller to notify the PU identified by 'nodeID' that a suspend has been requested. This will allow said PU to gracefully suspend itself by calling XPm_SelfSuspend for each of its CPU nodes, or else call XPm_AbortSuspend with its PU node as argument and specify the reason.

Parameters

<i>target</i>	Node ID of the PU node to be suspended
<i>ack</i>	Requested acknowledge type
<i>latency</i>	Maximum wake-up latency requirement in us(micro sec)
<i>state</i>	Instead of specifying a maximum latency, a PU can also explicitly request a certain power state.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

Note

If 'ack' is set to PM_ACK_CB_STANDARD, the requesting PU will be notified upon completion of suspend or if an error occurred, such as an abort or a timeout.

XStatus XPM_SelfSuspend (const enum XPMNodIdl nid, const u32 latency, const u8 state, const u64 address)

This function is used by a CPU to declare that it is about to suspend itself. After the PMU processes this call it will wait for the requesting CPU to complete the suspend procedure and become ready to be put into a sleep state.

Parameters

<i>nid</i>	Node ID of the CPU node to be suspended.
<i>latency</i>	Maximum wake-up latency requirement in us(microsecs)
<i>state</i>	Instead of specifying a maximum latency, a CPU can also explicitly request a certain power state.
<i>address</i>	Address from which to resume when woken up.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

Note

This is a blocking call, it will return only once PMU has responded

XStatus XPM_ForcePowerDown (const enum XPMNodIdl target, const enum XPMRequestAck ack)

One PU can request a forced poweroff of another PU or its power island or power domain. This can be used for killing an unresponsive PU, in which case all resources of that PU will be automatically released.

Parameters

<i>target</i>	Node ID of the PU node or power island/domain to be powered down.
<i>ack</i>	Requested acknowledge type

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

Note

Force power down may not be requested by a PU for itself.

XStatus XPM_AbortSuspend (const enum XPMAbortReason reason)

This function is called by a CPU after a XPM_SelfSuspend call to notify the power management controller that CPU has aborted suspend or in response to an init suspend request when the PU refuses to suspend.

Parameters

<i>reason</i>	Reason code why the suspend can not be performed or completed <ul style="list-style-type: none"> • ABORT_REASON_WKUP_EVENT : local wakeup-event received • ABORT_REASON_PU_BUSY : PU is busy • ABORT_REASON_NO_PWRDN : no external powerdown supported • ABORT_REASON_UNKNOWN : unknown error during suspend procedure
---------------	--

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

Note

Calling PU expects the PMU to abort the initiated suspend procedure. This is a non-blocking call without any acknowledge.

XStatus XPM_RequestWakeUp (const enum XPMNodeId target, const bool setAddress, const u64 address, const enum XPMRequestAck ack)

This function can be used to request power up of a CPU node within the same PU, or to power up another PU.

Parameters

<i>target</i>	Node ID of the CPU or PU to be powered/woken up.
<i>setAddress</i>	Specifies whether the start address argument is being passed. <ul style="list-style-type: none"> • 0 : do not set start address • 1 : set start address
<i>address</i>	Address from which to resume when woken up. Will only be used if <i>set_address</i> is 1.
<i>ack</i>	Requested acknowledge type

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

Note

If acknowledge is requested, the calling PU will be notified by the power management controller once the wake-up is completed.

XStatus XPM_SetWakeUpSource (const enum XPMNodId target, const enum XPMNodId wkup_node, const u8 enable)

This function is called by a PU to add or remove a wake-up source prior to going to suspend. The list of wake sources for a PU is automatically cleared whenever the PU is woken up or when one of its CPUs aborts the suspend procedure.

Parameters

<i>target</i>	Node ID of the target to be woken up.
<i>wkup_node</i>	Node ID of the wakeup device.
<i>enable</i>	Enable flag: <ul style="list-style-type: none"> • 1 : the wakeup source is added to the list • 0 : the wakeup source is removed from the list

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

Note

Declaring a node as a wakeup source will ensure that the node will not be powered off. It also will cause the PMU to configure the GIC Proxy accordingly if the FPD is powered off.

XStatus XPM_SystemShutdown (*u32 type, u32 subtype*)

This function can be used by a privileged PU to shut down or restart the complete device.

Parameters

<i>restart</i>	Should the system be restarted automatically? <ul style="list-style-type: none"> • PM_SHUTDOWN : no restart requested, system will be powered off permanently • PM_RESTART : restart is requested, system will go through a full reset
----------------	---

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

Note

In either case the PMU will call XPM_InitSuspendCb for each of the other PUs, allowing them to gracefully shut down. If a PU is asleep it will be woken up by the PMU. The PU making the XPM_SystemShutdown should perform its own suspend procedure after calling this API. It will not receive an init suspend callback.

XStatus XPM_SetConfiguration (*const u32 address*)

This function is called to configure the power management framework. The call triggers power management controller to load the configuration object and configure itself according to the content of the object.

Parameters

<i>address</i>	Start address of the configuration object
----------------	---

Returns

XST_SUCCESS if successful, otherwise an error code

Note

The provided address must be in 32-bit address space which is accessible by the PMU.

void XPm_InitSuspendCb (const enum XPmSuspendReason *reason*, const u32 *latency*, const u32 *state*, const u32 *timeout*)

Callback function to be implemented in each PU, allowing the power management controller to request that the PU suspend itself.

Parameters

<i>reason</i>	Suspend reason: <ul style="list-style-type: none"> • SUSPEND_REASON_PU_REQ : Request by another PU • SUSPEND_REASON_ALERT : Unrecoverable SysMon alert • SUSPEND_REASON_SHUTDOWN : System shutdown • SUSPEND_REASON_RESTART : System restart
<i>latency</i>	Maximum wake-up latency in us(micro secs). This information can be used by the PU to decide what level of context saving may be required.
<i>state</i>	Targeted sleep/suspend state.
<i>timeout</i>	Timeout in ms, specifying how much time a PU has to initiate its suspend procedure before it's being considered unresponsive.

Returns

None

Note

If the PU fails to act on this request the power management controller or the requesting PU may choose to employ the forceful power down option.

void XPm_AcknowledgeCb (const enum XPmNodeId *node*, const XStatus *status*, const u32 *oppoint*)

This function is called by the power management controller in response to any request where an acknowledge callback was requested, i.e. where the 'ack' argument passed by the PU was REQUEST_ACK_CB_STANDARD.

Parameters

<i>node</i>	ID of the component or sub-system in question.
<i>status</i>	Status of the operation: <ul style="list-style-type: none"> • OK: the operation completed successfully • ERR: the requested operation failed
<i>oppoint</i>	Operating point of the node in question

Returns

None

Note

None

void XPM_NotifyCb (const enum XPMNodId *node*, const u32 *event*, const u32 *oppoint*)

This function is called by the power management controller if an event the PU was registered for has occurred. It will populate the notifier data structure passed when calling XPM_RegisterNotifier.

Parameters

<i>node</i>	ID of the node the event notification is related to.
<i>event</i>	ID of the event
<i>oppoint</i>	Current operating state of the node.

Returns

None

Note

None

XStatus XPM_RequestNode (const enum XPMNodId *node*, const u32 *capabilities*, const u32 *qos*, const enum XPMRequestAck *ack*)

Used to request the usage of a PM-slave. Using this API call a PU requests access to a slave device and asserts its requirements on that device. Provided the PU is sufficiently privileged, the PMU will enable access

to the memory mapped region containing the control registers of that device. For devices that can only be serving a single PU, any other privileged PU will now be blocked from accessing this device until the node is released.

Parameters

<i>node</i>	Node ID of the PM slave requested
<i>capabilities</i>	Slave-specific capabilities required, can be combined <ul style="list-style-type: none"> • PM_CAP_ACCESS : full access / functionality • PM_CAP_CONTEXT : preserve context • PM_CAP_WAKEUP : emit wake interrupts
<i>qos</i>	Quality of Service (0-100) required
<i>ack</i>	Requested acknowledge type

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

Note

None

XStatus XPM_ReleaseNode (const enum XPMNodId *node*)

This function is used by a PU to release the usage of a PM slave. This will tell the power management controller that the node is no longer needed by that PU, potentially allowing the node to be placed into an inactive state.

