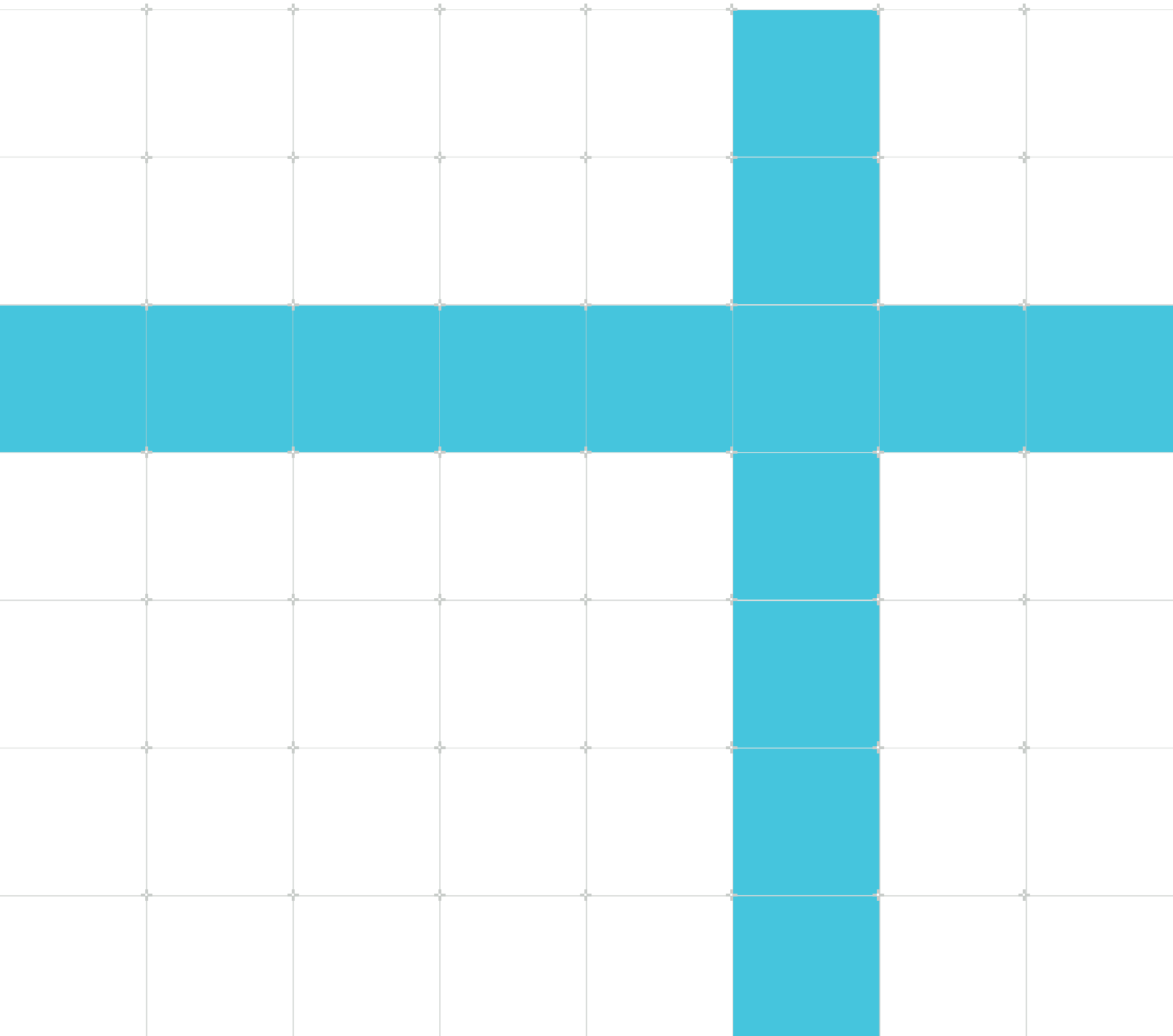
SystemReady Pre-Silicon BSA and SBSA Integration and Testing Guide

Version 2.1

Non-Confidential

Issue 02
102858_0201_02_en



SystemReady Pre-Silicon BSA and SBSA Integration and Testing Guide

Release information

Document history

Issue	Date	Confidentiality	Change
0201-02	5 June 2025	Non-Confidential	Appendix addition
0201-01	21 January 2025	Non-Confidential	Appendix addition
0200-01	24 December 2024	Non-Confidential	Major update
0100-01	4 April 2024	Non-Confidential	Minor update
0100	2 September 2022	Non-Confidential	First release

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Confidentiality Status

Unrestricted Access is an Arm internal classification.

