See all versions of this document

# Versal ACAP System Software Developers Guide

UG1304 (v2020.2) March 1, 2021

XILINX®

# Revision History

The following table shows the revision history for this document.

| Section | Revision Summary |
| --- | --- |
| **03/01/2021 Version 2020.2** | |
| Chapter 3: Development Tools | Added AI Engine Development Environment. |
| Chapter 6: Software Design Paradigms | Updated Virtualization with Hypervisor and Use of Hypervisors. |
| **11/24/2020 Version 2020.2** | |
| PLM Boot and Configuration | Updated JTAG information and added more PLM error codes.<br>Added PLM Watch Dog Timer details for PLM Interface (XilPLMI).<br>Updated Deferred Image Loading and Deferred Image Handoff details for XilLoader.<br>Updated PLM Build Flags and Memory and Registers Reserved for PLM in PLM Usage. |
| Chapter 9: Security | Updated details of ECDSA-RSA hardware accelerator and AES-GCM hardware Crypto block. |
| Chapter 10: Versal ACAP Platform Management | Updated XPm_DevIoctl EEMI API. |
| **07/16/2020 Version 2020.1** | |
| Initial release. | N/A |

# Table of Contents

Chapter 1

# Overview

## Introduction to Versal ACAP

Versal™ adaptive compute acceleration platforms (ACAPs) combine Scalar Engines, Adaptable Engines, and Intelligent Engines with leading-edge memory and interfacing technologies to deliver powerful heterogeneous acceleration for any application. Most importantly, Versal ACAP hardware and software are targeted for programming and optimization by data scientists and software and hardware developers. Versal ACAPs are enabled by a host of tools, software, libraries, IP, middleware, and frameworks to enable all industry-standard design flows.

Built on the TSMC 7 nm FinFET process technology, the Versal portfolio is the first platform to combine software programmability and domain-specific hardware acceleration with the adaptability necessary to meet today's rapid pace of innovation. The portfolio includes six series of devices uniquely architected to deliver scalability and AI inference capabilities for a host of applications across different markets—from cloud—to networking—to wireless communications—to edge computing and endpoints.

The Versal architecture combines different engine types with a wealth of connectivity and communication capability and a network on chip (NoC) to enable seamless memory-mapped access to the full height and width of the device. Intelligent Engines are SIMD VLIW AI Engines for adaptive inference and advanced signal processing compute, and DSP Engines for fixed point, floating point, and complex MAC operations. Adaptable Engines are a combination of programmable logic blocks and memory, architected for high-compute density. Scalar Engines, including Arm® Cortex®-A72 and Cortex-R5F processors, allow for intensive compute tasks.

The Versal AI Core series delivers breakthrough AI inference acceleration with AI Engines that deliver over 100x greater compute performance than current server-class of CPUs. This series is designed for a breadth of applications, including cloud for dynamic workloads and network for massive bandwidth, all while delivering advanced safety and security features. AI and data scientists, as well as software and hardware developers, can all take advantage of the high-compute density to accelerate the performance of any application.

The Versal Prime series is the foundation and the mid-range of the Versal platform, serving the broadest range of uses across multiple markets. These applications include 100G to 200G networking equipment, network and storage acceleration in the Data Center, communications test equipment, broadcast, and aerospace & defense. The series integrates mainstream 58G transceivers and optimized I/O and DDR connectivity, achieving low-latency acceleration and performance across diverse workloads.

The Versal Premium series provides breakthrough heterogeneous integration, very high-performance compute, connectivity, and security in an adaptable platform with a minimized power and area footprint. The series is designed to exceed the demands of high-bandwidth, compute-intensive applications in wired communications, data center, test & measurement, and other applications. Versal Premium series ACAPs include 112G PAM4 transceivers and integrated blocks for 600G Ethernet, 600G Interlaken, PCI Express® Gen5, and high-speed cryptography.

The Versal architecture documentation suite is available at: https://www.xilinx.com/versal.

## Navigating Content by Design Process

Xilinx® documentation is organized around a set of standard design processes to help you find relevant content for your current development task. All Versal™ ACAP design process Design Hubs can be found on the Xilinx.com website. This document covers the following design processes:

- **System and Solution Planning:** Identifying the components, performance, I/O, and data transfer requirements at a system level. Includes application mapping for the solution to PS, PL, and AI Engine.
  - Chapter 5: Software Development Flow
  - Chapter 6: Software Design Paradigms
  - Chapter 7: Boot and Configuration
  - Chapter 8: Platform Loader and Manager
  - Chapter 9: Security
  - Chapter 10: Versal ACAP Platform Management
  - Chapter 11: Target Development Platforms
- **Embedded Software Development:** Creating the software platform from the hardware platform and developing the application code using the embedded CPU. Also covers XRT and Graph APIs.
  - Chapter 5: Software Development Flow
  - Chapter 3: Development Tools

- Chapter 4: Software Stack
- Chapter 6: Software Design Paradigms

# About This Guide

This guide focuses on the Versal ACAP system software development environment, and is the first document that software developers should read. This document includes the following:

- **Chapter 2: Programming View of Versal ACAP:** Provides an overview of system software, and relevant aspects of the Versal ACAP hardware.
- **Chapter 3: Development Tools:** Describes the available Xilinx tools and flows for programming the Versal device.
- **Chapter 4: Software Stack:** Provides an overview of the various software stacks available for the Versal devices.
- **Chapter 5: Software Development Flow:** Explains the bare-metal software development for the real-time processing unit (RPU) and the application processing unit (APU) using the Vitis™ IDE, as well as Linux software development for the APU using PetaLinux and Vitis tools.
- **Chapter 6: Software Design Paradigms:** Describes the Xilinx Versal device architecture that supports heterogeneous multiprocessor engines for different tasks.
- **Chapter 7: Boot and Configuration:** Presents the type of boot modes that Versal ACAP supports, along with an overview of the boot and configuration process for both secure and non-secure boot modes.
- **Chapter 8: Platform Loader and Manager:** Explains the role of the platform loader and manager (PLM) that runs on the platform processing unit (PPU) in the platform management controller (PMC). The PLM performs boot and configuration of the Versal ACAP, and then continuously monitors services after the initial boot and configuration of the Versal device.
- **Chapter 9: Security:** Details the Versal device features that you can leverage to address security during boot time and run time of an application.
- **Chapter 10: Versal ACAP Platform Management:** Describes the role of the platform management in managing resources, such as power, clock, reset, and pins optimally.
- **Chapter 11: Target Development Platforms:** Describes the boards and kits available for Versal ACAP.
- **Appendix A: Libraries:** Details the libraries and APIs available for Versal ACAP.

*Chapter 2*

# Programming View of Versal ACAP

Versal™ ACAP devices include five types of programmable processors. Each type of processor provides different computation capabilities to meet different requirements of the overall system:

- **Arm® Cortex®-A72 dual-core processor in the processor system (PS):** Typically used for control-plane applications, operating systems, communications interfaces, and lower level or complex computations.
- **Arm Cortex-R5F dual-core processor in the PS:** Typically used for applications requiring safety and determinism.
- **MicroBlaze™ processors in the programmable logic (PL):** (Optional) Typically used for data manipulation and transport, non-vector-based computation, and interfacing to the PS and other components on Versal ACAP.
- **RCU and PPU units in the PMC:** Typically used for executing platform loader and manager (PLM) code.
- **AI Engines:** Typically used for compute-intensive functions in vector implementations.

Another key block in Versal ACAP is the platform management controller (PMC). Typically used for executing PLM code. This block is responsible for boot, configuration, partial-reconfiguration, and life cycle management tasks such as security. For more information about PMC management of boot and partial reconfiguration, see Chapter 7: Boot and Configuration.

This chapter briefly discusses:

- PS with dual-core Cortex-A72 and dual-core Cortex-R5F processors
- MicroBlaze processor in the PL
- Linux and bare-metal software stacks used with the processors
- Boot and configuration information
- Additional features relevant to a software engineer

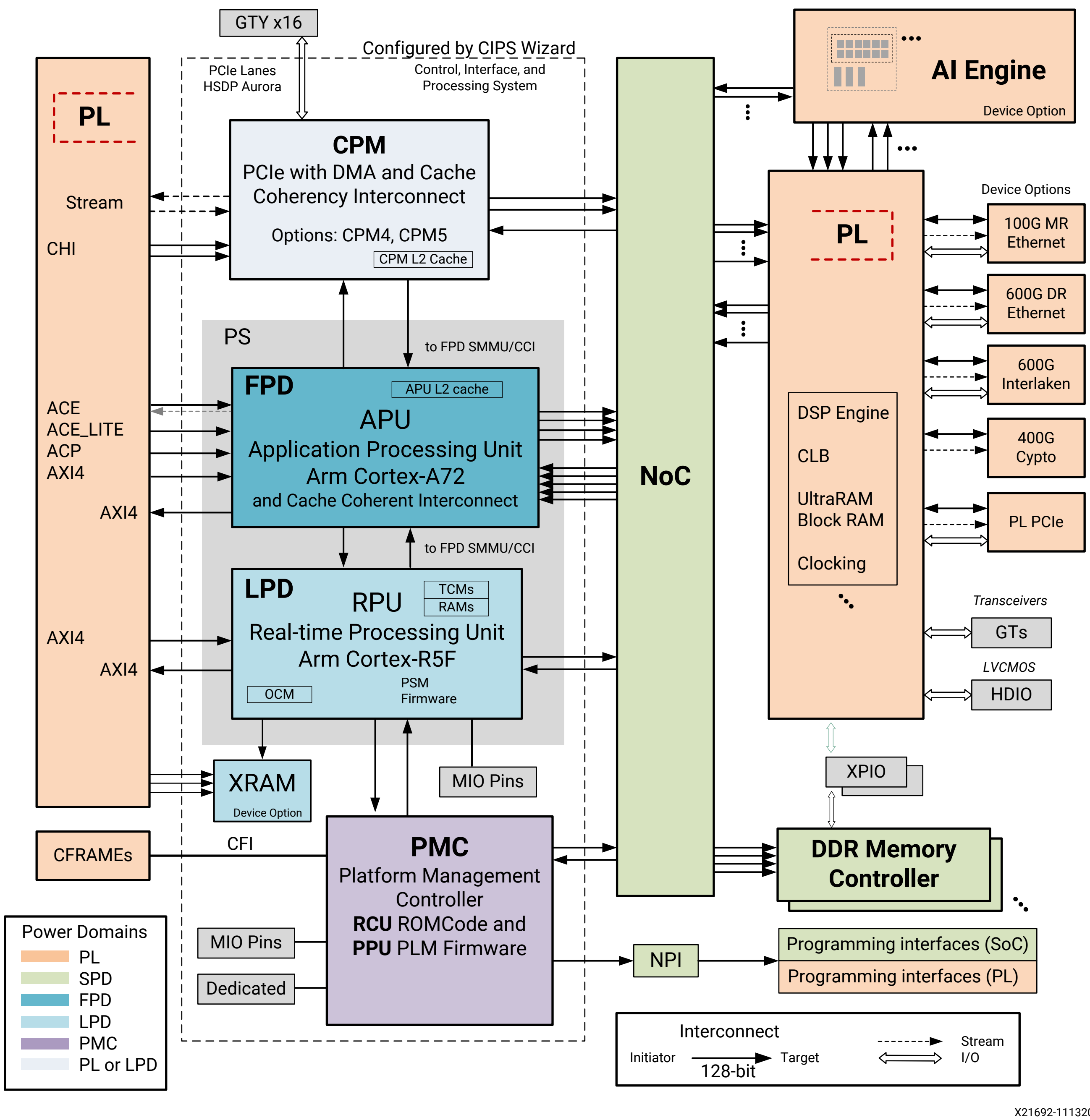
For details about additional features such as the PMC, DDR bus width, number of DDR controllers, interconnect for CCIX and PCIe (CPM), and PCI Express®, see the *Versal Architecture and Product Data Sheet: Overview* (DS950). The *Versal ACAP Technical Reference Manual* (AM011) includes details on PMC/PS-centric content with a hardware architecture section that includes links to documents that describe other integrated hardware and peripherals including the CPM, DDR, AI Engine, PL, and more.

# Hardware Overview

This section provides an overview of the Versal ACAP hardware view components.

***Note*:** For more detailed information about the Versal ACAP hardware, refer to the *Versal ACAP Technical Reference Manual* (AM011).

*Figure 1:* **System-level Interconnect Architecture**



### Key Hardware Components

The following list describes the largest hardware view components:

- **AI Engine:** The AI Engine is used for custom acceleration and compute engines.
- **APU:** The application processing unit (APU) consists of Cortex-A72 processor cores, L1/L2 caches, and related functionality. The Cortex-A72 cores and caches are part of Arm MPCore IP.

  Versal ACAP uses a dual-core Cortex-A72 processor system with 1 MB L2 cache. The Cortex-A72 cores implement Armv8 64-bit architecture. The Cortex-A72 MPCore does not have integrated generic interrupt controller (GIC), so an external GIC IP is used. For more information, refer to "APU Processor Features" in *Versal ACAP Technical Reference Manual* (AM011).
- **AXI Interconnect:** The Advanced eXtensible Interface interconnect connects one or more AXI memory-mapped master devices to one or more memory-mapped slave devices. The AXI interfaces conform to the AMBA® AXI version 4 specifications from Arm, including the AXI4-Lite control register interface subset.
- **CPM:** The interconnect for Cache Coherent Interconnect for Accelerators (CCIX) and PCIe® (CPM) module is the primary PCIe interface for the processing system. There are two integrated blocks for PCIe in the CPM, supporting up to Gen4 x16. You can configure both of the integrated blocks for PCIe as an endpoint. Furthermore, you can configure each integrated block as a root port that contains direct memory access (DMA) logic. The CPM also incorporates CCIX functionality to allow a PL accelerator to act as a CCIX compliant accelerator.
- **PL:** The programmable logic (PL) is a scalable structure that includes adaptable engines and intelligent engines that can be used to construct accelerators, processors, or almost any other complex functionality. It is configured using the Vivado® tools. The architect determines the components to be available in the PL design. For example, the MicroBlaze processor is just an IP core, so you can optionally add MicroBlaze processors to the design. For more info, refer to "Programmable Logic" in *Versal ACAP Technical Reference Manual* (AM011).
- **PMC:** The platform management controller (PMC) handles device management control functions such as device reset sequencing, initialization, boot, configuration, security, power management, dynamic function eXchange (DFX), health-monitoring, and error management. You can boot the device in either secure or non-secure mode. For more information, refer to "Platform Management Controller" in *Versal ACAP Technical Reference Manual* (AM011).
- **NoC Interconnect:** The NoC is the main interconnect and contains a vertical component (VNoC) and a horizontal component (HNoC).
  - HNoC is integrated in the horizontal super row/region (HSR). The HSR includes blocks such as XPIO, hard DDR memory controller, PLL, System Monitor satellites, HBM, and AI Engine.

- VNoC integration includes global-clk-column, and System Monitor satellites. In SSI technology, VNoCs are connected across super logic region (SLR) boundaries. Microbumps and buffers for this reside in the Thin-HNoC. Configuration data between SSI technology master and slaves travels over the NoC.

- **RPU:** The real-time processing unit (RPU) is a dual-core Cortex-R5F processor, based on the Armv7-R architecture, which can run as either two independent cores or in a lock-step configuration. For more information, refer to Platform Management in *Versal ACAP Technical Reference Manual* (AM011).

### Additional Hardware Components

- **Peripheral Controllers:** The input/output units (IOPs) present in the PS module contains the low speed peripherals in low power domain (LPD) and high-speed peripherals in full power domain (FPD). Some IOPs are present in the PMC power domain (PPD) peripherals.

  For more information, refer to "I/O Peripherals" in *Versal ACAP Technical Reference Manual* (AM011).

- **Interconnects and Buses:** Versal ACAP has following additional interconnects and buses:

  - **NPI:** The NoC programming interface, a 32-bit programming interface to the NoC and several attached units.

    For more information, refer to *Versal ACAP Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* (PG313).

  - **APB:** The advanced peripheral (APB) bus is a 32-bit single-word read/write bus interface. This bus is used to access control registers in the functional units, i.e., subsystem units. These control registers are used to program the functional units. The APB switch is used as the main switch in the following four areas:
    - PMC
    - LPD
    - FPD
    - CPM

  - **CFI:** The configuration frame interface (CFI) transports the configuration information contained in the boot image from the PMC to its destination within the Versal device. CFI provides a dedicated high-bandwidth 128-bit bus to PL for configuration and readback. For more information, refer "CFI" in *Versal ACAP Technical Reference Manual* (AM011).

- **System Watchdog Timer:** The system watchdog (SWDT) timer is used to detect and recover from various malfunctions. The watchdog timer can be used to prevent system lockup (when the software becomes trapped in a deadlock). For more information, refer to "System Watchdog Timer" in *Versal ACAP Technical Reference Manual* (AM011).

- **Clocks:** Versal ACAP has the following clocks:

  - PMC and PS clocks

  - CPM clocks
  - NoC, AI Engine, and DDR memory controller clocks
  - PL clocks

  For more information, refer to the "Clocks" chapter in *Versal ACAP Technical Reference Manual* (AM011).

- **Memory:** Versal device has following list of memories:
  - **DDR Memory:** Up to 12 GB of RAM is supported. This DDR memory is external to the device.
  - **On-chip memory (OCM) in the PS:** This memory is 256 KB in size, and is also accessible by the RPU.
  - **Tightly coupled memory (TCM) in the RPU:** This memory is 256 KB and is mainly used by the RPU but can be accessed by the APU.
  - **Battery-backed RAM (BBRAM):** This memory stores the advanced encryption standard (AES) 256-bit key.
  - **Accelerator RAM (XRAM):** 4 MB memory size present in some Versal devices.
  - **eFUSE:** 2,048-bits of user memory, stores multiple keys, and stores security configuration settings
- **Reset:** Versal ACAP has several layers of resets with overlapping effects. The highest-level resets are generally aligned with power domains, then power island resets, and finally the individual functional unit resets. In some cases, functional units have local resets that affects part of the block. The reset hierarchy:
  - Subsystem resets (power domains)
  - Power-island resets
  - Functional unit (block) resets
  - Partial resets of a block (some cases)

  For more information, refer to the "Resets" section in *Versal ACAP Technical Reference Manual* (AM011).

- **Virtualization:** The Versal device includes the following three hardware components for virtualization:
  - CPU virtualization
  - Memory virtualization
  - Interrupt virtualization

For more information, refer to "Memory Virtualization" in *Versal ACAP Technical Reference Manual* (AM011).

- **Security and Safety:** The Versal device has the following security management and safety features:
  - Secure key storage and management
  - Tamper monitoring and response
  - User access to Xilinx hardware cryptographic accelerators
  - Xilinx memory protection unit (XMPU) and Xilinx peripheral protection unit (XPPU) provides hardware-enforced isolation.
  - TrustZone

  For more information, refer to "Platform Management Controller" in *Versal ACAP Technical Reference Manual* (AM011) and also Chapter 9: Security. For XMPU and XPPU, refer to "Memory Protection" in *Versal ACAP Technical Reference Manual* (AM011).

## Processing System

The processing system (PS) has the following components:

- Dual-core Arm Cortex-A72 processor with 1 MB L2 cache with error correction code (ECC)
- Dual 32-bit Cortex-R5F processor cores based on the Arm® v7-R architecture and supports both RPU with 128 KB TCM with error correction code (ECC) and Single Lockstep R5 with 128 KB TCM with ECC.
- 256 KB on-chip memory with error correction code (ECC)
- Arm CoreSight™ debug and trace (DAP) with TMC, STM, ATM, and APM
- System memory management unit (SMMU)
- CCI
- One universal serial bus (USB) 2.0
- PCIe RP/EP in CPM (device dependent)
- Two gigabit Ethernet MAC with TSN support
- One low-power domain DMA (LPD-DMA)
- Performance I/O
- Two controller area network-flexible data rates (CAN-FD), two serial peripheral interfaces (SPI), two I2C, and two UART controllers
- PS management controller (PSM)

***Note*:** The PS has access to shared external DDR.

### *Application Processing Unit*

The application processing unit (APU) consists of an Arm Cortex-A72 processor, L1/L2 caches, and related functionality. The Cortex-A72 cores and caches are part of the Arm processor MPCore IP, with integrated L1 and L2 caches.

The Versal device uses a dual-core Cortex-A72 with 1 MB L2 cache.

The Cortex-A72 cores implement Armv8 64-bit architecture. The Cortex-A72 MPCore processor does not have an integrated generic interrupt controller (GIC), so an external GIC IP is used.

The APU includes the following features:

- Dual-core Cortex-A72 core class with 1 MB L2 with error correction code (ECC). The L1 caches include I-cache of 48 KB in size and for I-cache and D-cache of 32 KB in size. L1 caches include error correction code (ECC).
- GIC-500 interrupt controller
- Per core power-gating support
- TrustZone support

### *Real-Time Processing Unit*

The Versal ACAP APU provides improved performance at an improved safety level. However, for real-time applications which require a higher level of safety (e.g., ASIL-C/SIL3), reliability, and determinism, real-time processing unit (RPU) is used with a lockstep processor subsystem.

The RPU architecture specification consists of RPU cores, TCMs, and on-chip memory. The following list describes the main RPU features.

- Dual 32-bit Cortex-R5F cores based on Arm v7-R architecture and supports lock-step or split mode options
- 128 KB TCM per Cortex-R5F processor in split mode.
- Option to combine 256 KB of TCM in lock-step mode.
- 256 KB of on-chip memory with error correction code (ECC) accessible by both the RPU and the APU.
- 32 KB L1 instruction cache with error correction code (ECC) or parity and 32 KB L1 data cache with error correction code (ECC)
- Generic interrupt controller (GIC) to support GIC architecture
- Per lock-step power-gating support
- TCM and OCM power-gating
- TrustZone aware

### Tightly Coupled Memory Interface

The per-core TCM can be configured for the following modes:

- **Split, performance mode:** While running the RPU in split mode, each core can only access 128 KB of TCM.
- **Lock-step, safety mode:** While running in the lock-step mode, the RPU has access to the entire 256 KB of TCM.

### RPU Configuration Options

The following table and figure describe the four configuration options of the RPU.

*Table 1:* **RPU Configuration Options**

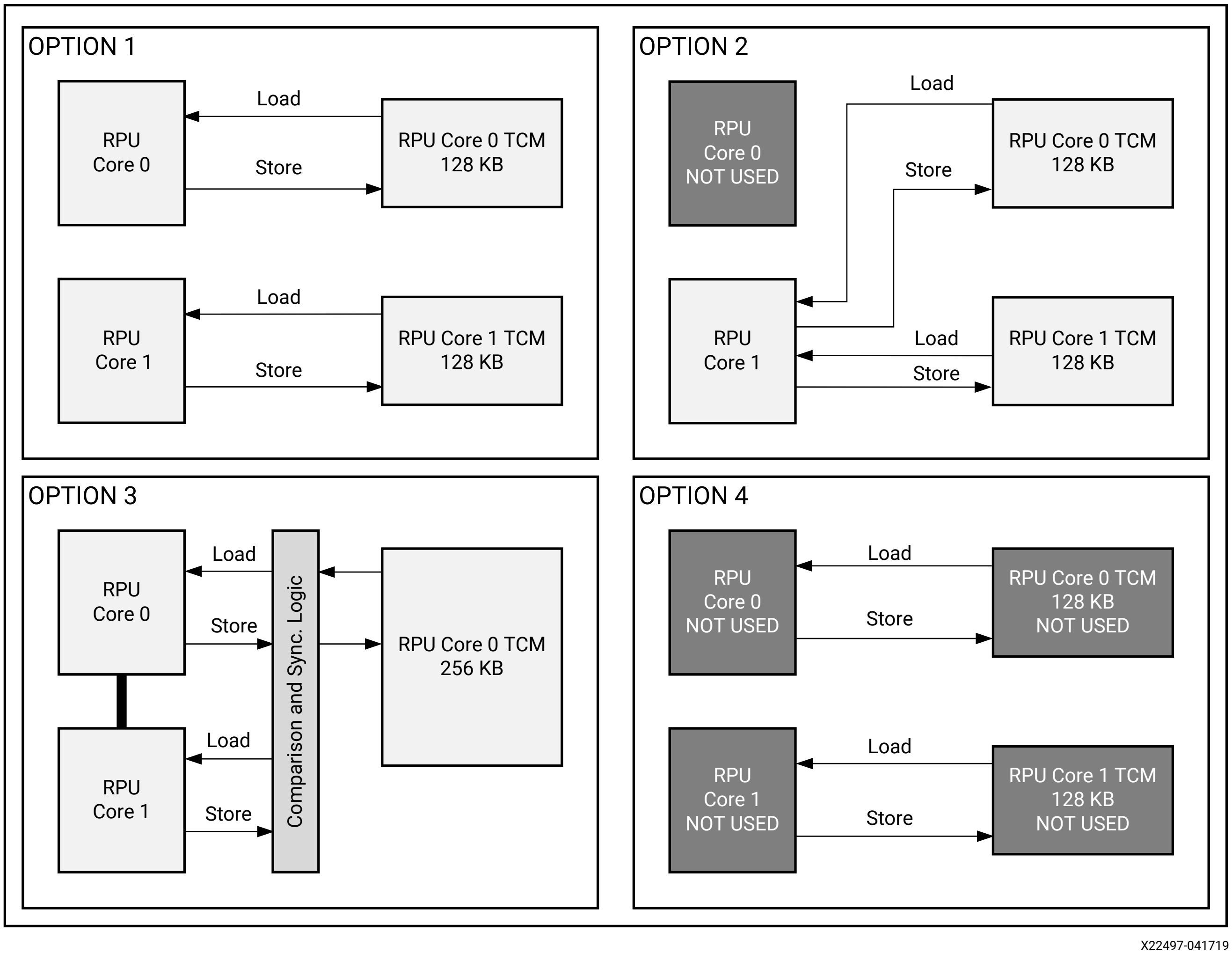
| Configuration Option | Description | Core Running |
|---|---|---|
| **Option 1:** Split mode | High-performance mode.<br>In this configuration, both real-time cores work independently, each using separate TCMs. | RPU core 0 and RPU core 1 |
| **Option 2:** Split mode, only one core used | High-performance mode.<br>In this configuration, RPU core 0 can be held in reset, while RPU core 1 runs independently using all 256 KB of TCM. | RPU core 1 only |
| **Option 3:** Lock-step mode | Safety mode.<br><br>***Note*:** The lock-step mode is typically used for safety-critical deterministic applications.<br><br>In this configuration, both cores run in parallel with each other, with integrated comparator logic. The RPU core 0 is the master, and RPU core 1 is the checker.<br>The TCM is combined to give RPU core 0 a larger TCM.<br>Each core executes the same code. The inputs and outputs of the two cores are compared. If they do not match, then the comparator detects the error.<br>While two cores are used, the performance is of one core. | RPU core 0 only |
| **Option 4:** None | RPU is not used. | None |

**Notes:**

1. RPU core 0 TCM is the tightly coupled memory associated with Cortex-R5F core 0 core in split mode; RPU core 1 TCM is the tightly-coupled memory associated with the RPU_0 core in split mode.
2. RPU core 1 TCM is located slightly farther from the cores than RPU core 0 TCM, so there might be a slightly longer delay when cores access RPU core 1 TCM versus RPU core 0 TCM.

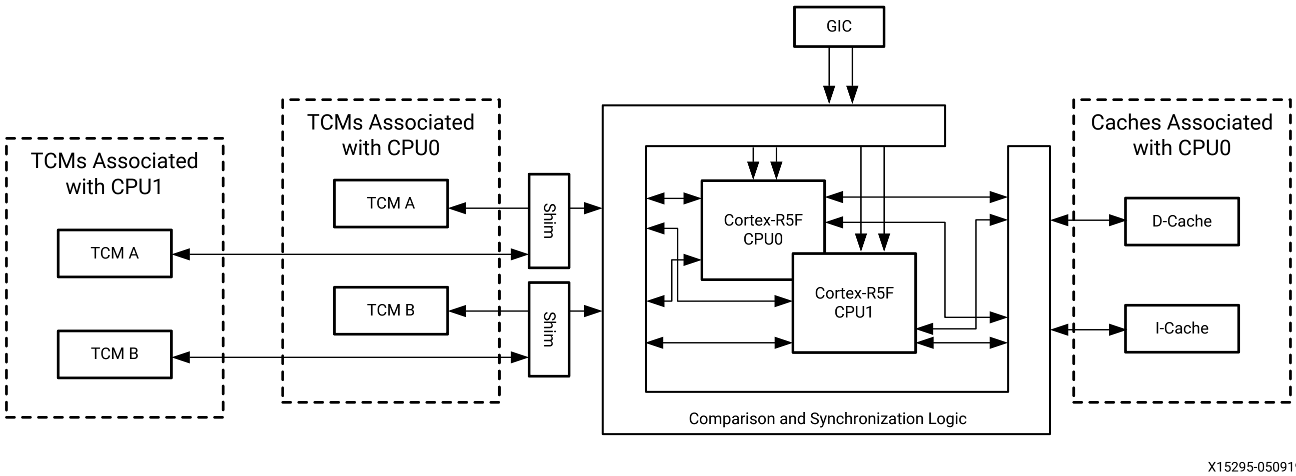
RPU cores can use the system watchdog timer (SWDT) to monitor functionality and check performance, through a periodic write to a timer register.

*Figure 2:* **RPU Configuration Options**




The following figure shows the resource sharing when the RPU cores are configured in lock-step mode.

*Figure 3:* **RPU Cortex-R5F Processor Lock-step Mode**




# Programmable Logic

You can use the Vivado IP integrator to configure the Versal ACAP programmable logic (PL). The PL is flexible, and the configuration uses building blocks or integrated components to create a customized design.

The PL is a complex structure that includes integrated and instantiated hardware accelerators, controllers, memories, and miscellaneous functional units.

- Integrated functional units include a multi-gigabit Ethernet MAC.
- Building blocks are used to instantiate functional units, and connect the integrated units to the interconnect and I/O structures. Building blocks include DSP, block RAM, UltraRAM, and clocking structures.
- Instantiated functional units are built using the PL integrated building blocks and could include:
  - **Interconnects:** AXI, NoC interconnect
  - **Platform control components:** PS configuration and reset
  - **Digital functional units:** Adders, counters, floating point unit (FPU), and video
  - **Radio frequency-oriented (RF) functional units:** RF usage functional units, long term evolution (LTE), and radio

## *MicroBlaze Processors in the Programmable Logic*

Optionally, architects can add MicroBlaze processors into the PL design. These MicroBlaze processors can run a variety of software, including Linux, bare-metal applications, FreeRTOS, or other custom software.

# Development Tools

This chapter focuses on Xilinx® tools and flows that are available for programming Versal™ ACAP. It also provides a brief description about the available open source tools that you can use for open source development on different processors of Versal ACAP.

A comprehensive set of tools for developing and debugging software applications on the Versal device includes:

- Hardware IDE
- Software IDE
- Compiler toolchain
- Debug and trace tools
- Simulators (for example, Quick Emulator (QEMU))
- Models and virtual prototyping tools (for example, emulation board platforms)

***Note*:** The level of integration and direct support with Versal ACAP using third-party tool solutions varies.

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 IDE. The IDE provides an intuitive GUI with powerful features.

The tools deliver a SoC-strength, IP- and system-centric, development environment built exclusively by Xilinx to address the productivity bottlenecks in system-level integration and implementation.

All commands and command options in the Vivado Design Suite use the native tool command language (Tcl) format, which can run on both, the Vivado IDE or the Vivado Design Suite Tcl shell. Analysis and constraint assignment is enabled throughout the design process. For example, you can run timing or power estimations after synthesis, placement, or routing. As 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 (an ASCII file defining the components and their connections) 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.

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

- The control, interfaces, and processing system IP (CIPS) configuration within IP integrator with graphical user interfaces to let you create and modify the CIPS within the IP integrator block design.
- Register transfer level (RTL) design in VHDL, Verilog, and SystemVerilog
- Quick integration and configuration of IP cores from the Xilinx IP catalog to create block designs through the Vivado IP integrator
- Vivado synthesis
- C-based sources in C, C++, and SystemC
- Vivado implementation for place and route
- Vivado serial I/O and logic analyzer for debugging
- Vivado power analysis
- Synopsys design constraints (SDC)-based Xilinx design constraints (XDC) for timing constraints entry
- Static timing analysis
- Flexible floorplanning
- Detailed placement and routing modification
- Vivado Tcl Store, which you can use to add to and modify the capabilities in the Vivado tool

You can download the Vivado Design Suite from Vivado Design Suite – HLx Editions.

# Vitis Software Platform

Versal ACAP designs are enabled by the Vitis™ tools, libraries, and IP. The Vitis IDE lets you program, run, and debug the different elements of a Versal ACAP AI Engine application, which can include AI Engine kernels and graphs, PL, high-level synthesis (HLS) IP, RTL IP, and PS applications. For detailed information on the Vitis IDE tool flow, refer to *Chapter 4: Design Flow* in *Versal ACAP Design Guide* (UG1273).

The Vitis software platform supports the following flows for software development:

- Accelerated Flow

- Embedded Flow

## Accelerated Flow

For acceleration, the Vitis development environment lets you build a software application using the OpenCL™ or the open source Xilinx Runtime (XRT) native API to run the hardware kernels on accelerator cards, such as the Xilinx Alveo™ Data Center accelerator cards. The Vitis core development kit also supports running the software application on a Linux-embedded processor platform such as Versal ACAP.

For the embedded processor platform, the Vitis execution model also uses the OpenCL™ API and the Linux-based XRT to schedule the hardware kernels and control data movement.

The Vitis tools support the Alveo PCIe-based cards, as well as the ZCU102 Base, ZCU102 Base with dynamic function eXchange (DFX), ZCU104 base, ZC702 base, ZC706 base embedded processor platforms, and the Versal ACAP VCK190 base and VMK180 base boards.

In addition to these off-the-shelf platforms, the Vitis tools support custom platforms.

The Vitis development environment supports application development for both data center and embedded processor platforms, allowing you to migrate data center applications to embedded platforms. The Vitis tool includes the `v++` compiler for the hardware kernel on all platforms, the `g++` compiler for compiling the application to run on an x86 host, and Arm® compiler for cross-compiling the application to run on the embedded processor of a Xilinx device.

***Note***: The `v++` compiler has two options: compile (that is `v++ -c` to Vitis HLS), system linking (`v++ -l`), and packaging (-p).

For more information, see the Vitis Accelerated Software Development Flow Documentation in the Application Acceleration Development flow of the *Vitis Unified Software Platform Documentation* (UG1416).

## Embedded Flow

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

The development environment lets you create software applications using a unified set of Xilinx tools for the Arm Cortex®-A72 and Cortex®-R5F processors, as well as the Xilinx MicroBlaze™ processors. The environment provides the following methods to create applications:

- Bare-metal and FreeRTOS applications for MicroBlaze processors

- Bare-metal, Linux, and FreeRTOS applications for APU
- Bare-metal and FreeRTOS applications for RPU
- User customization of PLM, which is primarily used to boot Versal ACAP
- Library examples are provided with Vitis (ready to load sources and build), as follows:
  - OpenCV
  - OpenAMP RPC
  - FreeRTOS “HelloWorld”
  - LWIP
  - Performance tests (Dhrystone, memory tests, peripheral tests)
  - Cryptographic hardware engine access
  - eFUSE and BBRAM programming
  - PLM

After a hardware design is created in the Vivado IDE, you can export a block design along with hardware design and bitstream files to the Vitis tool export directory directly from the Vivado Project Navigator.

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

- `.xpr`: Vivado project file
- `psv_init.tcl`, `psv_init_gpl.c`, `psv_init.c`, `psv_init.h`: Contain information required during PLM creation
- `psv_init.html`: Versal ACAP register summary viewer
- `.xsa`: Xilinx Support Archive for the design.

The Vitis environment can also generate:

- Programmable device image (PDI) for secure and non-secure boot for the following processors:
  - Arm Cortex-A72
  - Cortex-R5F
  - MicroBlaze

The Vitis environment supports Linux application development and debugging.

For more information, see Vitis Embedded Software Development Flow Documentation in the *Vitis Unified Software Platform Documentation* (UG1416).

## Vitis Tools

The Vitis development environment provides the following tools for use in Xilinx embedded software development:

- **Xilinx System Debugger (XSDB):** Provides a command line interface to the Xilinx `hw_server`. The `hw_server` is the backend tool, that provides device/processor level debug capabilities to front-end tools such as XSDB, and the Vitis IDE. XSDB also provides various low-level debugging features not directly available in the Vitis development environment.
- **Flash programmer:** Used for programming the software application images into external flash devices including QSPI, OSPI, or eMMC.
- **Linker script generator:** Used for mapping your bare-metal/FreeRTOS application elf sections across the hardware memory space.
- **GNU compiler tool suite:** Includes tools such as the GNU compiler collection (GCC), AS, LD, and binutils that are used for compilation on all Xilinx processors.
- **Bootgen:** Used for generating the boot image known as the PDI.
- **Performance analysis tools:** Includes GPROF, TCF Profiler, and OProfile.
- **Debug/download tools:** Includes GNU debugger (GDB), XSDB, and flash writer.
- **Simulator:** QEMU
- **Xilinx software command-line tool (XSCT):** Xilinx Software Command-line Tool (XSCT) is an interactive and scriptable command-line interface to the Vitis IDE. As with other Xilinx tools, the scripting language for XSCT is based on the tools command language (Tcl). The tools helps to abstract away and group most of the common functions into logical wizards that even the novice can use.

  However, scriptability of a tool is also essential for providing the flexibility to extend what is done with that tool. It is particularly useful when developing regression tests that will be run nightly or running a set of commands that are used often by the developer.

  You can run XSCT commands interactively or script the commands for automation. XSCT supports the following actions:

  - Create hardware, domains, platform projects, system projects, and application projects
  - Manage repositories
  - Set toolchain preferences
  - Configure and build domains/BSPs and applications

- Download and run applications on hardware targets
- Create and flash boot images by running Bootgen and program_flash tools

This reference content is intended to provide the information you need to develop scripts for software development and debug targeting Xilinx processors.

For more information, refer to Xilinx Software Command-Line Tool in the Embedded Software Development flow of the *Vitis Unified Software Platform Documentation* (UG1416).

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

- **Design perspective views:** View, create and build the software C/C++projects. By default, it consists of an editor area and other views, such as Vitis projects, C/C++ projects to show the software projects present in the workspace, a navigation console, properties, tasks, make targets, outline, and search.
- **Debug view:** Helps debug software applications. You can customize the open source applications from the debug perspective and integrate with the Vitis environments.
- **Remote system explorer:** Connect and work with a variety of remote systems, for example, running Linux on a Versal device.

# 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 SoC devices, the solution works with the Xilinx hardware design tools to facilitate the development of open source Linux systems for Versal devices.

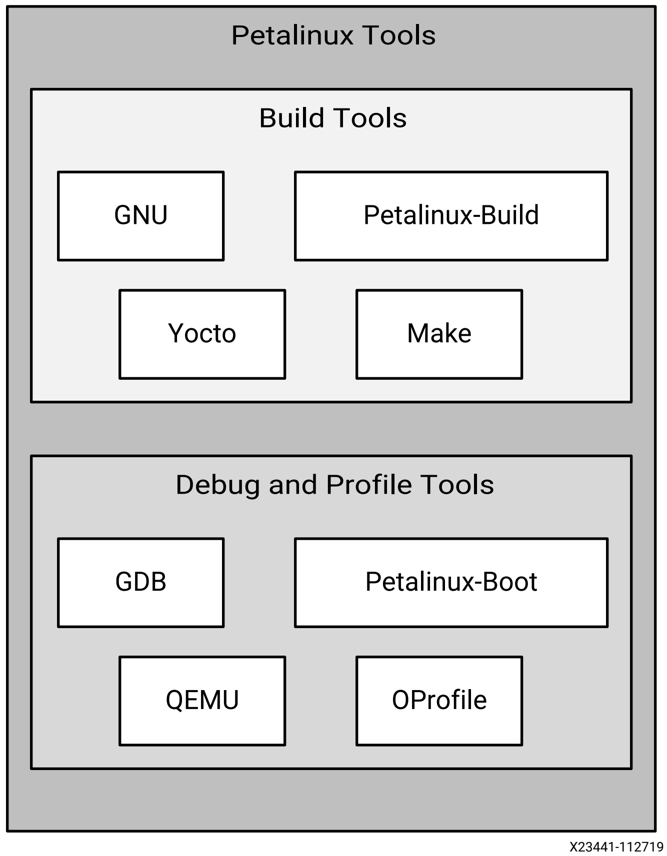
PetaLinux tools include the following:

- Build tools, such as GNU, `petalinux-build`, and make to build the kernel images and the application software.
- Debug tools, such as GNU debugger (GDB), `petalinux-boot`, and Oprofile for profiling. The following partial list shows the supported PetaLinux toolchain:
  - **GNU `gcc/g++` toolchain:** Xilinx Arm GNU tools.
  - `petalinux-build`: Used to build software image files.
  - **Make:** Make build for compiling the applications.
  - **GDB:** GDB tools for debugging.
  - `petalinux-package`: Used to create the boot image.
  - `petalinux-boot`: Used to boot Linux.