Parameters

<i>node</i>	Node ID of the PM slave.
-------------	--------------------------

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

Note

None

XStatus XPM_SetRequirement (const enum XPMNodeID nid, const u32 capabilities, const u32 qos, const enum XPMRequestAck ack)

This function is used by a PU to announce a change in requirements for a specific slave node which is currently in use.

Parameters

<i>nid</i>	Node ID of the PM slave.
<i>capabilities</i>	Slave-specific capabilities required.
<i>qos</i>	Quality of Service (0-100) required.
<i>ack</i>	Requested acknowledge type

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

Note

If this function is called after the last awake CPU within the PU calls SelfSuspend, the requirement change shall be performed after the CPU signals the end of suspend to the power management controller, (e.g. WFI interrupt).

XStatus XPM_SetMaxLatency (const enum XPMNodeID node, const u32 latency)

This function is used by a PU to announce a change in the maximum wake-up latency requirements for a specific slave node currently used by that PU.

Parameters

<i>node</i>	Node ID of the PM slave.
<i>latency</i>	Maximum wake-up latency required.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

Note

Setting maximum wake-up latency can constrain the set of possible power states a resource can be put into.

XStatus XPM_GetApiVersion (u32 * *version*)

This function is used to request the version number of the API running on the power management controller.

Parameters

<i>version</i>	Returns the API 32-bit version number. Returns 0 if no PM firmware present.
----------------	---

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

Note

None

XStatus XPM_GetNodeStatus (const enum XPMNodId *node*, XPM_NodeStatus *const *nodestatus*)

This function is used to obtain information about the current state of a component. The caller must pass a pointer to an [XPM_NodeStatus](#) structure, which must be pre-allocated by the caller.

Parameters

<i>node</i>	ID of the component or sub-system in question.
<i>nodestatus</i>	<p>Used to return the complete status of the node.</p> <ul style="list-style-type: none"> • status - The current power state of the requested node. <ul style="list-style-type: none"> ◦ For CPU nodes: <ul style="list-style-type: none"> ■ 0 : if CPU is powered down, ■ 1 : if CPU is active (powered up), ■ 2 : if CPU is suspending (powered up) ◦ For power islands and power domains: <ul style="list-style-type: none"> ■ 0 : if island is powered down, ■ 1 : if island is powered up ◦ For PM slaves: <ul style="list-style-type: none"> ■ 0 : if slave is powered down, ■ 1 : if slave is powered up, ■ 2 : if slave is in retention • requirement - Slave nodes only: Returns current requirements the requesting PU has requested of the node. • usage - Slave nodes only: Returns current usage status of the node: <ul style="list-style-type: none"> ◦ 0 : node is not used by any PU, ◦ 1 : node is used by caller exclusively, ◦ 2 : node is used by other PU(s) only, ◦ 3 : node is used by caller and by other PU(s)

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

Note

None

XStatus XPm_RegisterNotifier (**XPm_Notifier *const notifier**)

A PU can call this function to request that the power management controller call its notify callback whenever a qualifying event occurs. One can request to be notified for a specific or any event related to a specific node.

Parameters

<i>notifier</i>	<p>Pointer to the notifier object to be associated with the requested notification. The notifier object contains the following data related to the notification:</p> <ul style="list-style-type: none"> • <i>nodeID</i> : ID of the node to be notified about, • <i>eventID</i> : ID of the event in question, '-1' denotes all events (- EVENT_STATE_CHANGE, EVENT_ZERO_USERS, EVENT_ERROR_CONDITION), • <i>wake</i> : true: wake up on event, false: do not wake up (only notify if awake), no buffering/queueing • <i>callback</i> : Pointer to the custom callback function to be called when the notification is available. The callback executes from interrupt context, so the user must take special care when implementing the callback. Callback is optional, may be set to NULL. • <i>received</i> : Variable indicating how many times the notification has been received since the notifier is registered.
-----------------	---

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

Note

The caller shall initialize the notifier object before invoking the XPm_RegisterNotifier function. While notifier is registered, the notifier object shall not be modified by the caller.

XStatus XPm_UnregisterNotifier (XPm_Notifyer *const notifier)

A PU calls this function to unregister for the previously requested notifications.

Parameters

<i>notifier</i>	Pointer to the notifier object associated with the previously requested notification
-----------------	--

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

Note

None

XStatus XPM_GetOpCharacteristic (const enum XPMNodeID node, const enum XPMOpCharType type, u32 *const result)

Call this function to request the power management controller to return information about an operating characteristic of a component.

Parameters

<i>node</i>	ID of the component or sub-system in question.
<i>type</i>	Type of operating characteristic requested: <ul style="list-style-type: none"> • power (current power consumption), • latency (current latency in us to return to active state), • temperature (current temperature),
<i>result</i>	Used to return the requested operating characteristic.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

Note

None

XStatus XPM_ResetAssert (const enum XPMReset reset, const enum XPMResetAction assert)

This function is used to assert or release reset for a particular reset line. Alternatively a reset pulse can be requested as well.

Parameters

<i>reset</i>	ID of the reset line
<i>assert</i>	Identifies action: <ul style="list-style-type: none"> • PM_RESET_ACTION_RELEASE : release reset, • PM_RESET_ACTION_ASSERT : assert reset, • PM_RESET_ACTION_PULSE : pulse reset,

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

Note

None

XStatus XPM_ResetGetStatus (const enum XPMReset reset, u32 * status)

Call this function to get the current status of the selected reset line.

Parameters

<i>reset</i>	Reset line
<i>status</i>	Status of specified reset (true - asserted, false - released)

Returns

Returns 1/XST_FAILURE for 'asserted' or 0/XST_SUCCESS for 'released'.

Note

None

XStatus XPM_MmioWrite (const u32 address, const u32 mask, const u32 value)

Call this function to write a value directly into a register that isn't accessible directly, such as registers in the clock control unit. This call is bypassing the power management logic. The permitted addresses are subject to restrictions as defined in the PCW configuration.

Parameters

<i>address</i>	Physical 32-bit address of memory mapped register to write to.
<i>mask</i>	32-bit value used to limit write to specific bits in the register.
<i>value</i>	Value to write to the register bits specified by the mask.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

Note

If the access isn't permitted this function returns an error code.

XStatus XPM_MmioRead (const u32 address, u32 *const value)

Call this function to read a value from a register that isn't accessible directly. The permitted addresses are subject to restrictions as defined in the PCW configuration.

Parameters

address	Physical 32-bit address of memory mapped register to read from.
value	Returns the 32-bit value read from the register

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

Note

If the access isn't permitted this function returns an error code.

Error Status

Overview

This section lists the Power management specific return error statuses.

Macros

- #define `XST_PM_INTERNAL` 2000L
- #define `XST_PM_CONFLICT` 2001L
- #define `XST_PM_NO_ACCESS` 2002L
- #define `XST_PM_INVALID_NODE` 2003L
- #define `XST_PM_DOUBLE_REQ` 2004L
- #define `XST_PM_ABORT_SUSPEND` 2005L
- #define `XST_PM_TIMEOUT` 2006L
- #define `XST_PM_NODE_USED` 2007L

Macro Definition Documentation

#define XST_PM_INTERNAL 2000L

An internal error occurred while performing the requested operation.

#define XST_PM_CONFLICT 2001L

Conflicting requirements have been asserted when more than one processing cluster is using the same PM slave.

#define XST_PM_NO_ACCESS 2002L

The processing cluster does not have access to the requested node or operation.

#define XST_PM_INVALID_NODE 2003L

The API function does not apply to the node passed as argument.

#define XST_PM_DOUBLE_REQ 2004L

A processing cluster has already been assigned access to a PM slave and has issued a duplicate request for that PM slave.

#define XST_PM_ABORT_SUSPEND 2005L

The target processing cluster has aborted suspend.

#define XST_PM_TIMEOUT 2006L

A timeout occurred while performing the requested operation.

#define XST_PM_NODE_USED 2007L

Slave request cannot be granted since node is non-shareable and used.

Appendix J:

XilFPGA Library v2.0

Overview

The XilFPGA library provides an interface to the Linux or bare-metal users for configuring the programmable logic (PL) over PCAP from PS.

