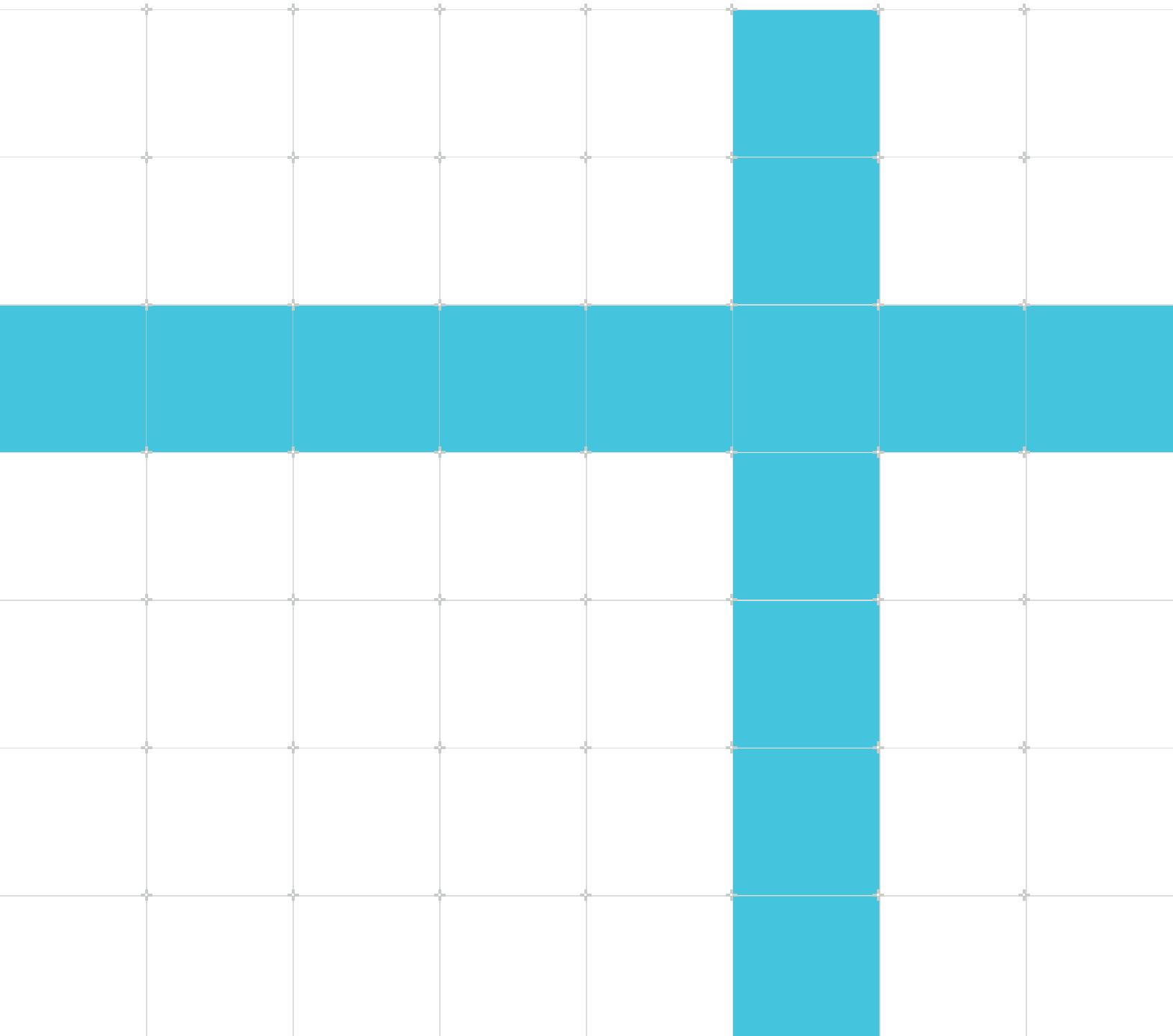
Learn the architecture - Arm System Architectures

Version 1.0

Non-Confidential

Issue 01
110303_0100_01_en



Learn the architecture - Arm System Architectures

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	19 May 2025	Non-Confidential	Initial release

Proprietary Notice

Confidentiality Status

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview of Arm system architectures

The Arm architecture offers a significant level of flexibility. This flexibility means that designers can develop processors specifically optimized for a wide range of applications, ranging from tiny embedded systems to supercomputers. These systems include much more than just a processor core. Additional components include memory and peripherals, as well as modules for power management, performance monitoring, and system debug. Connecting these components together requires hardware, and system software such as an operating system to manage it all.

Reusing components and software across systems minimizes both the elapsed time and the cost of development. Maintaining custom software becomes a significant burden if security updates are required throughout the product lifetime and across many generations. Well-established industry interoperability standards, such as PCIe, address some of these challenges.

To complement these industry standards, Arm provides a collection of open and free-to-use system architectures. Developed in collaboration with Arm's ecosystem partners, these system architectures help engineers design secure and efficient systems as easily as possible.

The Arm system architectures provide just enough standardization of components and interfaces to tackle common problems while enabling innovation. Over-definition restricts flexibility, while insufficient standardization creates unnecessary diversity increasing overall development costs. Arm does not mandate that you must use these system architectures. However, system designers might find that compliance is necessary for particular markets or beneficial when using off-the-shelf components and operating systems.

Arm invests to help partners benefit from developing compliant solutions. This investment includes:

- Contributing changes to upstream open-source projects such as the Linux kernel and trustedfirmware.org
- Developing checklists and automated test tools such as the Architectural Compliance Suites
- Running compliance programs, for example SystemReady

This guide introduces the system architecture specifications. It identifies the relationships between the different documents, and describes common usage scenarios.

The guide starts with the base standards that define the minimum hardware and firmware functionality needed to solve a high-level compatibility problem, such as supporting off-the-shelf operating systems. These standards combine industry and Arm-specific specifications with clarifications and guidance to define a system profile, or recipe, for a market-specific use case.

Later chapters provide a toolkit of lower-level firmware and hardware specifications, describing individual components and interfaces designed for Arm-based systems.

Some specifications are mandated by the base standards and remain relevant even in markets where they are not used. Other specifications are optional, and enable advanced or additional capabilities.