- **QEMU:** Emulator platform for the Versal device.
- **Oprofile:** Used for profiling.

The following figure shows the PetaLinux tool flow.

*Figure 4:* **PetaLinux Tool-Based Flow**




For more information, refer to the following:

- *PetaLinux Tools Documentation: Reference Guide* (UG1144)
- https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842250/PetaLinux on the Xilinx Wiki.

# Device Tree Generator

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

It is also possible to have various combinations of peripheral logic, each using a different configuration. For all the different combinations, the device tree generator (DTG) generates the .dts/.dtsi device tree files.

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

- `pl.dtsi`: Contains all the memory mapped peripheral logic (PL) IPs.
- `pcw.dtsi`: Contains the dynamic properties for the PS IPs.
- `system-top.dts`: Contains the memory, boot arguments, and command line parameters.
- `versal.dtsi`: Contains the PS-specific and CPU information.
- `versal-clk.dtsi`: Contains the clocks and power domain information for PS peripherals. For more information, see https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842279/Build+Device+Tree+Blob on the Xilinx Wiki.

# Open Source

The Arm GNU open source Linaro GCC toolchain is adopted for the Xilinx software development platform. The GNU tools for Linux hosts are available as part of the Vitis environment. This section details the open source GNU tools and Linux tools available for the processing clusters in Versal ACAPs.

## Xilinx Arm GNU Tools

The following Xilinx Arm GNU tools are available for compiling software for the APU, the RPU, and the embedded MicroBlaze processors:

- `arm-linux-gnueabihf-gcc`: Used for compiling Armv8 C code into 32-bit Linux applications with hard floating point instructions.
- `arm-linux-gnueabihf-g++`: Used for compiling Armv8 C++ code into 32-bit Linux applications.
- `aarch64-linux-gnu-gcc`: Used for compiling Armv8 C code into 64-bit Linux applications.
- `aarch64-linux-gnu-g++`: Used for compiling Armv8 C++ code into 64-bit Linux applications.
- `aarch64-none-elf-gcc`: Used for compiling Armv8 C code into 64-bit RTOS and bare-metal applications.
- `aarch64-none-elf-g++`: Used for compiling Armv8 C++ code into 64-bit RTOS and bare-metal applications.
- `armr5-none-eabi-gcc`: Used for compiling Armv7 C code into 32-bit RTOS and bare-metal applications.

- `armr5-none-eabi-g++`: Used for compiling Armv7 C++ code into 32-bit RTOS and bare-metal applications.
- `microblaze-xilinx-elf-gcc`: Used for compiling MicroBlaze™ C code.
- `microblaze-xilinx-elf-g++`: Used for compiling MicroBlaze C++ code.

***Note***: All GNU compilers are packaged with their associated assembler, linker, etc.

# Linux Software Development Using Yocto

Xilinx distributes open source, meta-xilinx Yocto/OpenEmbedded recipes on https://github.com/Xilinx, so you can work with in-house Yocto build systems to configure, build, and deploy Linux for Versal devices.

The meta-xilinx layer also provides a number of BSPs for Xilinx reference boards and vendor boards, such as ZC702, ZCU102, Ultra96, Zedboard, VCK190, and VMK180.

The meta-xilinx layer is compatible with Yocto/OpenEmbedded, adding recipes for various components. For more information, see https://git.yoctoproject.org/cgit/cgit.cgi/meta-xilinx/.

You can develop Linux software on the Arm Cortex-A72 processor using open source Linux tools. This section explains the Linux Yocto tools and its project development environment.

The following table lists the Yocto distribution.

*Table 2:* **Yocto Distribution**

| Distribution Type | Name | Description |
|---|---|---|
| Yocto Build System | 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 Packages | Perf | Profiling and tracing tool that comes bundled with the Linux Kernel. |
| | Ftrace | Refers to the ftrace function tracer but encompasses a number of related tracers along with the infrastructure used by all the related tracers. |
| | Oprofile | System-wide profiler that runs on the target system as a command-line application. |
| | Sysprof | System-wide profiler that consists of a single window with three panes, and buttons, which allow you to start, stop, and view the profile from one place. |
| | Blktrace | A tool for tracing and reporting low-level disk I/O. |

## Yocto Project Development Environment

The Yocto Project development environment allows Linux software to be developed for Versal ACAP through Yocto recipes distributed on https://github.com/Xilinx. You can use components from the Yocto project to design, develop, and build an open source-based Linux software stack.

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

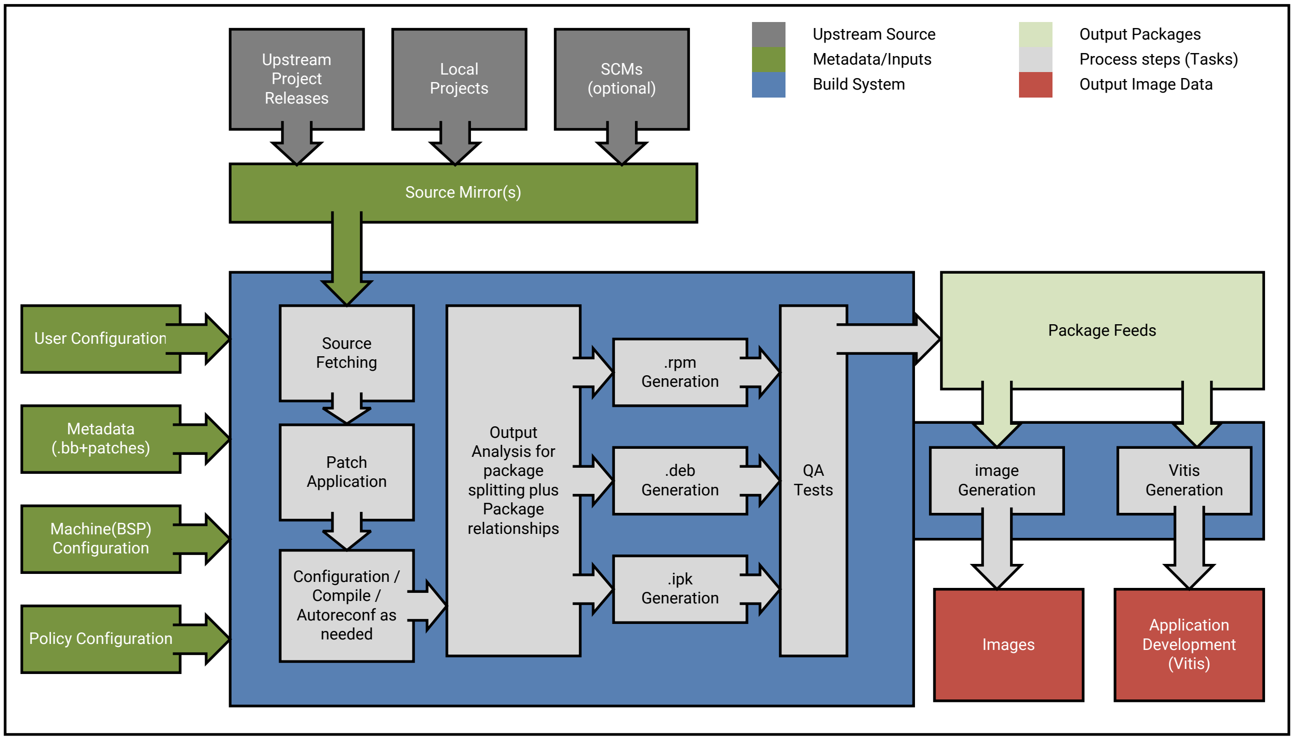
You can also change the build and hardware configuration through BSP.

Yocto combines a rich compiler and quality analyzing tools to build and test images. After the images pass the quality tests and package feeds required for Vitis tool generation are received, the Yocto distribution launches Vitis environment 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 want to use alternative UI frameworks, these components do not need to be installed.
- Creates a focused and stable core compatible with the OpenEmbedded project to easily and reliably build and develop Linux software.
- Supports a wide range of hardware and device emulation through the QEMU.

*Figure 5:* **Yocto Project Development Environment**




You can download the Yocto tools and the Yocto Project development environment from the https://www.yoctoproject.org/software-overview/downloads/.

For more information about Xilinx-supported Yocto features, see Yocto Features in *PetaLinux Tools Documentation: Reference Guide* (UG1144) and the https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841883/Yocto on the Xilinx Wiki.

# QEMU

QEMU is a fast functional and instruction accurate emulator that can be installed on both Linux and Windows hosts. The Xilinx QEMU features a large number of the same peripheral models as Xilinx silicon and can even inject real data from the host into emulation (for example, Ethernet, UART, storage, CAN-FD, etc.) to stimulate your applications. Xilinx has provided QEMU as a development platform for many generations including Versal ACAP, Zynq UltraScale+ MPSoC, Zynq-7000, and MicroBlaze processors.

The Xilinx QEMU is already integrated in Vitis, Petalinux and Yocto in every Xilinx release. For the most up to date development branch, you can download Xilinx QEMU, build and install from source.

To use this emulation platform you must be familiar with:

- Device architecture
- GNU debugger (GDB) for remote debugging
- Generation of guest software applications using Yocto, Xilinx PetaLinux and Vitis tools
- Linux Device Trees

This document focuses on the list of features supported for the emulation of Versal ACAP in Xilinx QEMU. The purpose of this section is to familiarize software application developers, system software developers, system hardware designers, and validation/verification designers with the basic information to use and debug software with QEMU.

### QEMU Model for Versal ACAP

The Xilinx Versal ACAP QEMU model supports the following resources:

- Central Processing Units (CPUs)
    - Application Processing Unit (APU)
        - 2 x Arm Cortex-A72 processor
    - Real-time Processing Unit (RPU)
        - 2 x Arm Cortex-R5F processor
    - Platform Management Controller (PMC)
        - Platform processing unit (PPU) 0 (MicroBlaze processor)
        - Platform processing unit (PPU) 1 (MicroBlaze processor)
    - PS management controller (PSM)
        - 1 x MicroBlaze processor
- Memory
    - On-chip memory (OCM)
    - Tightly coupled memory (TCM)
    - DDR
- Security Modules
    - Device Key Storage
        - Battery Backed Random Access Memory (BBRAM)
        - eFUSE (Electronic Fuse)
        - Physical Unclonable Function (PUF - API Only)

- Crypto Primitives
  - Advanced Encryption Standard - Galois/Counter Mode (AES-GCM)
  - Secure Hash Algorithm 3 (SHA-3)
  - RSA-4096
  - Elliptic Curve Digital Signature Algorithm (ECDSA)

- Peripheral and Controllers
  - 2 x Serial Peripheral Interface (SPI)
  - Octal SPI (OSPI)
    - OSPI DMA
  - 2 x SD
  - eMMC
  - 2 x CANFD
  - 2 x UART
  - 3 x Inter-Integrated Circuit (I2C)
    - 2 in PS
    - 1 in PMC
  - 2 x Gigabit Ethernet
  - 4 x triple timer counter (TTC)
  - Inter-processor interrupt (IPI)
  - (Xilinx Memory Protection Unit (XMPU)
  - (Xilinx Peripheral Protection Unit (XPPU)
  - System Memory Management Unit (SMMU)
  - General interrupt controller (GIC) v3
  - Direct memory access (DMAs)
  - Real-Time Clock (RTC)
  - USB (Host Mode Only)

### Using the QEMU

For more information on using the QEMU, see the *Xilinx Quick Emulator User Guide: QEMU*.

# AI Engine Development Environment

The Versal AI Core series delivers breakthrough AI inference acceleration with AI Engines that deliver over 100x greater compute performance than current server-class of CPUs. This series is designed for a breadth of applications, including cloud for dynamic workloads and network for massive bandwidth, all while delivering advanced safety and security features. AI and data scientists, as well as software and hardware developers, can all take advantage of the high compute density to accelerate the performance of any application. Given the AI Engine's advanced signal processing compute capability, it is well-suited for highly optimized wireless applications such as radio, 5G, backhaul, and other high-performance DSP applications.

AI Engines are an array of very-long instruction word (VLIW) processors with single instruction multiple data (SIMD) vector units that are highly optimized for compute-intensive applications, specifically digital signal processing (DSP), 5G wireless applications, and artificial intelligence (AI) technology such as machine learning (ML).

AI Engines provide multiple levels of parallelism including instruction-level and data-level parallelism. Instruction-level parallelism includes a scalar operation, up to two moves, two vector reads (loads), one vector write (store), and one vector instruction that can be executed—in total, a 7-way VLIW instruction per clock cycle. Data-level parallelism is achieved via vector-level operations where multiple sets of data can be operated on a per-clock-cycle basis. Each AI Engine contains both a vector and scalar processor, dedicated program memory, local 32 KB data memory, access to local memory in any of three neighboring directions. It also has access to DMA engines and AXI4 interconnect switches to communicate via streams to other AI Engines or to the programmable logic (PL) or the DMA. Refer to the *Versal ACAP AI Engine Architecture Manual* (AM009) for specific details on the AI Engine array and interfaces.

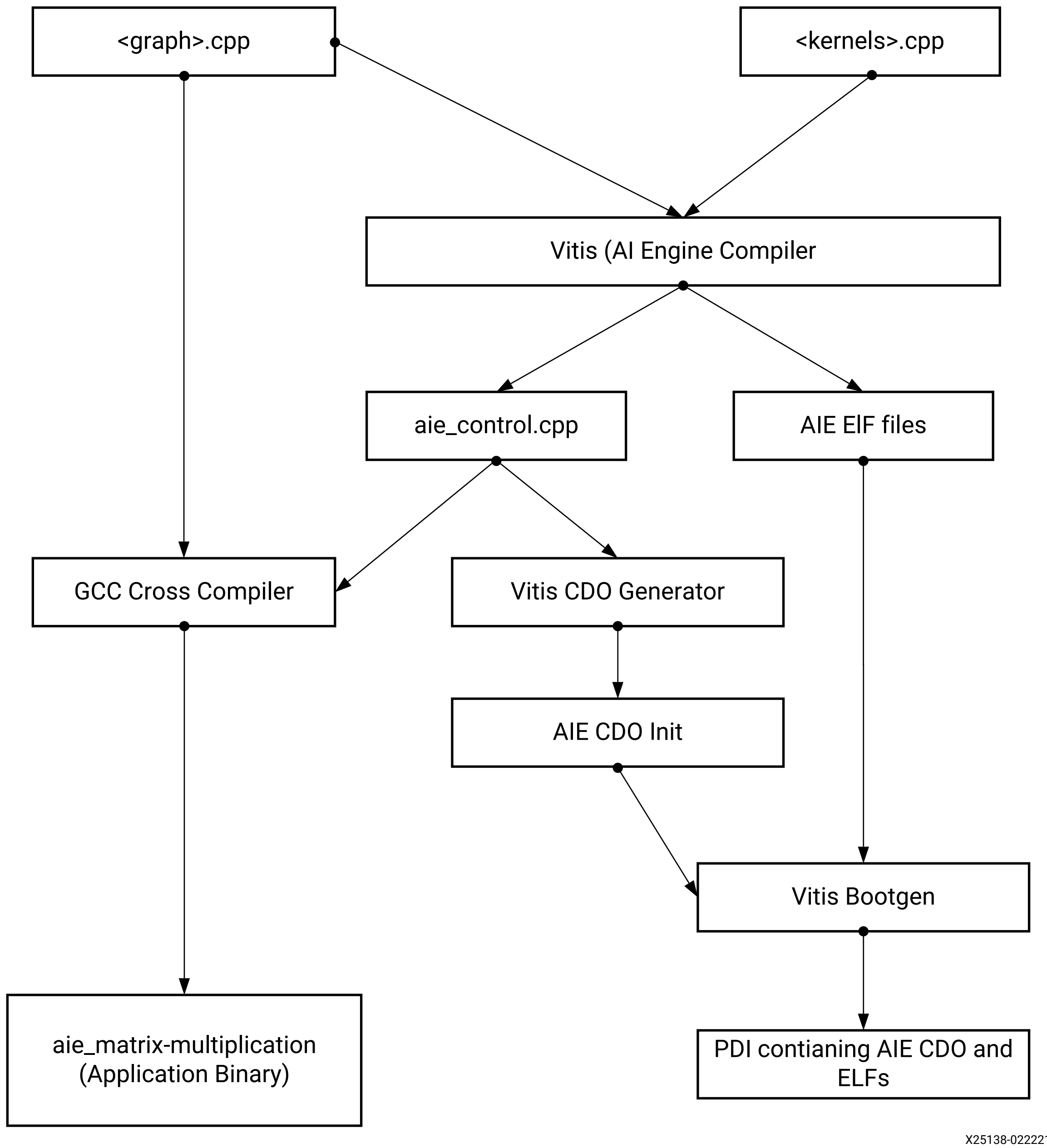
Xilinx® provides device drivers and libraries for user applications to access the Versal™ AI Engines. See *Versal ACAP AI Engine Programming Environment User Guide* (UG1076) to learn how to use the Xilinx® AI Engine program environment. The Vitis IDE lets you compile, simulate, and debug the different elements of a Versal ACAP AI Engine application. For detailed information on the Vitis IDE tool flow, refer to *Versal ACAP Design Guide* (UG1273). For detailed information on the Vitis AL Engine Tools and Flows, refer to *Versal ACAP AI Engine Programming Environment User Guide* (UG1076).

The following sections describe the software stack of the AI Engine.

## AI Engine Software Development Flow

The AI Engine application comprises kernels that run on the AI Engine tiles, and a control application that runs on the CPU. The following figure shows the flow of the AI Engine control application and ELFs generation.

*Figure 6:* **AI Engine Software Development Flow**



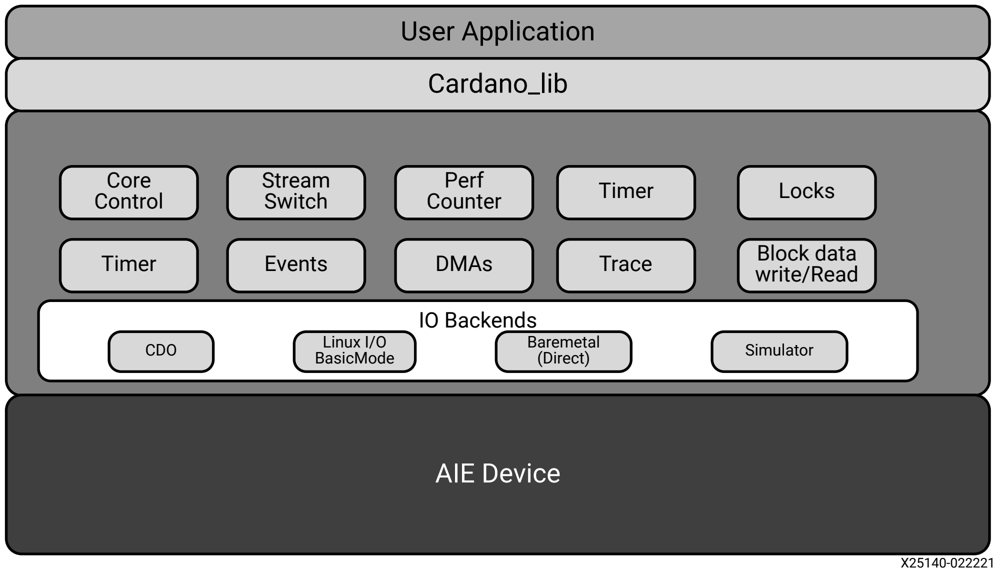

You define the `<graph>.cpp` and `<kernels>.cpp`. The AI Engine compiler takes these inputs to generate `aie_control.cpp`, which runs on either the APU or RPU to configure and monitor the AI Engine and the ELF files that run on the AI Engine tiles.

## AI Engine Runtime Stack

Your PS applications can call system libraries to access AI Engine registers and load the kernel ELF files.

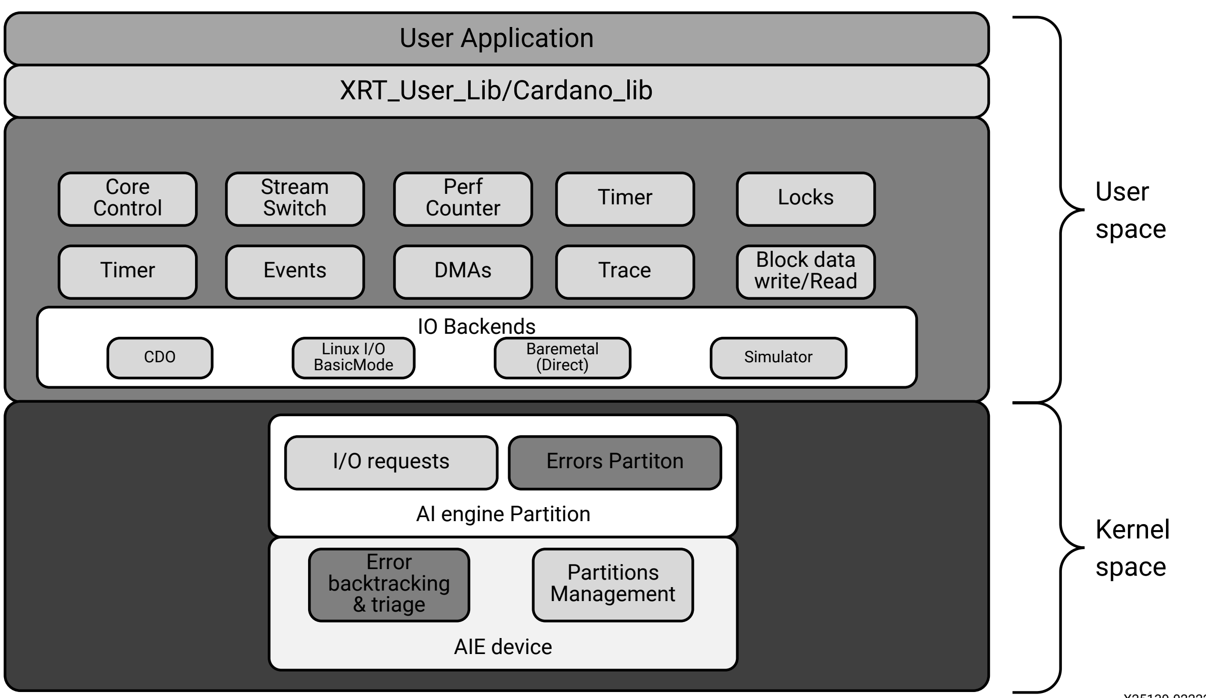
- **Bare-Metal:** Your application can use the `cardano_api.a` library to access AI Engine registers. The `cardano_api.a` library calls `aie_control` and the AI Engine driver, to access the AI Engine register and device memory.

*Figure 7:* **AI Engine Bare-Metal Runtime Stack**




- **Linux:** On Linux, an application can use AI Engine driver APIs or XRT APIs to interact with the AI Engine.

*Figure 8:* **AI Engine Linux Runtime Stack**



# Software Stack

This chapter provides an overview of the bare-metal, Linux, and FreeRTOS software stacks available for the Versal™ devices. Many third-party software stacks can also be used on Versal devices. For more information, refer to Embedded Software & Ecosystem page.

For more information about various software development tools used with the device, see Chapter 3: Development Tools.

For more information about bare-metal and Linux software application development, see Chapter 5: Software Development Flow.

## Bare-Metal Software Stack

Xilinx® provides a bare-metal software stack as part of the Vitis™ tools. The standalone software includes a simple, single-threaded environment that provides project domains such as standard input/output, and access to processor hardware features. The included board support packages (BSPs) and included libraries can be configured to provide the necessary functionality with the least overhead. You can locate the standalone drivers at the following path:

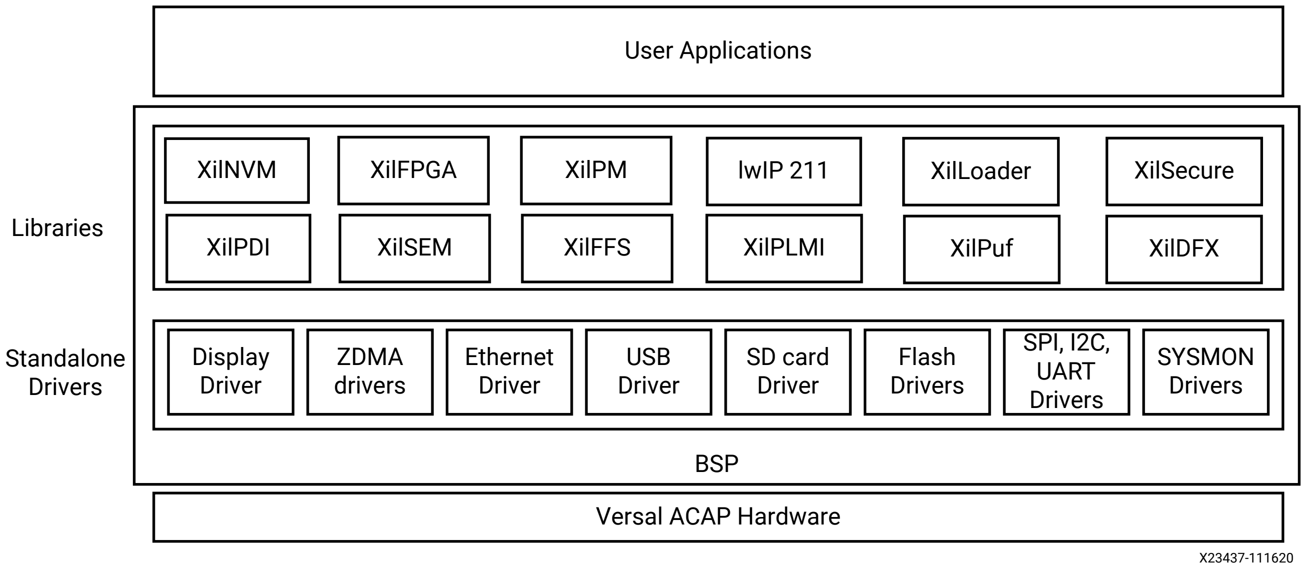
```
<Xilinx Installation Directory>\Vitis\<version>\data\embeddedsw
\XilinxProcessorIPLib\drivers
```

You can locate libraries in the following path:

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

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

*Figure 9:* **Bare-Metal Software Development Stack**




***Note*:** The software stack of libraries and drivers layer for bare-metal in the RPU is same in the APU.

The bare-metal stack key components include:

- Software drivers for peripherals including core routines needed for using the Arm® Cortex®-A72, and the Cortex-R5F processors in the PS, and MicroBlaze™ processors in the PL.
- bare-metal drivers for PS peripherals and optional PL peripherals.
- Standard C libraries: libc and libm, based on the open source Newlib library, ported to the Cortex-A72, Cortex-R5F, and the MicroBlaze processors.
- Additional middleware libraries that provide networking, file system, and encryption support.
- Application examples include test applications.

# The C Standard Library (libc)

The libc library contains standard functions that C programs can use. The following header files are included in the libc library:

- `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 go to code
- `stdint.h`: Standard integer types
- `stdio.h`: Standard I/O facilities
- `stdlib.h`: General utilities functions
- `time.h`: Time function

## The C Standard Library Mathematical Functions (libm)

The following table lists the libm mathematical C modules.

*Table 3:* **libm Modules by Function Types and 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

Standalone BSP is a simple, low-level software layer. This layer provides access to basic processor features, such as caches, interrupts and exceptions and the basic features of a hosted environment, such as standard input and output, profiling, abort, and exit.

The following libraries are a small number of those available with standalone BSP:

- Xilinx fat file system (XilFFS) library
- XilMailbox library
- Xilinx Power Management (XilPM)
- Xilinx Soft Error Mitigation (XilSEM) library
- Lightweight IP (lwIP)

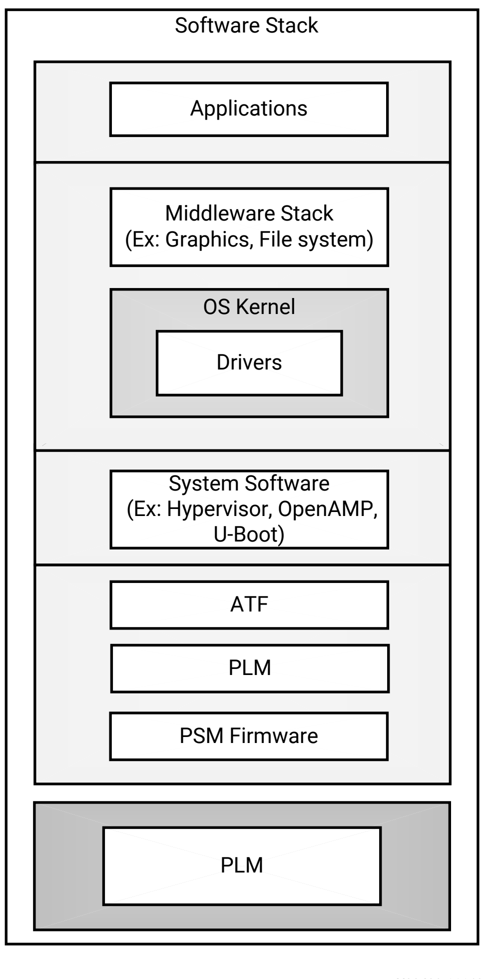
For more information on the available libraries, refer to Appendix A: Libraries and *OS and Libraries Document Collection* (UG643).

# Linux Software Stack

***Note***: The Arm AArch64 architecture is common between the Zynq UltraScale+ MPSoC APU and the Versal ACAP APU. The existing architectural reference nomenclature "ZynqMP" found the Linux source also applies to Versal devices.

The Linux OS supports the Versal device. Xilinx provides open source drivers for all peripherals in the PS, and key peripherals in the PL. The following figure illustrates the full software stack in the APU, including Linux and an optional hypervisor.

*Figure 10:* **Linux Software Development Stack**



Xilinx offers two tools to build and deploy custom Linux distributions for Versal devices: PetaLinux tools and the open source collaboration project, Yocto project. For more information, refer to https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841996/Linux on the Xilinx Wiki.

- **PetaLinux tools:** The PetaLinux tools offer a simplified commands set to quickly build embedded Linux. The tools include a branch of the Linux source tree, U-Boot as well as Yocto-based tools to make it easy to build complete Linux images, including the kernel, the root file system, device tree, and applications for Xilinx devices. The PetaLinux tools work with the all the same open source Linux components described immediately below. More information about using PetaLinux to build custom distributions can be found on the PetaLinux Tools page and https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842475/PetaLinux+Yocto+Tips on the Xilinx Wiki.
- **Yocto Project:** The Yocto Project can be used by more experienced users to highly customize embedded Linux for their boards. For those interested in the Yocto Project, the Xilinx Wiki has several articles and information pertaining Yocto for building Linux on Xilinx devices. The https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841883/Yocto on the Xilinx Wiki is a great place to start. Yocto board support packages are also available from the main Yocto tree.

You can leverage the Linux software stack for the Versal device in multiple ways.

- **Open Source Linux and U-Boot:** Xilinx offers release-specific prebuilt images for the Versal ACAP kits, VMK180 and VCK190 that can be found on the Versal ACAP Boards, Kits, and Modules page. The Linux Kernel sources including drivers, board configurations, and U-Boot updates for Versal devices are available from the https://github.com/Xilinx/linux-xlnx/, as well as, from the main Linux kernel and U-Boot repositories. For more information, refer to https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/115048462/Release+Notes+for+Open+Source+Components on the Xilinx Wiki.
- **Commercial Linux Distributions:** Along with open source Linux offerings, Xilinx works with several third-parties to offer other Linux solutions. Some commercial distributions also include support for the Versal devices, and they include advanced tools for Linux configuration, optimization, and debug. For more information, refer to the Embedded Software & Ecosystem page.

## FreeRTOS Software Stack

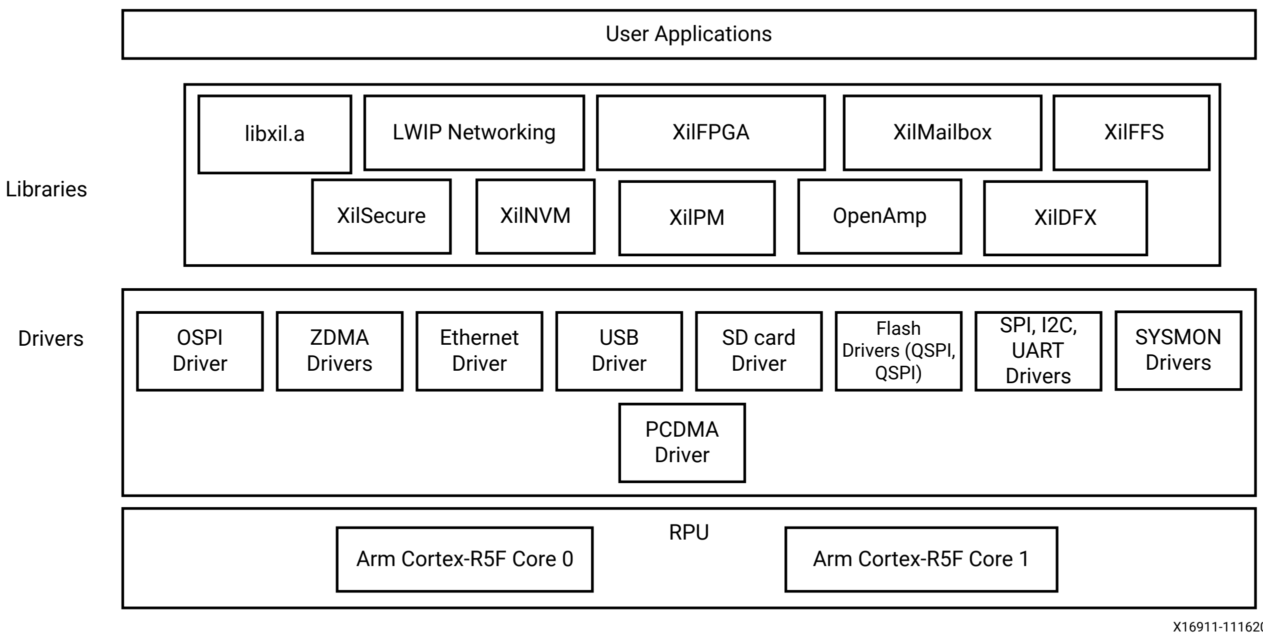
In a simpler RTOS, the RTOS itself is like a library that is linked together with the application and RTOS running in the same domain. In a more advanced RTOS, an optional process mode (e.g., VxWorks real-time processes, Nucleus Process Model, and QNX processes) is available with the same decoupling of applications and kernels as Linux. Xilinx provides a FreeRTOS BSP as a part of the Vitis software platform. The FreeRTOS BSP provides you a simple, multi-threading environment with basic features such as, standard input/output and access to processor

hardware features. The BSP and the included libraries are highly configurable to provide you the necessary functionality with the least overhead. The FreeRTOS software stack is similar to the bare-metal software stack, except that it contains the FreeRTOS library. Xilinx device drivers included with the standalone libraries can typically be used within FreeRTOS provided that only a single thread requires access to the device.

> **IMPORTANT!** *Xilinx bare-metal drivers are not aware of OS. They do not provide any support for mutexes to protect critical sections, and they do not provide any mechanism for semaphores for use during synchronization. While using the driver API with FreeRTOS kernel, you must take care of this aspect.*

The following figure illustrates the FreeRTOS software stack for the RPU.

*Figure 11:* **FreeRTOS Software Stack**




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

# Third-Party Software Stack

Many other embedded software solutions are also available from the Xilinx partner ecosystem. For more information, refer to:

- Embedded Software & Ecosystem page
- Xilinx Third Party-Tools page
- https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842463/3rd+Party+Operating+Systems on the Xilinx Wiki

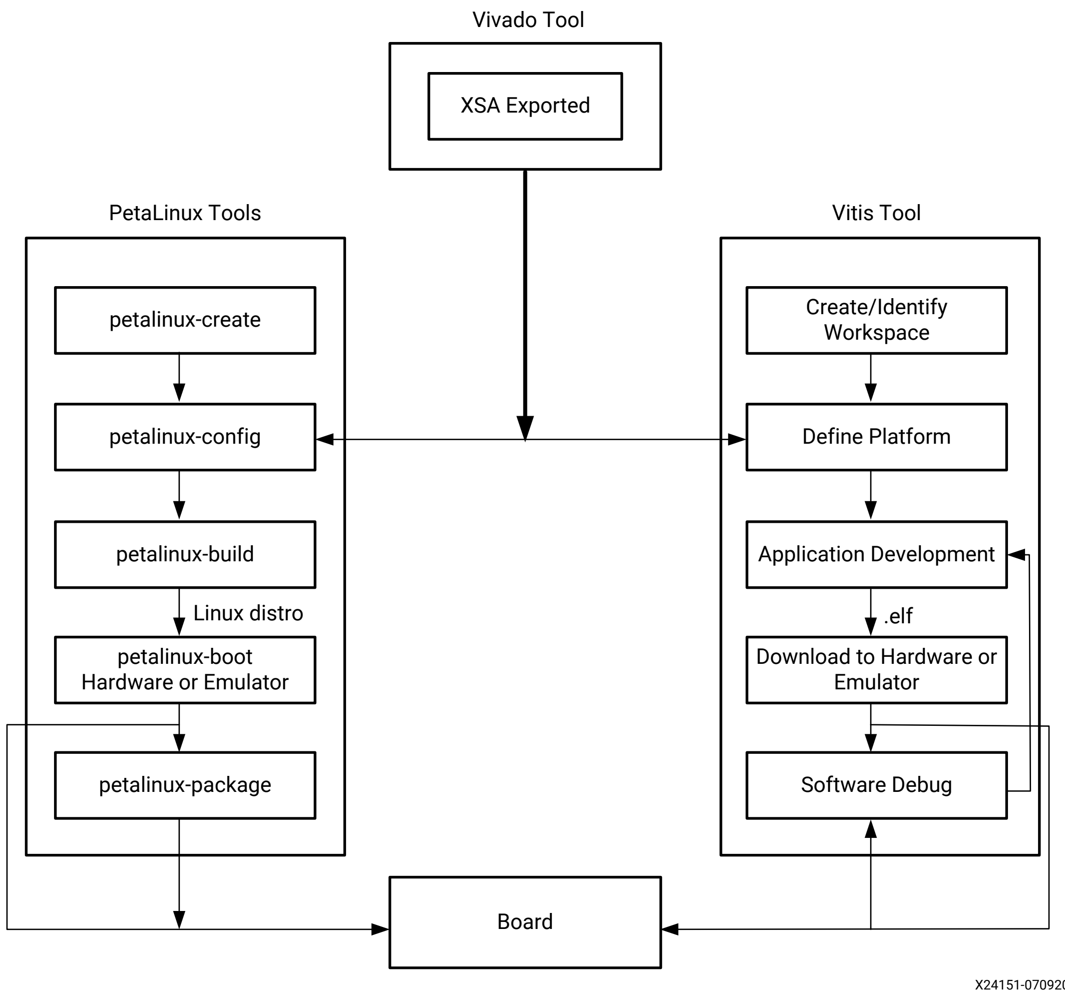
Chapter 5

# Software Development Flow

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

The following figure shows the basic software development flow.