The library is designed for Zynq® UltraScale+™ MPSoC to run on top of Xilinx standalone BSPs. It is tested for A53, R5 and MicroBlaze. In the most common use case, we expect users to run this library on PMU MicroBlaze with PMUFW to serve requests from Linux for bitstream programming. In this release, the XilFPGA library supports full, encrypted, authenticated bitstream download. In subsequent releases, the library may support partial bitstream loading.

Xilfpga library Interface modules

Xilfpga library uses the below major components to configure the PL through PS.

Processor Configuration Access Port (PCAP)

The processor configuration access port (PCAP) is used to configure the programmable logic (PL) through the PS.

CSU DMA driver

The CSU DMA driver is used to transfer the actual Bit stream file for the PS to PL after PCAP initialization.

Xilsecure_library

The LibXilSecure library provides APIs to access secure hardware on the Zynq® UltraScale+™ MPSoC devices. This library includes:

- SHA-256 hash function
- AES for symmetric key encryption
- RSA for authentication

Note

- The current version of library supports only Zynq® UltraScale+™ MPSoC devices.
- The XilFPGA library is capable of loading only .bin format files into PL. The library will not support the other file formats.
- Xilsecure_library is required only for the below use cases:
 - Encrypted bit-stream loading.
 - Authenticated bit-stream loading
- For Zynq® UltraScale+™ MPSoC devices, the required OCM memory for authentication bit-stream loading is 68Kb.

Design Summary

Xilfpga library acts as a bridge between the user application and the PL device. It provides the required functionality to the user application for configuring the PL Device with the required bit-stream. The figure below illustrates an implementation where the Xilfpga library needs the CSU DMA driver APIs to transfer the bit-stream from the DDR to the PL region. The Xilfpga library also needs the XilSecure library APIs to support while programming the authenticated and the encrypted bitstream files.

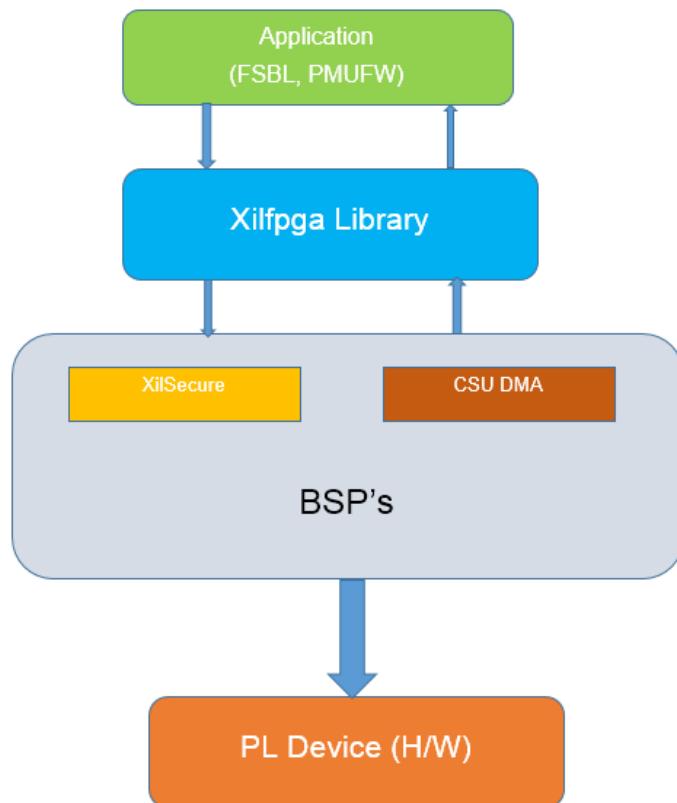


Figure 31.1: XilFPGA Design Summary

Flow Diagram

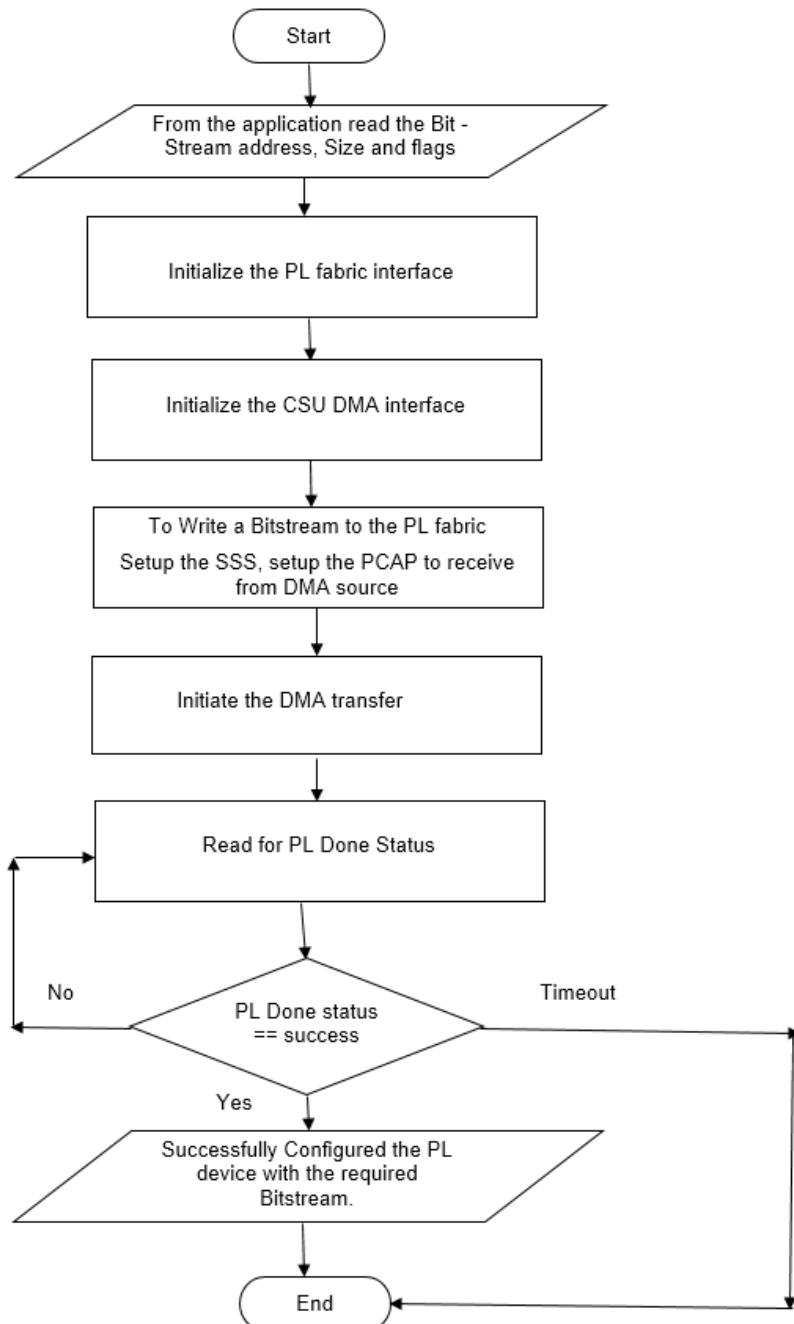


Figure 31.2: XilFPGA Library Workflow

Setting up the Software System

To use XilFPGA in a software application, you must first compile the XilFPGA library as part of software application.

1. Launch Xilinx SDK. Xilinx SDK prompts you to create a workspace.
 2. Select **File > New > Xilinx Board Support Package**. The **New Board Support Package** wizard appears.
 3. Specify a project name.
 4. Select **Standalone** from the **Board Support Package OS** drop-down list. The **Board Support Package Settings** wizard appears.
 5. Select the **xilfpga** library from the list of **Supported Libraries**.
 6. Expand the **Overview** tree and select **xilfpga**. The configuration options for xilfpga are listed.
 7. Configure the xilfpga by providing the base address of the Bit-stream file (DDR address) and the size (in bytes).
 8. Click **OK**. The board support package automatically builds with XilFPGA library included in it.
 9. Double-click the **system.mss** file to open it in the **Editor** view.
 10. Scroll-down and locate the **Libraries** chapter.
 11. Click **Import Examples** adjacent to the XilFPGA 2.0 entry.
-

Enabling Secure Mode in PMUFirmware

To support encrypted and authenticated bit-stream loading, you must enable secure mode in PMUFW.