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. Introduction....8
1.1 BSA/SBSA compliance....8
1.2 SystemReady pre-silicon compliance testing and the silicon journey....9
1.3 About this guide....10

2. BSA and SBSA overview....12
2.1 BSA requirements summary....12
2.1.1 PE architecture....13
2.1.2 Memory map....13
2.1.3 Interrupt controller....13
2.1.4 PPI assignments....14
2.1.5 System MMU and device assignment....14
2.1.6 Clock and timer subsystem....14
2.1.7 Wakeup semantics....14
2.1.8 Power state semantics....14
2.1.9 Watchdogs....15
2.1.10 Peripheral subsystems....15
2.1.11 Presenting an on-chip peripheral as a PCIe device....15
2.1.12 PCIe integration....15
2.2 SBSA requirements summary....16
2.2.1 SBSA Level 3....17
2.2.2 SBSA Level 4....17
2.2.3 SBSA Level 5....17
2.2.4 SBSA Level 6....18
2.2.5 SBSA Level 7....18

3. Design and integration guidance....20
3.1 Issue 1 - Non-contiguous configuration space as the ECAM region for endpoints or switches under the Root Port hierarchy....21
3.2 Issue 2 - Non-compliance with completion handling of configuration reads....22
3.3 Issue 3 - PCIe Root Port configuration space supports only 32-bit R/W access....23
3.4 Issue 4 - Root Port exposes ATS and PRI capabilities....24
3.5 Issue 5 - Enumeration code reports valid entries for device 1 to device 31 on bus 0....24

3.6 Issue 6 - ATC invalidate all command sent by SMMU does not automatically translate the address field.... 25
3.7 Issue 7 - Alternate Routing ID (ARI) mode handling.... 25
3.8 Issue 8 - Incorrect type (Type 0 or Type 1) conversion for downstream configuration requests.... 26
3.9 Issue 9 - Support for message-based interrupts.... 27
3.10 Issue 10 - Completion timeout response type.... 27
3.11 Issue 11 - PCIe response of Cpl-SC and not CplD-SC for bad ATS Translation Requests.... 28
3.12 Issue 12 - PCIe Root Port supports only 32-bit aligned R/W access.... 28
3.13 Issue 13 - Non-compliant UART.... 29

4. Pre-silicon BSA compliance testing.... 30
4.1 Why is pre-silicon BSA compliance testing needed?.... 30
4.2 Overview of the pre-silicon BSA compliance solution.... 31
4.3 What is ACS?.... 32
4.4 What are BSA/SBSA compliance tests?.... 32
4.5 Why is exerciser needed?.... 32
4.6 What is exerciser?.... 32
4.7 Compliance test software stack for exerciser with a UEFI shell application.... 34
4.8 How to access pre-silicon BSA compliance tests.... 35

5. Related information.... 36

A. Running BSA ACS tests on the Arm Neoverse N2 reference design (RD-N2) Fixed Virtual Platform (FVP) model.... 37
A.1 Setting up the RD-N2 FVP.... 38
A.1.1 Setting up the Workspace and Software Stack.... 38
A.1.2 Setup the Host Based Build Environment.... 39
A.1.3 Download the Reference Design Model.... 39
A.1.4 Modify and Build the Software Stack.... 39
A.1.5 Setting the PCIe Hierarchy.... 40
A.2 Baremetal BSA-ACS Run.... 40
A.2.1 Compile and build the BSA-ACS.... 40
A.2.2 Run the BSA-ACS.... 41
A.3 UEFI BSA-ACS RUN.... 42
A.3.1 EDK2 Setup and BSA-ACS Compilation.... 42
A.3.2 Run the BSA-ACS.... 44

B. Arm SystemReady Pre-Silicon Compliance Checklist.....45

1. Introduction

Arm SystemReady is a compliance program based on a set of hardware and firmware standards. SystemReady supports Arm-based servers, infrastructure edge, and embedded IoT systems to comply with specific requirements enabling compatibility and system interoperability with off the shelf standard operating system images.