*Figure 12:* **Software Development Flow**

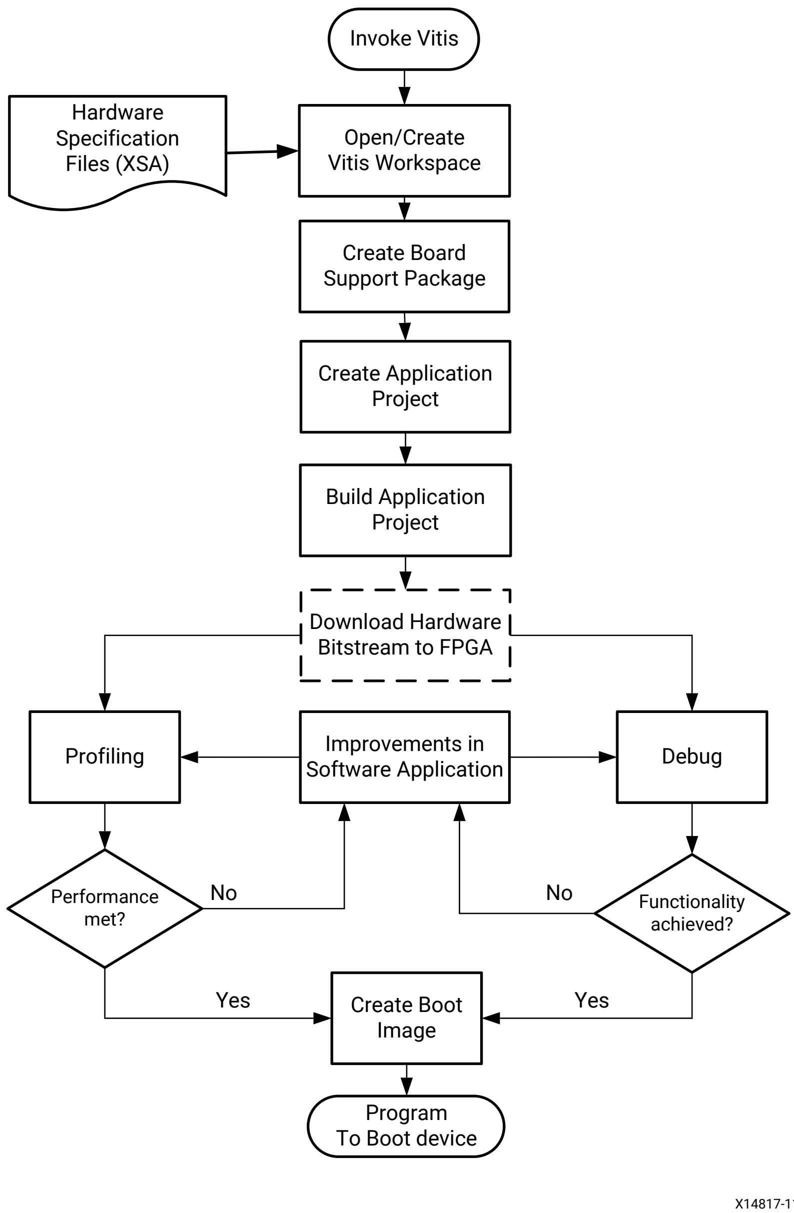



For more information, refer to Vitis Embedded Software Development Flow Documentation in the *Vitis Unified Software Platform Documentation* (UG1416)

# Bare-Metal Application Development in the Vitis Environment

The following figure shows the bare-metal application development flow.

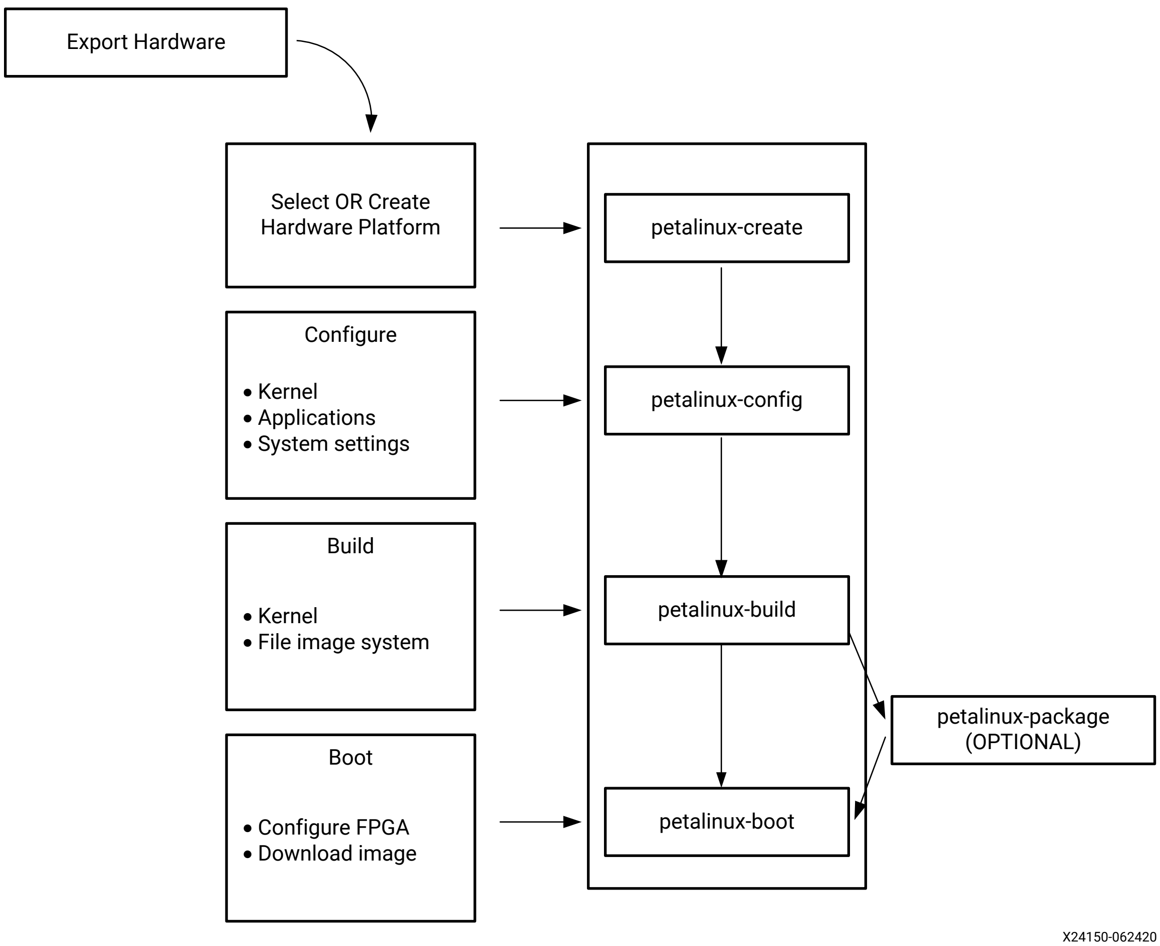
*Figure 13:* **Bare-Metal Application Development Flow**



# Linux Application Development Using PetaLinux Tools

The software development flow in the PetaLinux tools environment involves many stages. The following figure shows the primary stages within the PetaLinux tools application development flow.

*Figure 14:* **High-Level PetaLinux Tool-Based Software Development Flow**




The "Create Linux Images Using the PetaLinux" section in the *Xilinx Embedded Design Tutorials: Versal Adaptive Compute Acceleration Platform* (UG1305) shows you how to configure and build the Linux OS platform for Arm® Cortex®-A72 core-based APU on a Versal device, using the PetaLinux tools, along with the board-specific BSP.

For more information, refer to the Creating A New Project, Configure and Build the Project, and Boot and Packing the Project sections in *PetaLinux Tools Documentation: Reference Guide* (UG1144).

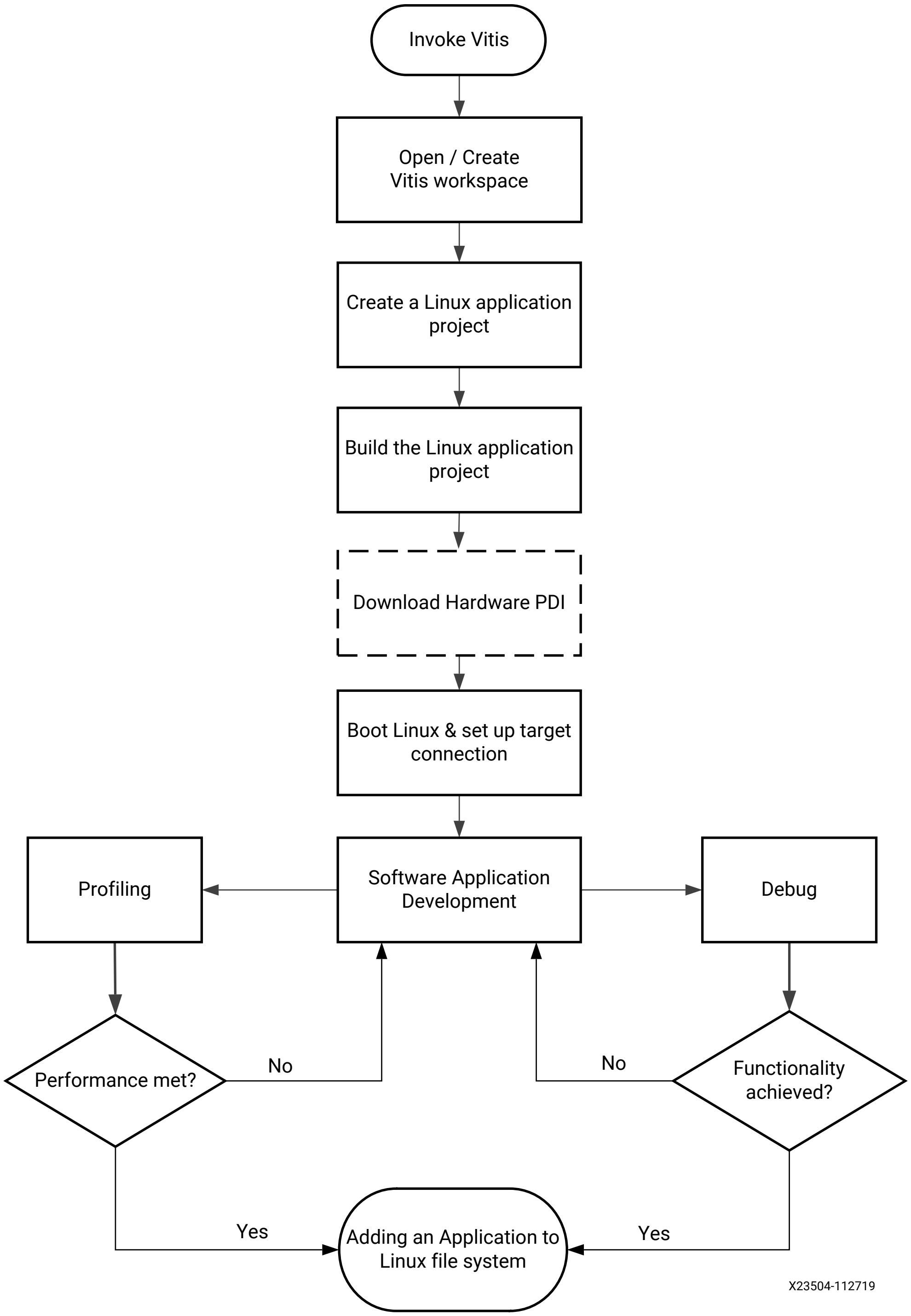
# Linux Application Development Using the Vitis Software Platform

The Vitis integrated design environment (IDE) facilitates the development of Linux user applications. This section provides an overview of the Linux application development flow from the Vitis tool.

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

***Note*:** These steps work only when the Vitis platform is built with the Linux domain being available.

*Figure 15:* **Develop Linux User Applications**



To complete the Linux application development flow from the Vitis tool which includes creating a software application, creating and building a sample project application, and debugging the application, see the *Xilinx Embedded Design Tutorials: Versal Adaptive Compute Acceleration Platform* (UG1305).

- For additional information on the Linux kernel and boot sequence, see Chapter 3: Development Tools.
- For more information, refer to *Creating and Adding Customs Modules* in *PetaLinux Tools Documentation: Reference Guide* (UG1144)

# Software Design Paradigms

The Versal™ device architecture supports heterogeneous multiprocessor engines targeted at different tasks. The main approaches for developing software to target these processors are by using the following:

- **Frameworks for Multiprocessor Development:** Describes the frameworks available for development on the Versal device.
- **Symmetric Multiprocessing:** Using SMP with PetaLinux is the most simple flow for developing an SMP design with a Linux operating system for the Versal device.
- **Asymmetric Multiprocessing :** 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.

## Frameworks for Multiprocessor Development

Xilinx® provides multiple frameworks to facilitate application development on Versal ACAP as follows:

- **Hypervisor Framework:** Xilinx supports the Xen hypervisor, a critical item needed to support virtualization on the Versal ACAP. For details, refer to Use of Hypervisors.
- **Security Framework:** The Versal device supports authentication, encryption, and other cryptographic features as a part of the security framework. To understand more about the security framework, see the Chapter 9: Security chapter.
- **TrustZone Framework:** TrustZone technology allows and maintains isolation between secure and non-secure hardware and software within the same system.

  Xilinx provides TrustZone support through the Arm® trusted firmware (ATF) to maintain isolation between secure and non-secure worlds. If implementing a trusted execution environment (TEE) on a Versal device, ATF is one of the major components of a TEE. See this whitepaper for an overview of a TEE architecture.
- **Multiprocessor Communication Framework:** Xilinx provides the OpenAMP framework for the Versal device to allow communication between the different processing units.

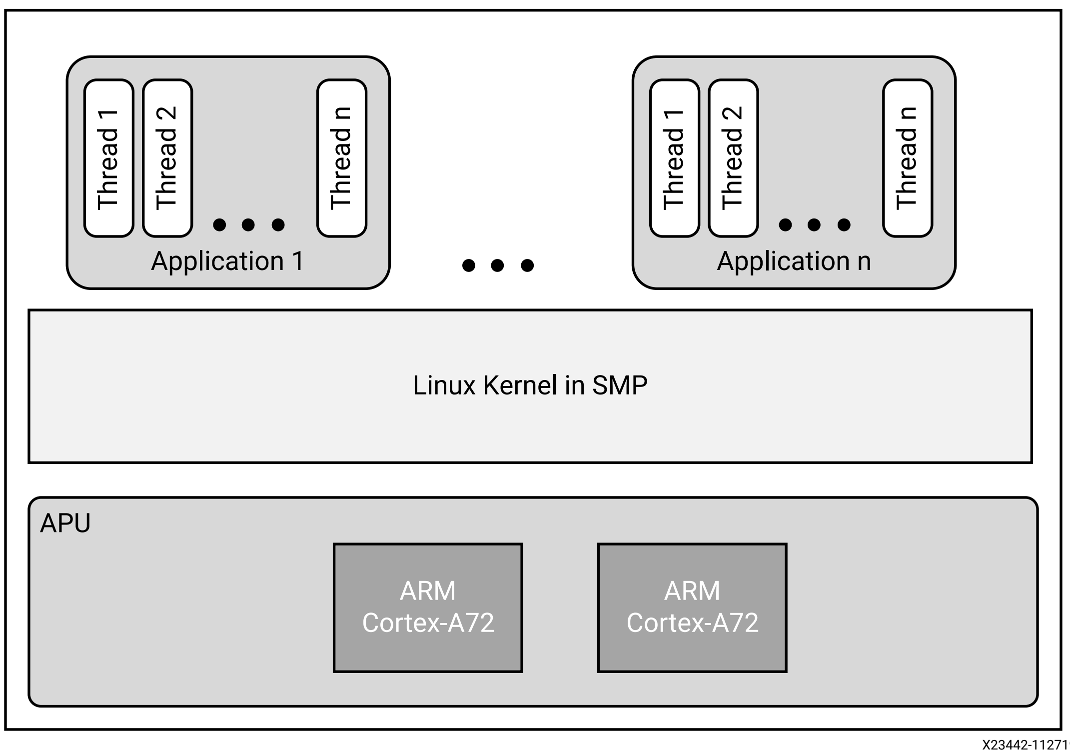
- **Power Management Framework:** The power management framework allows software components running across different processing units to communicate with the power management unit.

# Symmetric Multiprocessing

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

The APU in the Versal devices contains two homogeneous cache coherent Arm Cortex®-A72 processors that support SMP mode of operation using an OS, such as Linux or VxWorks®. Xilinx and its partners provide numerous operating systems that make it easy to leverage SMP in the APU. The following figure shows an example of Linux SMP with multiple applications running on a single OS.

*Figure 16:* **Example SMP Using Linux**

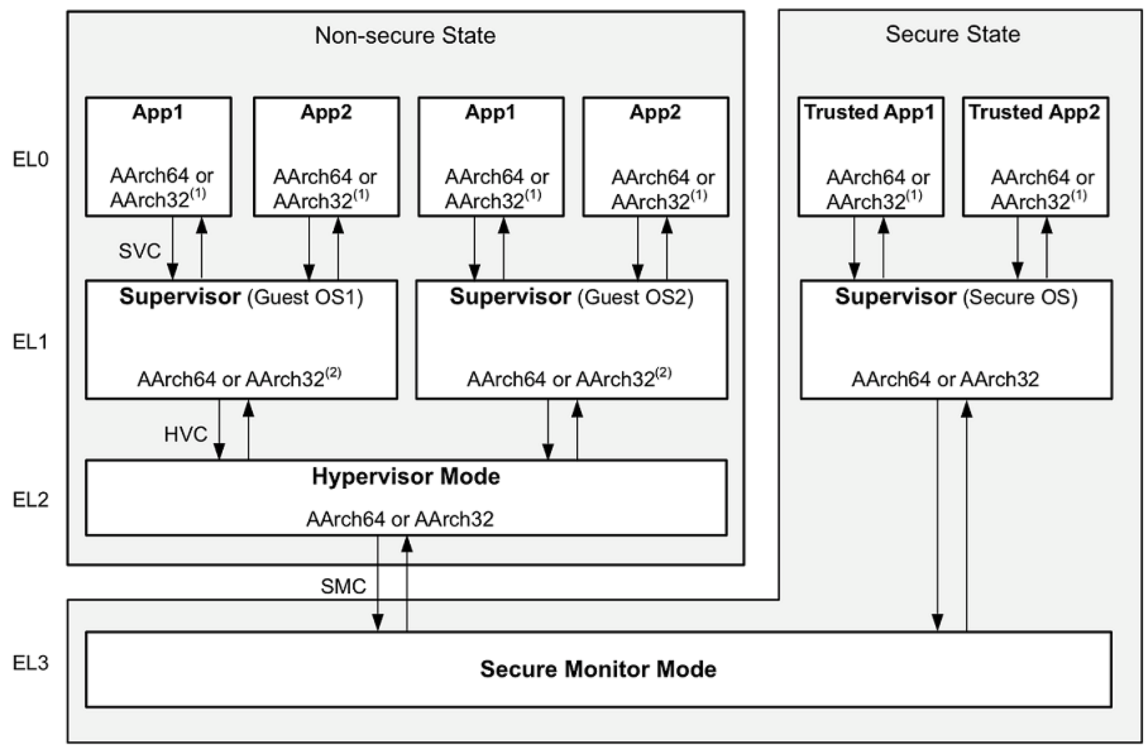



This might not be the best mode of operation when there are hard, real-time requirements because it ignores Linux application core affinity, which must be available to developers with the existing Xilinx software.

## Execution Domain and Images

A domain is a separate execution environment with its own address space and potentially attached devices. On a simple CPU, such as a minimal MicroBlaze processor without a memory management unit (MMU), there is only one domain. On more advanced CPUs, such as the APU, there are multiple domains, isolated by exception levels 0-3 (EL0-3) and some of those can have a segmented secure domain (SEL0-1) as shown in the following figure.

*Figure 17:* **Segmented Secure Domains on the APU**




The following table shows examples of what can run in the various domains on an APU.

*Table 4:* **Domain Examples**

| Domain | Non-Secure | Secure (TrustZone) |
|---|---|---|
| EL0 | Linux process/application<br>RTOS application process model | Secure or trusted application |
| EL1 | Linux kernel<br>RTOS kernel | OP-TEE OS |
| EL2 | Optional hypervisor (Xen), U-Boot | Future Arm processors will support this |
| EL3 | ATF | |

For CPU-like execution engines (APUs, RPUs, MicroBlaze processors, AI Engine, and etc.) an image is the compiled program that runs in a domain. Typically, the standard ELF format is used, but in some cases, it is converted to more compact formats such as a PDI.

The term, *image*, is also used for the "code/logic" running in the PL/FPGA. The PL image or bitstream is an overloaded term as it refers both to configuration information (e.g., initiating a device) as well as the customer code translated to PL fabric configuration.

The images are loaded into their domains in a few different ways:

- **During Boot:**
    - A PDI file in QSPI, SD, eMMC, SMAP, or OSPI
        - BootROM handles loading the PLM, while the PLM will handle loading the rest of images. Additionally, U-Boot will handle loading a hypervisor, if used, as well as an OS, such as Linux.
    - A JTAG debugger can place an image into memory.
- **During Run Time:**
    - An operating system can load an image from a filesystem (rootfs, remote filesystem, SD card, etc.):
        - Another process (fork/exec)
        - Another CPU (through OpenAMP)
    - A hypervisor could load another virtual machine (VM) and its associated set of software.
    - More advanced RTOSs with access to a filesystem can load images much like Linux.
    - PLM firmware can load an image dynamically during restart or at the request of a client.

Most of the time, an image is loaded into a domain and while the data in the image changes, the code does not change until the end of the lifecycle. There are exceptions to this, most notably when Linux dynamically loads a driver as a kernel module, when a DFX region is loaded into the PL, or when a firmware update occurs.

An application can have multiple images attached to it. These multi-image applications are used for example when a regular Linux process is using an accelerator that needs to be loaded at the same time.

# Asymmetric Multiprocessing

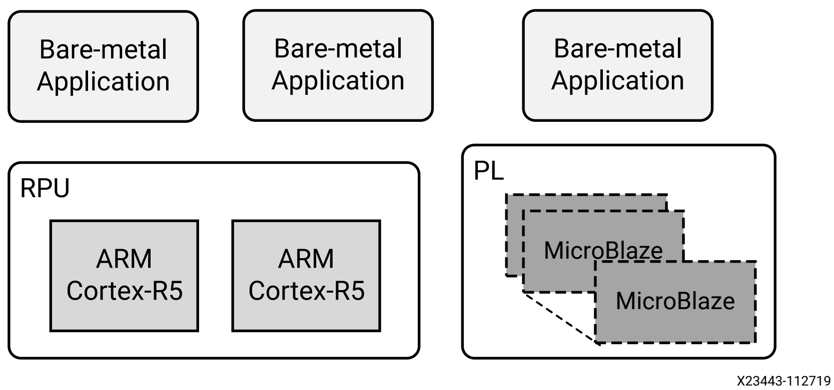
AMP uses multiple processors with precise control over what runs on each processor. Unlike SMP, there are different ways to use AMP. This section describes two ways of using AMP with varying levels of complexity.

In AMP, a software developer must decide what code has to run on each processor before compiling and creating a boot image. This includes both the APU and RPU processor clusters. For example, using AMP with the Arm Cortex-R5F processors (SMP is not supported) in the RPU enables developers to meet highly demanding, hard real-time requirements.

You can develop applications independently and program those applications to communicate with each other using inter-processing communication (IPC) options.

You can also apply this AMP method to applications running on the MicroBlaze processors in 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 to each other.

*Figure 18:* **AMP Example Using Bare-metal Applications Running in the RPU and PL**




## OpenAMP

Xilinx participates in OpenAMP, which is an open source project that provides support for APU and RPU communication. This communication path provides essential features that allow usage of the entire Versal ACAP. For example, using OpenAMP, the APU can load and unload the RPU software, and reset the RPU as needed.

The OpenAMP framework provides software components that enable development of software applications for AMP systems. The framework provides the following key capabilities:

- Provides lifecycle management and inter processor communication capabilities for management of remote compute resources and their associated software contexts.
- Provides a standalone library usable with RTOS and bare-metal software environments.
- Compatibility with upstream Linux remoteproc and RPMsg components

The OpenAMP supports the following configurations:

- Linux master/generic (bare-metal) remote
- Generic (bare-metal) master/Linux remote

- Generic (bare-metal) master/generic (bare-metal) remote

Proxy infrastructure and supplied demos showcase the ability of proxy on master to handle printf, scanf, open, close, read, and write calls from bare-metal based remote contexts.

For more advanced features, such as enhanced system management or higher level communications APIs, you might find helpful content within the OpenAMP community project, whose software targets Xilinx SoCs and others.

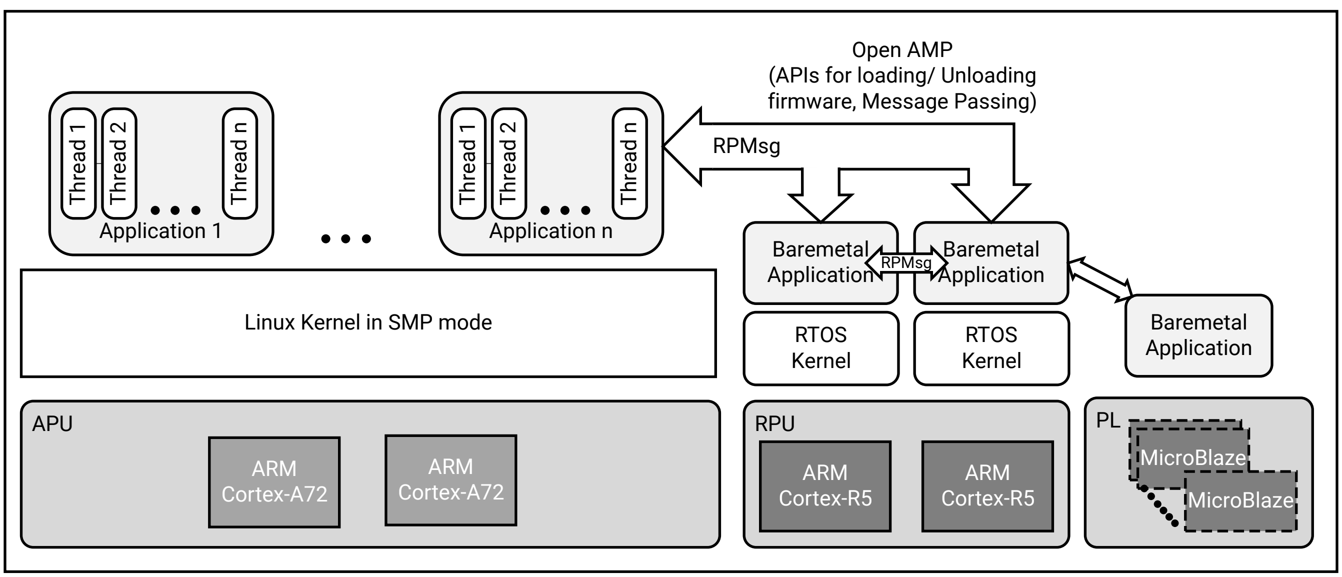
The OpenAMP framework provides mechanisms to do the following:

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

  The following diagram shows an example of an OpenAMP architecture on a Versal device.

  In this case, Linux applications running on the APU communicate with RPU through RPMsg protocol. Linux applications can load and unload applications on RPU with Remoteproc framework. This allows developers to load various dedicated algorithms to the RPU processing engines as needed with very deterministic performance. Notice that this architecture uses a mixture of SMP and AMP modes.

*Figure 19:* **Example with SMP and AMP Using OpenAMP Framework**




X23479-112719

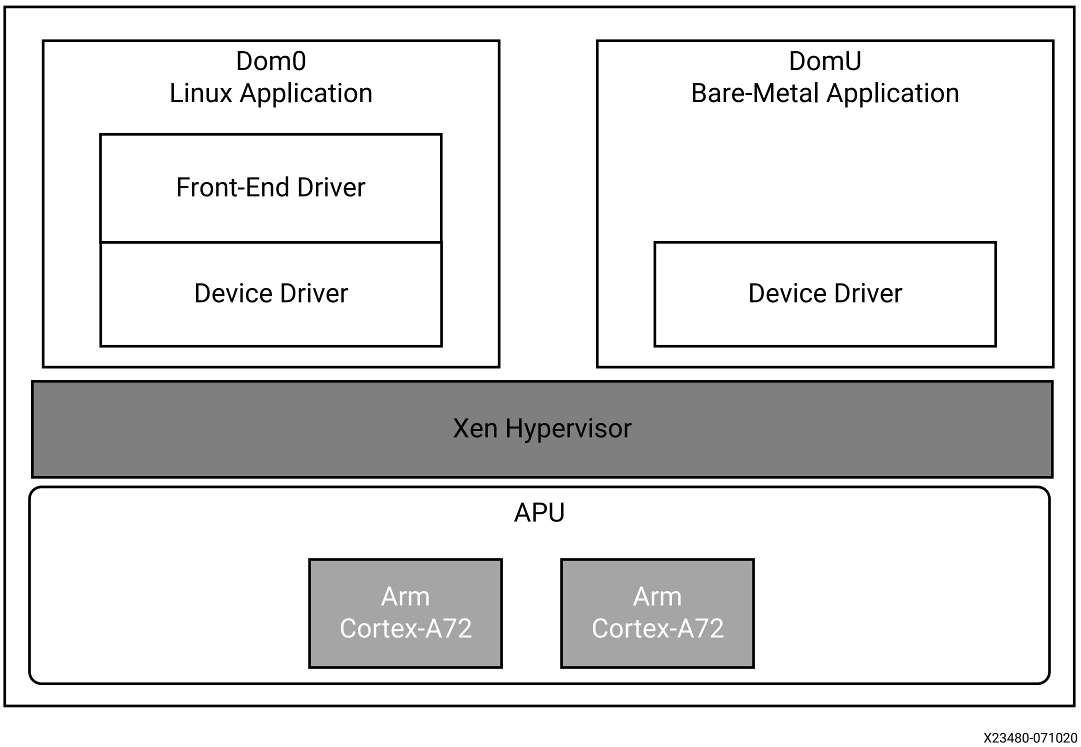
For more information, see https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841718/OpenAMP on the Xilinx Wiki.

## Virtualization with Hypervisor

Versal devices have hardware virtualization extensions on the Arm Cortex-A72 processors, Arm GIC-500 interrupt controller, and Arm System MMU (SMMU) that enables the use of hypervisors and enables greater hypervisor performance.

The following figure shows an example hypervisor architecture running on a Versal device. In this example, the hypervisor runs an SMP-capabable OS, such as Linux, an RTOS, or a bare-metal application.

*Figure 20:* **Example Hypervisor Architecture**




The addition of a hypervisor introduces a layer of software that can add design complexity to low-level system functions, such as peripheral and accelerators access. Xilinx recommends that developers initiate efforts early into these aspects of system architecture and implementation.

## Use of Hypervisors

Xilinx distributes a port for the Xen open source hypervisor for the Versal device. The 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 the CPU, memory, and interrupts. You can run multiple operating systems on the hypervisor, which are called domains, virtual machines (VMs), or a guest OS.

The Xen hypervisor provides the ability to concurrently run multiple operating systems and their standard applications with relative ease. It also makes it possible to give a guest OS direct access to specific peripherals. However, Xen does not provide a generic method to do so; peripheral-specific configurations are required.

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

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

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

Running multiple operating systems using Xen hypervisor involves setting up the host OS and adding one or more guest OS. For more information the Xen hypervisor, refer to https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842530/XEN+Hypervisor on the Xilinx Wiki.

Xen hypervisor is available as a selectable component within the PetaLinux tools; alternatively, you can download Xen hypervisor from Xilinx GIT. With Linux and Xen software that is made available by Xilinx, it is possible to build custom Linux guest configurations. Guest OS other than Linux require additional software and effort from third-parties. For more information, see the PetaLinux Tools page.

In addition to the Xen hypervisor, other hypervisors are available through various Xilinx ecosystem partners. Visit the Embedded Software page for further details.

Chapter 7

# Boot and Configuration

This chapter provides an overview of the boot and configuration process for Versal™ ACAP. This includes the boot device options, different stages of software involved in booting the platform and configuring different components.

The platform management controller (PMC) is responsible for boot and configuration and other post-boot tasks. A boot image, the programmable device image (PDI) is used to boot and configure Versal ACAP. A PDI contains platform loader and manager (PLM) executable, images, and configuration data. The PDI can be loaded from a boot device, such as an SD card, eMMC, OSPI, or QSPI, or it can be loaded through a slave interface, such as JTAG or SelectMAP. The BootROM starts the boot process and loads the PLM software that takes care of loading images and configuring the system based on images or configuration data in PDI.

***Note*:** For hardware-related information, see the *Versal ACAP Technical Reference Manual* (AM011).

## Versal ACAP Boot Process

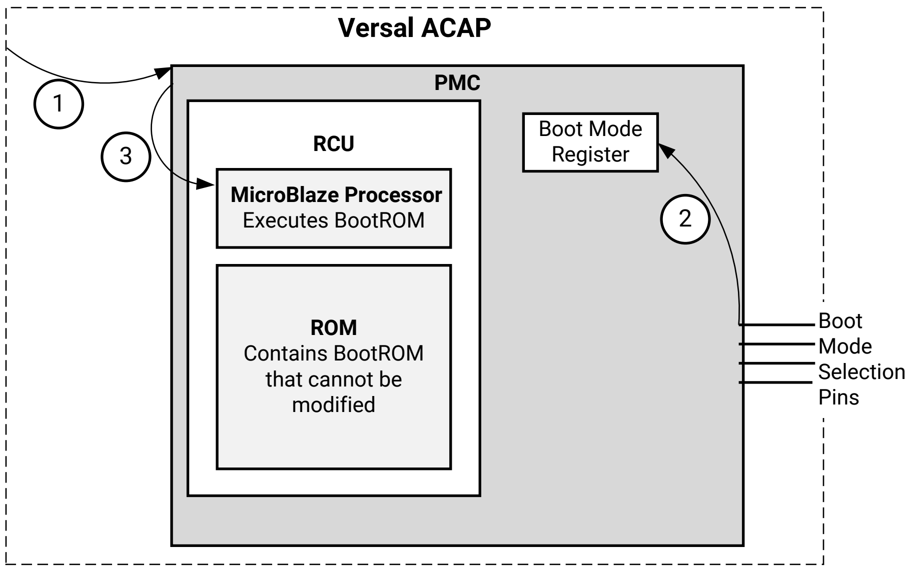
The boot process is divided into four phases that are independent of the selected boot mode:

- **Phase 1:** Pre-Boot (Power-up and Reset)
- **Phase 2:** Boot Setup (Initialization and boot header processing)
- **Phase 3:** Load Platform (Boot image processing and configuration)
- **Phase 4:** Post-Boot (Platform management and monitoring services)

The following sections provide a simplified overview the four phases.

## Phase 1: Pre-Boot

*Figure 21:* **Phase 1: Pre-Boot (Power-up and Reset)**

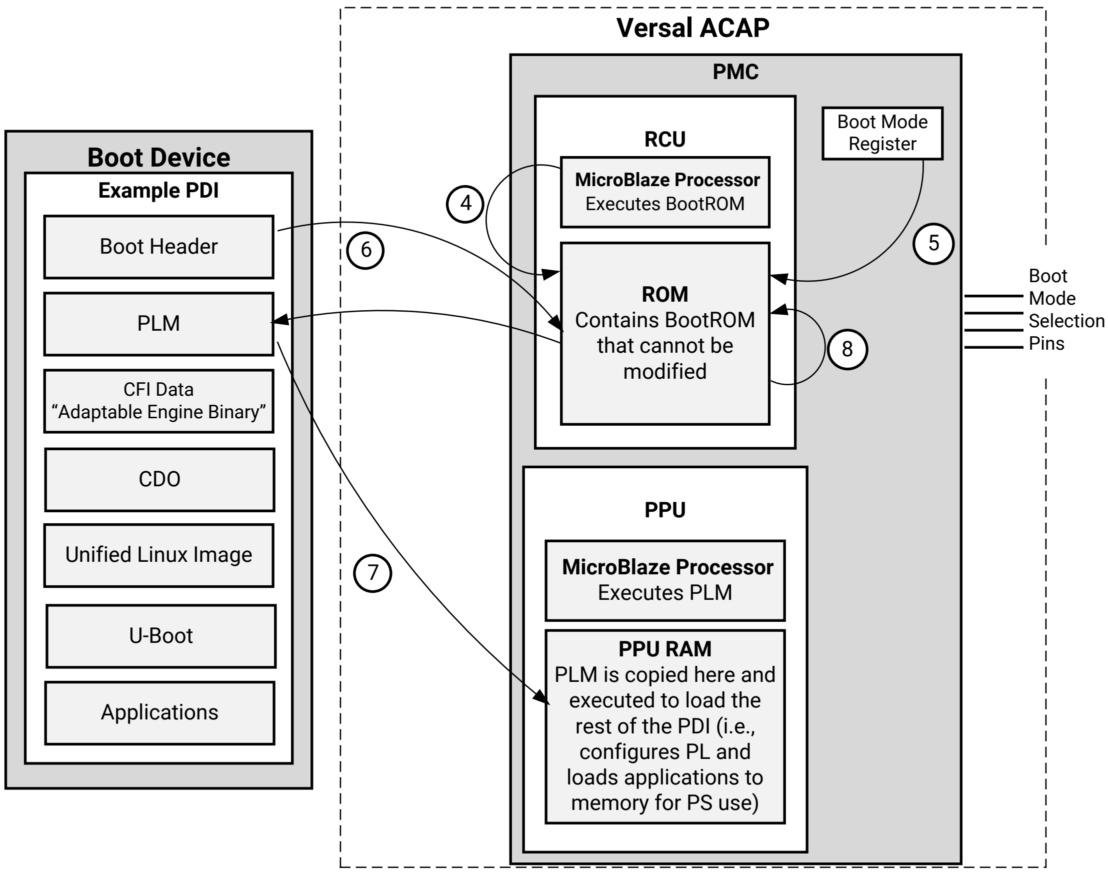



- **1:** The pre-boot phase is initiated when the PMC senses the PMC power and the external `POR_B` pin is released.
- **2:** PMC reads the boot mode pins and stores the value in the boot mode register.
- **3:** PMC sends the reset signal to the RCU.

  ***Note*:** All other processors and controller units remain in the reset state.

## Phase 2: Boot Setup

*Figure 22:* **Phase 2: Boot Setup**




- **4:** The RCU begins to execute the BootROM from the RCU ROM.
- **5:** The BootROM reads the boot mode register to select the boot device.
- **6:** The BootROM reads the PDI boot header in the boot device and validates it.
  - If the boot header is valid, the BootROM configures boot parameters based on the boot header data and then continues the boot process.
  - If the boot header is not valid, then the normal boot process changes to the fallback boot process.
- **7:** The BootROM releases the reset to the PPU, and loads the PLM from the PDI into the PPU RAM and validates it. After validation, the PPU is woken up (at this point, the PLM software starts executing, refer to point 9 in Phase 3: Load Platform.

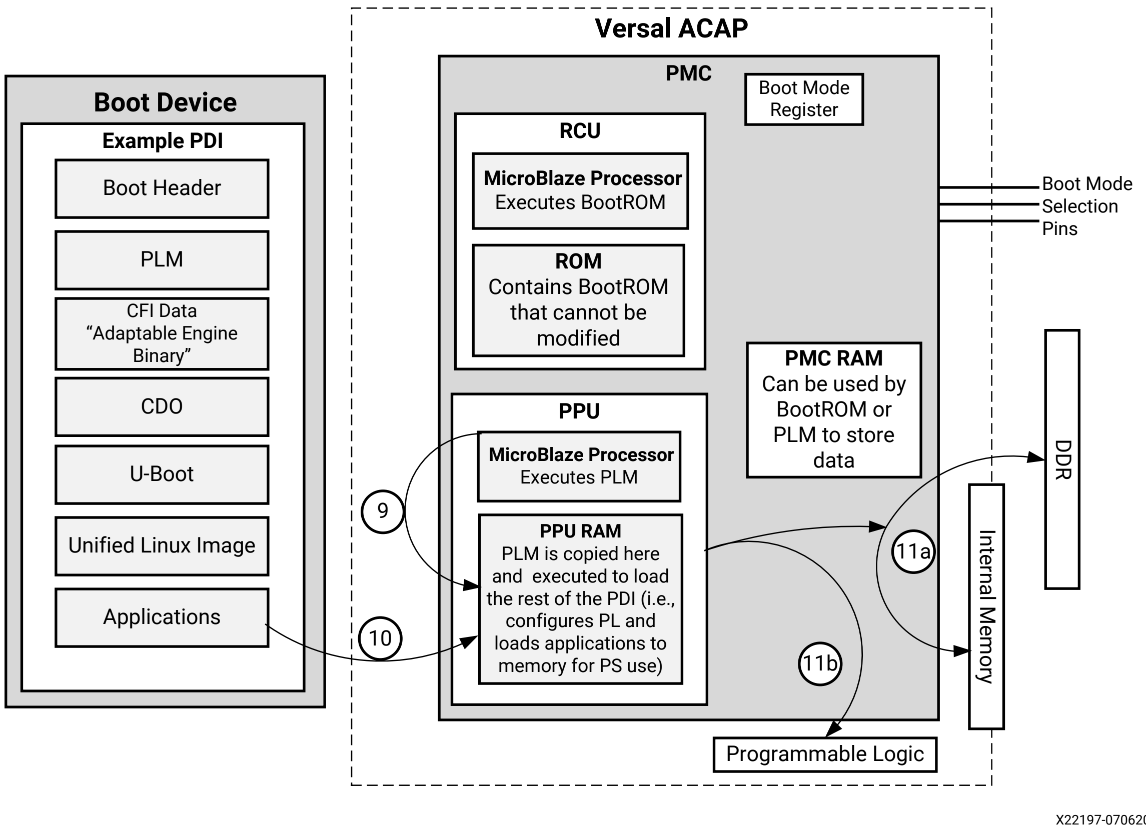
- **8:** The BootROM executable enters a sleep state. The BootROM executable continues to run until the next power-on-rset (POR) or system reset, and is responsible for post-boot platform tasks.