2. Base standards

System designers face a wide range of choices when developing computing systems. To make the process easier in common multi-vendor environments, various hardware and firmware standards ensure a consistent base level of compatibility across implementations. These standards leverage Arm-specific and industry-standard interfaces, defining market-specific profiles for both required and optional system-level capabilities.

2.1 Generic operating systems

Many computing scenarios require a standard off-the-shelf operating system (OS), such as Microsoft Windows or Linux. The OS depends on a critical set of underlying hardware and firmware capabilities.

Traditionally, vendors of Arm-based systems have extensively customized the OS for each individual system, but this approach presents challenges. Creating and maintaining these bespoke OS versions is often not cost-effective or practical, especially considering the growing need for long-term security updates.

The standards described here help solve this problem by defining a consistent set of firmware and hardware capabilities that are required to support common operating systems. The focus is on enabling fundamental system operation, allowing OS-specific driver mechanisms to support more advanced platform capabilities. These standards underpin the bands in the SystemReady program which helps partners achieve and declare compliance.

2.1.1 Firmware capabilities

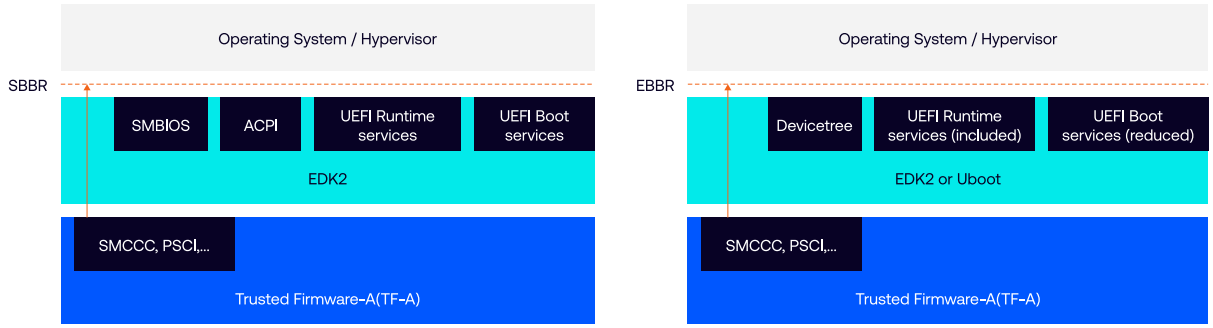
Firmware running on an application processor is essential to enable the system's boot process and overall operation. Base Boot Requirements (BBR) (DEN0044), together with market-specific supplements, defines a standard minimal set of boot and runtime services that firmware must provide. Different recipes meet the needs of various operating systems and hypervisors. The most common recipes are:

- SBBR recipe: for operating systems which require UEFI, ACPI, and SMBIOS when running on BSA-compliant hardware
- EBBR recipe: typically used by operating systems which use UEFI and Devicetree for hardware discovery

In many secure environments, the operating system needs assurances that the boot process has not been tampered with. Base Boot Security Requirements (BBSR) (DEN0107) provides additional standardization and guidance about this situation, including support for UEFI secure boot, UEFI authenticated variables, and secure firmware update.

The following diagram shows a typical open-source firmware stack implementation for the SBBR and EBBR recipes with the main firmware interfaces.

Figure 2-1: Typical SBBR and EBBR open-source firmware stack implementations



For more details about the firmware interfaces which make up BBR, including ACPI and Devicetree, see Platform firmware interfaces.

2.1.2 Hardware capabilities

Base System Architecture (BSA) (DEN0094), together with market-specific supplements, defines a basic set of hardware capabilities as seen by software running on an application processor.

The main BSA specification complements the BBR by describing the fundamental hardware required to run the core features of a standard OS, including boot, basic debugging, and hardware discovery. BSA requirements include processor features, memory subsystem, PCIe integration, interrupts, power states, and peripheral systems.

Compliance with BSA is generally required by operating systems which depend on ACPI-based firmware, for example Microsoft Windows and Linux distributions such as Red Hat Enterprise Linux. BSA can also benefit Linux distributions which use Devicetree firmware by minimizing or eliminating the need to customize Linux. This customization can significantly increase both initial development costs and ongoing maintenance expenses if upstream maintainers reject the changes.

Note

BSA is currently not required for Android-based systems, which have their own set of compatibility requirements.

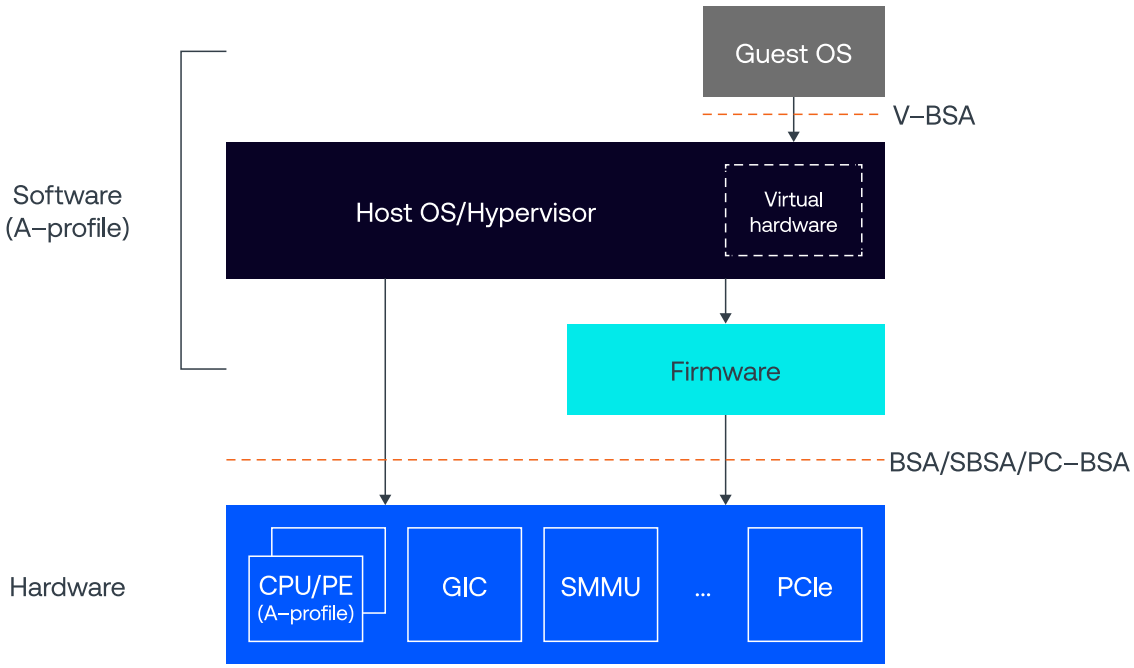
The following market-specific BSA supplements enable consistent implementation of additional hardware capabilities, allowing common software to be used across compliant systems:

- Server Base System Architecture (SBSA) (DEN0029): for servers, such as those used in a data center. SBSA Levels define progressively higher minimum support requirements for features including hypervisors, Reliability Availability and Serviceability (RAS) error reporting, system debug, and telemetry. Each successive SBSA Level improves overall functionality, performance, and security.

- PC Base System Architecture (PC-BSA) (DEN0151): for Personal Computers, such as laptops. Required features include basic hypervisor support, and security such as a Trusted Platform Module (TPM).
- Virtual Base System Architecture (V-BSA) (DEN0150): describes a logical view of the hardware presented by a hypervisor to a guest operating system.

The following diagram shows the hardware and software interfaces addressed by these market-specific BSA supplements:

Figure 2-2: Market-specific BSA supplements



If you produce silicon which does not comply to the relevant standards, costs and time to market can increase. Developing software workarounds is time consuming, and not always feasible. In severe scenarios, a silicon re-spin might be necessary for a viable product. To help silicon designers prevent these issues, the BSA Architecture Compliance Suite (ACS) offers a set of tests which can be conducted early in the design phase before tape-out. For more information, see the SystemReady Pre-Silicon Reference Guide BSA integration and compliance guide.

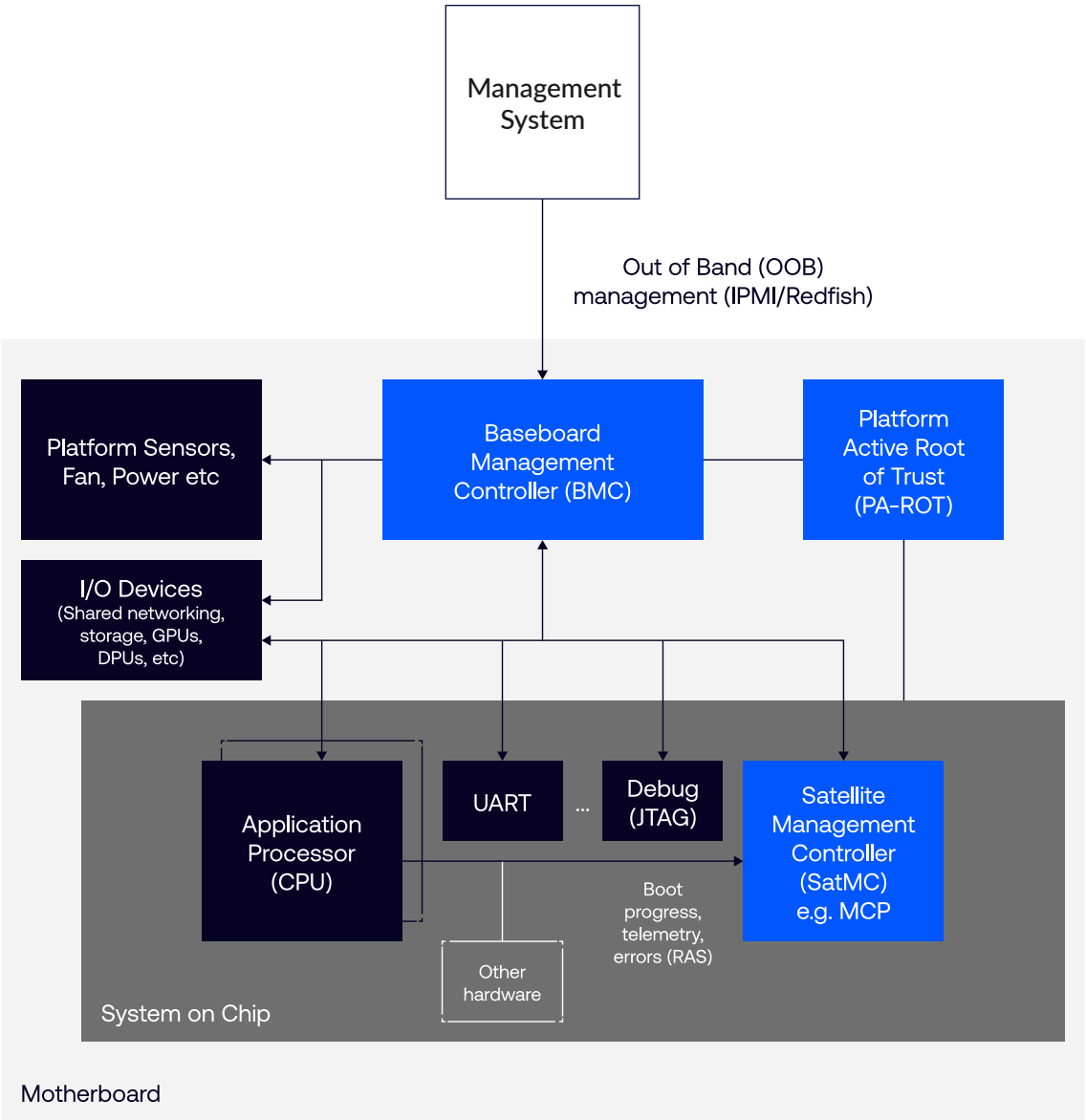
2.2 Server management

Servers designed for multi-server environments, such as data centers, usually have a Baseboard Management Controller (BMC). The BMC enables administrators to remotely oversee and manage the server independently from the host OS. The BMC is usually a specialized processor. Typical

BMC functions include monitoring hardware, updating firmware, configuring system settings, deploying operating systems, and debugging remotely.