1. Launch Xilinx SDK. Xilinx SDK prompts you to create a workspace.
2. Select **File > New > Application Project**. The **New Application Project** wizard appears.
3. Specify a project name.
4. Select **Standalone** from the **OS Platform** drop-down list.
5. Select a supported hardware platform.
6. Select **psu_pmu_0** from the **Processor** drop-down list.
7. Click Next. The **Templates** page appears.
8. Select **ZynqMP PMU Firmware** from the **Available Templates** list.
9. Click **Finish**. A PMUFW application project is created with the required BSPs.
10. Double-click the **system.mss** file to open it in the **Editor** view.

11. Click the **Modify this BSP's Settings** button. The **Board Support Package Settings** dialog box appears.
12. Select **xilfpga**. Various settings related to the library appears.
13. Select **secure_mode** and modify its value to **true**.
14. Click **OK** to save the configuration.

XilFPGA APIs

Overview

This chapter provides detailed descriptions of the XilFPGA library APIs.

Functions

- u32 [XFpga_PL_BitStream_Load](#) (u32 WrAddrHigh, u32 WrAddrLow, u32 WrSize, u32 flags)
- u32 [XFpga_PcapStatus](#) (void)

Function Documentation

u32 XFpga_PL_BitStream_Load (u32 WrAddrHigh, u32 WrAddrLow, u32 WrSize, u32 flags)

The API is used to load the user provided bitstream file into zynqmp PL region.

This function does the following jobs:

- Power-up the PL fabric.
- Performs PL-PS Isolation.
- Initialize PCAP Interface
- Write a bitstream into the PL
- Wait for the PL Done Status.
- Restore PS-PL Isolation (Power-up PL fabric).
- Performs the PS-PL reset.

Note

This function contains the polling implementation to provide the PL reset wait time due to this polling implementation the function call is blocked till the time out value expires or gets the appropriate status value from the PL Done Status register.

Parameters

<i>WrAddrHigh</i>	Higher 32-bit Linear memory space from where CSUDMA will read the data to be written to PCAP interface
<i>WrAddrLow</i>	Lower 32-bit Linear memory space from where CSUDMA will read the data to be written to PCAP interface
<i>WrSize</i>	Number of 32bit words that the DMA should write to the PCAP interface
<i>flags</i>	<p>Flags are used to specify the type of bitstream file.</p> <ul style="list-style-type: none"> ◦ BIT(0) - Bit-stream type <ul style="list-style-type: none"> ■ 0 - Full Bit-stream ■ 1 - Partial Bit-stream ◦ BIT(2) - Authentication <ul style="list-style-type: none"> ■ 1 - Enable ■ 0 - Disable ◦ BIT(3) - Encryption <ul style="list-style-type: none"> ■ 1 - Enable ■ 0 - Disable

Returns

- Error status based on implemented functionality (SUCCESS by default).

u32 XFpga_PcapStatus (void)

This function provides the STATUS of PCAP interface.

Parameters

<i>None</i>	
-------------	--

Returns

Status of the PCAP interface.

Appendix K:

XilSecure Library Reference

Overview

The XilSecure library provides APIs to access secure hardware on the Zynq® UltraScale+™ MPSoC devices and also provides an algorithm for SHA-2 hash generation.

The XilSecure library includes:

- SHA-3/384 engine for 384 bit hash calculation
- AES engine for symmetric key encryption and decryption
- RSA engine for asymmetry decryption

Note

The above libraries are grouped into the Configuration and Security Unit (CSU) on the Zynq UltraScale+ MPSoC device.

- SHA-2/256 algorithm for calculating 256 bit hash

Note

The SHA-2 hash generation is a software algorithm which generates SHA2 hash on provided data.

Source Files

The following is a list of source files shipped as a part of the XilSecure library:

- `xsecure_hw.h`: This file contains the hardware interface for all the three modules.
- `xsecure_sha.h`: This file contains the driver interface for SHA-3 module.
- `xsecure_sha.c`: This file contains the implementation of the driver interface for SHA-3 module.
- `xsecure_rsa.h`: This file contains the driver interface for RSA module.
- `xsecure_rsa.c`: This file contains the implementation of the driver interface for RSA module.
- `xsecure_aes.h`: This file contains the driver interface for AES module.
- `xsecure_aes.c`: This file contains the implementation of the driver interface for AES module
- `xsecure_sha2.h`: This file contains the interface for SHA2 hash algorithm.

- `xsecure_sha2_a53_32b.a`: Pre-compiled file which has SHA2 implementation for A53 32bit.
- `xsecure_sha2_a53_64b.a`: Pre-compiled file which has SHA2 implementation for A53 64 bit.
- `xsecure_sha2_a53_r5.a`: Pre-compiled file which has SHA2 implementation for r5.
- `xsecure_sha2_pmu.a`: Pre-compiled file which has SHA2 implementation for PMU.

AES-GCM

Overview

This block uses AES-GCM algorithm to encrypt or decrypt the provided data. It requires a key of size 256 bits and initialization vector(IV) is a symmetric key of size 96 bits.

XilSecure library supports the following features:

- Encryption of data with provided key and IV
- Decryption of data with provided key and IV
- Decryption of Zynq® Ultrascale+™ MPSoC boot image partition, where boot image is generated using bootgen.
 - Support for Key rolling
 - Operational key support
- Authentication using GCM tag.
- Key loading based on key selection, key can be either KUP key or device key.

For either encryption or decryption AES should be initialized first, the [XSecure_AesInitialize\(\)](#) API initializes the AES's instance with provided parameters as described.

AES Encryption Function Usage

When all the data to be encrypted is available, the [XSecure_AesEncryptData\(\)](#) can be used with appropriate parameters as described. When all the data is not available use the following functions in following order.

1. [XSecure_AesEncryptInit\(\)](#)
2. [XSecure_AesEncryptUpdate\(\)](#) - This API can be called multiple times till input data is completed.

AES Decryption Function Usage

When all the data to be decrypted is available, the [XSecure_AesDecryptData\(\)](#) can be used with appropriate parameters as described. When all the data is not available use the following functions in following order.

1. [XSecure_AesDecryptInit\(\)](#)

2. [XSecure_AesDecryptUpdate\(\)](#) - This API can be called multiple times till input data is completed. The GCM-TAG matching will also be verified and appropriate status will be returned.

Modules

- [AES-GCM API Example Usage](#)

Functions

- s32 [XSecure_AesInitialize](#) (XSecure_Aes *InstancePtr, XCsuDma *CsuDmaPtr, u32 KeySel, u32 *Iv, u32 *Key)
- void [XSecure_AesDecryptInit](#) (XSecure_Aes *InstancePtr, u8 *DecData, u32 Size, u8 *GcmTagAddr)
- s32 [XSecure_AesDecryptUpdate](#) (XSecure_Aes *InstancePtr, u8 *EncData, u32 Size)
- s32 [XSecure_AesDecryptData](#) (XSecure_Aes *InstancePtr, u8 *DecData, u8 *EncData, u32 Size, u8 *GcmTagAddr)
- s32 [XSecure_AesDecrypt](#) (XSecure_Aes *InstancePtr, u8 *Dst, const u8 *Src, u32 Length)
- void [XSecure_AesEncryptInit](#) (XSecure_Aes *InstancePtr, u8 *EncData, u32 Size)
- void [XSecure_AesEncryptUpdate](#) (XSecure_Aes *InstancePtr, const u8 *Data, u32 Size)
- void [XSecure_AesEncryptData](#) (XSecure_Aes *InstancePtr, u8 *Dst, const u8 *Src, u32 Len)
- void [XSecure_AesReset](#) (XSecure_Aes *InstancePtr)
- void [XSecure_AesWaitForDone](#) (XSecure_Aes *InstancePtr)

Function Documentation

s32 XSecure_AesInitialize (XSecure_Aes * InstancePtr, XCsuDma * CsuDmaPtr, u32 KeySel, u32 * Iv, u32 * Key)

This function initializes the instance pointer.