**Related Information**

Phase 4: Post-Boot

# Phase 3: Load Platform

*Figure 23:* **Phase 3: Load Platform**




- **9:** The PPU begins to execute the PLM from the PPU RAM.
- **10:** The PLM reads and processes the PDI components.
- **11:** The PLM configures other parts of the Versal device using the PDI contents.
  - **11a:** The PLM applies the configuration data to the following Versal ACAP components:
    - PMC, PS blocks (CDO files)
      - Multiplexed I/Os (MIOs), clocks, resets, and etc.

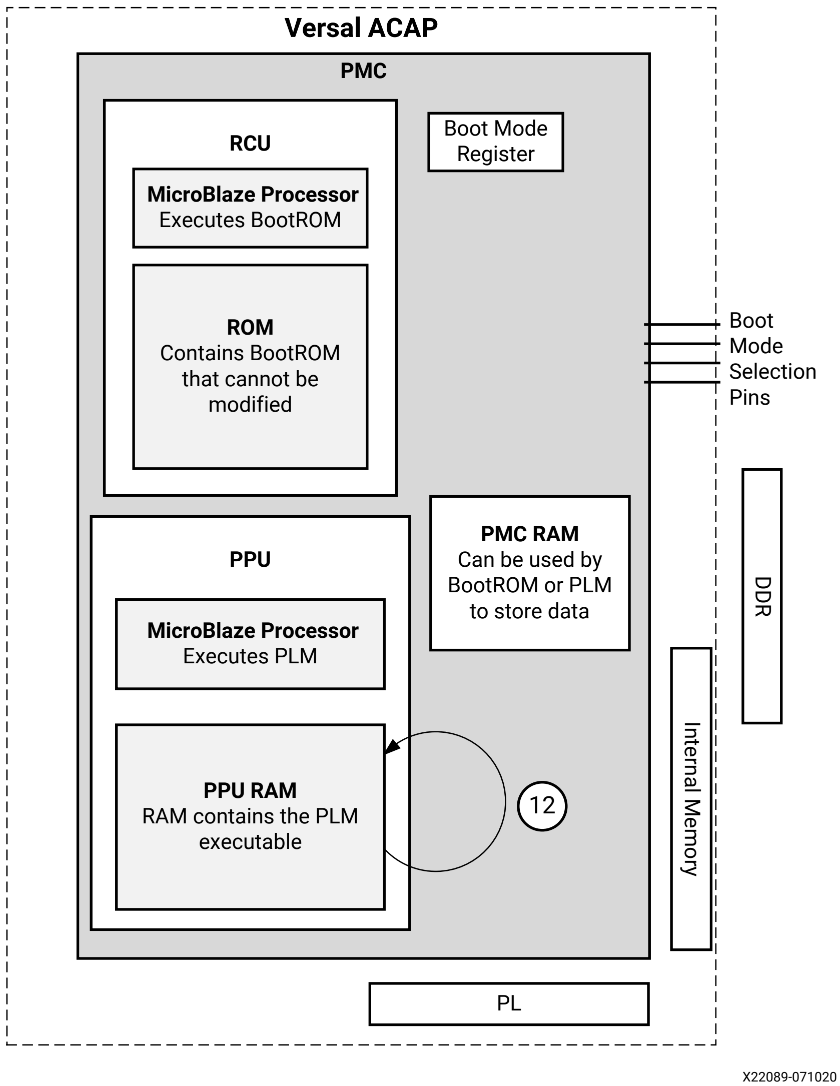
- NoC initialization and NPI components (NPI file)
    - DDR, NoC, GT, XPIPE, I/Os, clocking, and other NPI components
    - Adaptable Engine (PL) data (CFI file)
    - AI Engine configuration (AI Engine CDO)

      ***Note*:** The PMC triggers the scan clear of the individual programming control/status registers.

- **11b:** The PLM loads the applications and data for the Arm Cortex-A72 and Cortex-R5F processors to various memories specified by the ELF file. These memories include on-board DDR and internal memories, such as OCM and TCM.

## Phase 4: Post-Boot

*Figure 24:* **Phase 4: Post-Boot**




- **12:** The PLM continues to run until the next POR or system reset, and is responsible for post-boot platform management tasks. Post-boot services include DFX reconfiguration, power management, subsystem restart, error management, and safety and security services.

For a secondary boot device, the PLM:

1. Determines the specified secondary boot device from a field in the PDI meta header.
2. Uses the specified secondary boot device to load the specified PDI image.

# Boot Flow

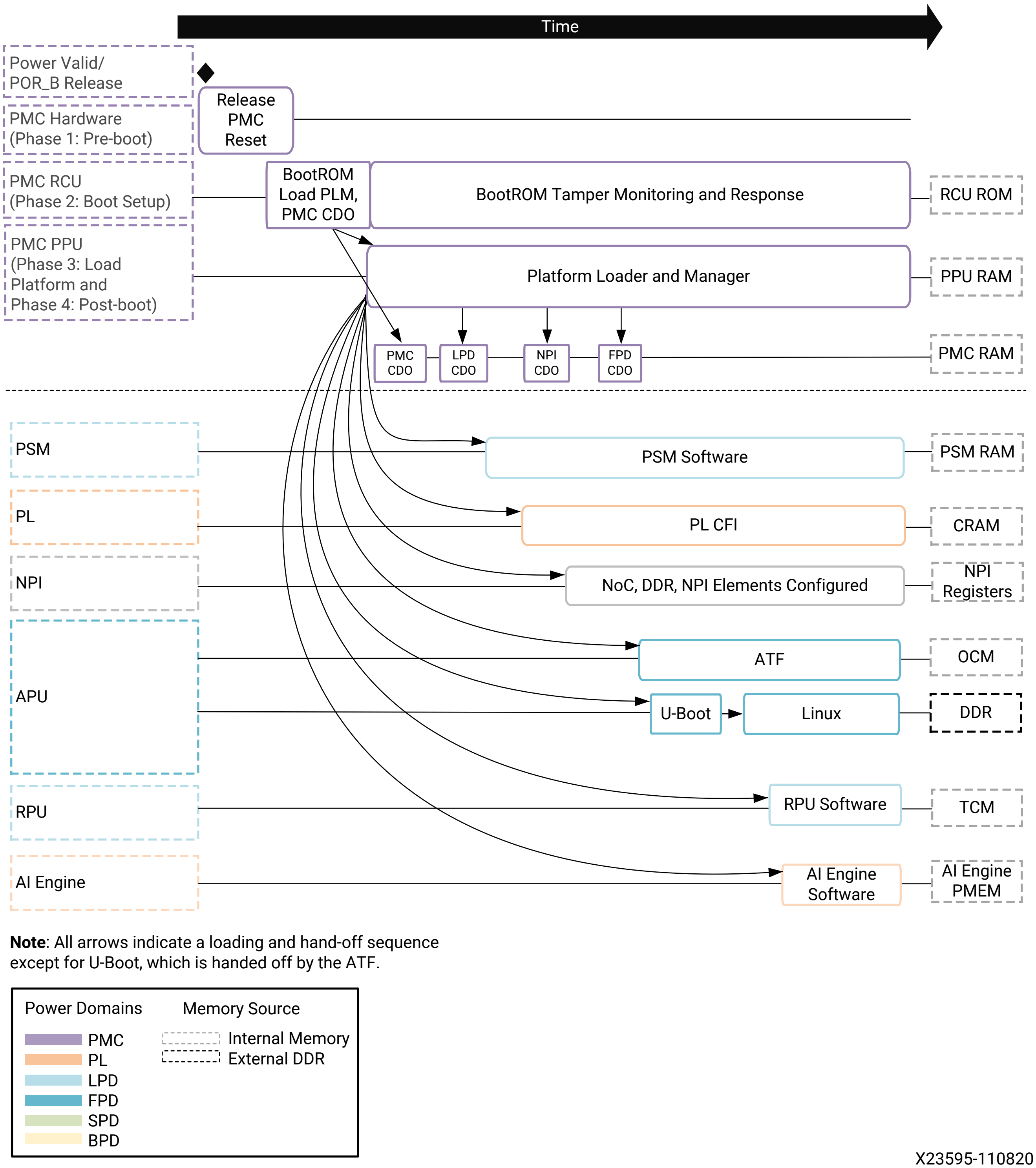
For system start-up, Versal ACAP must successfully initialize, boot, and configure from a supported boot source. There are two boot flows in the Versal ACAP architecture: secure and non-secure.

The following section describes the example boot sequences in which you can bring up various processors and execute the required boot tasks.

## Non-Secure Boot Flow

The following figure illustrates an example boot and configuration sequence and shows how the PLM loads the major partition components of the Versal ACAP software stack.

*Figure 25:* **Example Standard Boot Flow Processing Engines and Memory Sources**



In non-secure boot mode, BootROM loads PLM into PPU RAM and releases the PPU to begin PLM execution.. PLM continues loading of the images from PDI. In the above boot flow example, PLM initializes LPD, FPD and DDR through CDO files. As a part of LPD image, PLM loads and starts PSM Firmware. PLM loads PL CFI through RCDO file. PLM loads ATF, U-Boot and starts ATF (EL3-S) on APU. ATF starts U-Boot (EL2-NS). U-Boot then loads Linux and hand-off to it. PLM can also load and start RPU and AI Engine images.

***Note***: In symmetric multi-processing (SMP) mode, the OS manages the multiple Cortex-A72 processors.

## Secure Boot Flow

For information on the secure boot flow, refer to:

- Chapter 9: Security
- *Versal ACAP Technical Reference Manual* (AM011)

### *PDI Content Integrity and Support for Secure Boot*

The PDI content integrity is verified using checksums. Depending on the use case, executables and data objects in the PDI can be encrypted, authenticated, or both. For the artifacts to create the PDI, see Boot Image (PDI) Creation.

***Note***: In the boot process description, the term *validates* includes checksum verification and authentication as needed.

### *Boot Device Modes*

The following tables show the boot device choices for primary boot, and the boot device modes. For additional information, see the *Versal ACAP Technical Reference Manual* (AM011).

*Table 5:* **Primary Boot Devices**

| Boot Mode | Mode[3.0] Pin Setting | Data Bus Width | Secure Boot | Fallback Boot and MultiBoot | Search Offset Limit |
|---|---|---|---|---|---|
| eMMC1 (4.51) | `0110` | x1, x4, x8 | yes | yes | 8191 FAT files |
| JTAG | `0000` | x1 | no | no | N/A |
| Octal SPI single or dual-stacked [5] | `1000` | x8 | yes | yes | 8 Gb |
| Quad SPI24 single or dual-stacked[5] | `0001` | x1, x2, x4 | yes | yes | 128 Mb |
| Quad SPI24 dual-parallel | `0001` | x8 | yes | yes | 256 Mb |
| Quad SPI32 single or dual-stacked[5] | `0010` | x1, x2, x4 | yes | yes | 4 Gb |
| Quad SPI32 dual-parallel | `0010` | x8 | yes | yes | 8 Gb |
| SD0 (3.0) | `0011` | x4 | yes | yes | 8191 FAT files |

*Table 5:* **Primary Boot Devices** *(cont'd)*

| Boot Mode | Mode[3.0] Pin Setting | Data Bus Width | Secure Boot | Fallback Boot and MultiBoot | Search Offset Limit |
|---|---|---|---|---|---|
| SD1 (2.0) | `0101` | x4 | yes | yes | 8191 FAT files |
| SD1 (3.0) | `1110` | x4 | yes | yes | 8191 FAT files |
| SelectMAP | `1010` | x8, x16, x32 | yes | no | N/A |

**Notes:**

1. Execute in place (XIP) is not supported by Versal ACAP.
2. The legacy mode Linear Quad SPI (LQSPI) is not supported by Versal ACAP.
3. The "search offset limit" is used when the BootROM executable is searching the boot device for a PDI with a boot header and a PLM. This is used for fallback boot and MultiBoot.
4. JTAG and SelectMAP are slave boot modes. All other devices in this list are master boot modes.
5. For dual-stacked QSPI, only the first flash device can be accessed during the bootROM stage.

When selecting a boot device to implement in a board design, it is important to consider the post-boot use of shared multiplexed I/O pins and the voltage requirements of each boot mode. For more information, refer to the *Platform Management Controller* chapter in the *Versal ACAP Technical Reference Manual* (AM011).

# Secondary Boot Process and Device Choices

The boot process can be based on a primary boot device such as SD, eMMC, QSPI, OSPI, or slave interfaces such as JTAG, or SelectMAP. Optionally, the boot process starts using a primary boot device, and then is completed using a secondary boot device. In this case, the PLM loads images from primary boot device and configures the secondary boot device. The secondary boot device contains a PDI that includes images and configuration data that needs to be loaded from the secondary boot device. This secondary boot device does not contain a PLM or a boot header. The secondary boot process occurs if a secondary boot device is specified in a field in the PDI meta header. At POR/system reset, the boot process starts with the primary boot:

- **BootROM:**
  - Reads boot mode register to determine the primary boot device
  - Loads the PLM from the specified primary boot device into the PPU RAM
  - Releases the PPU to execute the PLM
- **PLM:**
  - Determines the specified secondary boot device from a field in PDI meta header
  - Uses the specified secondary boot device to load the remainder of the PDI content

***Note*:** The secondary boot device cannot be the same as the primary boot device.

The secondary boot devices include:

- eMMC0, eMMC1

  ***Note*:** There are two controllers (eMMC0 and eMMC1). Each controller allows two different sets of MIO to be assigned or an EMIO option. Primary boot on the first slot (eMMC0) is not supported. The second slot (eMMC1) cannot be used because it conflicts with the QSPI MIO pins. The first slot eMMC1 (MIO26-36) can be used for secondary boot.

- Ethernet

  ***Note*:** When Ethernet is used as a secondary boot device, Versal ACAP is first booted up to U-Boot using the primary boot device. U-Boot can then use Ethernet to complete the boot process.

- OSPI
- PCIe® interface
  1. First, the PLM is loaded from the primary boot device into the PPU RAM.
  2. Then, the PLM runs and initializes the Cache Coherent Interconnect for Accelerators (CCIX) PCIe Gen4 Module (CPM) block in PCIe Endpoint (EP) mode.
  3. Finally, the PCIe host, as a secondary boot device, loads the rest of the images.

     ***Note*:** PCIe interface is supported only as secondary boot device.

- QSPI
- SD0, SD1
- USB

  The secondary PDI is downloaded using dfu_util to a fixed DDR address (`0x50000000`). The dfu_util is an open source utility available for Windows and Linux. The PLM then processes the PDI.

  To indicate USB as a secondary boot mode, specify usb as the boot_device attribute in the BIF file.

- SelectMAP (SMAP)

  Use smap as the boot device attribute for SelectMAP as secondary boot mode in the BIF file.

# Fallback Boot and MultiBoot

Fallback boot allows Versal ACAP to automatically boot a different PDI than the initial PDI on the same primary boot device if the first PDI fails to boot. If an error occurs during booting the PDI load, the PLM increments the MultiBoot register by 1 and resets the device so that the BootROM can find the next good image.

MultiBoot allows the software developer to specify a different PDI than the initial PDI on the same primary boot device to use for the next Versal ACAP boot process.

To use fallback boot or MultiBoot, store multiple PDIs in the same primary boot device within the search limit for the device. For information, see the Primary Boot Devices table in Boot Device Modes.

## Octal-SPI and Quad-SPI Boot Devices

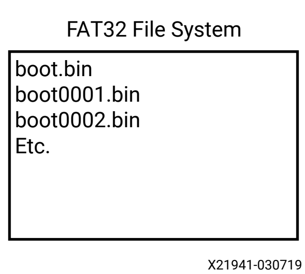
The BootROM executable supports fallback boot and MultiBoot on Octal-SPI and Quad-SPI boot devices. The BootROM searches QSPI and OSPI devices in 32k offsets, for example, if the multiboot register value is two, then it starts the image search from offset `0x10000 (2 * 32k)`.

## SD/eMMC Boot Devices

The BootROM executable supports fallback boot and MultiBoot on SD card and eMMC flash devices. The SD card or eMMC flash must be partitioned so that the first partition is a FAT 16/32 file system. Bootgen is used to create PDI files with the names: `boot.bin`, `boot0001.bin`, `boot0002.bin`, etc.

Except for the PMC_MULTI_BOOT value '0,' the PMC_MULTI_BOOT value is concatenated with first the string `boot`, then PMC_MULTI_BOOT, then .bin to create the specified PDI file name. For example, if PMC_MULTI_BOOT= 2, then the PDI file name is `boot0002.bin`. For command line users, the PDI file names are specified on the Bootgen command line. The PDI files are then copied to the FAT16/32 file system on the boot device. The search limit specified for the device corresponds to the maximum number in the file name, for example `boot8190.bin`.

*Figure 26:* **SD and eMMC FAT16/32 Files for Fallback Boot and MultiBoot**




### *SD/eMMC RAW Boot Mode*

A RAW partition is any partition on the eMMC that is not formatted. BootROM and PLM supports SD/eMMC in RAW format in addition to the existing file system mode support.

The BootROM searches for the boot image and finds it on the file system or in RAW. After the image is found, the BootROM updates the Multiboot register with the image information. The PLM continues the boot from the file system or RAW with the information from the MultiBoot register.

The MultiBoot register contains information such as whether the image is from the file system or RAW, image file number or the offset.

### Secondary Boot Combinations

- Any non-SD eMMC primary boot and SD/eMMC as secondary boot mode
- eMMC boot partition 1/2 as primary boot and eMMC user area as secondary boot
- eMMC boot partition 1, boot partition 2, and user area as primary boot modes

# Programmable Device Image

The boot image of Versal ACAP is called a PDI, which is a Xilinx file format processed by the PLM as part of the Versal ACAP boot process or partial configuration process. The PDI data is specific to the design requirements. The Versal ACAP boot image typically involves binaries used to boot and configure the platform, these binaries can include: bootloader, firmware, hardware configuration data, OS and user applications. Examples include Adaptable engine (PL) configuration file, PLM image, ATF, U-Boot, Linux kernels, rootfs, device tree, and standalone or RTOS applications.

A PDI file is a collection of images, where an image consists of one or more partitions. A partition can include:

- CDO file consisting of a list of commands executed in sequence
- Loadable data ( for example, APU/RPU ELFs or data)

There can be different PDI files based the use case.

- **PDI for Versal ACAP start-up:** Occurs during the power-on reset (POR).

  The PDI contains the following information needed to boot, configure, and manage Versal ACAP:

  - A boot header
  - A PLM

    ***Note*:** The PLM is read by the BootROM executable.
  - Additional subsystem images and image partitions used to configure a specific subsystem or engine.
  - Few of these images can also be part of another PDI targeted for secondary boot.
- **Subsystem Restart or DFX:** Occurs in some power management, warm restart, and partial reconfiguration scenarios.

This type of PDI contains the information needed to reconfigure Versal ACAP, and does not contain the boot header and PLM.

# Configuration Data Object

Configuration data object (CDO) files are intended to pass the configuration information generated from tools to the platform loader and manager (PLM). During the PDI load sequence, the PLM passes information from CDOs (CDO commands) to various components to configure the system.

Configuration can be generated as multiple CDO files, and each CDO file is included as a partition in the programmable device image (PDI) file. The Versal ACAP CDO is a list of commands and is meant to include pre-load or pre-boot requirements for any subsystem.

Adaptable Engine (PL) configuration data, which is generated as a raw file (CFI file), is passed as input to Bootgen. Bootgen includes this configuration data as a partition in the PDI, and creates a PDI image.

# Boot Image (PDI) Creation

The Xilinx boot image generation tool, Bootgen is essential in creating an image file (PDI) used to boot and configure Versal ACAP. Bootgen creates the boot image (PDI for Versal ACAP) by building the required boot header, appending tables that describe the images and partitions, and processing the input data files (ELF files, PL configuration data, and other binary files) to partitions. Bootgen has specific features to assign specific destination memory addresses or imposing alignment requirements for each partition. Bootgen also supports the encryption, authentication, and calculation of checksums on each partition.

An input file (*.bif) known as the boot image format (BIF) file serves as the input for Bootgen.

For detailed information on building a boot image, see the *Bootgen User Guide* (UG1283).

For Versal ACAP, the Vivado tool flow generates a PDI using Bootgen in the backend. This PDI contains hardware-specific user configuration data, Adaptable Configuration data, NPI Data, and a PLM. Typically, the PDI is outputted from the Vivado tools and then exported to software tools, such as the Vitis software platform or PetaLinux. These tools either reuse the PDI components or input the PDI, additional software executables, or images to create a larger PDI. The Vitis tool PDI contains a PLM (optionally modified), the configuration data generated by Vivado tools, and typically contain applications that run on the APU and/or RPU cores.

Bootgen is essential in creating an image file (PDI) to boot and reconfigure Versal ACAP. Bootgen places all of these binary images and generates a bootable image in a specific format that the PLM can interpret while loading the image.

The PDI created in the Vivado tool will contain the following images or configuration data:

- Boot header (part of PDI, based on BIF attributes)
- PLM (platform-independent)
- PMC CDO
- Meta headers
- CFI data
- LPD CDO and FPD CDO (based on CIPS configuration)
- PSM (subject to CIPS configuration)

Some data files or images, like LPD/FPD CDO, and PSM might not be required for PL only flow, and may not be created for some use-cases like PL only, which involves limited CIPS configuration for PMC and I/O.

### Extending the PDI generated by the Vivado Design Suite

Example files from the Vitis tool that can be input to Bootgen:

- PLM

  ***Note***: Bootgen supports the replacement of the PLM and PSM ELF files from the Vivado-generated PDI. This is only if these need to be modified for specific needs. Most use cases should work with the default PLM and PSM ELFs from the Vivado tool.

- Arm Cortex-A72/Cortex-R5F applications
- AI Engine configuration (DMAs, etc.)
- AI Engine ELFs
- U-Boot

Bootgen uses the following types of input to create the PDI:

- PDI generated from the Vivado tools.
- Other files from the Vitis tools, which includes an optionally modified PLM. The custom PLM replaces the default PLM. However, for most cases the default PLM can be used as is.
- Optionally, a user-created boot image format (BIF) file, which provides Bootgen with the instructions needed to create a Vitis tool PDI. If required (in few rare use-cases), Vivado generated BIF can be reused to create/update an updated PDI in case some Vivado-generated file is updated.

## Methods for Copying the PDI to a Primary Boot Device

For system configurations that use a master boot mode, the PDI file must be copied to the boot device. The following methods can be used to copy the PDI file to the master boot mode device:

- Vivado tool hardware device manager
- Vitis tool program flash utility
- Removable devices, such as an SD card, can be programmed separately, and then added to the board.
- Socketed and soldered devices, such as QSPI, can be off-board programmed, and then put onto the board.

***Note*:** When the PDI is copied to the boot device via the Vivado tools hardware device manager or the Vitis tool program flash utility, a cable is connected from the host PC to the PMC JTAG port, and the master boot device is programmed via the JTAG interface. JTAG can also be used to boot Versal ACAP to U-Boot, and then U-Boot can be used to program the master boot device.

## Software Developer Control of PDI File Creation via BIF File

A Xilinx boot image format (BIF) file is used to specify the subsystem images in the PDI and the subsystem image partitions. Each separate file within a subsystem image is stored in one of the subsystem image partitions. The BIF file is a data file in ASCII format. The BIF file tells Bootgen how to create the PDI by processing each of the input files.

***Note*:** The PLM is processed by the BootROM, so it is not formatted as a subsystem image or partition.

Command line users must modify the Vivado-generated BIF and run Bootgen to create the PDI from the instructions in the BIF files. You can optionally write your own BIF file and specify the Vivado generated PDI along with other input files to extend the PDI. Vitis IDE users can use a wizard to specify the required inputs for the BIF file, and then use the Vitis IDE to create the BIF file and run Bootgen to create the PDI.

***Note*:** The Bootgen wizard is not fully implemented for Versal ACAP. Currently, the wizard only supports PS and AI Engine partitions without any security features such as encryption, authentication, etc.

Chapter 8

# Platform Loader and Manager

The platform loader and manager (PLM) runs on the platform processing unit (PPU) in the platform management controller (PMC). It performs boot and configuration of the Versal device, and then continuously provides services after the initial boot and configuration.

During the initial boot, the BootROM decodes the programmable device image (PDI) and loads the PLM into the PPU RAM. The PPU PLM processes the PDI to boot up the entire system by loading the partitions present in the PDI. The PLM supports the loading of partial PDIs during run time. See *Versal ACAP Technical Reference Manual* (AM011) for more information.

## PLM Boot and Configuration

**BootROM, PLM Handoff State**

The BootROM loads the PLM into the PPU RAM from the boot device and is responsible for releasing the PPU from reset to start the PLM execution..

The PLM ELF is loaded to the PPU RAM. The PMC RAM is used to load the PMC DATA CDO file.

The state of the system at BootROM handoff is as follows:

- The PPU is sleeping with the reset released, in case of normal JTAG boot mode.
- The PPU RAM and PMC RAM are initialized with error code correction (ECC).
- The JTAG IDCODE instruction is always available regardless of the boot mode. Except the JTAG IDCODE instruction, all other JTAG instructions can be disabled when you program the required eFUSEs. If the eFUSEs are not programmed and a secure boot occurs, then only the base JTAG instructions are supported. When the AUTH_JTAG enable instruction is sent to the Versal ACAP in a secure boot mode, the full set of JTAG instructions (base +extended) can be enabled.
- The boot device is taken out of reset and initialized.
- The NoC default configuration is enabled for Slave Super Logic Region (SLRs), so that the master PLM can connect to slave PMC devices.

### PLM Subsystem

*Table 6:* **Components of the PLM Subsystem**

| File | Contents |
|---|---|
| PLM ELF | PLM ELF File |
| PMC CDO | PMC CDO file<br>• Device topology<br>• PMC Configuration: Register writes/polls for MIO, clocks, resets |

The BootROM loads the PLM ELF and PMC CDO files to the PPU RAM and PMC RAM, respectively.

After the BootROM handoff to the PLM:

- Initialize the PPU and register interrupts.
- Initialize the modules to register the CDO/IPI commands and handlers.
- Configure the PMC CDO file.
    - Device topology with PMC CDO commands to registers the nodes.
    - General/platform management calls to initialize the PMC and LPD MIO, clocks, etc.
        - PMC initialization for clocks, MIOs, resets.

### LPD Configuration

*Table 7:* **LPD Configuration**

| File | Contents |
|---|---|
| LPD CDO | • PS LPD PM init node commands (SC, LBIST, BISR, MBIST)<br>• LPD configuration: Register writes/polls |
| PSM ELF | PSM ELF file |

After initializing the PMC CDO:

- The PLM initializes the boot device and loads the LPD CDO file from the boot device.
- 
    - Configures the LPD CDO file.
    - Initiates the Scan Clear, BISRs, MBIST as required for LPD.
    - XilPM releases resets and powers up the nodes based on the CDO requirements.
- The PLM loads the PSM ELF file and waits until initialization is complete.

Before loading the LPD configuration, ensure that PMC is configured.

### PL Configuration

*Table 8:* **PL Configuration**

| File | Contents |
|---|---|
| PL CDO <.rcdo> | • The PM init node commands for scan clear, house cleaning, BISR of PL domain<br>• Register writes to CFU<br>• DMA Keyhole Xfer commands to load CFI data<br>• Register writes/polls to CFU<br>• If NPI not present:<br>    ◦ Global Signals (GMC_B, GRESTORE, GHIGH_B..): Register writes/polls<br>• Global Signals (GWE, EOS, EN_GLOb): Register writes/polls |
| NPI CDO <.rnpi> | NPI data<br>• PM init node commands (SC, BISR, MBIST)<br>• NPI data load: DMA Writes/register writes<br>• If CFI present:<br>    ◦ Global Signals (GMC_B, GRESTORE, GHIGH_B..:) Register writes/polls<br>• NPI Sequence: Register writes/polls<br>• If CFI present:<br>    ◦ Global Signals (GWE, EOS, EN_GLOb): Register writes/polls<br>• Isolation and PL reset commands |

The NPI data is generated by the Vivado tool for the various NPI blocks. The NPI blocks that are present in Versal ACAP include NoC, DDR, XPHY, XPIO, GTY, MMCMs, etc.

Before loading the PL configuration, ensure that PMC is configured.

### FPD Configuration

*Table 9:* **FPD Configuration**

| File | Contents |
|---|---|
| FPD CDO | • PS FPD PM init node commands (SC, BISR, MBIST)<br>• FPD configuration: Register writes/polls |

Before loading the FPD CDO, ensure that the PMC and LPD are configured.

### DFX Configuration

Dynamic Function eXchange (DFX) configuration enables one or more sub-regions of the device to be independently reprogrammed with new configuration data while all remaining regions (static or reconfigurable) remain active and unaffected. The DFX PDI can come either from PCIe, DDR, or the primary boot device. For loading the DFX PDIs, the XilLoader CDO commands are with the IPI interface.

### CPM Configuration

- CPM: CPM configuration CDO with register writes

Before loading the CPM CDO, ensure that the PMC, LPD and stage1 PL configuration are completed. Stage1 PL configuration should contain XPIPE, GT, and NoC configuration data in the NPI file.

### Processor Subsystem Configuration

The APU and RPU come under the processor- based subsystems. For all processor-based subsystems, ELF files and/or CDOs are present as a part of the image. Processor details are read from image headers and the processor is initialized using XilPM commands.

The configuration consists of the following files.

*Table 10:* **Processor Subsystem Configuration**

| File | Contents |
|---|---|
| PSM/RPU/APU CDO files | • (Optional) Set of PM commands with nodes and requirements |
| PSM/RPU/APU ELF files | • Cortex-R5F processor applications: Bare- metal/RTOS<br>• Cortex-A72 processor applications: ATF/U-Boot/Linux/Bare-metal |

- For loading Cortex-R5F processor applications, ensure that the LPD configuration is completed.
- For loading Cortex-A72 processor applications, ensure that the FPD configuration is completed.
- For loading Cortex-R5F/Cortex-A72 using DDR memory, ensure that the PL (NPI with DDR configuration) configuration is completed.
- For loading Cortex-R5F/Cortex-A72 processor applications to the DDR, enable the NoC path from the PMC to the DDR in the design.

### AI Engine Configuration

The AI Engine configuration consists of the following files.

*Table 11:* **AI Engine Configuration**

| File | Contents |
|---|---|
| AI Engine NPI CDO | AI Engine Global Configuration using NPI<br>• PLL configuration<br>• AI Engine scan clear and memory clear using PM initialization node commands |
| AI Engine ELF | AI Engine tile program and data memory |
| AI Engine CDO | AI Engine array configuration<br>• Program memory configuration<br>• Data memory configuration<br>• DMA, locks, stream switch configuration<br>• AI Engine module register configuration |

Before loading the AI Engine NPI CDO, ensure that PLM, LPD and PL (with NoC configuration in the NPI file) are completed. Also, enable the NoC path from the PMC to the AI Engine in the design for PLM to clear the AI Engine data memories.

# PLM Software Details

This section explains PLM responsibilities, architecture, and execution details.

## PLM Responsibilities

The PLM runs on the PMC PPU after the bootROM boots, and remains active throughout the lifetime of the system, beginning from the BootROM post-boot.

The PLM performs the system initialization and the boot and configuration of the Versal ACAP subsystems to include the APU, PL, and AI Engines. PLM handles authentication and decryption for secure boot as discussed in Chapter 9: Security. The PLM also takes care of power management, partial reconfiguration, error management, subsystem restart, and health monitoring.

The PLM responsibilities include:

- Secure/non-secure boot
    - System initialization
    - Initialize NoC, configure NoC programming interface (NPI), DDR, and CPM (optional)
    - Configure AI Engines
    - Load PS images on the APU (Arm® Cortex-A72 processors) and the RPU ( Cortex-R5F processors)

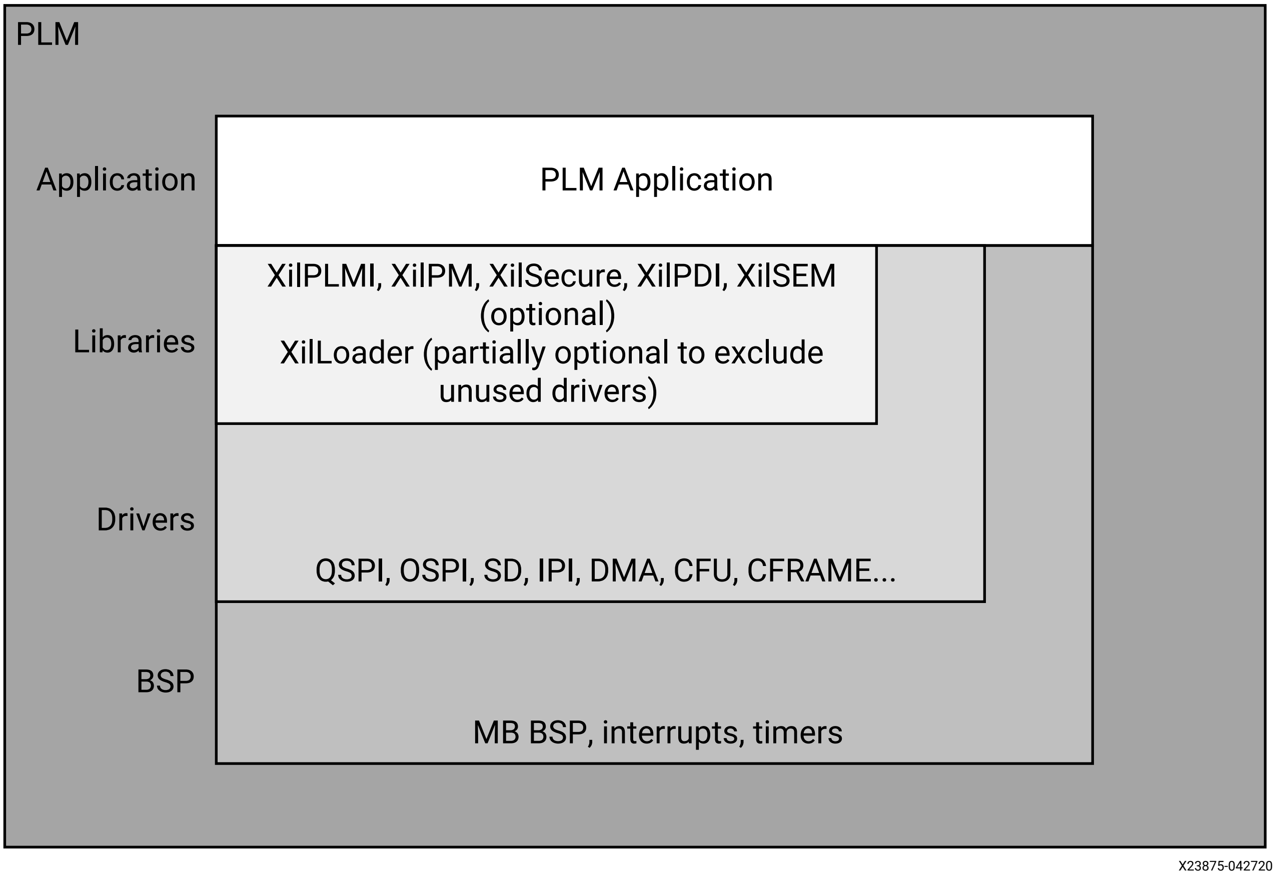
- The platform management tasks include:
    - Dynamic Function eXchange (DFX)
    - Error management
    - Power management
    - Subsystem restart
    - Health monitoring
    - Soft error mitigation (SEM)

## PLM Architecture

The PLM is designed with a modular subsystem-based configuration, and task-based services. Each functionally distinct feature is designed as a module. The PLM application layer contains the module initialization and start-up tasks. Depending on the modules selected in the configuration, different modules are initialized. Modules can register event handlers, IPI handlers, CDO commands, and scheduler events with the PLM interface layer.

The PLM reads the PDI from the boot device and configures the system with the payloads present in the PDI. The following figure shows components that are part of the PLM ELF file.

*Figure 27:* **PLM ELF Components**




***Note*:** For modularity and possible feature reuse by other system software modules, the following PLM application modules are placed into separate libraries. Currently, these libraries are intended to be internal-only libraries; no APIs from these libraries are intended to be directly called:

- XilPLMI
- XilPDI
- XilLoader

The following table describes these components.

*Table 12:* **PLM ELF Components**

| Software Component | Scope | Libraries/Drivers Used |
|---|---|---|
| PLM Application | • Start-up Events Registration<br>• Configuration to include modules and debug levels<br>• Event scheduling with queues | All libraries listed in this table. |

*Table 12:* **PLM ELF Components** *(cont'd)*

| Software Component | Scope | Libraries/Drivers Used |
|---|---|---|
| XilPLMI | • PLM interface layer<br>• Provides interface for parsing CDO files<br>• Provides an interface to register handlers for section data<br>• Provides an interface to register IPI events<br>• Provides an interface to register interrupt events<br>• Provides an interface to provide error responses, register error handlers<br>• Provides interfaces to write and read CFI data to and from PL respectively | BSP, PMC DMA, IPI, IOMODULE |
| XilLoader | • Responsible for loading boot PDI and subsystem images<br>• Provides an interface for loading subsystem images from PDI<br>• Interfaces with XilPLMI, XilSecure, XilPM and flash drivers | SD/eMMC, QSPI, OSPI, USB, XilPLMI, XilPM |
| XilPM | Provides an interface to create and manage subsystems, MIO, clocks, power, and reset nodes settings. | XilPLMI |
| XilSecure | • Interfaces with secure module Provides an interface to authenticate and decrypt Xilinx images | PMC DMA |
| XilSEM | • Provides handlers for single event upset (SEU) events in the configuration RAM and NPI registers<br>• Schedules handlers for SEU detection scan operations | XilSecure, XilPLMI, CFRAME, CFU |

# PLM Software

The PLM application performs processor initialization, module initialization, and executes start-up tasks.

- **Processor Initialization:** Includes initializing the stack and corresponding protections in hardware, enabling required interrupts and exceptions.
- **Start-up Tasks:** Start-up tasks include module initialization, executing PMC CDO, loading boot PDI, and running user hooks at predetermined places.
- **Modules:** Consists of module initialization functions that are included in the PLM to register the commands, interrupt handlers, and scheduler tasks with XilPLMI. The modules can:

- Register various commands using XilPLMI with the registered module ID. Commands can be part of PDI, or can come from the IPI message.
- Schedule tasks with PLMI.
- Register interrupt handlers for any PLM interrupts.

- **PLM Hooks:** In the PLM application, hook functions are present in predefined places for users to add their own code. Predefined user hooks are present at:

*Table 13:* **PLM Hooks**

| Function | Description |
|---|---|
| XPlm_HookBeforePlmCdo | Executed before processing the PMC CDO file. |
| XPlm_HookAfterPlmCdo | Executed after processing the PMC CDO file. |
| XPlm_HookAfterBootPdi | Executed after loading the Boot PDI. |

- **Exception Handling:** The exception handler is invoked when an exception occurs in the PPU. The handler logs the exceptions.