To establish a standard set of capabilities for the industry, the Server Base Manageability Requirements (SBMR) (DEN0069) specifies hardware and software interface requirements and provides guidance for SBSA-compliant Arm servers. It explains how to represent Arm-specific details using industry-standard protocols such as IPMI, Redfish, MCTP and PLDM. The following figure shows the main components of the architecture:

Figure 2-3: Typical server management architecture using BMC



An optional Satellite Management Controller (SatMC), usually a dedicated M-profile microcontroller integrated with the application processors, provides visibility into the SoC. The SatMC performs functions including:

- Monitoring boot progress
- Collecting SoC telemetry data such as temperature and power consumption
- Tracking Reliability Availability and Serviceability (RAS) errors

You can use the Manageability Control Processor (MCP) component featured in Arm Compute Subsystems (CSS) such as Neoverse V2 to implement the SatMC function.

The Platform Active Root of Trust (PA-ROT) authenticates any firmware loaded from flash before execution, and actively monitors the firmware store to detect any unauthorized changes. The PA-RoT can be either a discrete system component or integrated in the BMC.

3. Platform firmware interfaces

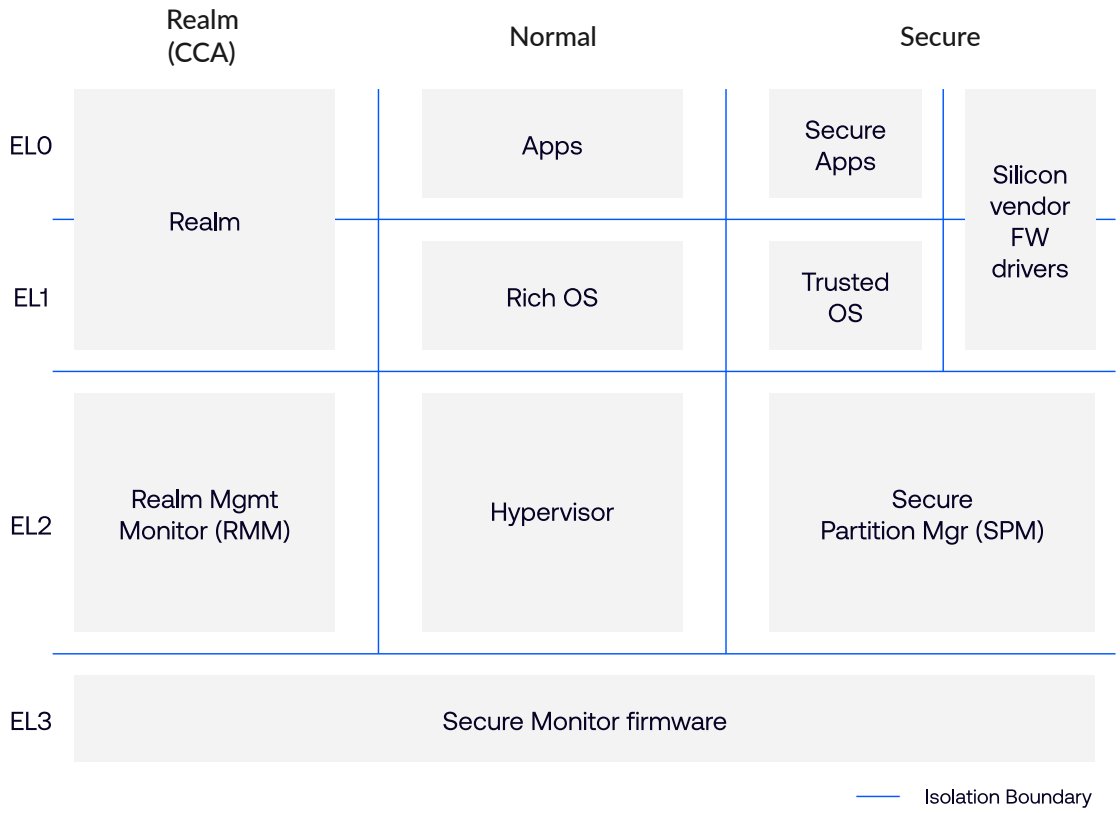
Platform firmware is low-level software that supports higher-level system software such as an OS or hypervisor. This section describes the standard interfaces provided by the Platform firmware, focusing on A-profile. Some of these interfaces are needed to meet the Base Boot Requirements (BBR) described in Base standards. Others extend beyond the BBR with additional capabilities.

Arm A-profile processors use Exception levels and Security states to provide hardware-enforced isolation between different execution modes. For more information, see the following guides:

- Learn the architecture - AArch64 Exception Model
- Learn the architecture - TrustZone for AArch64
- Learn the architecture - AArch64 Virtualization
- Learn the architecture - Realm Management Extension

The following figure shows how software is typically divided between different Exception levels and Security states:

Figure 3-1: Typical division of software between different Exception levels and Security states



The Normal world runs the traditional rich software stack. This typically includes a large application set and a complex operating system such as Linux. This software stack might also include a hypervisor, or additional firmware, or both. These software stacks are large and complex. While efforts can be made to secure them, the size of the attack surface means that they are more vulnerable to attack.

The Secure and Realm worlds are managed by a smaller and simpler software stack. This software stack has a considerably lower attack surface, which helps reduce vulnerability to attack. The main difference between the use of Realms and Secure world is the design intent:

- Secure world, also known as TrustZone, is generally used by platform-specific services owned by parties close to the system development, such as Silicon Providers (SiPs) and Original Equipment Manufacturers (OEMs). First introduced in Armv6K and enhanced subsequently, the Secure world is widely adopted in Arm-based devices across all industries.
- Realms are a more recent hardware extension to Armv9-A and enable the Arm Confidential Compute Architecture (CCA). Realms enable general developers to execute code on a system without being involved in complex business relationships with the developers of the underlying

compute system, such as a cloud server system. Arm CCA allows Realms and their resources to be dynamically created and destroyed on demand under the control of the Normal world host.