Parameters

<i>InstancePtr</i>	Pointer to the XSecure_Aes instance.
<i>CsuDmaPtr</i>	Pointer to the XCsuDma instance.
<i>KeySel</i>	Key source for decryption, can be KUP/device key <ul style="list-style-type: none"> • XSECURE_CSU_AES_KEY_SRC_KUP :For KUP key • XSECURE_CSU_AES_KEY_SRC_DEV :For Device Key
<i>Iv</i>	Pointer to the Initialization Vector for decryption
<i>Key</i>	Pointer to Aes decryption key in case KUP key is used. Passes Null if device key is to be used.

Returns

XST_SUCCESS if initialization was successful.

Note

All the inputs are accepted in little endian format, but AES engine accepts the data in big endianess, this will be taken care while passing data to AES engine.

void XSecure_AesDecryptInit (XSecure_Aes * InstancePtr, u8 * DecData, u32 Size, u8 * GcmTagAddr)

This function initializes the AES engine for decryption.

Parameters

<i>InstancePtr</i>	Pointer to the XSecure_Aes instance.
<i>DecData</i>	Pointer in which decrypted data will be stored.
<i>Size</i>	Expected size of the data in bytes.
<i>GcmTagAddr</i>	Pointer to the GCM tag which needs to be verified during decryption of the data.

Returns

None

Note

If data is encrypted using XSecure_AesEncrypt API then GCM tag address will be at the end of encrypted data. EncData + Size will be the GCM tag address.

s32 XSecure_AesDecryptUpdate (XSecure_Aes * InstancePtr, u8 * EncData, u32 Size)

This function is used to update the AES engine for decryption with provided data.

Parameters

<i>InstancePtr</i>	Pointer to the XSecure_Aes instance.
<i>EncData</i>	Pointer to the encrypted data which needs to be decrypted.
<i>Size</i>	Expected size of data to be decrypted in bytes.

Returns

Final call of this API returns the status of GCM tag matching.

- XSECURE_CSU_AES_GCM_TAG_MISMATCH: If GCM tag is mismatched
- XST_SUCCESS: If GCM tag is matching.

Note

When Size of the data equals to size of the remaining data that data will be treated as final data. This API can be called multiple times but sum of all Sizes should be equal to Size mention in init. Return of the final call of this API tells whether GCM tag is matching or not.

s32 XSecure_AesDecryptData (*XSecure_Aes * InstancePtr, u8 * DecData, u8 * EncData, u32 Size, u8 * GcmTagAddr*)

This function decrypts the encrypted data provided and updates the DecData buffer with decrypted data.

Parameters

<i>InstancePtr</i>	Pointer to the XSecure_Aes instance.
<i>DecData</i>	Pointer to a buffer in which decrypted data will be stored.
<i>EncData</i>	Pointer to the encrypted data which needs to be decrypted.
<i>Size</i>	Size of data to be decrypted in bytes.

Returns

This API returns the status of GCM tag matching.

- XSECURE_CSU_AES_GCM_TAG_MISMATCH: If GCM tag was mismatched
- XST_SUCCESS: If GCM tag was matched.

Note

When [XSecure_AesEncryptData\(\)](#) API is used for encryption In same buffer GCM tag also be stored, but Size should be mentioned only for data.

s32 XSecure_AesDecrypt (*XSecure_Aes * InstancePtr, u8 * Dst, const u8 * Src, u32 Length*)

This function will handle the AES-GCM Decryption.

Parameters

<i>InstancePtr</i>	Pointer to the XSecure_Aes instance.
<i>Src</i>	Pointer to encrypted data source location
<i>Dst</i>	Pointer to location where decrypted data will be written.
<i>Length</i>	Expected total length of decrypted image expected.

Returns

returns XST_SUCCESS if successful, or the relevant errorcode.

Note

This function is used for decrypting the Image's partition encrypted by Bootgen

void XSecure_AesEncryptInit (XSecure_Aes * *InstancePtr*, u8 * *EncData*, u32 *Size*)

This function is used to initialize the AES engine for encryption.

Parameters

<i>InstancePtr</i>	Pointer to the XSecure_Aes instance.
<i>EncData</i>	Pointer of a buffer in which encrypted data along with GCM TAG will be stored. Buffer size should be Size of data plus 16 bytes.
<i>Size</i>	A 32 bit variable, which holds the size of the input data to be encrypted.

Returns

None

Note

If all the data to be encrypted is available at single location One can use [XSecure_AesEncryptData\(\)](#) directly.

void XSecure_AesEncryptUpdate (XSecure_Aes * *InstancePtr*, const u8 * *Data*, u32 *Size*)

This function is used to update the AES engine with provided data for encryption.

Parameters

<i>InstancePtr</i>	Pointer to the XSecure_Aes instance.
<i>Data</i>	Pointer to the data for which encryption should be performed.
<i>Size</i>	A 32 bit variable, which holds the size of the input data in bytes.

Returns

None

Note

When Size of the data equals to size of the remaining data to be processed that data will be treated as final data. This API can be called multiple times but sum of all Sizes should be equal to Size mentioned at encryption initialization ([XSecure_AesEncryptInit\(\)](#)). If all the data to be encrypted is available at single location Please call [XSecure_AesEncryptData\(\)](#) directly.

void XSecure_AesEncryptData (XSecure_Aes * *InstancePtr*, u8 * *Dst*, const u8 * *Src*, u32 *Len*)

This Function encrypts the data provided by using hardware AES engine.

Parameters

<i>InstancePtr</i>	A pointer to the XSecure_Aes instance.
<i>Dst</i>	A pointer to a buffer where encrypted data along with GCM tag will be stored. The Size of buffer provided should be Size of the data plus 16 bytes
<i>Src</i>	A pointer to input data for encryption.
<i>Len</i>	Size of input data in bytes

Returns

None

Note

If data to be encrypted is not available at one place one can call [XSecure_AesEncryptInit\(\)](#) and update the AES engine with data to be encrypted by calling [XSecure_AesEncryptUpdate\(\)](#) API multiple times as required.

void XSecure_AesReset (XSecure_Aes * *InstancePtr*)

This function resets the AES engine.

Parameters

<i>InstancePtr</i>	is a pointer to the XSecure_Aes instance.
--------------------	---

Returns

None

void XSecure_AesWaitForDone (XSecure_Aes * *InstancePtr*)

This function waits for AES completion.

Parameters

<i>InstancePtr</i>	Pointer to the XSecure_Aes instance.
--------------------	--------------------------------------

Returns

None

AES-GCM API Example Usage

The `xilsecure_aes_example.c` file illustrates AES usage with decryption of a Zynq® UltraScale+™ MPSoC boot image placed at a predefined location in memory. You can select the key type (device key or user-selected KUP key). The example assumes that the boot image is present at `0x0400000` (DDR); consequently, the image must be loaded at that address through JTAG. The example decrypts the boot image and returns `XST_SUCCESS` or `XST_FAILURE` based on whether the GCM tag was successfully matched.

The Multiple key(Key Rolling) or Single key encrypted images will have the same format. The images include:

- Secure header - This includes the Dummy AES Key of 32byte + Block 0 IV of 12byte + DLC for Block 0 of 4byte + GCM tag of 16byte(Un-Enc).
- Block N - This includes the Boot Image Data for Block N of n size + Block N+1 AES key of 32byte + Block N+1 IV of 12byte + GCM tag for Block N of 16byte(Un-Enc).

The Secure header and Block 0 will be decrypted using Device key or user provided key. If more than one block is found then the key and IV obtained from previous block will be used for decryption.

Following are the instructions to decrypt an image:

1. Read the first 64 bytes and decrypt 48 bytes using the selected Device key.
2. Decrypt Block 0 using the IV + Size and the selected Device key.
3. After decryption, you will get the decrypted data+KEY+IV+Block Size. Store the KEY/IV into KUP/IV registers.
4. Using Block size, IV and the next Block key information, start decrypting the next block.

5. If the current image size is greater than the total image length, perform the next step. Else, go back to the previous step.