## *PLM Execution Flow*

PLM execution is based on tasks. Initial start-up tasks are added to the PLM task queue after initializing the processor and programmable interval timers (MB internal). Start-up tasks include module initialization, executing PMC CDO, loading boot PDI and running user hooks at predetermined places. Post start-up events, the PLM enters service mode, where it enters sleep and waits for events. When woken up, the PLM enters the interrupt context, services the interrupt, and goes back to task queue to check for any task. When the task queue is empty, it goes to sleep.

The following sequence of events occur in the PLM:

1. Initialize processor, register interrupt handlers, enable interrupts
2. Execute start-up tasks
    - Initialize modules
        - Initialize modules, such as XilPLMI, XilPM, and XilLoader
        - For every module: Register CDO commands and interrupt handlers
    - Process the PMC CDO stored in the PMC RAM
    - Load the rest of the images in the boot PDI
    - Execute user hooks
3. Task dispatch loop (Wait for event)
    - Execute any tasks added to the task queue

## PLM Errors

When errors are detected during PDI load, the PLM writes the error code to the PLM_FW_ERROR register.

Alternatively, you can read the PLM error from the JTAG Error Status command. The error is logged in the following format.

```
Error code: 0xXXXXYYYY
```

- **XXXX:** Major error code. The PLM/XilLoader/XPLMI error codes are defined in `xplmi_status.h`.
- **YYYY:** Minor error code. This is the Libraries/Drivers error code defined in each module.

### PLM Error Codes

The following table lists the PLMI, PLM, XilLoader, and XilSecure major error codes.

*Table 14:* **PLM Error Codes**

| Value | Description |
|---|---|
| `0x0` | XPLM_SUCCESS: Success |
| `0x1` | XPLM_FAILURE: Used internally for small functions |
| `0x2` | XPLMI_TASK_INPROGRESS: Used internally to indicate task is in progress |
| `0x100` | XPLMI_ERR_DMA_LOOKUP: Error when DMA driver lookup fails |
| `0x101` | XPLMI_ERR_DMA_CFG: Error when DMA driver configuration fails |
| `0x102` | XPLMI_ERR_DMA_SELFTEST: Error when DMA Self test fails.<br>Error occurs when DMA is in reset, and PLM tries to initialize it. |
| `0x103` | XPLMI_ERR_IOMOD_INIT: Error when IOModule driver look up fails |
| `0x104` | XPLMI_ERR_IOMOD_START: Error when IOModule driver startup fails |
| `0x105` | XPLMI_ERR_IOMOD_CONNECT: Error when IOModule driver connection fails |
| `0x106` | XPLMI_ERR_MODULE_MAX: Error when PLMI module is not registered<br>Can occur when invalid CDO CMD is processed by XilPLMI |
| `0x107` | XPLMI_ERR_CMD_APIID: Error when valid module and unregistered CMD ID is processed by XilPLMI |
| `0x108` | XPLMI_ERR_CMD_HANDLER_NULL: Error when no command handler is registered by module for CDO CMD |
| `0x109` | XPLMI_ERR_CMD_HANDLER: Error returned by the CDO CMD handler<br>For error returned by the CMD, check the PLM minor code. |
| `0x10A` | XPLMI_ERR_RESUME_HANDLER: Error returned by the CDO CMD resume handler<br>For error returned by the CMD, check the PLM minor code. |
| `0x10B` | XPLMI_ERR_CDO_HDR_ID: Error when valid CDO header ID is not present in CDO header<br>Can happen when different partition type is processed as CDO |
| `0x10C` | XPLMI_ERR_CDO_CHECKSUM:: Error when CDO header checksum is wrong<br>Can happen when CDO header is corrupted |
| `0x10D` | XPLMI_ERR_UART_DEV_PM_REQ: Error when XilPM request device for UART fails<br>PM error code is present in PLM minor code. |

*Table 14:* **PLM Error Codes** *(cont'd)*

| Value | Description |
|---|---|
| 0x10E | XPLMI_ERR_UART_LOOKUP: Error when UART driver lookup fails |
| 0x10F | XPLMI_ERR_UART_CFG: Error when UART driver configuration fails |
| 0x110 | XPLMI_ERR_SSI_MASTER_SYNC: Error when SSI technology slave sync fails with master |
| 0x111 | XPLMI_ERR_SSIT_SLAVE_SYNC: Error when SSI technology master times out waiting for slaves sync point |
| 0x112 | XPLMI_ERR_INVALID_LOG_LEVEL: Error when invalid log level is received in Logging command |
| 0x113 | XPLMI_ERR_INVALID_LOG_BUF_ADDR: Error when invalid log buffer address is received in Logging command |
| 0x114 | XPLMI_ERR_INVALID_LOG_BUF_LEN: Error when invalid log buffer length is received in Logging command |
| 0x115 | XPLMI_ERR_IPI_CMD: Error when command execution through IPI is not supported |
| 0x116 | XPLMI_ERR_REGISTER_IOMOD_HANDLER: Error when registering IOModule Handler |
| 0x117 | XPLMI_ERR_WDT_PERIODICITY: Invalid Periodicity parameter for SetWDT command |
| 0x118 | XPLMI_ERR_WDT_NODE_ID: Invalid Node ID parameter for SetWDT command |
| 0x119 | XPLMI_ERR_WDT_LPD_NOT_INITIALIZED: LPD MIO is used for WDT but LPD is not initialized |
| 0x200 | XPLM_ERR_TASK_CREATE: Error when task create fails<br>This can happen when maximum tasks are created. |
| 0x201 | XPLM_ERR_PM_MOD: Error initializing the PM Module |
| 0x202 | XPLM_ERR_LPD_MOD: Error initializing the LPD modules |
| 0x203 | XPLM_ERR_EXCEPTION: Exception has occurred during PLM initialization<br>EAR and ESR are printed on the UART console if enabled. |
| 0x204 | XPLM_ERR_NPLL_LOCK: Unable to lock NOC PLL for master SLR devices |
| 0x205 | XPLM_ERR_STL_MOD: Error initializing the STL module |
| 0x300 | XLOADER_UNSUPPORTED_BOOT_MODE: Error for unsupported boot mode<br>This error occurs if invalid boot mode is selected or boot mode peripheral is not selected in the CIPS wizard. |
| 0x302 | XLOADER_ERR_IMGHDR_TBL: Multiple conditions can cause this error:<br>• If image header table has the wrong checksum<br>• If PLM is unable to read the image header table |
| 0x303 | XLOADER_ERR_IMGHDR: Error if image header checksum fails |
| 0x304 | XLOADER_ERR_PRTNHDR: Error if partition header checksum fails |
| 0x305 | XLOADER_ERR_WAKEUP_A72_0: Error waking up the A72-0 during handoff<br>Check the PLM minor code for the PM error code. |
| 0x306 | XLOADER_ERR_WAKEUP_A72_1: Error waking up the A72-1 during handoff<br>Check the PLM minor code for the PM error code. |
| 0x307 | XLOADER_ERR_WAKEUP_R5_0: Error waking up the R5-0 during handoff<br>Check the PLM minor code for the PM error code. |
| 0x308 | XLOADER_ERR_WAKEUP_R5_1: Error waking up the R5-1 during handoff.<br>Check the PLM minor code for PM error code. |
| 0x309 | XLOADER_ERR_WAKEUP_R5_L: Error waking up the R5-L during handoff<br>Check the PLM minor code for the PM error code. |

*Table 14:* **PLM Error Codes** *(cont'd)*

| Value | Description |
|---|---|
| 0x30A | XLOADER_ERR_WAKEUP_PSM: Error waking up the PSM during handoff<br>Check the PLM minor code for the PM error code. |
| 0x30B | XLOADER_ERR_PL_NOT_PWRUP: Error powering up the PL |
| 0x30C | XLOADER_ERR_UNSUPPORTED_OSPI: Error due to unsupported OSPI flash |
| 0x30D | XLOADER_ERR_UNSUPPORTED_OSPI_SIZE: Error due to unsupported OSPI flash size |
| 0x30E | XLOADER_ERR_OSPI_INIT: Error when OSPI driver lookup fails<br>This error occurs when OSPI is not selected in CIPS. |
| 0x30F | XLOADER_ERR_OSPI_CFG: Error when OSPI driver CFG fails |
| 0x310 | XLOADER_ERR_OSPI_SEL_FLASH: Error when OSPI driver is unable to select flash<br>Check minor code for the OSPI driver error code. |
| 0x311 | XLOADER_ERR_OSPI_READID: Error when OSPI ReadID fails |
| 0x312 | XLOADER_ERR_OSPI_READ: Error when OSPI driver read fails<br>Check minor code for the OSPI driver error code. |
| 0x313 | XLOADER_ERR_OSPI_4BMODE: Error when OSPI is unable to enter/exit 4B mode |
| 0x314 | XLOADER_ERR_QSPI_READ_ID: Error when QSPI read fails |
| 0x315 | XLOADER_ERR_UNSUPPORTED_QSPI: Error when QSPI flash is not supported |
| 0x316 | XLOADER_ERR_QSPI_INIT: Error when QSPI driver look up or confguration fails |
| 0x317 | XLOADER_ERR_QSPI_MANUAL_START: Error when QSPI driver manual start fails |
| 0x318 | XLOADER_ERR_QSPI_PRESCALER_CLK: Error when QSPI driver Prescalar setting fails |
| 0x319 | XLOADER_ERR_QSPI_CONNECTION: Error when invalid QSPI connection listed other than single, dual, or stacked |
| 0x31A | XLOADER_ERR_QSPI_READ: Error when QSPI driver read fails |
| 0x31B | XLOADER_ERR_QSPI_LENGTH: Error when QSPI read length is greater than flash size |
| 0x31C | XLOADER_ERR_SD_INIT: Error when SD mount fails |
| 0x31D | XLOADER_ERR_SD_F_OPEN: Error when SD file open fails<br>This can happen when file is not present or read from SD fails. File system error code is present in the PLM minor code. |
| 0x31E | XLOADER_ERR_SD_F_LSEEK: Error when f_seek fails while reading from SD card |
| 0x31F | XLOADER_ERR_SD_F_READ: Error while reading from SD card |
| 0x320 | XLOADER_ERR_IMG_ID_NOT_FOUND: Error when Image ID is not found in subsystem while reloading image |
| 0x321 | XLOADER_ERR_TCM_ADDR_OUTOF_RANGE: Error while loading to TCM and if address is out of range |
| 0x322 | XLOADER_ERR_CFRAME_LOOKUP: Error when CFRAME driver look up fails |
| 0x323 | XLOADER_ERR_CFRAME_CFG: Error when CFRAME driver CFG fails |
| 0x324 | XLOADER_ERR_UNSUPPORTED_SEC_BOOT_MODE: Error due to unsupported secondary boot mode |
| 0x325 | XLOADER_ERR_SECURE_METAHDR: Error when meta header secure validations fail |
| 0X326 | XLOADER_ERR_GEN_IDCODE: Error caused due to mismatch in IDCODEs |
| 0x327 | XLOADER_ERR_USB_LOOKUP: Error when USB lookup fails |
| 0x328 | XLOADER_ERR_USB_CFG: Error when USB configuration initialize fails |
| 0x329 | XLOADER_ERR_USB_START: Error when USB fails to start |
| 0x32A | XLOADER_ERR_DFU_DWNLD: Error when PDI fails to download |

*Table 14:* **PLM Error Codes** *(cont'd)*

| Value | Description |
|---|---|
| `0x32B` | XLOADER_ERR_DEFERRED_CDO_PROCESS: Error occurred while processing CDO but error is deferred till whole CDO processing is completed |
| `0x32C` | XLOADER_ERR_SD_LOOKUP: Error when SD look up fails |
| `0x32D` | XLOADER_ERR_SD_CFG: Error when SD configuration fails |
| `0x32E` | XLOADER_ERR_SD_CARD_INIT: Error when SD card init fails |
| `0x32F` | XLOADER_ERR_MMC_PART_CONFIG: Error when MMC switch to user area in raw boot mode fails |
| `0x330` | XLOADER_ERR_SEM_STOP_SCAN: Error while stopping the XilSEM Scan |
| `0x331` | XLOADER_ERR_SEM_CFR_INIT: Error while starting the XilSEM Scan |
| `0x332` | XLOADER_ERR_DELAY_ATTRB: Error when both delay handoff and copy to image |
| `0x333` | XLOADER_ERR_NUM_HANDOFF_CPUS: Error when number of CPUs exceed maximum count |
| `0x334` | XLOADER_ERR_OSPI_CONN_MODE: Error when OSPI mode is not supported |
| `0x335` | XLOADER_ERR_OSPI_SEL_FLASH_CS1: Error when OSPI driver is unable to select flash CS1. Check minor code for OSPI driver error code. |
| `0x336` | XLOADER_ERR_OSPI_SDR_NON_PHY: Error when OSPI driver is unable to set the controller to SDR NON PHY mode |
| `0x337` | XLOADER_ERR_OSPI_COPY_OVERFLOW: Error when source address in OSPI copy exceeds flash size |
| `0x338` | XLOADER_ERR_SD_F_CLOSE: Error on closure of file in SD filesystem modes |
| `0x339` | XLOADER_ERR_SD_UMOUNT: Error on unmounting filesystem |
| `0x33A` | XLOADER_ERR_DMA_XFER: DMA Transfer failed |
| `0x33B` | XLOADER_ERR_DMA_XFER_SD_RAW: DMA Transfer failed in SD Raw |
| `0x33C` | XLOADER_ERR_CONFIG_SUBSYSTEM: Error while configuring subsystem |
| `0x33D` | XLOADER_ERR_COPY_TO_MEM: Error on copying image to DDR with the copy to memory attribute enabled |
| `0x33E` | XLOADER_ERR_DELAY_LOAD: When the image has delay load attribute set and the boot source is SMAP, SBI, PCIE or JTAG, the image is copied to PMC RAM to free it from the SBI buffers. Errors occurred during such copies to PMC RAM is denoted using this error code |
| `0x33F` | XLOADER_ERR_ADD_TASK_SCHEDULER: Error while adding task to the scheduler |
| `0x340` | XLOADER_ERR_SD_MAX_BOOT_FILES_LIMIT: Error code returned when search for bootable file crosses max limit |
| `0x341` | XLOADER_ERR_UNSUPPORTED_QSPI_FLASH_SIZE: Error when QSPI flash Size is not supported |
| `0x342` | XLOADER_ERR_PM_DEV_PSM_PROC: Failed in XPM Request Device for PM_DEV_PSM_PROC |
| `0x343` | XLOADER_ERR_PM_DEV_IOCTL_RPU0_SPLIT: Failed in XPM Device Ioctl for RPU0_0 in SPLIT mode |
| `0x344` | XLOADER_ERR_PM_DEV_IOCTL_RPU1_SPLIT: Failed in XPM Device Ioctl forRPU0_1 in SPLIT mode |
| `0x345` | XLOADER_ERR_PM_DEV_IOCTL_RPU0_LOCKSTEP: Failed to XPM Device Ioctl for RPU0_0 in LOCKSTEP mode |
| `0x346` | XLOADER_ERR_PM_DEV_IOCTL_RPU1_LOCKSTEP: Failed to XPM Device Ioctl for RPU0_1 in LOCKSTEP mode |
| `0x347` | XLOADER_ERR_PM_DEV_TCM_0_A: Failed to XPM Request Device for PM_DEV_TCM_0_A |
| `0x348` | XLOADER_ERR_PM_DEV_TCM_0_B: Failed to XPM Request Device for PM_DEV_TCM_0_B |
| `0x349` | XLOADER_ERR_PM_DEV_TCM_1_A: Failed to XPM Request Device for PM_DEV_TCM_1_A |
| `0x34A` | XLOADER_ERR_PM_DEV_TCM_1_B: Failed to XPM Request Device for PM_DEV_TCM_1_B |
| `0x34B` | XLOADER_ERR_PM_DEV_DDR_0: Failed to XPM Request Device for PM_DEV_DDR_0 |

*Table 14:* **PLM Error Codes** *(cont'd)*

| Value | Description |
|---|---|
| 0x34C | XLOADER_ERR_PM_DEV_QSPI: Failed to XPM Request Device for PM_DEV_QSPI |
| 0x34D | XLOADER_ERR_PM_DEV_SDIO_0: Failed to XPM Request Device for PM_DEV_SDIO_0 |
| 0x34E | XLOADER_ERR_PM_DEV_SDIO_1: Failed to XPM Request Device for PM_DEV_SDIo_1 |
| 0x34F | XLOADER_ERR_PM_DEV_USB_0: Failed to XPM Request Device for PM_DEV_USB_0 |
| 0x350 | XLOADER_ERR_PM_DEV_OSPI: Failed to XPM Request Device for PM_DEV_OSPI |
| 0x600 | XLOADER_ERR_INIT_GET_DMA: Failed to get DMA instance at time of initialization |
| 0x601 | XLOADER_ERR_INIT_INVALID_CHECKSUM_TYPE: Only SHA3 checksum is supported |
| 0x602 | XLOADER_ERR_INIT_CHECKSUM_COPY_FAIL: Failed when copying checksum from flash device |
| 0x603 | XLOADER_ERR_INIT_AC_COPY_FAIL: Failed when copying AC from flash device |
| 0x604 | XLOADER_ERR_INIT_CHECKSUM_INVLD_WITH_AUTHDEC: Failed as checksum was enabled with authentication and encryption enabled |
| 0x605 | XLOADER_ERR_DMA_TRANSFER: DMA Transfer failed while copying |
| 0x606 | XLOADER_ERR_IHT_AUTH_DISABLED: Authentication is not enabled for Image Header table |
| 0x607 | XLOADER_ERR_IHT_GET_DMA: Failed to get DMA instance for IHT authentication |
| 0x608 | XLOADER_ERR_IHT_COPY_FAIL: Failed when copying IHT AC from flash device |
| 0x609 | XLOADER_ERR_IHT_HASH_CALC_FAIL: Failed to calculate hash for IHT authentication |
| 0x60A | XLOADER_ERR_IHT_AUTH_FAIL: Failed to authenticate IHT |
| 0x60B | XLOADER_ERR_HDR_COPY_FAIL: Failed when copying IH/PH from flash device |
| 0x60C | XLOADER_ERR_HDR_AES_OP_FAIL: Failed due to AES init or Decrypt init or key selection failure |
| 0x60D | XLOADER_ERR_HDR_DEC_FAIL: Failed to decrypt image header/partition |
| 0x60E | XLOADER_ERR_HDR_AUTH_FAIL: Failed to authenticate image header/partition |
| 0x60F | XLOADER_ERR_HDR_NOT_SECURE: Neither authentication nor encryption is enabled for image header/ partition |
| 0x610 | XLOADER_ERR_HDR_GET_DMA: Failed to get DMA instance for image header/partition authentication/ decryption |
| 0x611 | XLOADER_ERR_HDR_HASH_CALC_FAIL: Failed to calculate hash for image header/partition authentication |
| 0x612 | XLOADER_ERR_HDR_NOT_ENCRYPTED: Image header/partition header is not encrypted |
| 0x613 | XLOADER_ERR_HDR_AUTH_DISABLED: Authentication disabled for image header/partition header |
| 0x614 | XLOADER_ERR_SEC_IH_READ_VERIFY_FAIL: Failed to read image header and verify checksum |
| 0x615 | XLOADER_ERR_SEC_PH_READ_VERIFY_FAIL: Failed to read partition header and verify checksum |
| 0x616 | XLOADER_ERR_PRTN_HASH_CALC_FAIL: Hash calculation failed for partition authentication |
| 0x617 | XLOADER_ERR_PRTN_AUTH_FAIL: Partition authentication failed |
| 0x618 | XLOADER_ERR_PRTN_HASH_COMPARE_FAIL: Partition hash comparison failed |
| 0x619 | XLOADER_ERR_PRTN_DECRYPT_FAIL: Partition decryption failed |
| 0x61A | XLOADER_ERR_AHWROT_EFUSE_AUTH_COMPULSORY: PPK Programmed but eFUSE authentication is disabled |
| 0x61B | XLOADER_ERR_AHWROT_BH_AUTH_NOT_ALLOWED: PPK Programmed and BH authentication is enabled |
| 0x61C | XLOADER_ERR_AUTH_EN_PPK_HASH_ZERO: PPK not programmed and authentication is enabled |
| 0x61D | XLOADER_ERR_SHWROT_ENC_COMPULSORY: Encryption is disabled |

*Table 14:* **PLM Error Codes** *(cont'd)*

| Value | Description |
|---|---|
| 0x61E | XLOADER_ERR_KAT_FAILED: Known answer tests (KAT) failed |
| 0x61F | XLOADER_ERR_DATA_COPY_FAIL: Data copy to internal memory failed |
| 0x620 | XLOADER_ERR_METAHDR_LEN_OVERFLOW: Failed when total size is greater than Metahdr length |
| 0x621 | XLOADER_ERR_AUTH_JTAG_EFUSE_AUTH_COMPULSORY: Jtag Authentication failed when PPK not programmed |
| 0x622 | XLOADER_ERR_AUTH_JTAG_DISABLED: Jtag Authentication disable efuse bit is set |
| 0x623 | XLOADER_ERR_AUTH_JTAG_PPK_VERIFY_FAIL: Jtag Authentication failed when verification of PPK |
| 0x624 | XLOADER_ERR_AUTH_JTAG_SIGN_VERIFY_FAIL: Jtag Authentication failed when verification of signature failed |
| 0x625 | XLOADER_ERR_AUTH_JTAG_EXCEED_ATTEMPTS: Jtag Authentication failed more than once |
| 0x626 | XLOADER_ERR_AUTH_JTAG_GET_DMA: Failed to get DMA instance for JTAG authentication |
| 0x627 | XLOADER_ERR_AUTH_JTAG_HASH_CALCULATION_FAIL: Hash calculation failed before signature verification |
| 0x628 | XLOADER_ERR_AUTH_JTAG_DMA_XFR: Failed to get Auth Jtag data with DMA transfer |
| 0x2XYZ | • 0x2 indicates failure in CDO command<br>• X indicates command module.<br>◦ 1 - PLM<br>◦ 2 -PM<br>◦ 3 - XilSEM<br>◦ 7 - Loader<br>◦ 8 - Error<br>• YZ indicate handler ID of the CDO command. |

# PLM Interface (XilPLMI)

The PLM Interface (XilPLMI) is a low-level interface layer for the PLM main application and other PLM modules. XilPLMI provides the common functionality required for the modules that run with PLM. Each new module can register itself with the supported command handlers. These command handlers are executed based on the request received from other modules or from other subsystems.

XilPLMI also includes the CDO parser that parses and executes any CDO commands. XilPLMI implements the Generic Module that includes generic commands to be used by all other modules. The XilPLMI layer provides:

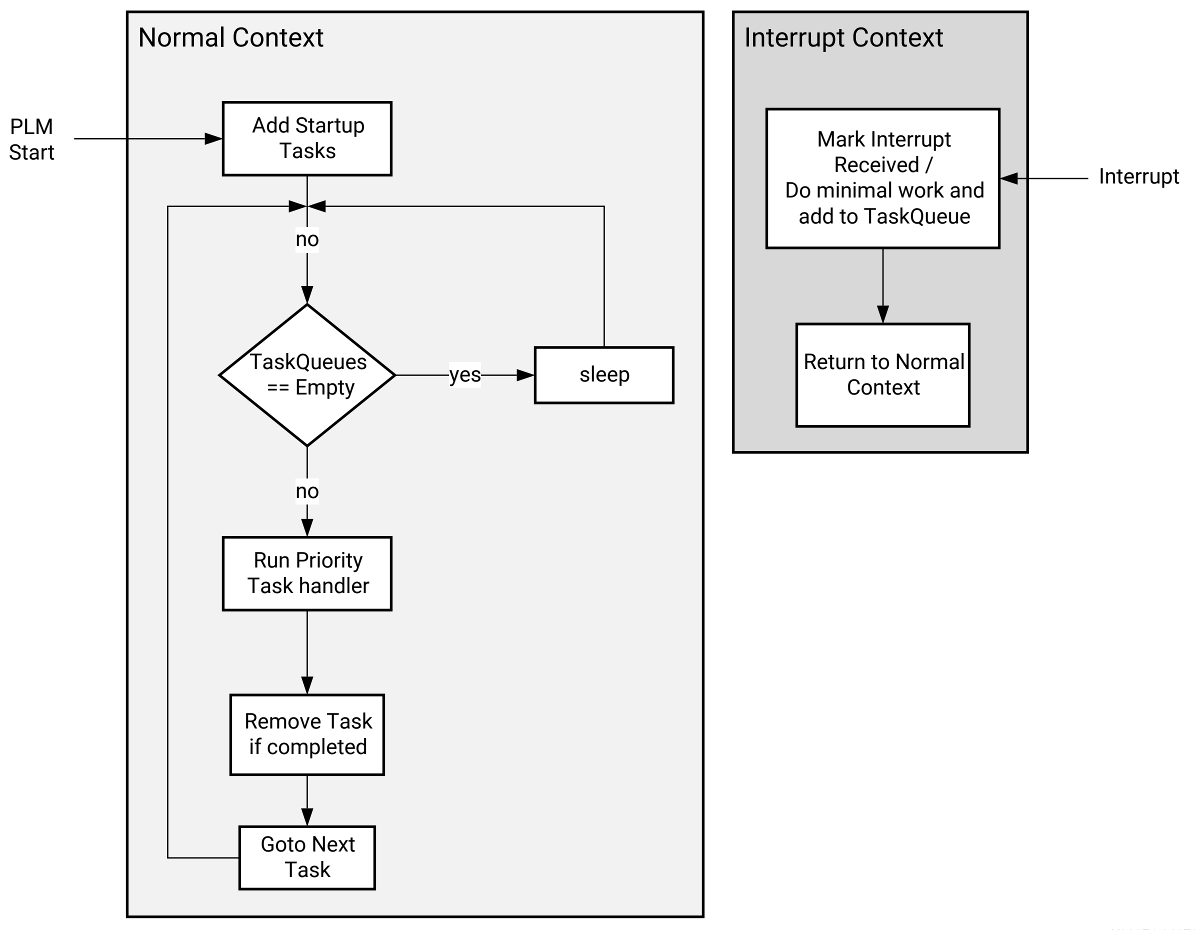
- Interface for parsing CDO files
- Implementation for general PLM CDO commands
- Interface to register handlers for commands that can be part of CDO and IPI
- Interface to set error actions and register notifications for error events

- Interface to schedule timer events
- Debug print levels and common utilities
- **Task Dispatcher Loop:** XilPLMI uses a very simple run-to-completion, time-limited priority task loop model, to get real-time behavior. The main program is a simple loop that looks up the next task from a queue of tasks, and calls the function to execute the task.

  This model is simple in the sense that because all tasks are executed until they are done, there are no critical regions and no locking needed. Any code in the PLM can access any global data without having to worry about any other tasks being in the middle of updating the data.

  The complexity with the run-to-completion model comes when a particular task needs to run longer than for the maximum allowed time limit (GTL). If that happens, split the tasks into multiple events.

*Figure 28:* **Task Dispatch Loop**




- **Modules:** XilPLMI provides an interface layer for modules to register the commands for PLM. Commands can come from the CDO or IPI.

- **Configurable Parameters:**
  - **Modules:** Configurable to include or exclude modules, boot drivers based on your needs.
  - **Debug Prints:** Configurable to include multiple levels of print statements. For more information, refer to PLM Build Flags.
- **CDO:** The CDO file contains the configuration information generated by the tools in the form of CDO commands. The module that supports the CDO registers itself to the PLM during the module initialization phase. XilPLMI provides the API to parse the CDO file and propagates the commands and its payload to the respective modules.
- **IPI Handling:** The PLM handles IPI interrupts, so that messages can be exchanged between the PLM and other processor on the Versal device. Data that is sent through IPI follows the CDO format.
- **Scheduler:** This is a simple timer-based function scheduler to support execution of periodic tasks. A scheduler is required by modules such as XilSEM to support periodic tasks such as SEU detection scan operations.

  ***Note*:** Accuracy of the PMC IRO clock affects the scheduler accuracy.
- **PLM Watchdog Timer:** PLM has the framework to update the PLM health periodically. PLM toggles Multiplexed I/Os (MIO) for this purpose when you use an external Watchdog Timer (WDT). PMC MIO is preferred but if you use a LPD MIO, then the WDT will be disabled when the PS is going down.

  You can enable the PLM WDT by using the `SetWdt CDO` command with the parameters `MIO PIN` and `periodicity`. Before running the `SetWdt CDO` command, ensure that you configure MIO as GPIO and load the corresponding PMC / LPD CDO. PDI loading is considered as Configuration mode and WDT is not supported during Configuration mode for the 2020.2 release. Other operations considered as configuration mode are boot, partial PDI loading, and subsystem restart/resume. The minimum scheduler period is 15 ms.

  The implementation is as follows:
  - **Normal Context:**
    - PLM will set a variable to indicate ALIVE in between every task.
    - During PDI load, PLM mode is set to Configuration mode, and then reset to Operational mode when PDI is loaded.
    - If LPD MIO is used, WDT is disabled during LPD shutdown.
    - If any task takes longer than the WDT periodicity, the ALIVE bit will be set and external WDT can reset the device.
  - **Interrupt Context:**
    - Current scheduler period is 10 ms. This period indicates that for every 10 ms, the PLM gets a timer interrupt to schedule the tasks.

- The WDT handler is called in the scheduler.
- When the WDT is enabled, the WDT handler performs following operation periodically based on configured periodicity (for example, for a periodicity of 100 ms, the following tasks run ~10 ms before expiry of periodicity)
  - When in configuration mode, the WDT handler toggles the MIO pin irrespective of the PLM alive status.
  - When in operation mode, the WDT handler toggles the MIO pin only when PLM alive status is set and clear the PLM alive status.

| Command: Set PLM WDT | | | |
|---|---|---|---|
| Reserved[31:24]=0 | Length[23:16]=2 | PLM=1 | CMD_SET_PLM_WDT=22 |
| NODE Idx - PMC MIO PIN / LPD MIO PIN | | | |
| [31:16] Reserved | | [15:0]Periodicity in ms - Default = 100ms | |

## PLM Event Logging

Event logging is categorized into the following features:

- Logging of PLM terminal prints
- Logging of trace events

### *Logging PLM Terminal Prints*

The PLM supports logging of PLM terminal prints to memory. This feature helps you easily debug if the UART is not present. This logging is based on the log level that you enabled. Each print message is logged with the PLM timestamp. The following are the features that are supported as part of logging:

- **Configure print log level:** While building the PLM, you can decide the log level to use. Based on the PLM, the ELF file is generated. The PLM provides the provision to change the log level by sending the IPI command to it. The IPI command can reduce/expand the log level.
- **Configure debug log buffer memory:** By default, the log buffer is 16K in size in the PMC RAM, and the PLM logs the prints to the PMC RAM memory. This feature allows changing the debug log memory to a different memory location.
- **Retrieve debug log buffer:** You can retrieve the logged prints to the memory of your choice.
- **Retrieve debug log buffer information:** You can retrieve information of PLM prints logging.
- **Retrieve terminal prints:** You can retrieve terminal prints with the following command: `xsdb > plm log`

## Logging Trace Events

The PLM supports the logging of trace events to memory. This memory is different than the memory used for logging the PLM prints. Currently, the PLM logs only PDI load trace events and is timestamped. The following features are supported by the PLM as part of tracing:

- **Configure trace log buffer memory:** By default, the PLM logs the trace events to the PMC RAM memory. This feature allows you to change the trace log memory to a different memory location.
- **Retrieve trace log buffer:** Retrieve the logged trace events to a memory of your choice.
- **Retrieve trace log buffer information:** Retrieve information of trace events logging.

For examples of logging trace events, see Event Trace Log Command Examples.

### Trace Events Log Format

*Table 15:* **Trace Events Log Format**

| Structure | |
|---|---|
| Trace event length including timestamp | Trace ID |
| Time stamp in milliseconds | |
| Time stamp in fraction | |
| Payload | |
| ... | |

Currently, the PLM logs only the list of images that are loaded as part of trace events logging.

*Table 16:* **Trace Event Table**

| Trace Event ID | Trace Event Length | Payload |
|---|---|---|
| XPLMI_TRACE_LOG_LOAD_IMAGE = 0x1 | 3 | Image ID |

### Event Trace Log Command Examples

The following examples show the structure of the event trace logs.

#### Example: Change the Log-Level to Debug Information

| Structure |
|---|
| CMD - 0x020113 (Event logging command) |
| SubCmd - 0x1 (Sub command to change log level) |
| Arg1 - 0x4 (Change to Debug Info) |

### Example: Change Debug Log Buffer Address

The following command structure changes the log buffer address from 1M with a size of 512K.

| Structure |
|---|
| CMD - 0x040113 (Event logging command) |
| SubCmd – 0x2 (Sub command to change log buffer address) |
| Arg1 - 0x0 (High Address) |
| Arg2 – 0x100000 (Low Address 1M) |
| Arg3 – 0x80000 (Size of log buffer 512K) |

### Example: Copy the Logged Data

| Structure |
|---|
| CMD - 0x030113 (Event logging command) |
| SubCmd – 0x3 (Sub command to copy the log buffer to below specified address) |
| Arg1 - 0x0 (High Address) |
| Arg2 – 0x100000 (Low Address 1M) |

### Example: Get the Log Buffer Details

| Structure |
|---|
| CMD - 0x010113 (Event logging command) |
| SubCmd – 0x7 (Sub command to get the event log buffer details) |
| Response Payload0: Command Status |
| Response Payload1: High Address |
| Response Payload2: Low Address |
| Response Payload3: Buffer offset till where the log is valid |
| Response Payload4: Log Buffer length |
| Response Payload5: Indicates that the log buffer is full |

## *Event Logging IPI Command*

*Table 17:* **Event Logging Command**

| Structure | | | |
|---|---|---|---|
| Reserved[31:24]=0 | Length[23:16]=4 | PLM=1 | CMD_EVENT_LOGGING=19 |
| Sub Command | | | |
| Argument 1 | | | |
| Argument 2 | | | |
| Argument 3 | | | |

The PLM supports logging PLM Terminal Prints and Trace Events to separate buffers. This command configures the buffers for logging, and also retrieves buffer information.

Sub Command information is as follows:

- **1U:** Use this sub command to configure the log level of the PLM terminal during run time. This log level can be less than or equal to the log level set at compile time.
  - **Argument 1:** Log Level. This argument can be one or a combination of the following entities:
    - **0x1U:** Unconditional messages (DEBUG_PRINT_ALWAYS)
    - **0x2U:** General debug messages (DEBUG_GENERAL)
    - **0x4U:** More debug information (DEBUG_INFO)
    - **0x8U:** Detailed debug information (DEBUG_DETAILED)
- **2U:** Use this sub command to configure the debug log buffer memory. By default this memory is configured to PMC RAM. You can change this memory as per your need.
  - **Argument 1:** High Address where PLM terminal prints are logged
  - **Argument 2:** Low Address where PLM terminal prints are logged
  - **Argument 3:** Length of Debug log buffer
- **3U:** Use this sub command is to copy the logged PLM prints from the Debug log buffer to where you need them.
  - **Argument 1:** High Address to which the log buffer data is copied
  - **Argument 2:** Low Address to which the log buffer data is copied
- **4U:** Use this sub command to retrieve the debug log buffer memory details where the PLM prints are logged. The response is as follows:
  - **Payload 0:** Command status
  - **Payload 1:** Debug log buffer High Address where the logging occurs
  - **Payload 2:** Debug log buffer Low Address where the logging occurs
  - **Payload 3:** Debug log buffer offset till where the logging occurs
  - **Payload 4:** Debug log buffer length
  - **Payload 5:** Indicates whether the debug log buffer is full
- **5U:** Use this sub command to configure the trace log buffer memory. By default, this memory is configured to PMC RAM. You can change this memory based on your need.

    - **Argument 1:** High Address where Trace events are logged
    - **Argument 2:** Low Address where Trace events are logged. Length of the Trace log buffer

- **6U:** Use this sub command to copy the logged trace events from the Trace log buffer to where you want them to be.
    - **Argument 1:** High Address to which you want to copy the trace log buffer data
    - **Argument 2:** Low Address to which you want to copy the trace log buffer data
- **7U:** Use this sub command is to retrieve the details of the trace log buffer memory where the trace events are logged. The response is as follows.
- **Payload 0:** Command status
- **Payload 1:** Trace log buffer High Address where the logging occurs
- **Payload 2:** Trace log buffer Low Address where the logging occurs
- **Payload 3:** Trace log buffer offset till where the logging occurred
- **Payload 4:** Trace log buffer length
- **Payload 5:** Indicates whether the trace log buffer is full

### Error Manager

The Error Management module in the PLM initializes and handles the hardware-generated errors across the Versal platform, and provides an option to customize these error actions. The error management support is one of the important run-time platform management activity supported by PLM. In the hardware, there are two error status registers for PMC and two error status registers for PSM that contain the type of error occurrence. You can enable or disable an error from interrupting the PMC/PSM MicroBlaze processor. The Error Manager code present in PLMI, maintains an Error Table that contains Handler Pointer, Error Action and Subsystem (to shutdown or restart in case of SUBSYSTEM SHUTDOWN or SUBSYSTEM RESTART error action) information for each error that is routed to PMC and PSM. You can set any supported error action for each of the errors to take an appropriate action when an error occurs.

The possible error actions include:

- Generation of a power-on-reset (POR)
- Generation of a system reset
- Assertion of error out signal on the device
- No action (This disables all actions on the error and clear the corresponding status bit)

The PLM Error Manager provides APIs for assigning a default error action in response to an error. During initialization of the PLM modules, the PLMI initializes the Error Manager, enables errors, and sets error action for each error in accordance with the Error Table structure defined in the `xplmi_err.c` file.

### Error Management Hardware

The Versal device has a dedicated error handler to aggregate and handle fatal errors across the SoC. The Error Manager handles the fatal errors using its hardware to trigger either SRST/PoR/Error out, or an interrupt to PMC/PSM.

For more information, refer to the *Versal ACAP Technical Reference Manual* (AM011).

### Error Management API Calls

The PLM Error Manager supports the following APIs:

- **XPlmi_EmSetAction:** Use this API to set up the error action. This API takes Error Node ID, Error Mask, Action ID and Error Handler if the Action ID is Interrupt to PMC as input arguments. When this function is called, it disables the existing error actions, and sets the action specified in Action ID for the given error.
- **XPlmi_EmDisable:** Use this API to disable all error actions for a given Error ID. This API takes Error ID as the input argument.

### Error Management CDO Commands

The following CDO commands are supported by the PLM Error Management module.

#### Set EM Action

*Table 18:* **Command: Set EM Action**

| Structure | | | |
|---|---|---|---|
| Reserved[31:24]=0 | Length[23:16]=3 | EM=8 | CMD_SET_EM_ACTION=1 |
| Error Event ID | | | |
| Reserved | | Action ID | |
| Error Mask | | | |

Use this command to set the error action for the specified Error Event ID and Error Mask. Refer to the following Error Event ID Table for list of Error Event IDs supported and Error Masks supported. Error management APIs are not supported over IPI, at present.

- **Actions:**
  - **Power On Reset:** 0x1
  - **System Reset:** 0x2

- **Error Out:** 0x4
- **None:** 0x7. Disable all actions on the Event and clear error status

***Note***: For PSM error events, the command returns failure if LPD is not initialized.

### Register Notifier for EM Events

*Table 19:* **Command: Register Notifier**

| Structure | | | |
|---|---|---|---|
| Reserved[31:24]=0 | Length[23:16]=4 | PMC_XILPM=2 | CMD_PM_REGISTER_NOTIFIER=5 |
| Node ID (Error Event ID) | | | |
| Event Mask (Error Mask) | | | |
| Argument 1 | | | |
| Argument 2 | | | |

EM supports notifying a subsystem when registered error occurs, using the register notifier API supported by XilPM. Use this command to register for notifications when registered errors occur. Refer to the Error Event ID Table for a list of supported Error Event IDs and Error Masks.

- Node ID: Can either be a Device ID or Error Event ID. Use an Error Event ID for registering error events.
- Event Mask
  - For Device ID: Event Type
  - For Error Event ID: Error Mask
- Argument 1
  - For Device ID: Wake
- Argument 2
  - For Device ID: Enable

Register notifier of an event of Error Event ID enables the error event by clearing the corresponding PMC/PSM_ERR#N_STATUS bit and writes to the corresponding PMC/PSM_IRQ#N_EN. The notifier returns an event index (which is a bit that notify callback sets) to indicate the occurrence of the event.