3.1 Transitioning isolation boundaries

The Arm architecture includes the SVC, HVC, and SMC instructions to enable the PE to move between Exception levels:

- The Supervisor Call (SVC) instruction enables unprivileged user-space code in EL0 to call the operating system in EL1 or EL2. The use of registers is defined by the OS and is typically based on the Arm Procedure Call Standard.
- The Hypervisor Call (HVC) instruction enables EL1 software to request a transition to EL2. For example, an OS might use the `HVC` instruction to request hypervisor services.
- The Secure Monitor Call (SMC) instruction enables EL1 and EL2 software to request a transition to EL3. For example, Secure Monitor firmware might use the `SMC` instruction.

Only the software running at EL3 can switch between Normal, Secure and Realm Security states. To ensure basic interoperability with Secure Monitor firmware, a standard SMC Calling Convention (SMCCC) (DEN0028) is defined. The SMCCC describes how code uses registers to pass arguments and return results. One of the arguments specified by the SMCCC is a 32-bit Function Identifier passed on every SMC and HVC call. The calling software uses this Function Identifier to identify the firmware service. Ranges of Function Identifiers are allocated to standard platform services. Some ranges are reserved for vendor-specific services, though it is advisable to use these only when software is tightly integrated and there is no risk of identifier conflicts.

The SMCCC can also be used with the Hypervisor Call (HVC) instruction to trigger EL2 software, such as the services provided by a hypervisor. These services are typically hypervisor-specific, although Arm Paravirtualized Time for Arm-based Systems (DEN0057) defines a standardized interface for measuring stolen time on virtualized system.

The Firmware Framework for A-profile (FF-A) builds on the SMCCC to provide a generalized method for exchanging messages between Exception levels across Normal and Secure worlds. This enables vendor services to be isolated in de-privileged sandboxes, preventing code bloat in EL3 firmware. FF-A provides a standardized method for discovering service partitions using a Universally Unique IDentifier (UUID), interrupting and resuming blocking calls, and message passing. FF-A also describes the function of the Secure Partition Manager (SPM) which serves as a lightweight hypervisor for the Secure world partitions.

The following documents describe FF-A:

- The core specification: Arm Firmware Framework for Arm A-profile (DEN0077)
- A memory management supplement: FF-A Memory Management Protocol (DEN0140)
- A secure partition lifecycle supplement: FF-A SP Lifecycle (DEN0143)

The Arm Confidential Compute Architecture (CCA) provides a protected execution environment called Realms. The core component, the Realm Management Monitor (RMM), is firmware managing

the execution of the Realm VMs and their interaction with the Normal world hypervisor. The RMM interfaces are SMCCC-based, with the Normal world communicates with RMM through the Realm Management Interface (RMI), and the RMM interacts with Realm VMs via the Realm Service Interface (RSI).

For more information about the Arm CCA, see Learn the architecture - Introducing Arm Confidential Compute Architecture, Learn the architecture - Arm Confidential Compute Architecture software stack and Realm Management Monitor specification (DEN0137).

3.2 Secure and monitor firmware interfaces

System designers can partition sensitive device hardware and software resources so that they are isolated from the Normal world. This helps protect against a wide range of possible software attacks.

The following table lists a set of standard interfaces provided by secure and monitor firmware. Many of these depend on capabilities provided by the underlying hardware, with the firmware offering a secure and abstract interface to the hardware. The detail of this hardware interface is implementation dependent. Typically, these services are used by the operating system or intermediate firmware running in the Normal world.

Standard	Description
Power State Coordination Interface (PSCI) (DEN0022)	System and CPU state transitions. For example, requesting that cores be powered up or down, or transferring secure context between cores.
True Random Number Generator (TRNG) (DEN0098)	Obtain a conditioned entropy source from a secure back end. For example, for generating seeds or keys.
Errata Management Firmware Interface (DEN0100)	Discover CPU errata and whether they are mitigated in firmware.
Software Delegated Exception Interface (SDEI) (DEN0054)	Mechanism for an OS to selectively mask interrupts prior to NMI being supported in hardware (2021 extensions).
DRTM Architecture for Arm (DEN0113)	Establish trust in the boot chain (Dynamic Root of Trust Measurement).
Platform Security Firmware Update for the A-profile Arm Architecture (DEN0118)	Securely write new images into the platform firmware store, for example flash.
TPM Service Command Response Buffer Interface over FF-A (DEN0138)	Interact with a standard Trusted Platform Module, for example to use and store private keys.
PCI Configuration Space Access Firmware Interface (DEN0115)	Alternative firmware-based method for managing PCI configuration on hardware which does not support the standard Enhanced Configuration Access Mechanism (ECAM).
SMCCC (General Service Queries) (DEN0028)	Discover firmware support for specific CPU features and security vulnerability workarounds.

Note

In older documentation, you might see the term “standard secure services”. This document uses the term “secure and monitor firmware interfaces”, which means the same as “standard secure services”.

3.3 Normal world firmware

Many systems run firmware in the Normal world, presenting industry standard interfaces found in other CPU architectures such as x86. Open source reference firmware implementations include:

- Tianocore EDK II, which supports both ACPI-based and Devicetree-based firmware models
- Uboot, which supports Devicetree-based firmware models

Key functionality provided by Normal world firmware includes booting the operating system and helping it discover hardware.

3.3.1 Operating system boot

Unified Extensible Firmware Interface (UEFI) is an open industry standard maintained by UEFI Forum. It defines a standard boot environment for handing over control from SoC-specific firmware boot loaders to the operating system. It is supported by common operating systems including Linux and Microsoft Windows. Services provided include:

- Console input and output for use by the OS loader for debug purposes early in the boot process, before it has loaded its own device drivers
- Booting from block devices or the network
- Accessing files from an EFI file system during boot, for example configuration information, or executables such as EFI applications and OS boot loaders.
- Network device access during boot

Aside from booting, UEFI defines several runtime services which remain available after OS boot, for example:

- Reading and writing firmware non-volatile variables, including configuring UEFI Secure Boot
- Reading and updating the system real time clock
- Requesting the system is shutdown or reset

UEFI also provides the low-level interface for accessing other normal world firmware services such as ACPI and SMBIOS.

3.3.2 Hardware discovery

Operating systems need to know about the hardware present in the system. For example, an OS needs to know if a generic UART exists, and the memory addresses for accessing its registers. Some buses, such as PCIe, provide standard mechanisms for discovering and enumerating connected devices. However, other hardware such as an I^2C sensor, might not be discoverable

in this way. Even with PCIe the operating system needs to know the base address of the PCIe configuration registers used for enumeration.