6. If there are failures, an error code is returned. Else, the decryption is successful.

The contents of the `xilsecure_aes_example.c` file are shown below:

```

int SecureAesExample(void)
{
    u8 *Dst = (u8 *)0x04100000;
    XCsuDma_Config *Config;

    int Status;

    Config = XCsuDma_LookupConfig(0);
    if (NULL == Config) {
        xil_printf("config failed \n\r");
        return XST_FAILURE;
    }

    Status = XCsuDma_CfgInitialize(&CsuDma, Config, Config->BaseAddress);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    /*
     * Download the boot image elf in DDR, Read the boot header
     * assign Src pointer to the location of FSBL image in it. Ensure
     * that linker script does not map the example elf to the same
     * location as this standalone example
     */
    u32 FsblOffset = XSecure_In32((UINTPTR)(ImageOffset + HeaderSrc0ffset));

    u32 FsblLocation = ImageOffset + FsblOffset;

    u32 FsblLength = XSecure_In32((UINTPTR)(ImageOffset + HeaderFsblLenOffset));

    /*
     * Initialize the Aes driver so that it's ready to use
     */
    XSecure_AesInitialize(&Secure_Aes, &CsuDma, XSECURE_CSU_AES_KEY_SRC_KUP,
                          (u32 *)csu_iv, (u32 *)csu_key);

    Status = XSecure_AesDecrypt(&Secure_Aes, Dst, (u8 *)(UINTPTR)FsblLocation,
                               FsblLength);

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

    return XST_SUCCESS;
}

```

Note

The `xilsecure_aes_example.c` example file is available in the `<library-install-path>\examples` folder. Where `<library-install-path>` is the XilSecure library installation path.

RSA

Overview

The xsecure_rsa.h file contains hardware interface related information for RSA device. This block decrypts data based on Rivest-Shamir-Adelman (RSA)-4096 algorithm. It is an asymmetry algorithm.

Initialization & Configuration

The Rsa driver instance can be initialized by using the [XSecure_RsaInitialize\(\)](#) function.

The method used for RSA decryption needs pre-calculated value of $R^2 \bmod N$, which is generated by bootgen and is present in the signature along with modulus and exponent. If you do not have the pre-calculated exponential value pass NULL, the controller will take care of exponential value.

Note

- From public key modulus, exponent should be extracted. If image is created using bootgen all the fields are available in the boot image.
- For matching, PKCS padding scheme has to be applied in the manner while comparing the data hash with decrypted hash.

Modules

- [RSA API Example Usage](#)

Functions

- s32 [XSecure_RsaInitialize](#) (XSecure_Rsa *InstancePtr, u8 *Mod, u8 *ModExt, u8 *ModExpo)
- s32 [XSecure_RsaDecrypt](#) (XSecure_Rsa *InstancePtr, u8 *EncText, u8 *Result)
- u32 [XSecure_RsaSignVerification](#) (u8 *Signature, u8 *Hash, u32 HashLen)

Function Documentation

s32 XSecure_RsaInitialize (*XSecure_Rsa * InstancePtr, u8 * Mod, u8 * ModExt, u8 * ModExpo*)

This function initializes a specific Xsecure_Rsa instance so that it is ready to be used.

Parameters

<i>InstancePtr</i>	Pointer to the XSecure_Rsa instance.
<i>Mod</i>	A character Pointer which contains the key Modulus.
<i>ModExt</i>	A Pointer to the pre-calculated exponential($R^2 \text{ Mod } N$) value. <ul style="list-style-type: none"> • NULL - if user doesn't have pre-calculated $R^2 \text{ Mod } N$ value, control will take care of this calculation internally.
<i>ModExpo</i>	Pointer to the buffer which contains key exponent.

Returns

XST_SUCCESS if initialization was successful.

Note

Modulus, ModExt and ModExpo are part of partition signature when authenticated boot image is generated by bootgen, else all of them should be extracted from the key.

s32 XSecure_RsaDecrypt (*XSecure_Rsa * InstancePtr, u8 * EncText, u8 * Result*)

This function handles the RSA decryption from end to end.

Parameters

<i>InstancePtr</i>	Pointer to the XSecure_Rsa instance.
<i>EncText</i>	Pointer to the buffer which contains the input data to be decrypted.
<i>Result</i>	Pointer to the buffer where resultant decrypted data to be stored .

Returns

XST_SUCCESS if decryption was successful.

u32 XSecure_RsaSignVerification (u8 * *Signature*, u8 * *Hash*, u32 *HashLen*)

This function verifies the RSA decrypted data provided is either matching with the provided expected hash by taking care of PKCS padding.

Parameters

<i>Signature</i>	Pointer to the buffer which holds the decrypted RSA signature
<i>Hash</i>	Pointer to the buffer which has hash calculated on the data to be authenticated.
<i>HashLen</i>	<p>Length of Hash used.</p> <ul style="list-style-type: none"> • For SHA3 it should be 48 bytes • For SHA2 it should be 32 bytes

Returns

XST_SUCCESS if decryption was successful.

RSA API Example Usage

The `xilsecure_rsa_example.c` file deals with RSA based authentication of FSBL in a Zynq® UltraScale+™ MPSoC boot image. The boot image signature is decrypted using RSA- 4096 algorithm. Resulting digest is matched with SHA digest calculated on the FSBL using SHA-3 driver.

The authenticated boot image should be loaded in memory through JTAG and address of the boot image should be passed to the function. By default, the example assumes that the authenticated image is present at location 0x04000000 (DDR), which can be changed as required.

The contents of the `xilsecure_rsa_example.c` file are shown below.

```
u32 SecureRsaExample(void)
{
    u32 Status;

    /*
     * Download the boot image elf at a DDR location, Read the boot header
     * assign Src pointer to the location of FSBL image in it. Ensure
     * that linker script does not map the example elf to the same
     * location as this standalone example
     */
    u32 FsblOffset = XSecure_In32((UINTPTR)(ImageOffset + HeaderSrcOffset));

    xil_printf(" Fsbl Offset in the image is %0x ",FsblOffset);
    xil_printf(" \r\n ");

    u32 FsblLocation = ImageOffset + FsblOffset;

    xil_printf(" Fsbl Location is %0x ",FsblLocation);
    xil_printf(" \r\n ");

    u32 TotalFsblLength = XSecure_In32((UINTPTR)(ImageOffset +
        HeaderFsblTotalLenOffset));
```

```

u32 AcLocation = FsblLocation + TotalFsblLength - XSECURE_AUTH_CERT_MIN_SIZE;

xil_printf(" Authentication Certificate Location is %0x ",AcLocation);
xil_printf(" \r\n ");

u8 BIHash[XSECURE_HASH_TYPE_SHA3] __attribute__ ((aligned (4)));
u8 * SpkModular = (u8 *)XNULL;
u8 * SpkModularEx = (u8 *)XNULL;
u32 SpkExp = 0;
u8 * AcPtr = (u8 *)(UINTPTR)AcLocation;
u32 ErrorCode = XST_SUCCESS;
u32 FsblTotalLen = TotalFsblLength - XSECURE_FSBL_SIG_SIZE;

xil_printf(" Fsbl Total Length(Total - BI Signature) %0x ",
    (u32)FsblTotalLen);
xil_printf(" \r\n ");

AcPtr += (XSECURE_RSA_AC_ALIGN + XSECURE_PPK_SIZE);
SpkModular = (u8 *)AcPtr;
AcPtr += XSECURE_FSBL_SIG_SIZE;
SpkModularEx = (u8 *)AcPtr;
AcPtr += XSECURE_FSBL_SIG_SIZE;
SpkExp = *((u32 *)AcPtr);
AcPtr += XSECURE_RSA_AC_ALIGN;

AcPtr += (XSECURE_SPK_SIG_SIZE + XSECURE_BHDR_SIG_SIZE);
xil_printf(" Boot Image Signature Location is %0x ",(u32)(UINTPTR)AcPtr);
xil_printf(" \r\n ");

/*
 * Set up CSU DMA instance for SHA-3 transfers
 */
XCsuDma_Config *Config;

Config = XCsuDma_LookupConfig(0);
if (NULL == Config) {
    xil_printf("config failed\r\n");
    return XST_FAILURE;
}

Status = XCsuDma_CfgInitialize(&CsuDma, Config, Config->BaseAddress);
if (Status != XST_SUCCESS) {
    return XST_FAILURE;
}

/*
 * Initialize the SHA-3 driver so that it's ready to use
 * Look up the configuration in the config table and then initialize it.
 */

XSecure_Sha3Initialize(&Secure_Sha3, &CsuDma);
XSecure_Sha3Start(&Secure_Sha3);

XSecure_Sha3Update(&Secure_Sha3, (u8 *)(UINTPTR)FsblLocation,
    FsblTotalLen);

XSecure_Sha3Finish(&Secure_Sha3, (u8 *)BIHash);

/*
 * Initialize the Rsa driver so that it's ready to use
 * Look up the configuration in the config table and then initialize it.
 */
XSecure_RsaInitialize(&Secure_Rsa, SpkModular, SpkModularEx,
    (u8 *)&SpkExp);

/*