The Register Notifier command works with the Notify Callback command.

For example, register notifier of Error Node GT_CR error event clears PMC_ERR1_STATUS.GT_CR, enables PMC_IRQ1_EN.GT_CR, and returns a number, for example, 5. Notify Callback sets bit 5 of the Event Status to indicate that the GT_CR error has occurred.

### Notify Callback

*Table 20:* **Command: Notify Callback**

| Structure | | | |
|---|---|---|---|
| Reserved[31:24]=0 | Length[23:16]=4 | PMC_LIBPM=2 | CMD_PM_NOTIFY_CALLBACK |
| Node ID (Error Event ID) | | | |
| Event Status (Error Mask) | | | |

On notify callback of an event of Error Event ID, the Error Node is disabled. For example, notification of the Error Node GT_CR error event disables the error by writing to PMC_IRQ1_DIS.GT_CR. You must re-register to be notified again.

### Error Events

For a complete list of the available error events, refer to the *Versal ACAP Technical Reference Manual* (AM011).

# XilLoader

The PDI contains all the images that must be loaded into the system. PLM reads the PDI from the boot device and loads the images to the respective memories based on the image header.

***Note*:** For more details on the PDI format, see the PDI section in Chapter 7: Boot and Configuration.

The XilLoader provides an interface for the modules to load/start/look up the images present in the PDI. XilLoader also interacts with XilSecure (for secure boot), XilPM (for subsystem bring up), and boot drivers (for boot).

The following XilLoader functions are available:

- **XLoader_LoadAndStartSubSystemPdi:** This function takes PDI pointer as input. This API is called to load the images present in the PDI image, and start the subsystem based on the hand-off information present in the PDI. Based on the Image and Partition information present in the PDI, this API reads and loads each image partition from the PDI source. The API checks if the image requires hand-off and calls the XilPM APIs to perform the handoff.

  The image can be a CDO partition or the subsystem image. If the image is a CDO partition, the API calls the CDO parser API from the PLMI. If the image is a subsystem executable partition, the API loads the partition to the respective subsystem's memory. After loading the image, this API reads the CPI ID and hand-off address from the PDI, and calls the XilPM APIs with this information. If there is any error while loading or starting the image, this API returns the appropriate error code. Otherwise, this API returns SUCCESS.

- **XLoader_RestartImage:** Each image has its own unique ID in the PDI. This function takes the Image ID as input to identify image in the PDI and restart it. Based on the Image ID, this API parses through the subsystem information stored during boot, and obtains the image number and the partition number that is present in the PDI. This API then reads the Image partitions and loads them. In addition, this API checks if the image requires hand-off, and calls the XilPM APIs accordingly. If there is any error while restarting the image, the API returns an appropriate error code. Otherwise, the API returns SUCCESS.

### Sequence of Operations

The following is the sequence of operations that happen between the loader and other components.

1. Based on the boot mode, the corresponding flash drivers are used to read the PDI.
2. Based on the image headers, the loader gets the CPU details for an image.
3. The loader uses the XilPlmi API for loading CDOs, and the XilPM API to start/stop the processors, and to initialize memory.
4. The loader uses XilSecure to perform checksum, authentication, or decryption on the image.

### Boot Drivers

XilLoader implements the following high-level boot device interface APIs that are called to initialize the boot device, and to load the images from the boot device to the corresponding subsystem memory. These APIs internally call the driver APIs as required.

- **Flash Drivers:** QSPI, OSPI, SD/eMMC drivers are used to read the images from the flash device.
- **PMC DMA:** APIs for PMC DMA related functionality.

## XilLoader/IPI CDO Commands

The following table provides the list of commands that are supported by the XilLoader.

*Table 21:* **XilLoader/IPI CDO Commands**

| Command | API ID |
|---|---|
| Reserved | 0 |
| Load Partial PDI | 1 |
| Load DDR Copy Image | 2 |

### Load Partial PDI

The Load Partial PDI CDO command with IPI interface is used to support partial reconfiguration from DDR and flash devices. Partial reconfiguration request comes through IPI command with source field specified as DDR/QSPI/OSPI flashes. The PdiSrc field will have the same values as boot mode values when used as primary boot device.

*Table 22:* **Load Partial PDI Structure**

| Structure | | | |
|---|---|---|---|
| Reserved[31:24]=0 | Length[23:16]=3 | XilLoader=7 | CMD_XILLOADER_LOAD_PPDI = 1 |
| PdiSrc – 0x1=QSPI24, 0x2=QSPI32, 0x8=OSPI, 0xF for DDR | | | |
| High PDI Address | | | |
| Low PDI Address | | | |

### Load DDR Copy Image

*Table 23:* **Load DDR Copy Image Structure**

| Structure | | | |
|---|---|---|---|
| Reserved[31:24]=0 | Length[23:16]=1 | XilLoader=7 | CMD_XILLOADER_LOAD_DDR_CPY_IMG = 2 |
| Image ID | | | |
| Function ID | | | |

## Deferred Image Loading

The PLM supports the Copy to Memory and Deferred Image Handoff features. Based on the scenario, you might have to defer loading a certain subsystem image in the PDI after a certain boot milestone, or defer the handing off to a certain subsystem image. This establishes a certain boot order and can be used to decrease boot time.

### Copy to Memory

Images with the `copy = <DDR address>` attribute specified against them in the BIF file are stored at the specified address in DDR and then loaded during boot PDI load. Specifying the `copy` attribute takes a backup of the image in DDR, while loading the image. Specifying `delay_load` in BIF for an image skips the loading of the image altogether.

Specifying `copy = <DDR address>, delay_load` in BIF takes the back up of the image in DDR and skips loading of the image altogether.

Images are loaded by an IPI command `0x10702` that takes the `image ID, function ID` as an argument. The IPI command loads and executes these images after all other images start their run. You cannot specify `delay_load` and `delay_handoff` simultaneously for an image.

However, you can specify `copy` and `delay_handoff` simultaneously for an image.

For partial PDIs, the Copy to Memory feature is not supported.

### Deferred Image Handoff

You can pass the `delay_handoff` attribute to the selected image IDs in the BIF file, which is an input to Bootgen. The PLM reads the `delay_handoff` attribute and defers the handoff of the images till the end of boot PDI load. In other words, the images with the `delay_handoff` attribute specified, only start running after the images without the `delay_handoff` and `copy = 1` attributes start their run. Deferred Image Handoff is supported for both full PDIs and partial PDIs.

# XilPM

Platform Management (XilPM) is a library that provides interfaces to create and manage subsystems, MIO, Clocks, Power and Reset settings of nodes. The following table provides the list of commands supported by this module. For details about Platform Management, refer to the Chapter 10: Versal ACAP Platform Management.

*Table 24:* **Platform Management Modules**

| Command Name | API ID |
|---|---|
| **Node-Related Commands** | |
| PM_GET_NODE_STATUS | 3 |
| PM_REQUEST_SUSPEND | 6 |
| PM_SELF_SUSPEND | 7 |
| PM_ABORT_SUSPEND | 9 |
| PM_REQUEST_WAKEUP | 10 |
| PM_SET_WAKEUP_SOURCE | 11 |
| PM_REQUEST_NODE | 13 |
| PM_RELEASE_NODE | 14 |
| PM_SET_REQUIREMENT | 15 |
| PM_SET_MAX_LATENCY | 16 |
| **Reset Control Commands** | |
| PM_RESET_ASSERT | 17 |
| PM_RESET_GET_STATUS | 18 |
| **Pin Control Commands** | |
| PM_PINCTRL_REQUEST | 28 |
| PM_PINCTRL_RELEASE | 29 |
| PM_PINCTRL_GET_FUNCTION | 30 |

*Table 24:* **Platform Management Modules** *(cont'd)*

| Command Name | API ID |
|---|---|
| PM_PINCTRL_SET_FUNCTION | 31 |
| PM_PINCTRL_CONFIG_PARAM_GET | 32 |
| PM_PINCTRL_CONFIG_PARAM_SET | 33 |
| **Generic Commands** | |
| PM_GET_API_VERSION | 1 |
| PM_REGISTER_NOTIFIER | 5 |
| PM_FORCE_POWERDOWN | 8 |
| PM_SYSTEM_SHUTDOWN | 12 |
| PM_INIT_FINALIZE | 21 |
| PM_GET_CHIPID | 24 |
| PM_QUERY_DATA | 35 |
| PM_IOCTL | 34 |
| **Clock Control Commands** | |
| PM_CLOCK_ENABLE | 36 |
| PM_CLOCK_DISABLE | 37 |
| PM_CLOCK_GETSTATE | 38 |
| PM_CLOCK_SETRATE | 39 |
| PM_CLOCK_GETRATE | 40 |
| PM_CLOCK_SETDIVIDER | 41 |
| PM_CLOCK_GETDIVIDER | 42 |
| PM_CLOCK_SETPARENT | 43 |
| PM_CLOCK_GETPARENT | 44 |
| PM_PLL_SET_PARAMETER | 48 |
| PM_PLL_GET_PARAMETER | 49 |
| PM_PLL_SET_MODE | 50 |
| PM_PLL_GET_MODE | 51 |

# XilSecure

The XilSecure library is a library of security drivers that access the hardened cryptographic cores to support the AES-GCM 256-bit/128-bit engine, the RSA/ECC engine that supports RSA-4096, RSA-3076, RSA-2048 as well as ECDSA NIST P-384 and NIST P-521, and the SHA3/384 engine.

For more information, see Chapter 9: Security.

# XilSEM

The Xilinx Soft Error Mitigation (XilSEM) library is a pre-configured, pre-verified solution to detect and optionally correct soft errors in Configuration Memory of Versal ACAPs.

See the *OS and Libraries Document Collection* (UG643) for more information.

# PLM Usage

This section describes building PLM software using the Vitis™ tool including the build flags to use, the PLM memory layout and PLM reserved memory and registers. To perform the PLM build using the Vitis tool, refer to the *Xilinx Embedded Design Tutorials: Versal Adaptive Compute Acceleration Platform* (UG1305).

## PLM Build Flags

The following table lists the important build flags in PLM and their usage. For a complete list of build flags, see the `xplmi_config.h` file in the XilPLMI library. To apply changes to the build flags, rebuild the BSP and the PLM application.

*Table 25:* **PLM Build Flags**

| Flag | Description | Prerequisites | Default Setting |
|---|---|---|---|
| PLM_PRINT | When enabled, prints the PLM header and any mandatory prints. | None | Disabled |
| PLM_DEBUG | Prints basic information and any error messages. | None | Enabled |
| PLM_DEBUG_INFO | Prints with format specifiers in addition to the basic information. | None | Disabled |
| PLM_DEBUG_DETAILED | Prints detailed information. | None | Disabled |
| PLM_PRINT_PERF | Prints the time taken for loading the partitions, images and tasks. | Any of the above-mentioned debug print flags need to be enabled | Enabled |
| PLM_QSPI_EXCLUDE | When this flag is enabled, the QSPI code is excluded. | None | Disabled |
| PLM_SD_EXCLUDE | When this flag is enabled, the SD code is excluded. | None | Disabled |
| PLM_SEM_EXCLUDE | When this flag is enabled, the XilSEM code is excluded. | None | Disabled |

*Table 25:* **PLM Build Flags** *(cont'd)*

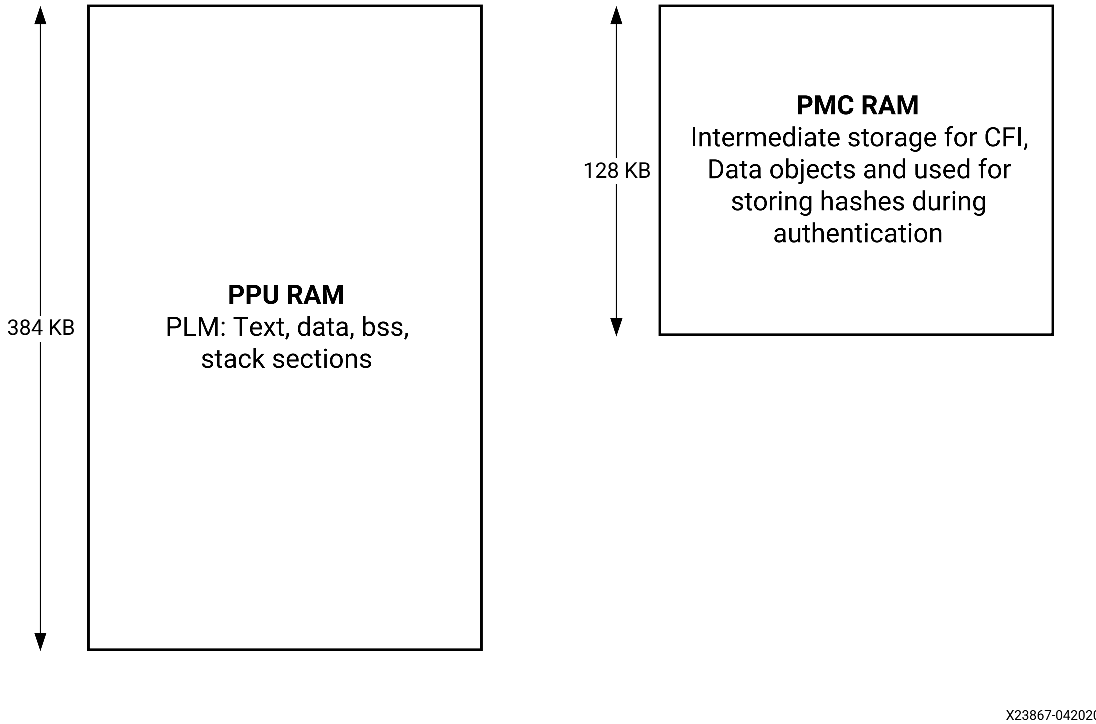
| Flag | Description | Prerequisites | Default Setting |
|---|---|---|---|
| PLM_DEBUG_MODE | Disables soft reset in case of error during Base PDI loading. This will be helpful in maintaining the system state for debugging. | None | Disabled |
| PLM_PRINT_PERF_POLL | Prints the time taken for any poll for MASK_POLL command. | None | Disabled |
| PLM_PRINT_PERF_DMA | Prints the time taken for PMC DMA, QSPI, OSPI | None | Disabled |
| PLM_PRINT_PERF_CDO_PROCESS | Prints the time taken to process the CDO file. | None | Disabled |
| PLM_PRINT_PERF_KEYHOLE | Prints the time taken to process keyhole command. Keyhole command is used for Cframe and slave SLR image loading. | None | Disabled |
| PLM_PRINT_PERF_PL | Prints the PL Power status and House clean status. | None | Disabled |

# PMC Memory Layout and PLM Footprint

This section contains the approximate details of the PMC memory layout and the PLM memory footprint, with the various PLM build options.

The following figure shows the PMC memory layout.

*Figure 29:* **PMC Memory Layout**




In the PLM, the PLM_DEBUG and PLM_PRINT_PERF build flags along with all modules, are enabled by default.

*Table 26:* **PLM Footprint**

| S. No | Feature/ Component | Size Occupied (KB) | Additional Notes | Remarks |
|---|---|---|---|---|
| 1 | PLM default build | 348 KB | Default PLM includes all PLM modules and basic PLM prints, and has time stamp enabled. (PLM_DEBUG and PLM_PRINT_PERF) | |
| 2 | PLM with more information debug prints enabled | 360 KB | Detailed debug prints are enabled when PLM_DEBUG_INFO flags are enabled. | This estimate is with the combination of (1) and (2) |

## Memory and Registers Reserved for PLM

The following registers and memory locations are reserved for PLM:

- **PPU RAM of size 384 KB:** Reserved for PLM for execution. This memory location is used for code, data, and BSS sections of the PLM ELF file.

- **PMC RAM of size 128 KB:** Used for tool generated data, for authentication/decryption chunks, slave boot modes, and event logging.
- **PMC_GLOBAL_GLOBAL_GEN_STORAGE0, and PMC_GLOBAL_GLOBAL_GEN_STORAGE1:** Registers reserved for the ROM to store the ROM execution time stamp. When the PLM is active, these two registers are read to obtain the ROM execution time.
- **PMC_GLOBAL_GLOBAL_GEN_STORAGE2:**

  Contains the device security status, updated by ROM. PLM uses this register to determine if KAT needs to be performed.
- **PMC_GLOBAL_GLOBAL_GEN_STORAGE4:**

  Used by PLM to store ATF handoff parameter address pointer.
- **PMC_GLOBAL_PERS_GLOB_GEN_STORAGE0:** Reserved for XilPM to save the status of each power domain initialization.
- **PMC_GLOBAL_PERS_GLOB_GEN_STORAGE1:** Reserved
- **PMC_GLOBAL_PERS_GLOB_GEN_STORAGE2:** Used by XilPM to store the reset reason.

# Services Flow

All interrupts are added as events and are run in the normal context. The PLM calls the appropriate handler based on the interrupts or inter-processor interrupts (IPIs) received.

# Exception Handling

The exception handler is invoked when an exception occurs in the PPU. This handler logs the exception data and sets the firmware error bit in the Error Manager.

If an exception occurs before the boot PDI programming is complete, the PLM Error Manager updates the PMC MultiBoot registers and triggers a system reset (SRST). However, if an exception occurs after the boot PDI programming is complete, then the PLM application runs in an infinite `while` loop.

# Security

This chapter describes the Versal™ device features that you can leverage to address security during the application boot time and run time.

Refer to the *Versal ACAP Security Manual* (UG1508) for the production readiness of the desired security feature as well as its detailed usage instructions. You can downloaded it from the Design Security Lounge (registration required).

## Security Features

The Versal device provides several security-related features. One of the biggest security features that Versal ACAP provides is the hardened cryptographic engines that support:

- Advanced encryption standard galois counter mode (AES-GCM) 128-bit and 256-bit
- RSA 2048, 3072, and 4096
- Elliptic curve cryptography (ECC) engine that supports multiple curves
- SHA3/384 Hashing

Because of the hardened cryptographic engines in Versal ACAP, Xilinx provides an associated set of security-related drivers that use the cryptographic engines either during secure boot or run time. During secure boot, the ROM, the PLM, and U-Boot can take advantage of these cryptographic features. During run time, these drivers can be accessed directly through a bare-metal application or indirectly depending on the architecture configuration. This can include using an OS, a hypervisor, trusted execution environment, etc. For example, in a Linux application, the application can call the Linux kernel, which would send an IPI request to the PLM where the security library runs. This is just one example of accessing the security libraries from run time; the options are numerous because Versal ACAP is highly configurable.

If there are any security features not provided by Xilinx, you can take advantage of the PL to implement additional security features or use the built-in Arm®v8 cryptographic extensions and the Arm NEON extensions in the Arm Cortex-A72 processors.

## Known Answer Tests

The Versal ACAP has the ability to perform KATs on the cryptographic engines before using them. The KAT checks the integrity of the hardened cryptographic engines before operating on the data.

The KAT includes the following tests:

- SHA3/384
- RSA-4096
- ECDSA with the NIST P-384 curve
- AES-GCM 256-bit with and without the DPA counter measure enabled

KATs can be run during boot. These APIs can be called explicitly from an application running on either Cortex-A72 or Cortex-R5F processors.

## Secure Boot

On Versal devices, secure boot ensures the confidentiality, integrity, and authentication of the firmware and software loaded onto the device. The root of trust starts with the BootROM, which authenticates and/or decrypts the PLM software. Now that the PLM software is trusted, the PLM handles loading the rest of the firmware and software in a secure manner. Secure boot is extremely important for two reasons.

1. Ensures that the software being loaded onto a device is allowed to be loaded, which prevents malicious code from running on the device
2. Protects the OEM IP because the software is stored in an encrypted fashion, which prevents the OEM IP from being stolen.

Additionally, if secure boot is not desired, then software can at least be validated with a simple checksum; however, keep in mind that the protections listed above do not apply when using this method of boot. The following table highlights the possible secure boot configurations.

*Table 27:* **Cumulative Secure Boot Operations**

| Boot Type | Operations | | | Hardware Crypto Engines |
|---|---|---|---|---|
| | Authentication | Decryption | Integrity (Checksum Verification) | |
| Non-secure boot | No | No | No | None |
| Asymmetric Hardware Root-of-Trust (A-HWRoT) | Yes (Required) | No | No | RSA/ECDSA along with SHA3 |
| Symmetric Hardware Root-of-Trust (S-HWRoT) (Forces decryption of PDI with eFUSE black key) | No | Yes (Required PLM and Meta Header should be encrypted with eFUSE KEK) | No | AES-GCM |
| A-HWRoT + S-HWRoT | Yes (Required) | Yes (Required) | No | RSA/ECDSA along with SHA3 and AES-GCM |
| Authentication + Decryption of PDI | Yes | Yes (Key source can be either from BBRAM or eFUSE) | No | RSA/ECDSA along with SHA3 and AES-GCM |
| Decryption (Uses user-selected key. The key source can be of any type such as BBRAM/ BHDR or even eFUSE) | No | Yes | No | AES-GCM |
| Checksum Verification | No | No | Yes | SHA3 |

The Versal ACAP system uses the following hardware cryptographic blocks in the secure boot process:

- **SHA Hardware Accelerator:** Calculates the SHA3/384 hash on images, used in conjugation with the RSA or elliptical curve cryptography (ECC) engine for signing.
- **ECDSA-RSA Hardware Accelerator:** Authenticates images using a public asymmetric key. Either RSA-4096 or ECDSA with curve NIST P-384 can be used.

  In addition to NIST-P384, NIST-P521 curve can also be used by the PLM for other images. P-381 is required for the MetaHeader, the PMC CDO, and the PLM. For all the other partitions, you can use P-521.
- **AES-GCM Hardened Crypto Block:** Decrypts images using a 256-bit key, and verifies the integrity of the decrypted image using the GCM tag.

  In addition to AES-GCM 256-bit, AES-GCM 128-bit can also be used by the PLM for other images. AES-GCM 256-bit is required for the MetaHeader, the PMC CDO, and the PLM. For all the other partitions, use AES-GCM 128-bit.

### Checksum Verification

Versal ACAP uses the SHA-3 digest to verify the data integrity of Versal device images. If the checksum verification is enabled in the PDI, the PLM calculates the SHA-3 digest and verifies it with the expected SHA-3 digest that is the part of PDI.

The checksum option does not work with A-HWRoT and/or S-HWRoT because these methods already perform data integrity checks.

## Asymmetric Hardware Root-of-Trust (A-HWRoT) (Authentication Required)

The Versal device image's SHA3 hash is signed with the private RSA/ECDSA key to generate a signature and is placed into the Versal device image. Upon boot, the SHA3 hash is calculated on the image, and the signature stored in the image is passed into the RSA/ECDSA engine using the public key. If both the calculated SHA3 hash and the verified signature match, the image is valid.

There are two public key types used in Versal ACAP: the primary public key (PPK) and the secondary public key (SPK). Each image is assigned its own or the same SPK. For example, the PLM could be assigned to use SPK0 and an application for the Cortex-A72 could be assigned the same SPK0 or its own SPK such as SPK1.

In the Versal device, there is storage for three PPK hashes in the eFUSE memory: PPK0, PPK1, and PPK2. If you program any of the PPK eFUSE bits, the A-HWRoT is forced at boot time, and therefore, all software needs to be authenticated before it is loaded into the Versal device. The asymmetric key pair can be either RSA 4096 or ECDSA–P384 curve. For the three PPK choices, a combination of RSA and ECDSA hash values are allowed to be programmed.

### RSA Engine

During boot, only RSA-4096 is supported. However, post boot the RSA engine is accessible and supports private and public key operations of sizes 2048, 3072, and 4096 bits.

### Elliptic Curve Cryptography Engine

During boot, the elliptic curve digital signature algorithm (ECDSA) is only supported with the NIST P-384 curve for the PLM, while other images can be signed using either the NIST P-384 curve or the NIST P-521 curve. However, post boot, the elliptic curve cryptography (ECC) engine is accessible, and supports a variety of curves in addition to the NIST P-384 curve. Supported curves are NIST P-384 and NIST P-521.

## Key Revocation

In eFUSEs, you have only three PPK choices to store the hash value of the primary public key and up to two of those values can be revoked. If another revocation occurs, the device is no longer bootable. If a PPK is compromised, you can revoke the public key by setting the corresponding PPK revocation bit in eFUSEs.

To revoke the SPK, you program the corresponding eFUSE bit in the Revocation ID. There are 256-bits [0-255] in total, so you can revoke the SPK 255 times. The 0-bit of Revocation ID represents KEY 0, the $32^{nd}$ bit of Revocation ID represents KEY 32, etc.

# Encryption

Versal devices include an AES-GCM hardware engine that supports confidentiality, authentication, and integrity. GCM assists in the authentication and integrity check. You can use the engine to encrypt or decrypt data with the use of a symmetric key and initialization vector pair boot or post-boot. The boot flow is required to use a key size of 256-bit and a 96-bit initialization vector, through loading of the PLM, and 256-bit or 128-bit key sizes are allowed for additional firmware and software. Post boot, the engine supports both 256-bit and 128-bit key sizes.

## DPA Counter Measure

In Versal devices, the AES engine has the capability of countermeasures, which means protection against DPA attacks. By default, the automotive devices (XA) include the DPA countermeasures. All other devices must be ordered with an ordering code to get DPA enabled devices.

As another counter measure against DPA attacks, boot images support key rolling to minimize the use of a single AES key. The DPA countermeasures can be used during boot as well as post-boot.

## Key Sources

### *Red Key*

The red key is stored as plain text in either BBRAM, eFUSEs, or the boot header.

### *Black Key*

The black key is produced after the red key is encrypted using the PUF generated key encryption key (KEK).

*Note*: The KEK is unique per Versal device and cannot be read out of the device.

The black key can be stored in eFUSE, BBRAM, or in the boot header for secure boot. If encrypt-only boot mode is selected, the black key can only be stored in eFUSEs .

### *User Key*

After the boot process, you have the option to load the AES engine with the keys stored in BBRAM, eFUSEs, as well as keys stored in memory that are specific to an operating system or an application.

### *Revocation*

The Revocation ID points to an eFUSE and if the eFUSE is *not* programmed, then keys using that Revocation ID are valid. Programming the eFUSE revokes the key associated with its associated eFUSEs. When revoking a key, it will also revoke an SPK because the Revocation ID is shared with the A-HWRoT boot mode.

## Symmetric Hardware Root-of-Trust (S-HWRot) Boot Mode (Encryption Required)

There are dedicated bits in eFUSEs that correspond to the S-HWRoT boot mode. If any one of these bits are set, then the boot image must be in the encrypted format and the key source is forced to be the eFUSE black key, where as remaining partitions can be either unencrypted or encrypted with any other valid key source.

*Chapter 10*

# Versal ACAP Platform Management

The Versal™ ACAP is a heterogeneous multiprocessor SoC that combines multiple user programmable processors, FPGA, and advanced power management capabilities.

Modern power efficient designs require usage of complex system architectures with several hardware options to reduce power consumption, and 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. In addition to power, there are other resources like clock, reset, and pins that need to be similarly managed. The challenge is to provide an intelligent software framework that complies to industry standard (IEEE P2415), and can handle all requests coming from multiple CPUs running different software. Xilinx® has created the platform management to support a flexible management control through the platform management controller (PMC).

The platform management handles several use case scenarios. For example, Linux provides basic power management capabilities, such as CPU frequency scaling, and CPU hot-plugging. The kernel relies on the underlining APIs to execute power management decisions, but most RTOS do not have this capability. Therefore, they rely on user implementation, which is made easier with use of the platform management framework.

Industrial applications, such as embedded vision, Advanced Driver Assistance, surveillance, portable medical, and Internet of Things (IoT) are increasing 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. Others, such as cloud and data center, have demanding cooling and energy costs, not including their need to reduce environmental cost. All these applications benefit from a flexible platform management solution.

### Key Features

The following are the key features of the platform management:

- Provides centralized power state, clock, reset, and pin configuration information using the PMC
- Supports Embedded Energy Management Interface (EEMI) APIs (IEEE P2415)
- Provides support for the Linux common clock framework
- Provides support for the Linux reset framework
- Manages power state, clock, and reset of all devices

- Provides support for Linux power management including:
  - Linux device tree power management
  - ATF/PSCI power management support
  - CPU frequency scaling
  - CPU hot-plugging
  - Suspend
  - Resume
  - Wake-up process management
  - Idle
- Provides direct control of the following power management features with more than 24 APIs:
  - Core management
  - Error management
  - Memories and peripherals management
  - Power, clock, reset
  - Processor unit suspend or wake-up management

# Versal ACAP Platform Management Overview

The Versal ACAP platform management implements the EEMI. Platform management allows software components running across different subsystems on Versal ACAP to issue or respond to platform management requests. The platform management also provides support through EEMI APIs to allow different processing units to configure clock characteristics, such as its enable/disable state, divisor, and the PLL to use. It also provides support through EEMI APIs for asserting/deasserting reset lines and configuring pins for use with different functional units.

# Versal ACAP Power Domains

The Versal device is divided into following power domains:

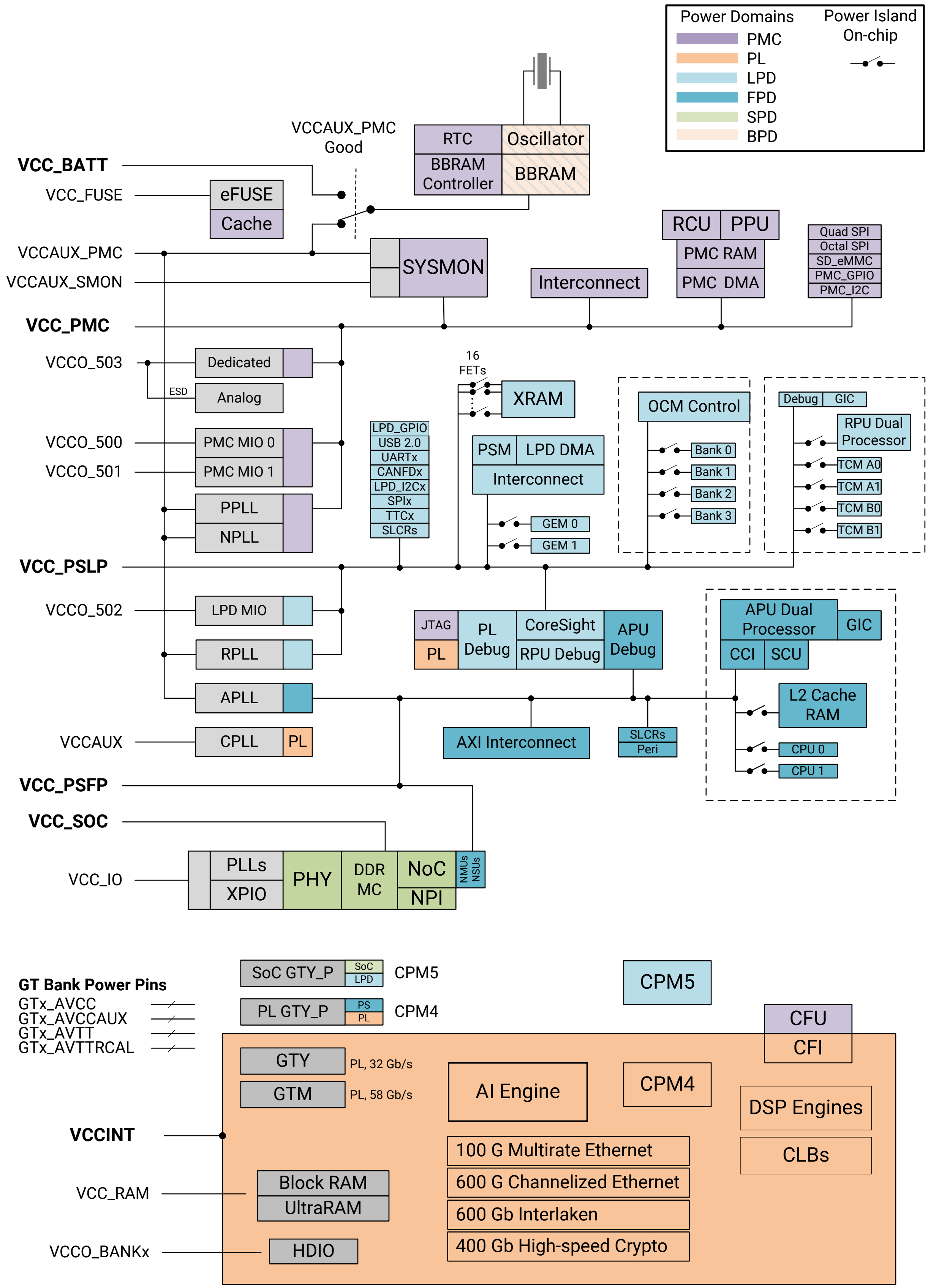
- **Full power domain (FPD):** Contains the Arm® Cortex-A72 application processor unit (APU).
- **Low power domain (LPD):** Contains the Arm Cortex-R5F real-time processor unit (RPU), and on-chip peripherals.

- **System power domain (SPD):** Contains the DDR controllers and NoC.
- **PL power domain:** Contains the PL and the AI Engine.
- **Battery power domain:** Contains the real-time clock (RTC) as well as battery-backed RAM (BBRAM).
- **PMC power domain:** Contains the platform management controller.

The battery and PMC power domains are not managed by the framework. Designs that want to take advantage of the platform management switching the power domains, must keep some power rails discrete. This allows individual rails to be powered off with the power domain switching logic.

The following figure illustrates the Versal device power domains and islands.

*Figure 30:* **Power Domains and Islands Diagram**



Because of the heterogeneous multi-core architecture of the Versal ACAP, no 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 controller (PMC). The PMC is the key component in coordinating the power management requests received from the other processing units, such as the APU or the RPU, and the coordination and execution from other processing units through the power management API.

Versal ACAP also supports inter-processor interrupts (IPIs), which are used as the basis for platform management related communication between the different processors. For more information, refer to the interrupts information in the *Versal ACAP Technical Reference Manual* (AM011).

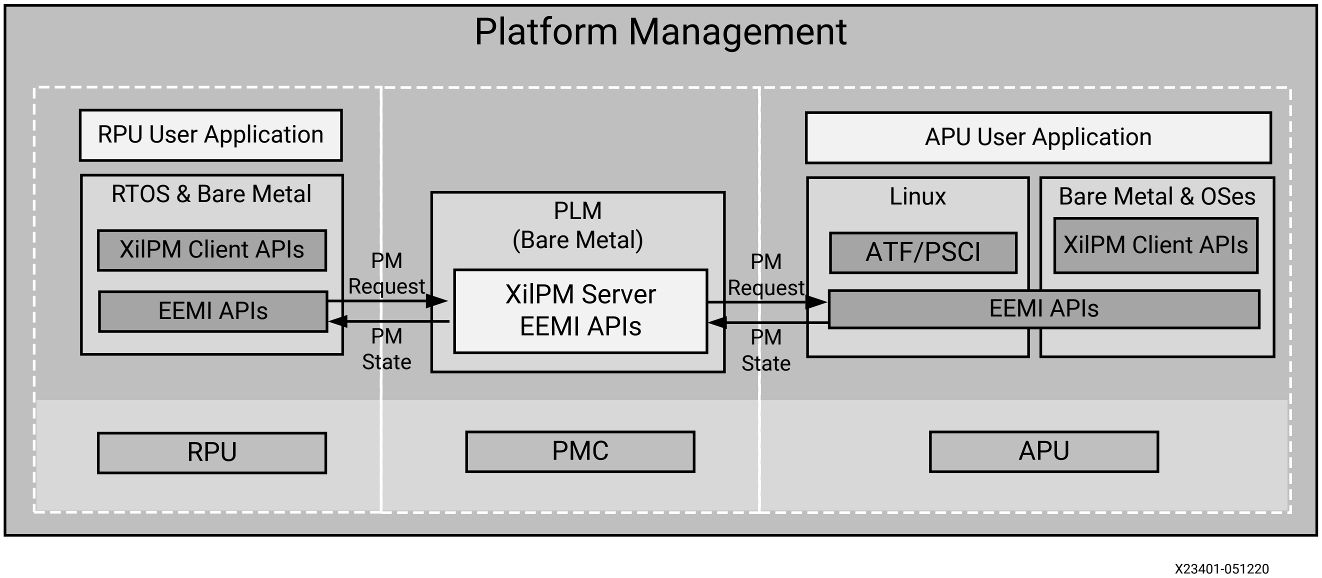
# Versal ACAP Platform Management Software Architecture

The Versal ACAP architecture employs a dedicated programmable unit that controls the power-up, power-down, and monitors of all system resources. You benefit from a system that is better equipped on handling platform management administration for a multiprocessor heterogeneous system. However, the system becomes more complex to operate. The platform management framework abstracts this complexity and exposes only the APIs you need to meet your power budget and efficiently manage resources.

The PMC consists of a PPU that runs the PLM. The PLM consists of several modules. One module, the XilPM server, implements a layer of the platform management framework. A second PPU MicroBlaze processor controls the PS management controller (PSM); PSM firmware is in charge of powering up and powering down power islands and power domains.

The PLM passes the topology of resources in the system to the platform management framework through CDOs. The platform management framework manages resources such as power domains, power islands, clocks, resets, pins and their relationship to CPU cores, memory, and peripheral devices.

*Figure 31:* **Platform Management Framework**




The Versal ACAP platform management framework is based on an implementation of EEMI (refer to the *Embedded Energy Management Interface EEMI API Reference Guide* (UG1200). APIs are available to the processing units to send messages to the PLM, as well as callback functions for the PLM to send messages to the processing units.

EEMI provides a common API that allows all software components to manage cores and peripherals. For power management, EEMI allows you to specify a high-level 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.

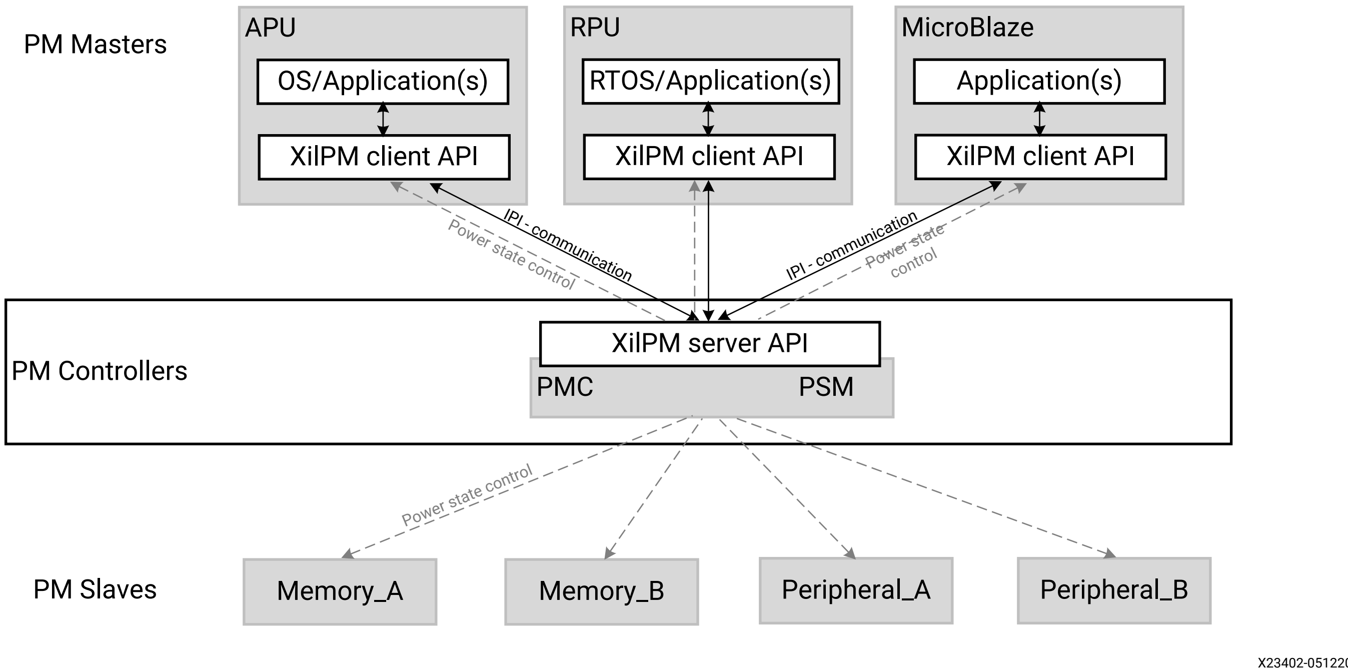
The Linux device tree provides a common description format for each device and its power characteristics. Linux also provides basic power management capabilities such as CPU and clock frequency scaling, and CPU hot-plugging. The kernel relies on the underlining APIs to execute power management decisions.