To avoid customized operating system images containing hardcoded information about hardware, operating systems designed to run on multiple systems typically use one of the following firmware-based mechanisms to assist with discovery:

- Advanced Configuration and Power Interface (ACPI): an open standard maintained by UEFI Forum. This allows firmware to present an abstracted view of hardware to the operating system, improving cross-compatibility and enabling an older operating system to run on new BSA-compliant hardware.
- Devicetree: a data structure passed by the boot loader to the OS, statically describing hardware devices and the connections between them.

The choice of ACPI or Devicetree is determined by the OS:

- Windows only supports ACPI
- Linux support varies by distribution:
  - Distributions targeting servers, for example Red Hat Enterprise Linux, typically use ACPI
  - Distributions targeting embedded applications tend to use Devicetree
- Android uses Devicetree

The following Arm-specific supplements complement the core ACPI specification:

Standard	Arm hardware binding
ACPI for Arm Components (DEN0093)	Miscellaneous Arm-licensed components, for example generic UART and watchdog.
Arm Functional Fixed Hardware (FFH) (DEN0048)	CPU power state enumeration and performance monitoring using AMUs, SMC, and HVC calls.
ACPI for Memory System Resource Partitioning and Monitoring (MPAM) (DEN0065)	MPAM memory system components
ACPI for CoreSight (DEN0067)	CoreSight debug trace components and their topology, for example the System Debug Bus and Embedded Trace Buffer.
ACPI for CoreSight Performance Monitoring Unit Architecture (DEN0117)	CoreSight PMU instances.
ACPI for Arm RAS (DEN0085)	Error sources, for example CPU, memory, and SMMU.
IO Remapping Table (IORT) (DEN0049)	Input/output topology of the system, PCIe, and System Memory Management Unit (SMMU).

ACPI also provides a standard interface for abstracting power and performance management.

3.4 System management controller firmware

Many SoC designs, including those built using Arm Compute Subsystems (CSS), feature additional hardware controllers dedicated to particular system management and security tasks. Typically, these are M-profile microcontrollers running their own specialized firmware.

To enable entities such as the host operating system to interoperate with the controller firmware, the following message-based interfaces are available:

- System Control and Management Interface (SCMI) (DEN0056): Mainly used for managing system power, it also features interfaces for resetting domains, accessing sensors and controlling pins such as General Purpose IO (GPIO). For more information, see Power Control.
- MPAM Firmware-backed (Fb) Profile (DEN0144): MPAM enables partitioning of constrained memory resources between different software running on the application processor. This interface provides an alternative method for managing MPAM configuration on hardware where the MPAM Memory System Controller (MSC) registers are not directly accessible by the application processor.

Messages are exchanged between entities using a variety of transport mechanisms, for example an area of shared memory. The details of these mechanisms can be discovered using ACPI or Devicetree. This provides system designers with the flexibility to create an implementation that meets the needs of their own particular use cases.

For example, many systems use the standard ACPI Platform Communications Channel to exchange SCMI messages between the OS and the SCP, and DMTF-defined Management Component Transport Protocol (MTCP) messages between the OS and MCP.

4. Hardware components and subsystems

Several standard hardware components and subsystems exist alongside the processor cores in a modern Arm-based System-on-Chip (SoC). These components support essential system functions such as interrupts, memory access, and power management, ensuring consistent operation for system software.

The AMBA hardware interfaces enable the integration of additional components including memory systems, peripherals, and accelerators.

4.1 System components

Several system components are defined with standardized hardware connections and a software programming model to complement the A-Profile architecture. Arm provides these as standalone IP blocks, or pre-integrated into Compute Subsystems (CSS) such as Neoverse CSS.

A Generic Interrupt Controller (GIC) takes interrupts from peripherals, prioritizes them, and delivers them to the appropriate processor core. It is the standard interrupt controller for Arm Cortex-A and Arm Cortex-R profile processors, supporting a range of systems from single-core devices to large multi-chip designs with hundreds of cores. For more information about the GIC, see Learn the architecture - Generic Interrupt Controller.

Arm provides several GIC models that implement a range of interrupt management solutions for all types of Arm Cortex multiprocessor systems. Standalone controllers range from the simplest CoreLink GIC-400 for systems with small CPU cores counts, to CoreLink GIC-700 for high-performance multi-chip systems.

The System Memory Management Unit (SMMU), also known as IOMMU, translates virtual addresses from DMA-capable devices to physical addresses. It performs a similar task to the MMU in a PE. An example implementation of an SMMU is Arm's CoreLink MMU-700. For more information about the SMMU, see Learn the architecture - SMMU Software Guide.

The CoreSight architecture provides a comprehensive debug and trace infrastructure for Arm processors, enabling both self-hosted and external debugging. Components included in CoreSight include:

- Cross Trigger Interface (CTI) and Cross Trigger Matrix (CTM) for coordinating debug activities across multiple processor cores
- Embedded Trace Macrocell (ETM) for generating trace data.

The CoreSight architecture supports several connection methods, including JTAG, and facilitates trace data capture both on-chip and off-chip using components such as the Trace Port Interface Unit (TPIU) and Embedded Trace Buffer (ETB). These components can be found in Arm products such as CoreSight SoC-600. For more information about CoreSight, see Learn the architecture - Introducing CoreSight debug and trace.

4.2 System support for A-profile features

Some optional features of A-profile cores depend on complementary support in other system components.

Memory System Resource Partitioning and Monitoring (MPAM) enables the partitioning and monitoring of memory resources across applications or virtual machines. The goal is to limit the performance impacts caused by resource contention in the memory system. The MPAM System Component Specification (IHI0099) describes the requirements for shared memory system components such as cache memories, interconnects and memory channel controllers. For more information about MPAM, see Learn the architecture - MPAM Overview.

Reliability, Availability and Serviceability (RAS) defines a framework for gathering and reporting hardware faults, allowing for a wide range of capabilities from basic “reset-on-error” recovery through to sophisticated error handling and pre-emptive replacement of failing components. The Arm Reliability, Availability, and Serviceability (RAS) System Architecture (IHI0100) defines the features of components that can detect errors, such as memory and I/O controllers. For more information, see Learn the Architecture - RAS Overview.