```

```
* Decrypt Boot Image Signature.  
*/  
if(XST_SUCCESS != XSecure_RsaDecrypt(&Secure_Rsa, AcPtr,  
    XSecure_RsaSha3Array))  
{  
    ErrorCode = XSECURE_IMAGE_VERIF_ERROR;  
    goto ENDF;  
}  
  
xil_printf("\r\n Calculated Boot image Hash \r\n ");  
int i= 0;  
for(i=0; i < 384/8; i++)  
{  
    xil_printf(" %0x ", BIHash[i]);  
}  
xil_printf(" \r\n ");  
  
xil_printf("\r\n Hash From Signature \r\n ");  
int ii= 128;  
for(ii = 464; ii < 512; ii++)  
{  
    xil_printf(" %0x ", XSecure_RsaSha3Array[ii]);  
}  
xil_printf(" \r\n ");  
  
/*  
 * Authenticate FSBL Signature.  
 */  
if(XSecure_RsaSignVerification(XSecure_RsaSha3Array, BIHash,  
    XSECURE_HASH_TYPE_SHA3) != 0)  
{  
    ErrorCode = XSECURE_IMAGE_VERIF_ERROR;  
}  
  
ENDF:  
    return ErrorCode;  
}
```

Note

The `xilsecure_rsa_example.c` example file is available in the `<library-install-path>\examples` folder. Where `<library-install-path>` is the XilSecure library installation path.

SHA-3

Overview

This block uses the NIST-approved SHA-3 algorithm to generate 384 bit hash on the input data. Because the SHA-3 hardware only accepts 104 byte blocks as minimum input size, the input data is padded with a 10*1 sequence to complete the final byte block. The padding is handled internally by the driver API.

Initialization & Configuration

The SHA-3 driver instance can be initialized using the [XSecure_Sha3Initialize\(\)](#) function.

A pointer to CsuDma instance has to be passed in initialization as CSU DMA will be used for data transfers to SHA module.

SHA-3 Functions Usage

When all the data is available on which sha3 hash must be calculated, the [XSecure_Sha3Digest\(\)](#) can be used with appropriate parameters, as described. When all the data is not available on which sha3 hash must be calculated, use the sha3 functions in the following order:

1. [XSecure_Sha3Start\(\)](#)
2. [XSecure_Sha3Update\(\)](#) - This API can be called multiple times till input data is completed.
3. [XSecure_Sha3Finish\(\)](#) - Provides the final hash of the data. To get intermediate hash values after each [XSecure_Sha3Update\(\)](#), you can call [XSecure_Sha3_ReadHash\(\)](#) after the [XSecure_Sha3Update\(\)](#) call.

Modules

- [SHA-3 API Example Usage](#)

Functions

- s32 [XSecure_Sha3Initialize](#) (XSecure_Sha3 *InstancePtr, XCsuDma *CsuDmaPtr)
- void [XSecure_Sha3Start](#) (XSecure_Sha3 *InstancePtr)

- void **XSecure_Sha3Update** (XSecure_Sha3 *InstancePtr, const u8 *Data, const u32 Size)
- void **XSecure_Sha3Finish** (XSecure_Sha3 *InstancePtr, u8 *Hash)
- void **XSecure_Sha3Digest** (XSecure_Sha3 *InstancePtr, const u8 *In, const u32 Size, u8 *Out)
- void **XSecure_Sha3_ReadHash** (XSecure_Sha3 *InstancePtr, u8 *Hash)

Function Documentation

s32 XSecure_Sha3Initialize (XSecure_Sha3 * InstancePtr, XCsuDma * CsuDmaPtr)

This function initializes a specific Xsecure_Sha3 instance so that it is ready to be used.

Parameters

<i>InstancePtr</i>	Pointer to the XSecure_Sha3 instance.
<i>CsuDmaPtr</i>	Pointer to the XCsuDma instance.

Returns

XST_SUCCESS if initialization was successful

Note

The base address is initialized directly with value from xsecure_hw.h

void XSecure_Sha3Start (XSecure_Sha3 * InstancePtr)

This function configures the SSS and starts the SHA-3 engine.

Parameters

<i>InstancePtr</i>	Pointer to the XSecure_Sha3 instance.
--------------------	---------------------------------------

Returns

None

void XSecure_Sha3Update (XSecure_Sha3 * InstancePtr, const u8 * Data, const u32 Size)

This function updates hash for new input data block.

Parameters

<i>InstancePtr</i>	Pointer to the XSecure_Sha3 instance.
<i>Data</i>	Pointer to the input data for hashing.
<i>Size</i>	Size of the input data in bytes.

Returns

None

Note

None

void XSecure_Sha3Finish (XSecure_Sha3 * *InstancePtr*, u8 * *Hash*)

This function sends the last data and padding when blocksize is not multiple of 104 bytes.

Parameters

<i>InstancePtr</i>	Pointer to the XSecure_Sha3 instance.
<i>Hash</i>	Pointer to location where resulting hash will be written

Returns

None

Note

None

void XSecure_Sha3Digest (XSecure_Sha3 * *InstancePtr*, const u8 * *In*, const u32 *Size*, u8 * *Out*)

This function calculates the SHA-3 digest on the given input data.

Parameters

<i>InstancePtr</i>	Pointer to the XSecure_Sha3 instance.
<i>In</i>	Pointer to the input data for hashing
<i>Size</i>	Size of the input data
<i>Out</i>	Pointer to location where resulting hash will be written.

Returns

None

void XSecure_Sha3_ReadHash (XSecure_Sha3 * InstancePtr, u8 * Hash)

Reads the SHA3 hash of the data. It can be called intermediately of updates also to read hashes.

Parameters

<i>InstancePtr</i>	Pointer to the XSecure_Sha3 instance.
<i>Hash</i>	Pointer to a buffer in which read hash will be stored.

Returns

None

Note

None

SHA-3 API Example Usage

The `xilsecure_sha_example.c` file is a simple example application that demonstrates the usage of SHA-3 device to calculate 384 bit hash on Hello World string. A more typical use case of calculating the hash of boot image as a step in authentication process using the SHA-3 device has been illustrated in the `xilsecure_rsa_example.c`.

The contents of the `xilsecure_sha_example.c` file are shown below:

```
int SecureHelloWorldExample()
{
    u8 HelloWorld[4] = {'h','e','l','l'};
    u32 Size = sizeof(HelloWorld);
    u8 Out[384/8];
    XCsuDma_Config *Config;

    int Status;

    Config = XCsuDma_LookupConfig(0);
    if (NULL == Config) {
        xil_printf("config failed\n\r");
        return XST_FAILURE;
    }

    Status = XCsuDma_CfgInitialize(&CsuDma, Config, Config->BaseAddress);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    /*
     * Initialize the SHA-3 driver so that it's ready to use
     */
    XSecure_Sha3Initialize(&Secure_Sha3, &CsuDma);
```

```
XSecure_Sha3Digest(&Secure_Sha3, HelloWorld, Size, Out);

xil_printf(" Calculated Digest \r\n");
int i= 0;
for(i=0; i< (384/8); i++)
{
    xil_printf(" %0x ", Out[i]);
}
xil_printf(" \r\n");

return XST_SUCCESS;
}
```

Note

The `xilsecure_sha_example.c` and `xilsecure_rsa_example.c` example files are available in the `<library-install-path>\examples` folder. Where `<library-install-path>` is the XilSecure library installation path.