You can create your own platform management applications using the XilPM client library, which provides access to more than 24 APIs. The APIs can be grouped into the following functional categories:

- Slave device power management, such as memories and peripherals
- Clock management
- Reset management
- Pin management
- Miscellaneous

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

*Figure 32:* **API-Based Platform Management Software Architecture**




**Related Information**

Platform Loader and Manager

# API Calls and Responses

## *Platform Management Communication Using IPIs*

In the Versal device, the platform management communication layer is implemented using inter-processor interrupts (IPIs), provided by the IPI block. For more details on IPIs, see the Interrupts chapter of the *Versal ACAP Technical Reference Manual* (AM011).

Each processing unit has a dedicated IPI channel with the PMC, 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 following processing of the API:

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

### *Platform Management Layers*

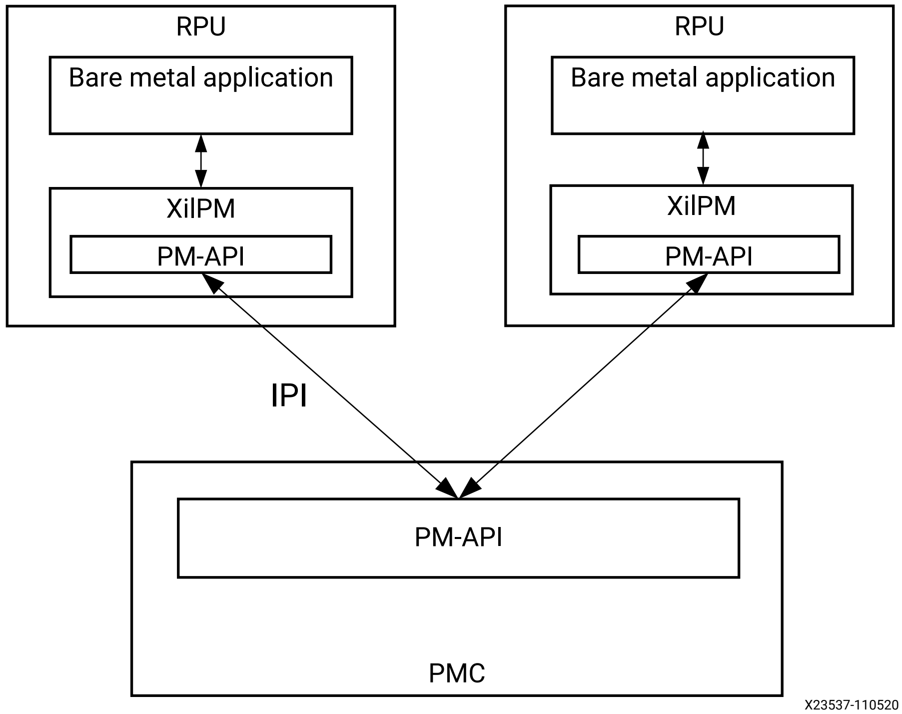
The following API layers are included in the platform management implementation for the Versal devices:

- **XilPM (client):** Library layer used by standalone applications in the different processing units, such as the APU and RPU.
- **XilPM (server):** Library layer part of the PLM firmware that handles the requests that are passed from the XilPM client layer through IPIs.
- **ATF:** The Arm trusted firmware (ATF) contains its own implementation of the client-side PM framework. ATF is currently used by the Linux OS.
- **PLM:** The PLM receives IPI packets and passes platform management requests to the XilPM server.
    - **XilPM (server):** Library layer part of the PLM firmware and handles the requests that are passed from XilPM client layer through IPIs.
- **PSM Firmware:** Invoked by the XilPM server to control power islands and power domains.

For more details on PMC, PSM, and power domain hardware topics, see the *Versal ACAP Technical Reference Manual* (AM011).

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

*Figure 33:* **API Layers Used Only With Bare-Metal Applications**

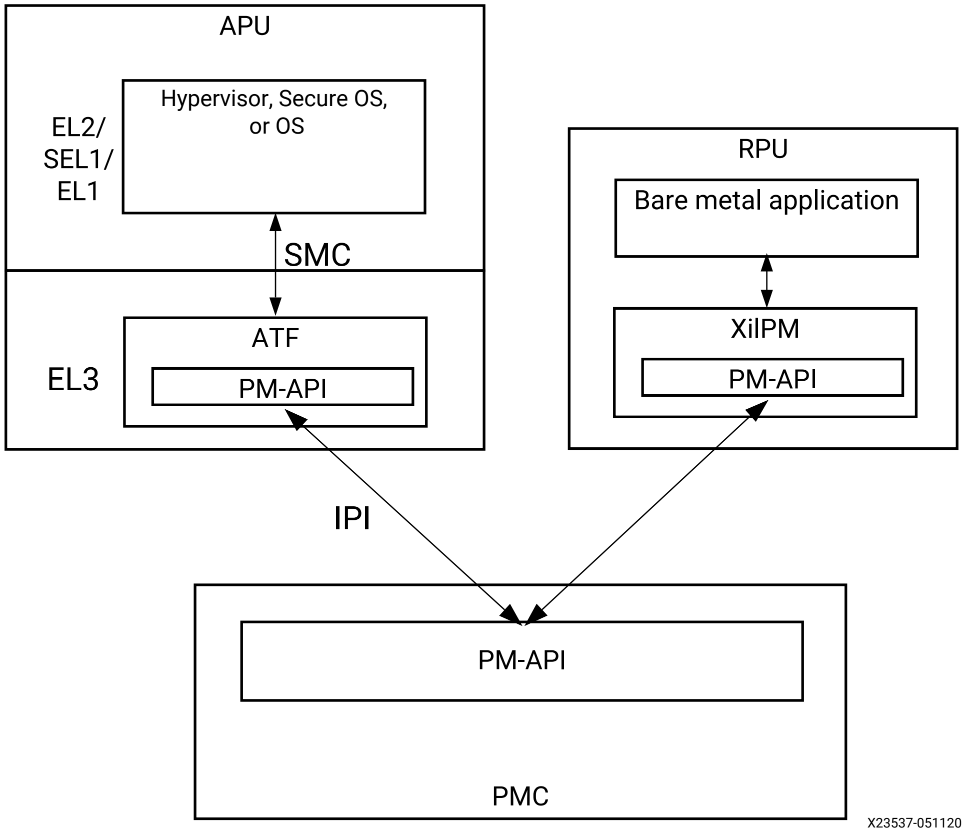



If the APU runs a complete software stack with an operating system, it does not use the XilPM library. Instead, the ATF running at EL3 implements the client-side XilPM API and provides a secure monitor call (SMC)-based interface to the software running at EL2, SEL1, or EL1 depending on the Versal ACAP device system architecture.

For more details on the Armv8 architecture and its different execution modes, see https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile.

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

*Figure 34:* **Platform Management Layers Involved When Running a Full Software Stack on the APU**




## Typical Platform Management API Call Flow

Any entity that is involved in power management is referred to as a node. The following sections describe how the platform management works with slave nodes allocated to each subsystem instead of the APU and the RPU.

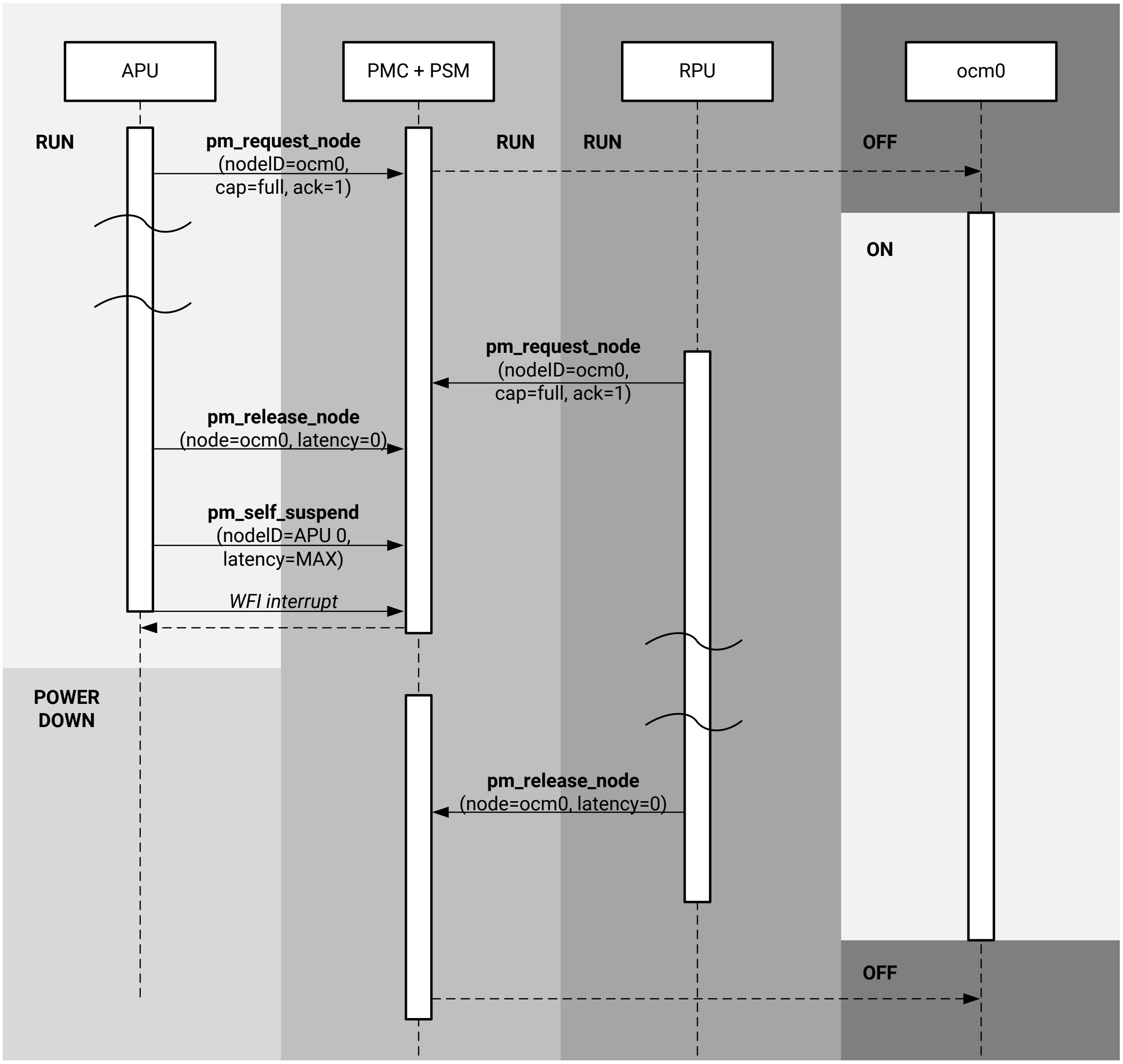
Generally, the APU or the RPU inform the PMC about their usage of a slave node, by requesting for it. The PMC is then informed about the capability requirement needed from the slave node. At this point, the PMC powers up the slave node, so it can be initialized by the APU or the RPU.

### *Requesting and Releasing Slave Nodes*

When a processing unit requires a peripheral or memory slave node, it must request the slave node using the power management API. After the slave node has performed its function and is no longer required, it can be released and powered off.

The following figure shows the call flow for a use case where the APU and the RPU are sharing an on-chip memory bank, `ocm0`.

*Figure 35:* **Platform Management Framework Call Sequence for APU and RPU Sharing an On-Chip 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. After the RPU also releases the `ocm0` node, then the PMC requests the PSM to power off the `ocm0` memory.

## Platform Management Default Subsystem

Currently, there is no support for isolation configuration, and all PM masters are part of the default subsystem.

In this configuration, both PM masters (APU and RPU) have permission to access all devices, so exercise caution on how PM slave devices are used by different masters.

### Activation of Subsystem

To activate a subsystem is to make it operational by requesting (using `XPm_RequestNode`) all its pre-allocation devices. This is an essential one-time operation that is required before the subsystem image can start execution. Every time a subsystem is restarted/shutdown, activation happens before execution of the new subsystem image.

When the application binaries (ELFs) are downloaded using Programmable Device Image (PDI) as a subsystem image, the PLM Firmware automatically activates the corresponding subsystem image using the subsystem image ID that is present in the BIF. Currently, this is the default subsystem as there is no support for custom subsystems. However, there can be debugging use-cases where you need to download and execute application binaries (ELFs) directly on a master using Xilinx System Debugger (XSDB), and not through the PDI-flow. In such cases, the PLM is not aware of the correspondingsubsystem image ID for ELFs. As a result, pre-allocation devices essential for a subsystem to be operational will not be requested, which may cause unforeseen issues. Therefore, before downloading any application binaries (ELFs) using XSDB, a one-time activation of a subsystem is required.

Currently, XSDB automatically requests PLM firmware to activate the default subsystem before downloading an application binary/ELF on a specific master. In future, when there are multiple/custom subsystems, you may need to explicitly activate the subsystem of choice for such debugging use-cases.

# Using the API for Power Management

This section contains detailed instructions on how to use the Xilinx platform management APIs to carry out common power management tasks. These APIs can be used by any bare-metal application that imports the XilPM client library.

## *Implementing Power Management on a Processor Unit*

The XilPM client library provides the functions that the standalone applications executing on a processor can use to initiate the power management API calls.

See *Zynq UltraScale+ MPSoC: Software Developers Guide* (UG1137) 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 the `XPm_InitXilpm` API. Because the argument to this API is a pointer to a proper IPI driver instance, there is a dependency for your design to have an IPI channel assigned to the PM master, so it could communicate with PMC.

For more information on IPIs, see the Interrupts information in the *Versal ACAP Technical Reference Manual* (AM011).

For more information about `XPm_InitXilpm`, see *OS and Libraries Document Collection* (UG643).

### Power Management Using XPm_InitFinalize

A subsystem sends the `XPm_InitFinalize(void)` requests the PLM firmware to finalize the subsystem configuration and power down unused nodes to maintain optimum power consumption.

For bare-metal application, an application developer needs to call `XPm_InitFinalize(void)` from the application.

For Linux applications, the platform management driver calls `XPm_InitFinalize()`.

A subsystem that is incapable of PM never sends this request. Therefore, its platform management devices remain powered up at all times, or until the PM subsystem itself is powered down.

If `XPm_InitFinalize()` is not called, the PLM firmware does not power down any device. The objective of `XPm_InitFinalize()` is to make the firmware aware that the caller subsystem of `XPm_InitFinalize()` is platform management capable (uses platform management APIs if it needs any device). `XPm_RequestNode()` will power up devices even if `XPm_InitFinalize()` is not invoked. Nodes will also be released through `XPm_ReleaseNode()` even if `XPmInitFinalize()` is not invoked `XPm_ReleaseNode()` also passes. However in this case, only the use count is decremented, and no power down operation is performed.

#### Reason Why PLM Firmware Does Not Power Down Device When XPm_InitFinalize Not Called

Consider a APU subsystem that is PM capable (uses PM APIs), and a RPU subsystem that is PM incapable (does not use PM API). Assume that both subsystems use TCM.

As the APU subsystem is PM-capable, it calls both `XPm_RequestNode(TCM)` and `XPm_InitFinalize()`. The PLM firmware then knows that the APU subsystem is PM capable, and calls `XPm_RequestNode()` when it requires a device.

In contrast, because the RPU is PM unaware, applications might be using devices without requesting the PM API, as Xilinx allows applications to run without using the XilPM library. The PLM firmware is aware that the RPU subsystem is running, but remains unaware about the devices that are used by the RPU. Therefore, the PLM firmware does not power down any device until each subsystem has called `XPm_InitFinalize()`.

Call `XPm_InitFinalize()` for proper power management and to obtain desired power values. The PLM firmware power downs all unused nodes when you call `XPm_InitFinalize()` from each subsystem. The PLM firmware also allows you to power down any device when you call the `XPm_ReleaseNode()` API.

A PM-capable subsystem sends an `XPm_InitFinalize()` request after initializing the sub-system. The PLM firmware then begins to power down the PM devices in this subsystem whenever they are not being used.

## Working with Slave Devices

The Versal ACAP platform management contains functions dedicated to managing slave devices (also referred to as PM slaves), such as memories and peripherals. Subsystems use these functions to inform the PMC about the requirements (such as capabilities and wake-up latencies) for those devices. The PMC manages the system so that each device resides in the lowest possible power state, meeting the requirements from all eligible subsystems.

## Requesting and Releasing a Node

When a subsystem requires a device node, either peripheral or memory, the device must be requested using the power management API. After the device node has performed its function and is no longer required, it should be released, so the device can be powered off or used by other subsystems.

When you call the `XPm_InitFinalize()` API, the platform management firmware turns off devices (such as a peripheral or memory) that are not requested by any subsystem.

You must pass the PM_CAP_ACCESS argument to the REQUEST_NODE API to access the particular peripheral/memory, as otherwise it is not powered ON.

The device, clock, and the reset operation are also not allowed if the device is not requested.

### Device Request Permission

The PLM firmware checks permissions for any subsystem that requests a device. While creating a subsystem in the Vivado tool, you can assign a single peripheral to multiple subsystems, or to a single subsystem. This information is exported to PMC CDO, and the access for requesting a device is allowed based on this information from the CDO. By default, this permission is shared for all subsystems, that is, a single device can be requested by multiple subsystems. The exceptions are clock, reset and the power operation that are allowed only if a single subsystem has requested any of these resources.

***Note*:** All subsystems must call the `XPm_InitFinalize()` API when they have finished initializing all their devices and requested devices using `XPm_RequestNode()`. Otherwise, the PLM firmware does not power down any device.

The following sequence is the ideal sequence for calling using the PM API for device management is as follows:

1. Call `XPm_RequestNode()` on all required devices.
2. Initialize devices.
3. Call `XPM_InitFinalize()` to inform PLM firmware that all required devices are requested and initialized

It is not mandatory that `XPm_RequestNode()` has to be called before `XPm_InitFinalize()`. If `XPm_InitFinalize()` is called before `XPm_RequestNode()`, the PLM firmware powers down that device (as initial state is ON). If `XPm_RequestNode()` is called after `XPm_InitFinalize()`, the PLM firmware powers up the device.

Some device initialization is done through CDO. Therefore, if `XPm_InitFinalize()` is called before `XPm_RequestNode()`, it is possible that initialization is lost as device is powered down first and then powered up again. It is recommended that you call `XPm_InitFinalize()` once it has requested all required devices. Otherwise, you need to take care of initialization again.

When you call `XPm_ReleaseNode()`, be mindful that the device powers down, and initialization configuration might be lost.

### Request and Release a Device From a Standalone Application

The application needs to call `XPm_RequestNode()`. to request usage of peripheral/device. For example:

```
XStatus XPm_RequestNode (const u32 DeviceId, const u32 Capabilities, const
u32 QoS, const u32 Ack);
```

The arguments are as follows:

- **Device ID:** Device ID of the PM device to be requested.
- **Capabilities:** Device-specific capabilities are required, and can be combined. The capabilities include:
  - PM_CAP_ACCESS: Full access / functionality
  - PM_CAP_CONTEXT: Preserve context
  - PM_CAP_WAKEUP: Emit wake interrupts
  - PM_CAP_UNUSABLE: Runtime suspend (Device is requested for a subsystem but the device is clock disabled)

  For more information, see *OS and Libraries Document Collection* (UG643).
- **QoS:** Quality of Service (0-100) is required.

  ***Note*:** Currently, this argument is not available.
- **Ack:** Requested acknowledge type.

***Note***: This argument is used only for ZynqMP. For Versal devices, this argument value is always set to blocking.

If a device is already requested by a subsystem, you can call `XPm_SetRequirement()` to change its requirement. For example:

```
XStatus XPm_SetRequirement (const u32 DeviceId, const u32 Capabilities,
const u32 QoS, const u32 Ack);
```

The application must release the device when it is no longer required. To release the device, call `XPm_ReleaseNode()`. For example:

```
XStatus XPm_ReleaseNode (const u32 DeviceId);
```

### Changing Requirements

When a subsystem is using a PM slave, its requirement on the capability of the slave can change. For example, an interface port might go into a low power state, or even be completely powered off, if the interface is not being used. The subsystem can use `XPm_SetRequirement` to change the capability requirement of the PM slave. Typically, the subsystem would not release the PM slave if it will be changing the requirement again in the future.

The following example call changes the requirement for the node argument so it is powered up and accessible to the PM master.

```
XPm_SetRequirement(node, PM_CAP_ACCESS, 0, REQUEST_ACK_NO);
```

> **IMPORTANT!** *Setting the requirements of a node to zero is not equivalent to releasing the PM slave. By releasing the PM slave, a subsystem might be allowing other subsystems to use the device exclusively.*

When multiple subsystems share a PM slave (this applies mostly to memories), the PMC selects a power state of the PM slave that satisfies all requirements of the requesting subsystems.

The requirements on a PM slave include capability as well as latency requirements. Capability requirements may include a top capability state, some intermediate capability states, an inactive state (but with the configuration retained), and the off state. Latency requirement specifies the maximum time allowed for the PM slave to switch to the top capability state from any other state. If this time limit cannot be met, the PMC will leave the PM slave in the top capability state regardless of other capability requirements.

For more information about `XPM_SetRequirement`, see *OS and Libraries Document Collection* (UG643).

### Self-Suspending

A processing unit can be a cluster of CPUs. The APU is a dual-core Arm Cortex-A72 CPU and RPU a dual-core Arm Cortex-R5F CPU. The RPUs can run independently (split mode) or in fault tolerant mode (lockstep). Currently, these processing units are part of the Default Subsystem.

Any processing unit can suspend itself by calling the `XPm_SelfSuspend` API to inform PMC about its intent. The processing unit must also inform the target state as part of this call. Currently, CPU Idle target state is supported for PU Suspend.

There are two types of target states CPU Idle and Suspend to RAM.

Actions performed by the respective processing units, PMC and PSM are discussed as follows in each case.

- **CPU Idle:**
  - Rich OS-like Linux has the capability to idle and subsequently power-down cores not being used by any process. This helps in power saving. If the workload increases, the OS can also wake up the powered down core. The platform management provides API calls as part of ATF and the XilPM (Client) library.
  - A processing unit can invoke CPU Idle in a bare-metal APU application use case by using `XPm_SelfSuspend` with the target state set as PM_SUSPEND_STATE_CPU_IDLE.
- **Suspend to RAM:**
  - The platform management provides the capability to suspend an entire subsystem. If the subsystem uses DDR and no other subsystem uses DDR, XilPM puts the DDR into Self-Refresh mode. This mode eliminates need for the DDRMC to refresh the DRAM.
  - A PM master that is part of subsystem (to be suspended) can invoke subsystem suspend using the `XPm_SelfSuspend` call to the PMC by setting target state as PM_SUSPEND_STATE_SUSPEND_TO_RAM.

For more information about `XPm_SelfSuspend` and `XPm_SuspendFinalize`, see *OS and Libraries Document Collection* (UG643).

### Setting a Wake-up Source

The platform management provides the option to power down a PU or a subsystem. The platform management can even power down the entire FPD if none of the FPD devices are in use and existing latency requirements allow this. If the FPD is powered off and the APU is woken up by an interrupt triggered by a device in the LPD or PMC domain, 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.

Before suspending, the PU must call the `XPm_SetWakeUpSource` API and add the slaves as a wake-up source. The PU can then set requirement to zero. However, to set the requirement to zero, the following conditions should be met:

1. No other subsystem can share the devices. In the present case, because only default subsystem is supported, this is not a consideration.
2. No other PM master (that is in Running state) present in the same subsystem, as the PU that is self suspending should use the slave device.

Setting the requirement to zero indicates to the platform management that the subsystem does not require these slaves to be up. After the PU finalizes the suspend procedure, provided no devices under FPD are being used, the PMC powers the entire FPD and configures the GIC proxy to enable propagation of the wake event of the LPD or PMC domain slaves.

For more information about `XPm_SetWakeupSource`, see *OS and Libraries Document Collection* (UG643).

### Resuming Execution

A CPU can be woken up by:

- Wake interrupt triggered by a hardware resource
- Explicit wake request using the `XPm_RequestWakeup` client API

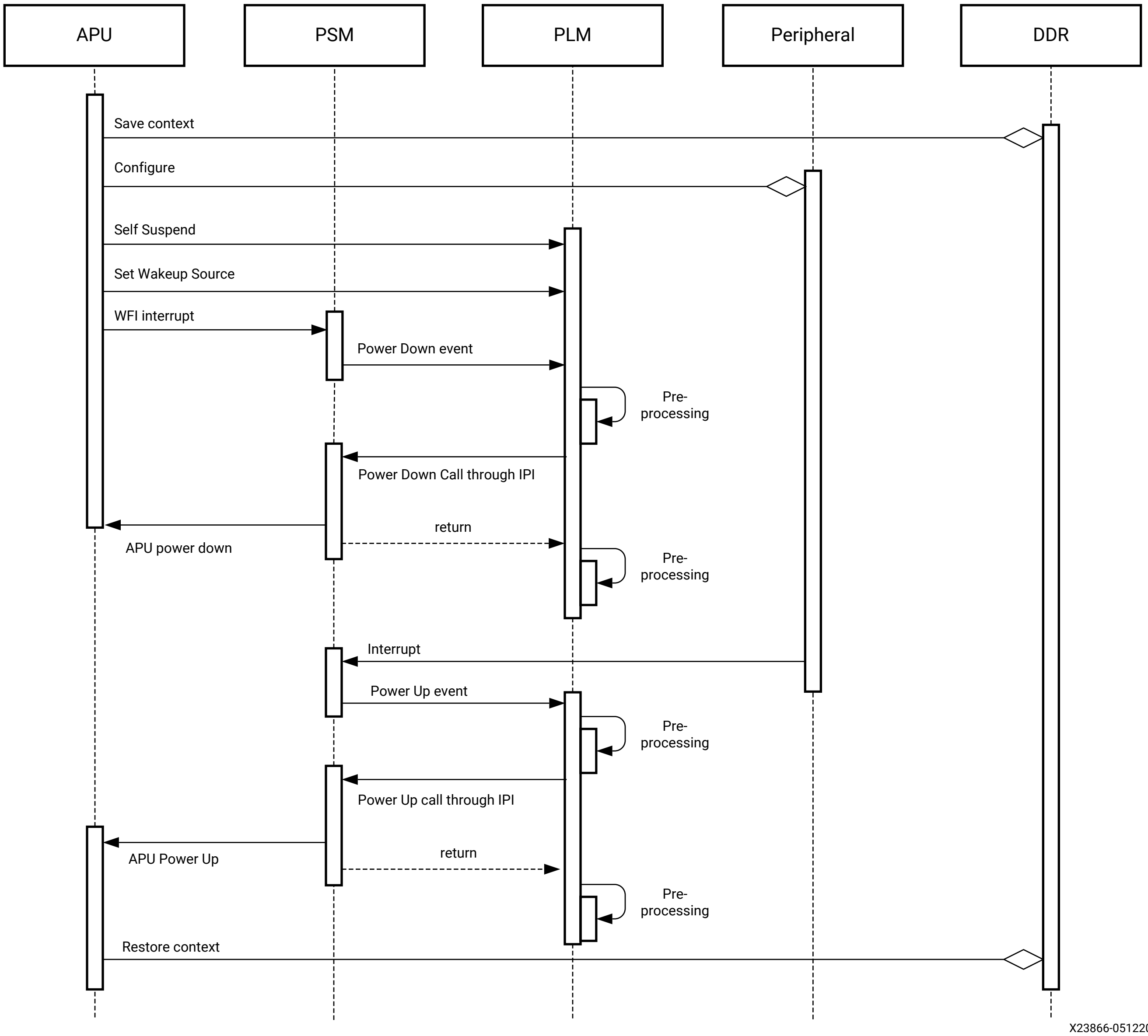
The CPU starts executing from the resume address provided with the `XPm_SelfSuspend` call.

For more information about `XPm_RequestWakeup` and `XPm_SelfSuspend`, see *OS and Libraries Document Collection* (UG643).

### Suspend-Resume Flow

The following figure shows the detailed interactions between different processors for a suspend resume use case.

*Figure 36:* **Versal ACAP APU Suspend/Resume Flow**




In this example, a peripheral is set up as a wake up source. Each step is explained as follows:

- **Suspend Flow:**
  1. APU saves the CPU context in DDR.
  2. Peripheral is configured to be used as a wake up source
  3. APU through ATF or XilPM client (if APU bare-metal application is running) informs PMC of its intent to suspend. PLM enables WFI interrupt.
  4. Informs PLM of the Wake Up Source.

5. APU interrupts PSM by going into WFI state. If APU runs a bare-metal application, PSM can be interrupted by using `XPm_SuspendFinalize` API.
6. After receiving the interrupt, a handshake between PSM and PLM occurs through the IPI. PSM powers down by power implementing power gating, asserting reset to APU and clock gating APU.

- **Wake Up Flow:**

1. The peripheral (configured by the APU to act as a wake up source) interrupts PSM.
2. PSM handshakes with PLM to initiate APU power up.
3. After the handshake, PSM powers up the APU. It disables power gating, deassert reset, and disables clock gating on the APU.
4. APU resumes context stored in the DDR.

### Suspending the Entire FPD Domain

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

For information on powering down the FPD, refer to `Suspend to RAM` in Self-Suspending.

### 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 PMC. The PMC releases all PM Slaves used by the APU automatically.

**IMPORTANT!** *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.*

For more information about `XPM_ForcePowerdown`, see *OS and Libraries Document Collection* (UG643).

## Using the API for Clock Management

There are EEMI APIs available that allow processing units to manage clocks in the system. Each clock is identified by a unique ClockId. A PM master can use the `XPm_Query` API to obtain a list of ClockIds accessible to that processing unit. Before a PM master can modify any attributes of a clock, the master should request the slave device that is associated with that clock. See the `XPm_RequestNode` API in *OS and Libraries Document Collection* (UG643) for more information.

Use the following APIs to get and set a clock's state:

- `XStatus XPm_ClockGetStatus(const u32 ClockId, u32 *const State)`
- `XStatus XPm_ClockEnable(const u32 ClockId)`
- `XStatus XPm_ClockDisable(const u32 ClockId)`

Use the following APIs to configure a clock to operate at a different divider value:

- `XStatus XPm_ClockGetDivider(const u32 ClockId, u32 *const Divider)`
- `XStatus XPm_ClockSetDivider(const u32 ClockId, const u32 Divider)`

Use the following APIs to configure clocks that could be driven by different parents:

- `XStatus XPm_ClockGetParent(const u32 ClockId, u32 *const ParentId)`
- `XStatus XPm_ClockSetParent(const u32 ClockId, const u32 ParentId)`

> **IMPORTANT!** *ParentId is an index to possible clocks that could be configured as a parent of ClockId. In Zynq UltraScale+ MPSoC, the definition of ParentId is not unified between Linux and standalone applications. In Versal ACAP, these EEMI APIs are unified in expecting ParentId to be an index value to possible parent clocks.*

Use the following APIs to configure the rate of reference clocks:

- `int XPm_ClockGetRate(const u32 ClockId, u32 *const Rate)`
- `int XPm_ClockSetRate(const u32 ClockId, const u32 Rate)`

> **IMPORTANT!** `XPm_ClockSetRate()` *can only be valid during CDO loading. You cannot set the clock rate from XilPM client API.*

Use the following APIs to configure PLLs that are identified by ClockId:

- `XStatus XPm_PllGetMode(const u32 ClockId, u32 *const Value)`
- `XStatus XPm_PllSetMode(const u32 ClockId, const u32 Value)`
- `XStatus XPm_PllGetParameter(const u32 ClockId, const enum XPm_PllConfigParams ParamId, u32 *const Value)`
- `XStatus XPm_PllSetParameter(const u32 ClockId, const enum XPm_PllConfigParams ParamId, const u32 Value)`

See *OS and Libraries Document Collection* (UG643) for more information about these APIs.

## Using the API for Reset Management

There are EEMI APIs available which allow PUs to manage reset lines in the system. Each reset is identified by a unique ResetId. In case of Linux as a PM master, a set of available ResetIds are obtained from the device tree. Before a PM master can modify state of a reset line, it should request the slave device that is associated with that reset. See the `XPm_RequestNode` API in *OS and Libraries Document Collection* (UG643) for more information.

Use the following APIs to manage resets:

- `XStatus XPm_ResetGetStatus(const u32 ResetId, u32 *const State)`
    - State is either 1 (asserted), or 2 (released)
- `XStatus XPm_ResetAssert(const u32 ResetId, const u32 Action)`
    - Action is either 0 (reset_release), 1 (reset_assert), or 2 (reset_pulse)

## Using the API for Pin Management

There are EEMI APIs available that allow PUs to manage pins in the system. Each pin is identified by a unique PinId. A PM master could obtain a list of PinIds accessible to that PU by using the `XPm_Query` API.

Use the following APIs to request or release control of a pin:

- `XStatus XPm_PinCtrlRequest(const u32 PinId)`
- `XStatus XPm_PinCtrlRelease(const u32 PinId)`

You can configure a pin for use by different functional units. Use the `XPm_Query` API to obtain a list of functional units accessible to a pin, which is identified by FunctionId. Request a pin before assigning a functional unit to a pin. Most functional units are associated with a device. Use the `XPm_RequestNode` API to request the device before assigning it to a pin.

- `XStatus XPm_PinCtrlGetFunction(const u32 PinId, u32 *const FunctionId)`
- `XStatus XPm_PinCtrlSetFunction(const u32 PinId, const u32 FunctionId)`

Use the following APIs to configure a pin for different operating characteristics. For a list of possible ParamIds and ParamVals, see *OS and Libraries Document Collection* (UG643).

- `XStatus XPm_PinCtrlGetParameter(const u32 PinId, const u32 ParamId, u32 *const ParamVal)`
- `XStatus XPm_PinCtrlSetParameter(const u32 PinId, const u32 ParamId, const u32 ParamVal)`

# Using Miscellaneous APIs

## *XPm_DevIoctl EEMI API*

The `XPm_DevIoctl` EEMI API allows a platform management master to perform specific operations to certain devices.

The following table lists the supported operations in Versal ACAP.

*Table 28:* **XPm_DevIoctl Operations**

| ID | Name | Description | Arguments | | | |
|---|---|---|---|---|---|---|
| | | | Node ID | Arg1 | Arg2 | Return Value |
| 0 | IOCTL_GET_RPU_OPER_MODE | Returns current RPU operating mode | NODE_RPU_0<br>NODE_RPU_1 | - | - | Operating mode:<br>0: LOCKSTEP<br>1: SPLIT |
| 1 | IOCTL_SET_RPU_OPER_MODE | Configures RPU operating mode | NODE_RPU_0<br>NODE_RPU_1 | Value of operating mode<br>0: LOCKSTEP<br>1: SPLIT | - | - |
| 2 | IOCTL_RPU_BOOT_ADDR_CONFIG | Configures RPU boot address | NODE_RPU_0<br>NODE_RPU_1 | Value to set for boot address<br>0: LOVEC/TCM<br>1: HIVEC/OCM | - | - |
| 3 | IOCTL_TCM_COMB_CONFIG | Configures TCM to be in split mode or combined mode | NODE_RPU_0<br>NODE_RPU_1 | Value to set (Split/ Combined)<br>0: SPLIT<br>1: COMB | - | - |
| 4 | IOCTL_SET_TAPDELAY_BYPASS | Enable/ disable tap delay bypass | NODE_QSPI | Type of tap delay<br>2: QSPI | Tapdelay Enable/ Disable<br>0: DISABLE<br>1: ENABLE | - |
| 6 | IOCTL_SD_DLL_RESET | Resets DLL logic for the SD device | NODE_SD_0,<br>NODE_SD_1 | SD DLL Reset type<br>0: ASSERT<br>1: RELEASE<br>2: PULSE | - | - |
| 7 | IOCTL_SET_SD_TAPDELAY | Sets input/ output tap delay for the SD device | NODE_SD_0,<br>NODE_SD_1 | Type of tap delay to set<br>0: INPUT<br>1: OUTPUT | Value to set for the tap delay | - |
| 12 | IOCTL_WRITE_GGS | Writes value to GGS register | - | GGS register index (0/1/2/3) | Register value to be written | - |

*Table 28:* **XPm_DevIoctl Operations** *(cont'd)*

| ID | Name | Description | Arguments | | | |
|---|---|---|---|---|---|---|
| | | | Node ID | Arg1 | Arg2 | Return Value |
| 13 | IOCTL_READ_GGS | Returns GGS register value | - | GGS register index (0/1/2/3) | - | Register value |
| 14 | IOCTL_WRITE_PGGS | Writes value to PGGS register | - | PGGS register index (0/1/2/3) | Register value to be written | - |
| 15 | IOCTL_READ_PGGS | Returns PGGS register value | - | PGGS register index (0/1/2/3) | - | Register value |
| 17 | IOCTL_SET_BOOT_HEALTH_STATUS | Sets healthy bit value to indicate boot health status to firmware | - | healthy bit value | - | - |

*Table 28:* **XPm_DevIoctl Operations** *(cont'd)*

| ID | Name | Description | Arguments | | | |
|---|---|---|---|---|---|---|
| | | | Node ID | Arg1 | Arg2 | Return Value |
| 19 | IOCTL_PROBE_COUNTER_READ | Read probe counter register of LPD/FPD | FPD/LPD power domain ID<br>0x4210002U for LPD<br>0x420C003U for FPD | Register configuration<br>- Counter Number (0 to 7 bit)<br>- Register Type (8 to 15 bit)<br>0 - LAR_LSR access (Request Type and Counter Number are ignored)<br>1 - Main Ctl (Counter Number is ignored)<br>2 - Config Ctl (Counter Number is ignored)<br>3 - State Period (Counter Number is ignored)<br>4 - PortSel<br>5 - Src<br>6 - Val<br>- Request Type (16 to 23 bit)<br>0 - Read Request<br>1 - Read Response<br>2 - Write Request<br>3 - Write Response<br>4 - lpd Read Request (For LPD only)<br>5 - lpd Read Response (For LPD only)<br>6 - lpd Write Request (For LPD only)<br>7 - lpd Write Response (For LPD only) | - | Register value |

*Table 28:* **XPm_DevIoctl Operations** *(cont'd)*

| ID | Name | Description | Arguments | | | |
|---|---|---|---|---|---|---|
| | | | Node ID | Arg1 | Arg2 | Return Value |
| 20 | IOCTL_PROBE_COUNTER_WRITE | Write probe counter register of LPD/FPD | FPD/LPD power domain ID<br>0x4210002U for LPD<br>0x420C003U for FPD | Register configuration<br>- Counter Number (0 to 7 bit)<br>- Register Type (8 to 15 bit)<br>0 - LAR_LSR access (Request Type and Counter Number are ignored)<br>1 - Main Ctl (Counter Number is ignored)<br>2 - Config Ctl (Counter Number is ignored)<br>3 - State Period (Counter Number is ignored)<br>4 - PortSel<br>5 - Src<br>- Request Type (16 to 23 bit)<br>0 - Read Request<br>1 - Read Response<br>2 - Write Request<br>3 - Write Response<br>4 - lpd Read Request (For LPD only)<br>5 - lpd Read Response (For LPD only)<br>6 - LPD Write Request (For LPD only)<br>7 - LPD Write Response (For LPD only) | Register value to be written | - |

*Table 28:* **XPm_DevIoctl Operations** *(cont'd)*

| ID | Name | Description | Arguments | | | |
|---|---|---|---|---|---|---|
| | | | Node ID | Arg1 | Arg2 | Return Value |
| 21 | IOCTL_OSPI_MUX_SELECT | Select OSPI AXI Multiplexer | NODE_OSPI | Operation mode<br>0: Select DMA<br>1: Select Linear<br>2: Get mode | - | Get mode<br>0: DMA<br>1: Linear |
| 22 | IOCTL_USB_SET_STATE | Set USB controller in different device power states | NODE_USB_0 | Requested power state<br>0: D0<br>1: D1<br>2: D2<br>3: D3 | - | - |
| 23 | IOCTL_GET_LAST_RESET_REASON | Get last reset reason of system | - | - | - | 0 – The POR button was pressed outside of the system<br>1 – An internal POR was caused by software<br>2 - One of the other SSIT slices caused a POR<br>3 - An error caused a POR<br>7 - JTAG TAP initiated system reset<br>8 - Error initiated system reset<br>9 - Software initiated system reset<br>10 - One of the other SSIT slices caused a system reset<br>15 – Invalid reset reason |
| 24 | IOCTL_AIE_ISR_CLEAR | Clear AIE NPI Interrupts | DEV_AIE (0x18224072U) | 4-bit NPI Interrupt Clear Mask (wtc)<br>Bit<3-0> correspond to Interrupt<3-0> | - | - |

## *XPm_GetOpCharacteristic EEMI API*

The `XPm_GetOpCharacteristic` EEMI API allows a PM master to request PMC to return information about an operating characteristic of a component. Currently, the following device characteristics can be requested:

- Temperature of the SoC
- Wake-up latency of Cores, Power Islands and rails, and PLL locking time

This API also allows registering notification by an application through the `XPm_RegisterNotifier` API. Using this API, a PU can request the PMC to call its notify callback whenever a qualifying event occurs. you can request to be notified for a specific or any event related to a specific node. The following two qualifying events would cause the invocation of the callback function:

- State change of the node for which callback is registered
- Zero users of the node for which callback is registered

In addition, if you are no longer interested in receiving notifications about events related to the node that the callback was previously requested, you can unregister it.