The Realm Management Extension (RME) defines the hardware-based isolated execution environment known as Realms. It underpins the Arm Confidential Compute Architecture (CCA) but can also be used for other purposes. The Arm Realm Management Extension (RME) System Architecture (DEN0129 defines the required system properties, including definition of resources, capabilities, and components. For more information, see Learn the architecture - Realm Management Extension.

4.3 Power control

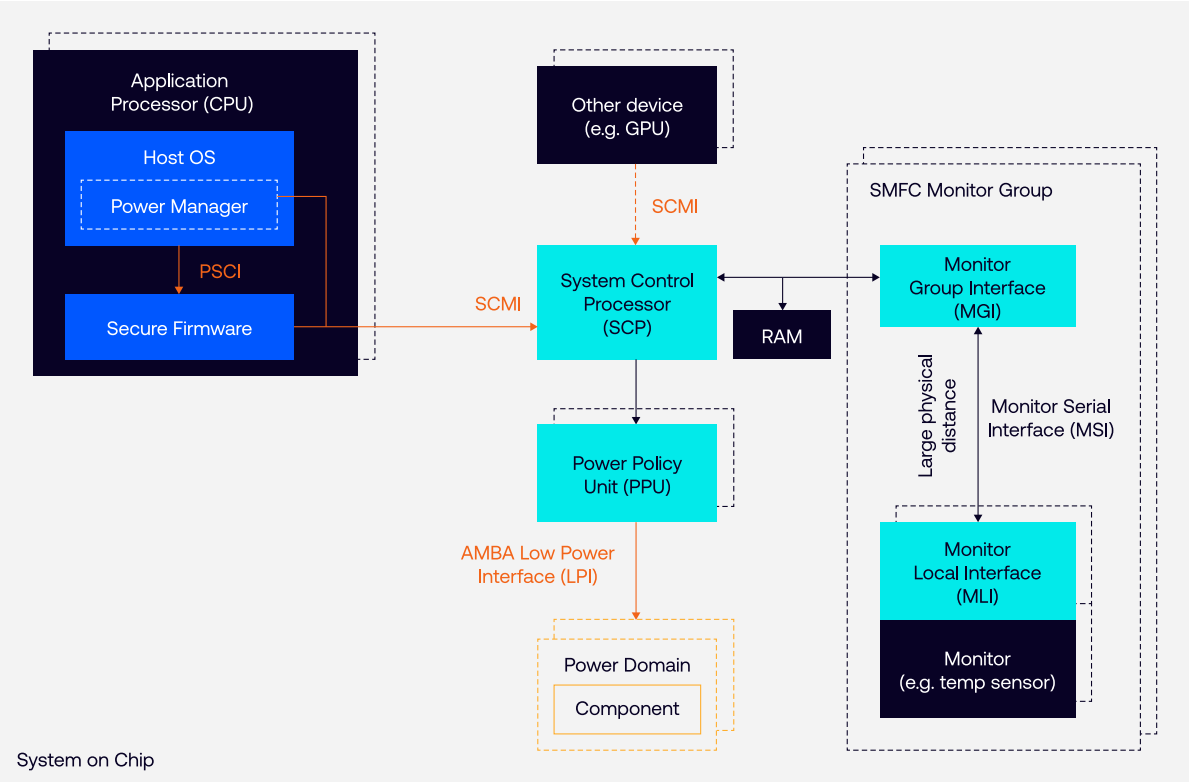
The Arm architecture is designed to be highly energy efficient. However, simply using individually power-efficient components is not sufficient for best results. Minimizing system power requires intricate and coordinated clock, power, and thermal management.

The Power Control System Architecture (PSCA) (DEN0050) defines a flexible framework for system power control integration using standard infrastructure components and low-power interfaces. It enables a generic OS to delegate SoC-specific power and performance control to a specialized subsystem, while allowing the OS to remain responsible for assessing the performance needs of workloads and translating them to desired performance levels.

The PSCA is important in most markets, including mobile and servers, and is implemented in many Arm Compute Systems (CSS).

The following diagram summarizes the main components of the PSCA:

Figure 4-1: PCSA main components



The key elements of the PCSA are as follows:

- System Control Processor (SCP). A processor, for example a microcontroller, which offloads and abstracts details away from the host Operating System Power Manager (OSPM).
    - The SCP operates independently from, and reacts faster than the OS kernel. It securely arbitrates power requests from multiple sources, including when the main application processors are themselves in a low-power state or subject to thermal throttling.
    - A complex system may use additional, distributed secondary SCPs to provide functional encapsulation of subsystems and lower-latency local control. Product documentation refers to these as Local Control Processors (LCPs).
- Power Policy Unit (PPU). Hardware which converts power domain policy to basic power control signals (AMBA Low Power Interface) for each power domain. See Arm Power Policy Unit Architecture Specification (DEN0051) for more information about the PPU.
- System Monitoring Control Framework (SMCF). Defines how data from distributed sensors can be collected and trigger alerts in the SCP, for example when a temperature threshold is breached. The SMCF is designed to manage a large and diverse set of on-chip sensors. It does this by presenting software with a standard interface to control the monitors, regardless of type, and reducing the software load of sampling and data collection. See System Monitoring Control Framework Architecture Specification, (DEN0108) for more information.

Other entities, including the application processor system software, communicate with the SCP using the System Control and Management Interface (SCMI) (DEN0056) to perform various functions including:

- Discovery of power and performance domains in the platform
- Setting performance levels and power states for each domain
- Obtaining sensor information, for example temperature data

When the operating system power manager requires a change in CPU and system power states, such as shutting down a core, the application processor EL3/Monitor firmware should be called using PSCI. For more information about PSCI, see Secure and monitor firmware interfaces. This allows the firmware to perform additional operations before passing the request to the SCP using SCMI. This typically includes saving and restoring EL3 monitor context, and managing secure-world interrupts. The OS can use SCMI directly for other operations.

For a more detailed overview of SCMI, see the Power and Performance Management using Arm SCMI white paper.

4.4 Connecting functional blocks together

The Advanced Microcontroller Bus Architecture (AMBA) family of specifications define standard interfaces between functional components on a system-on-a-chip (SoC). Widely adopted, feature rich, and architecture neutral, AMBA has a comprehensive and thriving ecosystem of third-party partners offering compatible products and solutions. Examples include peripheral IP, modelling, and validation tools.