SHA-2

Overview

This is an algorithm which generates 256 bit hash on the input data.

SHA-2 Function Usage

When all the data is available on which sha2 hash must be calculated, the `sha_256()` can be used with appropriate parameters, as described. When all the data is not available on which sha2 must be calculated, use the sha2 functions in the following order:

1. `sha2_starts()`
2. `sha2_update()` - This API can be called multiple times till input data is completed.
3. `sha2_finish()` - Provides the final hash of the data.

To get intermediate hash values after each `sha2_update()`, you can call `sha2_hash()` after the `sha2_update()` call.

Modules

- [SHA-2 Example Usage](#)
-

Functions

- `void sha_256 (const unsigned char *in, const unsigned int size, unsigned char *out)`
- `void sha2_starts (sha2_context *ctx)`
- `void sha2_update (sha2_context *ctx, unsigned char *input, unsigned int ilen)`
- `void sha2_finish (sha2_context *ctx, unsigned char *output)`
- `void sha2_hash (sha2_context *ctx, unsigned char *output)`

Function Documentation

void sha_256 (const unsigned char * *in*, const unsigned int *size*, unsigned char * *out*)

This function calculates the hash for the input data using SHA-256 algorithm. This function internally calls the sha2_init, updates and finishes functions and updates the result.

Parameters

<i>In</i>	Char pointer which contains the input data.
<i>Size</i>	Length of the input data
<i>Out</i>	Pointer to location where resulting hash will be written.

Returns

None

void sha2_starts (sha2_context * *ctx*)

This function initializes the SHA2 context.

Parameters

<i>ctx</i>	Pointer to sha2_context structure that stores status and buffer.
------------	--

Returns

None

void sha2_update (sha2_context * *ctx*, unsigned char * *input*, unsigned int *ilen*)

This function adds the input data to SHA256 calculation.

Parameters

<i>ctx</i>	Pointer to sha2_context structure that stores status and buffer.
<i>input</i>	Pointer to the data to add.
<i>Out</i>	Length of the input data.

Returns

None

void sha2_finish (sha2_context * ctx, unsigned char * output)

This function finishes the SHA calculation.

Parameters

<i>ctx</i>	Pointer to sha2_context structure that stores status and buffer.
<i>output</i>	Pointer to the calculated hash data.

Returns

None

void sha2_hash (sha2_context * ctx, unsigned char * output)

This function reads the SHA2 hash, it can be called intermediately of updates to read the SHA2 hash.

Parameters

<i>ctx</i>	Pointer to sha2_context structure that stores status and buffer.
<i>output</i>	Pointer to the calculated hash data.

Returns

None

SHA-2 Example Usage

The `xilsecure_sha2_example.c` file contains the implementation of the interface functions for SHA driver. When all the data is available on which sha2 must be calculated, the `sha_256()` function can be used with appropriate parameters, as described. But, when all the data is not available on which sha2 must be calculated, use the sha2 functions in the following order:

- `sha2_update()` can be called multiple times till input data is completed.
- `sha2_context` is updated by the library only; do not change the values of the context.

The contents of the `xilsecure_sha2_example.c` file are shown below:

```

u32 XSecure_Sha2_Hash_Gn()
{
    sha2_context Sha2;
    u8 Output_Hash[32];
    u8 IntermediateHash[32];
    u8 Cal_Hash[32];
    u32 Index;
    u32 Size = XSECURE_DATA_SIZE;
    u32 Status;

    /* Generating SHA2 hash */
    sha2_starts(&Sha2);
    sha2_update(&Sha2, (u8 *)Data, Size - 1);

    /* If required we can read intermediate hash */
    sha2_hash(&Sha2, IntermediateHash);
    xil_printf("Intermediate SHA2 Hash is: ");
    for (Index = 0; Index < 32; Index++) {
        xil_printf("%02x", IntermediateHash[Index]);
    }
    xil_printf("\n");

    sha2_finish(&Sha2, Output_Hash);

    xil_printf("Generated SHA2 Hash is: ");
    for (Index = 0; Index < 32; Index++) {
        xil_printf("%02x", Output_Hash[Index]);
    }
    xil_printf("\n");

    /* Convert expected Hash value into hexa */
    Status = XSecure_ConvertStringToHexBE(XSECURE_EXPECTED_SHA2_HASH,
                                           Cal_Hash, 64);
    if (Status != XST_SUCCESS) {
        xil_printf("Error: While converting expected "
                  "string of SHA2 hash to hexa\n\r");
        return XST_FAILURE;
    }

    /* Compare generated hash with expected hash value */
    for (Index = 0; Index < 32; Index++) {
        if (Cal_Hash[Index] != Output_Hash[Index]) {
            xil_printf("Error: SHA2 Hash generated through "
                      "XilSecure library does not match with "
                      "expected hash value\n\r");
            return XST_FAILURE;
        }
    }

    return XST_SUCCESS;
}

```

Note

The `xilsecure_sha2_example.c` example file is available in the `<library-install-path>\examples` folder. Where `<library-install-path>` is the XilSecure library installation path.

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see [Xilinx Support](#).

Solution Centers

See the [Xilinx Solution Centers](#) for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (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 the Xilinx Documentation Navigator, click the **Design Hubs View** tab.
- On the Xilinx website, see the [Design Hubs](#) page.

Note: For more information on Documentation Navigator, see the [Documentation Navigator](#) page on the Xilinx website.

References

Xilinx References

1. *Xilinx Third-Party Licensing Solution Center* (includes [UG763](#))
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 Documents

8. *Quick EMUlator (QEMU) User Guide* ([UG1169](#))
9. *UltraScale Architecture and Product Overview* ([DS890](#))
10. *Zynq UltraScale+ MPSoC Technical Reference Manual* ([UG1085](#))
11. *Zynq UltraScale+ MPSoC Register Reference* ([UG1087](#))
12. *UltraScale Architecture System Monitor Guide* ([UG580](#))
13. *Zynq UltraScale+ MPSoC OpenAMP Getting Started Guide* ([UG1186](#))
14. *Zynq UltraScale+ MPSoC Power Management Framework* ([UG1199](#))
15. *Embedded Energy Management Interface Specification* ([UG1200](#))
16. *UltraFast Embedded Design Methodology Guide* ([UG1046](#))
17. *Zynq-7000 Embedded Design Tutorial* ([UG1165](#))
18. *Zynq UltraScale+ MPSoC Packaging and Pinout* ([UG1075](#))
19. *Zynq-7000 All Programmable SoC Software Developers Guide* ([UG821](#))
20. [Vivado Design Suite Documentation](#)

SDK and PetaLinux Documents

21. *Xilinx Software Developer Kit Help* ([UG782](#))
22. *OS and Libraries Document Collection* ([UG643](#))
23. *Embedded Design Tools* [Download](#)

24. *PetaLinux Tools Documentation Reference Guide* ([UG1144](#))
25. *Xilinx Software Development Kit: System Performance* ([UG1145](#))

Xilinx IP Documents

26. *LogiCORE IP AXI Central Direct Memory Access Product Guide* ([PG034](#))
27. *LogiCORE IP AXI Video Direct Memory Access Product Guide* ([PG020](#))

Miscellaneous Links

28. [Xilinx Github](#)
29. [Embedded Development](#)
30. [meta-xilinx](#)
31. [PetaLinux Software Development](#)
32. [Zynq UltraScale+ Silicon Devices Page](#)
33. Xilinx Answer: [66249](#)
34. [Vivado Quick Take Video: Vivado PS Configuration Wizard Overview](#)

Third-Party References

35. [Lauterbach Technologies](#)
36. [ARM Trusted Firmware](#)
37. [Xen Hypervisor](#)
38. [ARM Developer Center](#)
39. [ARM Cortex-A53 MPCore Processor Technical Reference Manual](#)
40. [Yocto Product Development](#)
41. [GNU FTP](#)
42. *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 2015-2017 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, 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. AMBA, AMBA Designer, ARM, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, and MPCore are trademarks of ARM in the EU and other countries. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. PCI, PCIe, and PCI Express are trademarks of PCI-SIG and used under license.