***Note*:** Currently, only event and device nodes are supported.

For more information, refer to *OS and Libraries Document Collection* (UG643).

## *XPm_Query EEMI API*

The `XPm_Query` EEMI API allows a platform management master to request specific configuration information from the PLM.

The following table lists the supported operations in Versal™ ACAP.

*Table 29:* **XPm_Query Operations**

| ID | Name | Description | Arguments | | |
|---|---|---|---|---|---|
| | | | Arg1 | Arg2 | Return Value |
| 1 | XPM_QID_CLOCK_GET_NAME | Get the string name associated with a clock id | Clock id | - | Data[0]: Success/ Failure<br>Data[1-4]: String name |

*Table 29:* **XPm_Query Operations** *(cont'd)*

| ID | Name | Description | Arguments | | |
|---|---|---|---|---|---|
| | | | Arg1 | Arg2 | Return Value |
| 2 | XPM_QID_CLOCK_GET_TOPOLOGY | Get a clock's topology | Clock id | Topology node index | Data[0]: Success/Failure<br>Data[1-3]: 3 nodes of topology start from node index (passed in Arg 2) |
| 3 | XPM_QID_CLOCK_GET_FIXEDFACTOR_PARAMS | Get Fixed Factor value | Clock id | - | Data[0]: Success/Failure<br>Data[1]: Fixed factor value |
| 4 | XPM_QID_CLOCK_GET_MUXSOURCES | Get clock's multiplexer sources | Clock id | Parent node index | Data[0]: Success/Failure<br>Data[1-3]: Parents id of a clock, starts from parent index (passed in Arg 2) |
| 5 | XPM_QID_CLOCK_GET_ATTRIBUTES | Get clock's attributes | Clock id | - | Data[0]: Success/Failure<br>Data[1] Bit(0): Valid/Invalid clock<br>Data[1] Bit(1): Initial enable requirement<br>Data[1] Bit(2): Clock type (output/external)<br>Data[1] Bit(14:19): Clock node type<br>Data[1] Bit(20-25): Clock node subclass<br>Data[1] Bit(26:31): Clock node class |

*Table 29:* **XPm_Query Operations** *(cont'd)*

| ID | Name | Description | Arguments | | |
|---|---|---|---|---|---|
| | | | Arg1 | Arg2 | Return Value |
| 6 | XPM_QID_PINCTRL_GET_NUM_PINS | Get the number of pins available for configuration | - | - | Data[0]: Success/ Failure<br>Data[1]: Number of pins |
| 7 | XPM_QID_PINCTRL_GET_NUM_FUNCTIONS | Get the total number of functional units available | - | - | Data[0]: Success/ Failure<br>Data[1]: Number of functions |
| 8 | XPM_QID_PINCTRL_GET_NUM_FUNCTION_GROUPS | Get the number of groups that a function id belongs | Function id | - | Data[0]: Success/ Failure<br>Data[1]: Number of groups |
| 9 | XPM_QID_PINCTRL_GET_FUNCTION_NAME | Get the string name associated with a functional unit | Function id | - | Data[0]: Success/ Failure<br>Data[1-3]: String name |
| 10 | XPM_QID_PINCTRL_GET_FUNCTION_GROUPS | Get group ids that a function id belongs | Function id | Index | Data[0]: Success/ Failure<br>Data[1-3]: 6 groups start from index (Arg2) , each group is of 16 bits |
| 11 | XPM_QID_PINCTRL_GET_PIN_GROUPS | Get group ids that a pin id could belong | Pin id | Index | Data[0]: Success/ Failure<br>Data[1-3]: 6 groups start from index (Arg 2), each group is of 16-bits |

*Table 29:* **XPm_Query Operations** *(cont'd)*

| ID | Name | Description | Arguments | | |
|---|---|---|---|---|---|
| | | | Arg1 | Arg2 | Return Value |
| 12 | XPM_QID_CLOCK_GET_NUM_CLOCKS | Get the number of clocks | - | - | Data[0]: Success/ Failure<br>Data[1]: Number of clocks |
| 13 | XPM_QID_CLOCK_GET_MAX_DIVISOR | Get the maximum divisor value of a clock | Clock id | - | Data[0]: Success/ Failure<br>Data[1]: Maximum divisor value |
| 14 | XPM_QID_PLD_GET_PARENT | Get the parent of the PL Device Node | PLDevice Id | - | Data[0]: Success/ Failure<br>Data[1]: PlDevice Parent |

### *XPm_ActivateSubsystem API*

The XPm_ActivateSubsystem API allows the XSDB master to request PMC to activate a subsystem. See Activation of Subsystem for more details.

This API activates a subsystem by requesting all its pre-allocation devices that are essential for it to be operational. This API accepts a target subsystem ID that needs activation, as an input argument. The format for activating a subsystem is as follows:

```
XPm_ActivateSubsystem(u32 Subsystem ID)
```

This command is only allowed from the XSDB master. Currently, only the default subsystem is supported and is automatically activated before downloading any application binaries on a specific master.

## XilPM Client Implementation Details

The system layer of the platform management is implemented on the Versal ACAP using the IPI. To issue an EEMI API call, a PU writes the API data (API ID and arguments) into the IPI request buffer and then triggers the IPI to the PMC.

After the PMC processes the request, it sends the acknowledgment depending on the particular EEMI API and provided arguments.

### *Payload Mapping for API Calls to PMC*

The following data uniquely identifies each EEMI API call:

- EEMI API identifier (ID)
- EEMI API arguments

See *OS and Libraries Document Collection* (UG643) for a list of all API identifiers as well as API argument values.

Before initiating an IPI to the PMC, the PU writes 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 three values.

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

### *Payload Mapping for API Callbacks from the PMC*

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

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

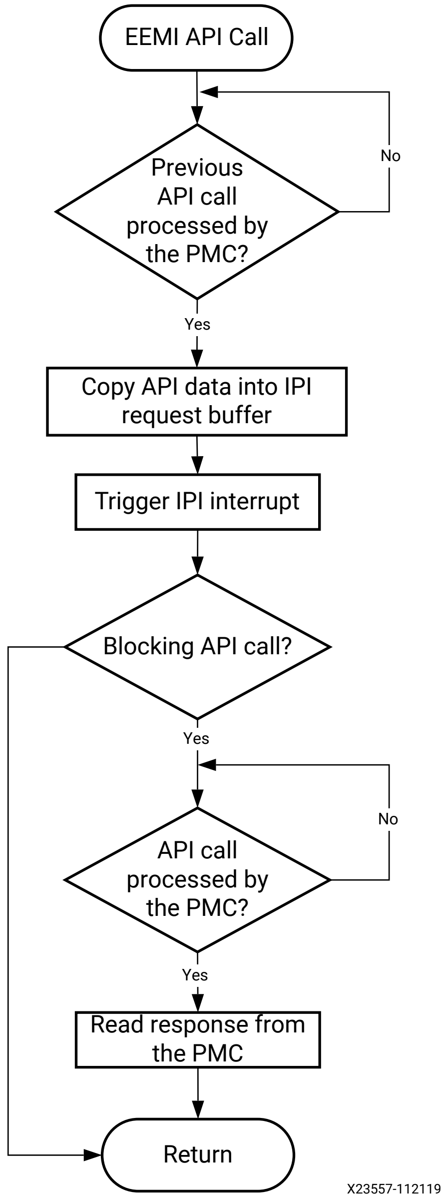
See *OS and Libraries Document Collection* (UG643) for a list of all API identifiers as well as API argument values.

### *Issuing EEMI API Calls to the PMC*

Before issuing an API call to the PMC, a PU must wait until its previous API call is processed by the PMC. Implement a check for completion of a PMC action by reading the corresponding IPI observation register.

Issue an API call by populating the IPI payload buffer with API data and triggering an IPI interrupt to the PMC. For a blocking API call, the PMC responds 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 PMC if a PM operation was unsuccessful. The PU must wait until the PMC has finished processing the API call prior to reading the response buffer, to ensure that the data in the response buffer is valid.

*Figure 37:* **Example Flow of Issuing API Call to the PMC**




## *Handling API Callbacks from the PMC*

The PMC invokes callback functions to the PU by populating the IPI buffers with the API callback data and triggering an IPI interrupt to the PU. 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.

The XilPM library contains an implementation of this behavior.

## PM Features in Linux

Linux executes on at EL1, 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 use the EEMI services provided by the PMC.

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 Platform Management Interface*

#### System Power States

You can request to change the power state of a system or the entire system. Platform management facilitates the switching of the system or subsystem to the new power state.

##### Suspend

When the CPU and most of the peripherals are powered down, the kernel is suspended. The system run states needed to resume from suspend are stored in the DRAM, which is put into self-refresh mode.

- **Required Kernel Configurations:**
  - Power management options
    - [*] Suspend to RAM and standby
    - [*] User space wakeup sources interface interface
    - [*] Device power management core functionality
  - Device Drivers
    - SoC (System on Chip)-specific drivers
  - Firmware Drivers
    - Zynq MPSoC Firmware Drivers
      - -*- Enable Xilinx Zynq MPSoC firmware interface

***Note*:** Any device can prevent the kernel from suspending.

- **Example Code:** To suspend the kernel:

```
$ echo mem > /sys/power/state
```

See also https://wiki.archlinux.org/index.php/Power_management/Suspend_and_hibernate.

#### 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.
  - **Required Kernel Configurations:** Same as Suspend
  - **Example Code:** To set RTC to wake up the APU after 10 seconds:

```
$ echo enabled > /sys/devices/platform/amba/ff000000.serial/tty/ttyAMA0/
power/wakeup
```

- **RTC:** If enabled as a wake-up source, the kernel will resume from the suspend mode when the RTC timer expires. Note that the RTC wake-up source is enabled by default.
  - **Required Kernel Configurations:** Same as Suspend
  - **Example Code:** To wake up the APU on UART input:

```
$ echo enabled > /sys/devices/platform/amba/ff000000.serial/tty/ttyAMA0/
power/wakeup
```

- **GPIO:** If enabled as a wake-up source, a GPIO event will trigger the kernel to resume from the suspend mode.
  - **Required Kernel Configurations:**
    - 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
  - **Example Code:** To wake up the APU on the GPIO pin:

```
$ echo enabled > /sys/devices/platform/gpio-keys/power/wakeup
```

## Power Management for the CPU

### CPU Hot-Plugging

You may take one or more APU cores on-line and off-line as needed using the CPU Hot-plug control interface. The required kernel configurations are as follows:

- Kernel Features
    - [*] Support for hot-pluggable CPUs

For example, to take CPU1 off-line, run the following command:

```
echo 0 > /sys/devices/system/cpu/cpu1/online
```

For more information, refer to the following resources:

- https://www.kernel.org/doc/Documentation/cpu-hotplug.txt
- http://lxr.free-electrons.com/source/Documentation/devicetree/bindings/arm/idle-states.txt

### CPU Idle

If enabled, the kernel might cut power to individual APU cores when they are idling. The required kernel configurations are as follows:

- CPU Power Management
- CPU Idle
- [*] CPU idle PM support
- [*] Ladder governor (for periodic timer tick)
- -*- Menu governor (for tickless system)
- Arm CPU Idle Drivers
    - [*] Generic ARM/ARM64 CPU idle Driver
    - [*] PSCI CPU idle Driver
- **Example 1:** The following code example is the sysfs interface for cpuidle.

```
$ ls -lR /sys/devices/system/cpu/cpu0/cpuidle/
/sys/devices/system/cpu/cpu0/cpuidle/:
drwxr-xr-x 2 root root 0 Jun 10 21:55 state0
drwxr-xr-x 2 root root 0 Jun 10 21:55 state1
/sys/devices/system/cpu/cpu0/cpuidle/state0:
-r--r--r-- 1 root root 4096 Jun 10 21:55 desc
-rw-r--r-- 1 root root 4096 Jun 10 21:55 disable
-r--r--r-- 1 root root 4096 Jun 10 21:55 latency
-r--r--r-- 1 root root 4096 Jun 10 21:55 name
-r--r--r-- 1 root root 4096 Jun 10 21:55 power
-r--r--r-- 1 root root 4096 Jun 10 21:55 residency
-r--r--r-- 1 root root 4096 Jun 10 21:55 time
```

```
-r--r--r-- 1 root root 4096 Jun 10 21:55 usage
/sys/devices/system/cpu/cpu0/cpuidle/state1:
-r--r--r-- 1 root root 4096 Jun 10 21:55 desc
-rw-r--r-- 1 root root 4096 Jun 10 21:55 disable
-r--r--r-- 1 root root 4096 Jun 10 21:55 latency
-r--r--r-- 1 root root 4096 Jun 10 21:55 name
-r--r--r-- 1 root root 4096 Jun 10 21:55 power
-r--r--r-- 1 root root 4096 Jun 10 21:55 residency
-r--r--r-- 1 root root 4096 Jun 10 21:55 time
-r--r--r-- 1 root root 4096 Jun 10 21:55 usage
```

where:

- **desc:** Small description about the idle state (string)
- **disable:** Option to disable this idle state (bool)
- **latency:** Latency to exit out of this idle state (in microseconds)
- **name:** Name of the idle state (string)
- **power:** Power consumed while in this idle state (in milliwatts)
- **time:** Total time spent in this idle state (in microseconds)
- **usage:** Number of times this state was entered (count)

- **Example 2:** The following code example 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 */
```

For more information, refer to the following resources:

- 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

#### CPU Frequencies

Enable this feature to permit the CPU cores to switch between different operation clock frequencies.

- **Required Kernel Configurations:**
  - 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

- **Example Commands:** The following are some commands related to CPU frequency scaling:
  - Look up the available CPU speeds:

    ```
    cat /sys/devices/system/cpu/cpu*/cpufreq/scaling_cpu_freq
    ```

  - Select the 'userspace' governor for CPU frequency control:

    ```
    echo userspace > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
    ```

  - Look up the current CPU speed (same for all cores):

    ```
    cat /sys/devices/system/cpu/cpu*/cpufreq/scaling_cpu_freq
    ```

  - Change the CPU speed (same for all cores):

    ```
    echo <freq> > /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed
    ```

  For details on adding and changing CPU frequencies, see the Linux kernel documentation on Generic Operating Points.

## Power Management for the Devices

### Clock Gating

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

The required kernel configurations are as follows:

- Common Clock Framework
  - [*] Support for Xilinx Zynq MP UltraScale+ clock controllers

### Runtime PM

Powers off devices when they are not being used.

***Note*:** Individual drivers might or might not support run-time power management.

The required kernel configurations are as follows:

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

#### Global General Storage Registers

Four 32-bit storage registers are available for general use. Their values are not preserved after software reboots. The following table lists the global general storage registers.

*Table 30:* **Global General Storage Registers**

| Device Node | Valid Value Range |
|---|---|
| /sys/firmware/zynqmp/ggs0 | 0x00000000 - 0xFFFFFFFF |
| /sys/firmware/zynqmp/ggs1 | 0x00000000 - 0xFFFFFFFF |
| /sys/firmware/zynqmp/ggs2 | 0x00000000 - 0xFFFFFFFF |
| /sys/firmware/zynqmp/ggs3 | 0x00000000 - 0xFFFFFFFF |

To read the value of a global storage register:

```
cat /sys/firmware/zynqmp/ggs0
```

To write the mask and value of a global storage register:

```
echo 0xFFFFFFFF 0x1234ABCD > /sys/firmware/zynqmp/ggs0
```

#### Persistent Global General Storage Registers

Four 32-bit persistent global storage registers are available for general use. Their values are preserved after software reboots. The following table lists the persistent global general storage registers.

*Table 31:* **Persistent General Global Storage Registers**

| Device Node | MMIO Register | Valid Value Range |
|---|---|---|
| /sys/firmware/zynqmp/pggs0 | PMC_GLOBAL_PERS_GLOB_GEN_STORAGE3 | `0x00000000` to `0xFFFFFFFF` |
| /sys/firmware/zynqmp/pggs1 | PMC_GLOBAL_PERS_GLOB_GEN_STORAGE4 | `0x00000000` to `0xFFFFFFFF` |
| /sys/firmware/zynqmp/pggs2 | PSM_GLOBAL_PERS_GLOB_GEN_STORAGE0 | `0x00000000` to `0xFFFFFFFF` |
| /sys/firmware/zynqmp/pggs3 | PSM_GLOBAL_PERS_GLOB_GEN_STORAGE1 | `0x00000000` to `0xFFFFFFFF` |

To read the value of a persistent global storage register:

```
cat /sys/firmware/zynqmp/pggs0
```

To write the mask and value of a persistent global storage register:

```
echo 0xFFFFFFFF 0x1234ABCD > /sys/firmware/zynqmp/pggs0
```

The following registers are reserved.

**IMPORTANT!** *Xilinx recommends that you do not use reserved registers.*

*Table 32:* **Reserved Storage Registers**

| Register | Description |
|---|---|
| PMC_GLOBAL_GLOBAL_GEN_STORAGE0, PMC_GLOBAL_GLOBAL_GEN_STORAGE1 | Contains the ROM execution time stamp. When PLM is active, it reads these two registers to obtain the execution time of ROM. Registers can be used after loading boot PDI. |
| PMC_GLOBAL_GLOBAL_GEN_STORAGE2 | Contains device security status, updated by ROM. PLM uses this register to determine if KAT needs to be performed. |
| PMC_GLOBAL_GLOBAL_GEN_STORAGE4 | Used by PLM to store ATF handoff parameter address pointer. |
| PMC_GLOBAL_PERS_GLOB_GEN_STORAGE0 | Reserved for XilPM to save the status of each power domain initialization. |
| PMC_GLOBAL_PERS_GLOB_GEN_STORAGE1 | Not yet used in PLM but intended for data sharing between PLM and debugger. |

## Debug Interface

The PM platform driver exports a standard debugfs interface to access all EEMI services. The interface is only intended for testing and does not contain any checking regarding improper usage, and the number, type and valid ranges of the arguments. Be aware that invoking EEMI services directly through this interface can very easily interfere with the kernel power management operations, resulting in unexpected behavior or system crash. Zynq MP debugfs interface is enabled by default in defconfig.

- **Required Kernel Configurations:**
  - Kernel hacking
    - Compile-time checks and compiler options
      - [*] Debug Filesystem
  - Firmware Drivers
    - Zynq MPSoC Firmware Drivers
      - [*] Enable Xilinx Zynq MPSoC firmware interface
      - [*] Enable Xilinx Zynq MPSoC firmware debug APIs

You can invoke any EEMI API except for:

- Self Suspend
- Request Wake-up the APU
- System Shutdown
- Force Power Down the APU

## Command Line Input

You can invoke an EEMI service by writing the EEMI API ID, followed by up to four arguments, to the debugfs interface node.

### API ID

The Function ID can be an EEMI API function name or ID number, type `string` or type `integer`, respectively.

### Arguments

The number and type of the arguments depend on the selected API function. You must provide all arguments 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 *Embedded Energy Management Interface EEMI API Reference Guide* (UG1200).

## Command List

### Get API Version

Get the API version.

```
echo pm_get_api_version > /sys/kernel/debug/zynqmp-firmware/pm
```

### Force Power Down

Force another PU to power down.

```
echo pm_force_powerdown <node> > /sys/kernel/debug/zynqmp-firmware/pm
```

### Request Node

Request to use a node.

```
echo pm_request_node <node> > /sys/kernel/debug/zynqmp-firmware/pm
```

#### Release Node

Free a node that is no longer being used.

```
echo pm_release_node <node> > /sys/kernel/debug/zynqmp-firmware/pm
```

#### Set Requirement

Set the power requirement on the node.

```
echo pm_set_requirement <node> <capabilities> > /sys/kernel/debug/zynqmp-
firmware/pm
```

#### Set Max Latency

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

```
echo pm_set_max_latency <node> <latency> > /sys/kernel/debug/zynqmp-
firmware/pm
```

#### Get Node Status

Get status information of a node

```
echo pm_release_node <node> > /sys/kernel/debug/zynqmp-firmware/pm
```

***Note***: Any PU can check the status of any node, regardless of the node assignment.

#### Get Operating Characteristic

Get operating characteristic information of a node.

```
echo pm_get_operating_characteristic <node> > /sys/kernel/debug/zynqmp-
firmware/pm
```

#### Reset Assert

Assert or deassert on specific reset lines.

```
echo pm_reset_assert <reset> <action> > /sys/kernel/debug/zynqmp-firmware/pm
```

#### Reset Get Status

Get the status of the reset line.

```
echo pm_reset_get_status <reset> > /sys/kernel/debug/zynqmp-firmware/pm
```

#### Get Chip ID

Get the chip ID.

```
echo pm_get_chipid > /sys/kernel/debug/zynqmp-firmware/pm
```

#### Get Pin Control Functions

Get current selected function for given pin.

```
echo pm_pinctrl_get_function <pin-number> > /sys/kernel/debug/zynqmp-
firmware/pm
```

#### Set Pin Control Functions

Set requested function for given pin.

```
echo pm_pinctrl_set_function <pin-number> <function-id> >
/sys/kernel/debug/zynqmp-firmware/pm
```

#### Get Configuration Parameters for the Pin

Get the value of the requested configuration parameter for given pin.

```
echo pm_pinctrl_config_param_get <pin-number> <parameter to get> >
/sys/kernel/debug/zynqmp-firmware/pm
```

#### Set Configuration Parameters for the Pin

Set value of requested configuration parameter for given pin.

```
echo pm_pinctrl_config_param_set <pin-number> <parameter to set> <param
value> >
/sys/kernel/debug/zynqmp-firmware/pm
```

#### Control Device and Configurations

Control device and configurations and get configurations values.

```
echo pm_ioctl <node id> <ioctl id> <arg1> <arg2> >
/sys/kernel/debug/zynqmp-firmware/pm
```

#### Query Data

Request data from firmware.

```
echo pm_query_data <query id> <arg1> <arg2> <arg3> >
/sys/kernel/debug/zynqmp-firmware/pm
```

#### Enable Clock

Enable the clock for a given clock node ID.

```
echo pm_clock_enable <clock id> > /sys/kernel/debug/zynqmp-firmware/pm
```

#### Disable Clock

Disable the clock for a given clock node ID.

```
echo pm_clock_disable <clock id> > /sys/kernel/debug/zynqmp-firmware/pm
```

#### Get Clock State

Get the state of clock for a given clock node ID.

```
echo pm_clock_getstate <clock id> > /sys/kernel/debug/zynqmp-firmware/pm
```

#### Set Clock Divider

Set the divider value of clock for a given clock node ID.

```
echo pm_clock_setdivider <clock id> <divider value> >
/sys/kernel/debug/zynqmp-firmware/pm
```

#### Get Clock Divider

Get the divider value of clock for a given clock node ID.

```
echo pm_clock_setdivider <clock id> <divider value> >
/sys/kernel/debug/zynqmp-firmware/pm
```

#### Set Clock Rate

Set the clock rate for a given clock node ID.

```
echo pm_clock_setrate <clock id> <clock rate> >
/sys/kernel/debug/zynqmp-firmware/pm
```

#### Get Clock Rate

Set the clock rate for a given clock node ID.

```
echo pm_clock_setrate <clock id> <clock rate> >
/sys/kernel/debug/zynqmp-firmware/pm
```

#### Set Clock Parent

Set the parent clock for a given clock node ID.

```
echo pm_clock_setparent <clock id> <parent clock id> >
/sys/kernel/debug/zynqmp-firmware/pm
```

#### Get Clock Parent

Get the parent clock for a given clock node id.

```
echo pm_clock_getparent <clock id> > /sys/kernel/debug/zynqmp-firmware/pm
```

***Note***: Clock ID definitions are available in the following TXT file of the clock bindings documentation:`Documentation/devicetree/bindings/clock/xlnx,versal-clk.txt`

#### Self Suspend

Notify the PMC that this PU is about to suspend itself.

```
$ echo pm_self_suspend <node> > /sys/kernel/debug/zynqmp-firmware/pm
```

#### Request Wake-Up

Request another PU to wake up from suspend state.

```
$ echo pm_request_wakeup <node> <set_address> <address> >
/sys/kernel/debug/zynqmp-firmware/pm
```

#### Set Wake-Up Source

Set up a node as the wake-up source.

```
$ echo pm_set_wakeup_source <target> <wkup_node> <enable> >
/sys/kernel/debug/zynqmp-firmware/pm
```

## *PM Linux Drivers*

The Versal ACAP power management for Linux is encapsulated in a power management driver, power domain driver, and platform firmware driver. The system-level API functions are exported and can be called by other Linux modules with GPL compatible license.

### PM Firmware Driver

The function declarations are available in the following location: `include/linux/firmware/xlnx-zynqmp.h`

The function implementations are available in the following location: `drivers/firmware/xilinx/zynqmp/zynqmp*.c`

For proper driver intialization, you must provide the correct node in the Linux device tree . The firmware driver relies on the firmware node to detect the presence of PLM, 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 the following properties:

- **Compatible:** Must contain `xlnx,versal-firmware`
- **Method:** The method of calling the PM framework firmware should be smc.

  ***Note***: Additional information is available in the following txt file of Linux Documentation: `Documentation/devicetree/bindings/firmware/xilinx/xlnx,zynqmp-firmware.txt`

Example:

```
firmware {
                versal_firmware: versal-firmware {
                compatible = "xlnx,versal-firmware";
                method = "smc";
                };
                };
```

***Note***: Power management and power domain driver binding details are available in the following files of Linux documentation:

- `Documentation/devicetree/bindings/power/reset/xilinx/xlnx,zynqmp-power.txt`
- `Documentation/devicetree/bindings/power/reset/xilinx/xlnx,zynqmp-genpd.txt`

### Request and Release a Device From Linux

The Linux ZynqMP (see https://github.com/torvalds/linux/blob/master/drivers/soc/xilinx/zynqmp_pm_domains.c) manages device usage request, and sets requirement and release of a device.

You must provide the Device ID as the power domain ID in the respective device node. The `zynqmp-firmware` node works as the power domain provider. For example:

```
firmware {
zynqmp_firmware: zynqmp-firmware {
...
#power-domain-cells = <1>;
...
};
};
usb0 {
...
power-domains = <&zynqmp_firmware 0x18224018U>;
...
};
```

The Linux ZynqMP power domain driver sets PM_CAP_ACCESS when the device is in use. During runtime suspend, the power domain driver sets the requirement to PM_CAP_UNUSABLE (where only the clock is disabled). If the device is not in use, the power domain driver releases it. In this case, if there is no other subsystem using the device, the platform management firmware power gates, disables the clock, and resets the device.

## Arm Trusted Firmware

The Arm trusted firmware (ATF) executes at 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 PMC.

## *ATF Application Binary Interface*

All APU executable layers below EL3 might indirectly communicate with the PMC through ATF. The ATF receives all calls made from the lower ELs, consolidates all requests, and sends the requests to the PMC.

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.

The EEMI API implementation for the APU is compliant only with the SMC64 calling convention. EEMI API calls made from the hypervisor, secure OS, or OS and 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.

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, you must check that the EEMI API version implemented in the ATF matches the version implemented in the PMC firmware. EEMI API version is a 32-bit value separated in higher 16-bits of MAJOR and lower 16-bits of MINOR part. Both fields must be the same between the ATF and the PMC firmware.

#### How to Check the EEMI API Version

The EEMI version implemented in the ATF is defined in the local EEMI_API_VERSION flag. The rich OS can invoke the PM_GET_API_VERSION function to retrieve the EEMI API version from the PMC. If the versions are different, this function reports an error.

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

### Checking the Chip ID

Linux or another rich OS can invoke the PM_GET_CHIPID function using SMC to retrieve the chip ID information from the PMC. The return values are as follows:

- TAP idcode register
- TAP version register

For more details, see the *Versal ACAP Technical Reference Manual* (AM011).

# Power State Coordination Interface

The Power State Coordination Interface (PSCI) 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 only supports PSCI v0.2 (with no backward compatible support for v0.1).

The Linux kernel comes with standard support for PSCI. For information about the binding between the kernel and the ATF/PSCI, see https://www.kernel.org/doc/Documentation/devicetree/bindings/arm/psci.txt.

The following table lists the PSCI v0.2 functions that ATF supports.

*Table 33:* **PSCI v0.2 Functions Supported by ATF**

| Function | Description | Supported? |
|---|---|---|
| PSCI Version | Return the implemented PSCI version. | 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 wake-up event. | Yes |
| CPU On | Power up a core. This call is used to power up cores that either:<br>• Have not yet been booted into the calling supervisory software.<br>• 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 a hotplug. A core that is powered down by CPU_OFF can only be powered up again in response to a CPU_ON. | Yes |
| Affinity Info | Enable the caller to request status of an affinity instance. | Yes |
| Migrate (Optional) | This is used to ask a uniprocessor trusted OS to migrate its context to a specific core. | Yes |
| Migrate Info Type (Optional) | This function allows a caller to identify the level of multicore support present in the trusted OS. | Yes |
| Migrate Info Up CPU (Optional) | For a uniprocessor Trusted OS, this function returns the current resident core. | Yes |
| System Off | Shut down the system. | Yes |
| System Reset | Reset the system. | Yes |

*Table 33:* **PSCI v0.2 Functions Supported by ATF** *(cont'd)*

| Function | Description | Supported? |
|---|---|---|
| PSCI Features | Introduced in PSCI v1.0.<br>Query API that allows discovering whether a specific PSCI function is implemented and its features. | Yes |
| CPU Freeze (Optional) | Introduced in PSCI v1.0.<br>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.<br>Will place a core into an IMPLEMENTATION DEFINED low-power state. Unlike CPU_SUSPEND, the caller does not need to specify a power state parameter. | No |
| Node HW State (Optional) | Introduced in PSCI v1.0.<br>This function is intended to return the true hardware state of a node in the power domain topology of the system. | Yes |
| System Suspend (Optional) | Introduced in PSCI v1.0.<br>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.<br>This function allows setting the mode used by CPU_SUSPEND to coordinate power states. | No |
| PSCI Stat Residency (Optional) | Introduced in PSCI v1.0.<br>Returns the amount of time the platform has spent in the given power state because cold boot. | Yes |
| PSCI Stat Count (Optional) | Introduced in PSCI v1.0.<br>Return the number of times the platform has used the given power state because of cold boot. | Yes |

# PS Management Controller Firmware

The PS management controller (PSM) is a separate, MicroBlaze processor, with a triple modular redundancy MicroBlaze implementation in the PS domain. The PSM firmware, that runs on the PSM, is in charge of managing power PS power islands and domains. The PMC leverages functions implemented in the PSM firmware to power up or down the PS power islands and domains. These functions are exposed only to the PLM.

When an EEMI request results in changing the state of a power island or a power domain, the PLM triggers an interrupt handler in the PSM to handle that operation. There is also an IPI channel assigned to the PSM to communicate with the PLM on completion of the operation.

## Relationship with PLM Firmware

The EEMI service handlers are implemented in the PLM, as one of the modules called a PM Controller (there are other modules running in the PLM to handle other types of services). For more details, see the Chapter 7: Boot and Configuration.

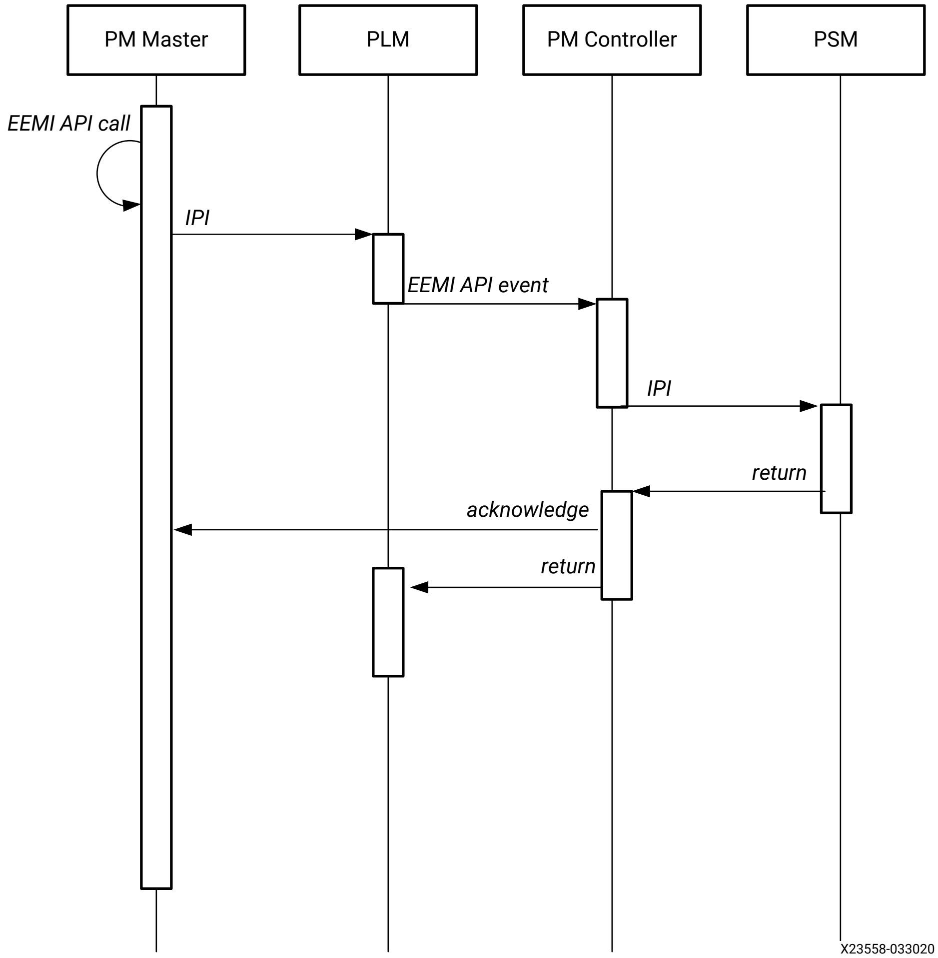
#### Embedded Energy Management Interface (EEMI)

EEMI API events are software-generated events. The events are triggered using IPI interrupts when a PM master initiates an EEMI API call to the PMC. The PM Controller handles the EEMI request. An EEMI request often triggers a change in the power state, or change in configuration of a resource.

### *General Flow of an EEMI API Call*

The following figure shows the typical API sequence call, starting with the call initiated by a PM master (such as another processing unit).

*Figure 38:* **EEMI API Call Sequence**




The figure shows four actors; the first actor represents the PM master, which is either the RPU, APU, or a MicroBlaze processor core. The remaining three actors are the PLM firmware, PMC, and PSM.

First, the PLM firmware receives the IPI interrupt. After the interrupt has been identified as a power management-related interrupt, the IPI arguments are passed to the XilPM server. The XilPM server then processes the API call. If necessary, the XilPM server can call the PSM firmware to perform some power management operations, such as powering on or powering off a power island, or a power domain.

*Chapter 11*

# Target Development Platforms

This chapter describes the various development platforms available for Versal ACAP, such as boards and kits.

The following platforms are available:

- **AI Core Series:** High-compute series with medium density programmable logic and connectivity capability coupled with AI and DSP acceleration engines.
- **Prime Series:** Mid-range series with medium density PL, signal processing, and connectivity capability.
- **Premium Series:** High-end, high-bandwidth series, rich in networking interfaces, security engines, and providing high compute density.

## Boards and Kits

Xilinx provides the Versal ACAP device evaluation kit for developers.

Currently, there are two evaluation kits available for Versal ACAP devices:

- **VCK190:** AI Core series evaluation kit. For more details on the VCK190 board, refer to the *VCK190 Evaluation Board User Guide* (UG1366).
- **VMK180:** Prime series evaluation kit. For more details on the VMK180 board, refer to *VMK180 Evaluation Board User Guide* (UG1411).

*Appendix A*

# Libraries

See *OS and Libraries Document Collection* (UG643) for information on API reference for the following Versal ACAP libraries.

- XilFFS
- XilMailbox
- XilPM
- XilSEM

*Appendix B*

# Additional Resources and Legal Notices

## Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

## Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

***Note*:** For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

## References

These documents provide supplemental material useful with this guide:

### Xilinx References

1. PetaLinux Tools
2. Xilinx Vivado Design Suite – HLx Editions
3. Xilinx Third-Party Tools
4. Embedded Software & Ecosystem

### Devices Documentation

1. *Versal ACAP AI Engine Architecture Manual* (AM009)
2. *Versal ACAP Technical Reference Manual* (AM011)
3. *Versal Architecture and Product Data Sheet: Overview* (DS950)
4. *Versal ACAP Programmable Network on Chip and Integrated Memory Controller LogiCORE IP Product Guide* (PG313)
5. *OS and Libraries Document Collection* (UG643)
6. *Versal ACAP AI Engine Programming Environment User Guide* (UG1076)
7. *Embedded Energy Management Interface EEMI API Reference Guide* (UG1200)
8. *Versal ACAP Design Guide* (UG1273)
9. *Bootgen User Guide* (UG1283)
10. *Xilinx Embedded Design Tutorials: Versal Adaptive Compute Acceleration Platform* (UG1305)
11. *VCK190 Evaluation Board User Guide* (UG1366)
12. *Versal ACAP QEMU User Guide* (UG1372)
13. *VMK180 Evaluation Board User Guide* (UG1411)

### Tools and PetaLinux Documentation

1. *Zynq UltraScale+ MPSoC: Software Developers Guide* (UG1137)
2. *PetaLinux Tools Documentation: Reference Guide* (UG1144)
3. *Bootgen User Guide* (UG1283)
4. *Vitis Unified Software Platform Documentation* (UG1416)

### Miscellaneous Links

1. https://github.com/Xilinx/linux-xlnx/
2. https://github.com/torvalds/linux/blob/master/drivers/soc/xilinx/zynqmp_pm_domains.c
3. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842279/Build+Device+Tree+Blob
4. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841996/Linux#Linux-XilinxLinux

5. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842250/PetaLinux
6. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842316/Linux+Prebuilt+Images
7. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841718/OpenAMP
8. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842530/XEN+Hypervisor
9. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841883/Yocto

**Third-Party References**

1. https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile
2. https://www.yoctoproject.org/software-overview/downloads/
3. https://git.yoctoproject.org/cgit/cgit.cgi/meta-xilinx/
4. https://www.kernel.org/doc/Documentation/devicetree/bindings/arm/psci.txt
5. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0022c/index.html
6. https://www.kernel.org/doc/Documentation/cpuidle/core.txt
7. https://www.kernel.org/doc/Documentation/cpuidle/driver.txt
8. https://www.kernel.org/doc/Documentation/cpuidle/governor.txt
9. https://www.kernel.org/doc/Documentation/cpuidle/sysfs.txt
10. https://www.kernel.org/doc/Documentation/cpuidle/sysfs.txt
11. https://www.kernel.org/doc/Documentation/devicetree/bindings/opp/opp.txt
12. http://lxr.free-electrons.com/source/Documentation/devicetree/bindings/arm/idlestates.txt
13. https://www.kernel.org/doc/Documentation/cpu-hotplug.txt
14. https://wiki.archlinux.org/index.php/Power_management/Suspend_and_hibernate

# Please Read: Important Legal Notices

### AUTOMOTIVE APPLICATIONS DISCLAIMER



### Copyright