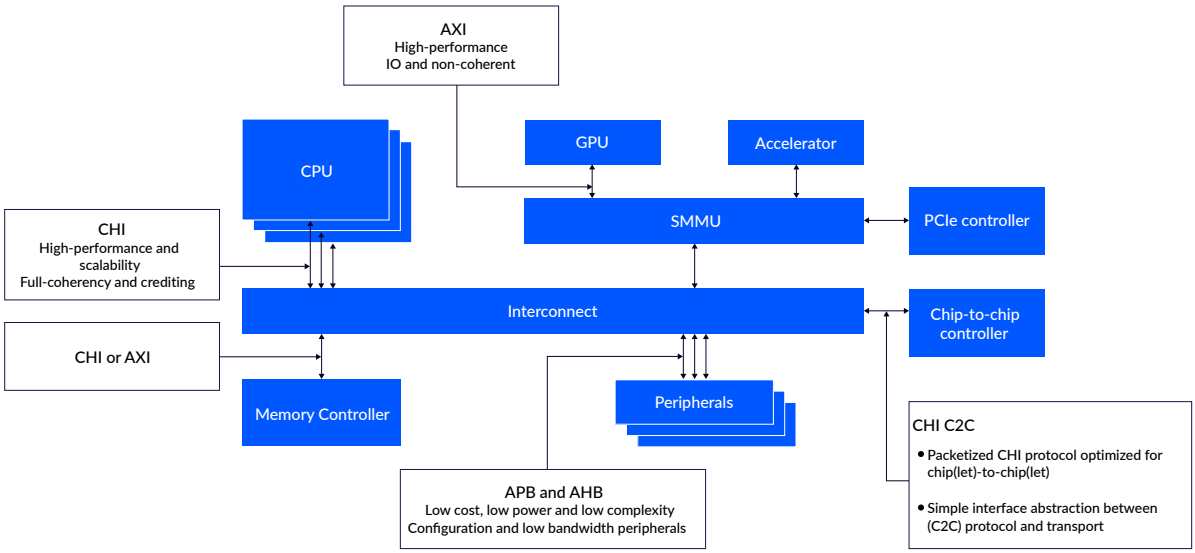
4.4.1 Key specifications

Processors and other entities such as accelerators require efficient access to memory and memory-mapped devices, including peripherals.

AMBA has significantly evolved over many generations to address these needs. AMBA offers a range of interface options that support non-coherent, IO-coherent, and fully coherent traffic, at various levels of power, performance, and area requirements.

The following figure shows the AMBA interfaces:

Figure 4-2: AMBA interfaces



The AMBA Coherent Hub Interface (CHI) (IHI0050) offers high performance with full cache coherency, ensuring all processors in the system see the same data. Some of the key features include:

- Support for high frequency, non-blocking coherent data transfer between many processors.
- A layered model to enable separation of communication and transport protocols for flexible topologies, including a cross bar, ring, or mesh.
- Cache stashing to allow accelerators or IO devices to store critical data in a CPU cache for low-latency access.
- Far atomic operations to enable the interconnect to perform high-frequency updates to shared data.
- End-to-end data protection and poisoning signaling.
- Realm management support for confidential compute.

The CHI protocol requires a coherent interconnect block to connect multiple components within a chip. An example interconnect from Arm which features CHI interfaces is the Arm Neoverse CMN-S3. For more information about the operation of CHI, see Learn the architecture - Introducing AMBA CHI.

There are several ways to extend the interconnect beyond the die, each offering different design trade-offs. A SoC design may implement several interfaces for different functions. Using CHI end-to-end lets architectural features span chip and chiplet boundaries, enabling a common memory and security model. This approach avoids protocol conversions, incompatibilities, and extra latency. To facilitate this, the AMBA CHI Chip-to-Chip (C2C) Architecture Specification (IHI0098) defines how the CHI protocol is encapsulated into fixed-size containers for transport over an appropriate layer. Example transports include UCIe streaming between chiplet dies and PCIe physical layer between chip packages. The IP blocks implementing the CHI-C2C packetization and the chosen

transport layer are typically connected on-chip using the AMBA Credited eXtensible Stream (CXS) protocol (IHI0079).

The simpler AMBA Advanced eXtensible Interface (AXI) (IHI0022) provides a solution for high performance on-chip connections which only need IO-coherent or non-coherent access. Originally developed before CHI, AXI has been updated to include several performance and scalability features which align and complement AMBA CHI. The AXI Coherency Extension (ACE), which added full coherency to AXI, has been superseded by CHI. For more information about AXI, see Learn the architecture - An introduction to AMBA AXI.

Other interconnect interfaces include:

- AMBA Advanced High-Performance Bus (AHB) (IHI0033), which is generally used with M-profile.
- AMBA Advanced Peripheral Bus (APB) (IHI0024), which offers highly compact low-power implementations for low bandwidth peripherals and configuration registers.

4.4.2 Application-specific interfaces

In addition to load and store operations, AMBA includes the following specialized point-to-point interfaces targeting specific applications:

AMBA standard	Description
AXI-Stream (IHI0051)	Interface for transferring streams of arbitrary unidirectional data. AXI-Stream has numerous applications, including connecting GIC-600 blocks together.
Distributed Translation Interface (DTI) (IHI0088)	SMMU interface for querying address translations, which can be cached. DTI-ATS is used by a PCIe Root Port with Address Translation Services (ATS) support. DTI-TBU is used between internal SMMU components.
Local Translation Interface (LTI) (IHI0089)	SMMU interface for an I/O device to query address translations. LTI is simpler to use than DTI, but designed for short distances without any caching.
Advanced Trace Bus (ATB) (IHI0032)	Data-agnostic interface for transferring trace information between components.
Low Power Interface (LPI) (IHI0068)	Defines Q-Channel and P-Channel interfaces. LPI is designed to manage clock and power features of SoC components.
Generic Flash Bus (GFB) (IHI0083)	Interface between a generic flash controller, for example Arm CoreLink GFC-200, and a process-specific controller for flash or similar non-volatile memory.