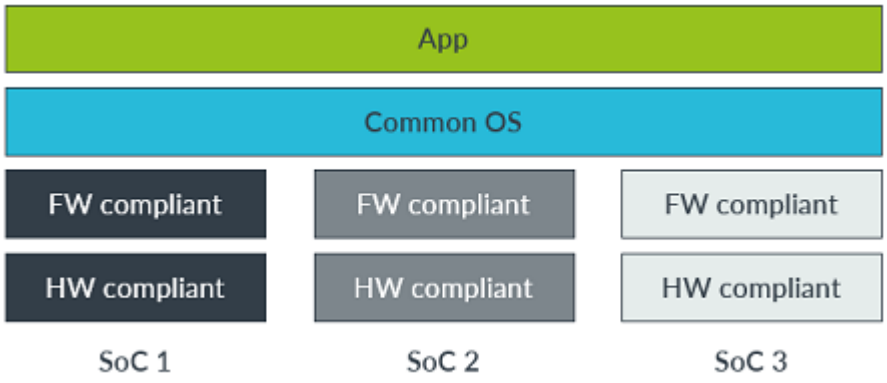
Systems that are designed for the end user to install and run generic off-the-shelf standard operating systems unmodified out-of-the-box on Arm-based devices provide a seamless experience for system integrators, software developers, and end users. Silicon vendors can rely on standardized layers and focus on innovating and deploying differentiating ones.

1.1 BSA/SBSA compliance

The SystemReady standards include the Base System Architecture (BSA) and Base Boot Requirements (BBR) specifications, and market-specific supplements such as the Server Base System Architecture (SBSA).

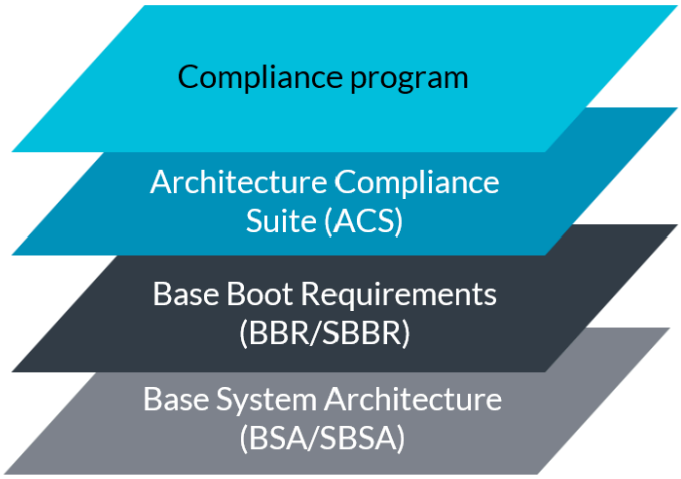
Compliance to the BSA specification and its supplements such as SBSA is a key requirement for SystemReady compliance. The following diagram shows an example software stack in BSA/SBSA compliant SoCs:

Figure 1-1: Software stack in BSA/SBSA compliant SoCs



The following diagram shows the top-level requirements for BSA/SBSA compliance:

Figure 1-2: Top-level requirements for BSA/SBSA compliance



BSA defines a system architecture and enables hardware compatibility. It defines the minimum CPU and system requirements to boot and run an operating system. This means that BSA compliance needs to be achieved in silicon hardware.

However, integration and hardware BSA compliance issues are common, leading to software-visible defects and interoperability issues. PCIe Enhanced Configuration Access Mechanism (ECAM), timers, and interrupts are challenging areas and often require hardware changes by silicon vendors and third-party IP vendors.

Non-compliant silicon is not competitive. Mitigating hardware compliance issues in software often requires patches, custom operating systems, or firmware workarounds. Developing these patches and workarounds is costly, time consuming, and not always feasible. Patches can be challenging to upstream and be approved by maintainers. The end-product may be sub-optimal, uncompetitive, or not acceptable at all. The worst-case scenario may require silicon re-spin or risk non-adoption.

1.2 SystemReady pre-silicon compliance testing and the silicon journey

When developing a System on Chip (SoC) to be SystemReady compliant, silicon vendors must consider BSA compliance throughout all pre-silicon development phases. This is because BSA defines a system architecture, and compliance needs to be achieved in silicon hardware.

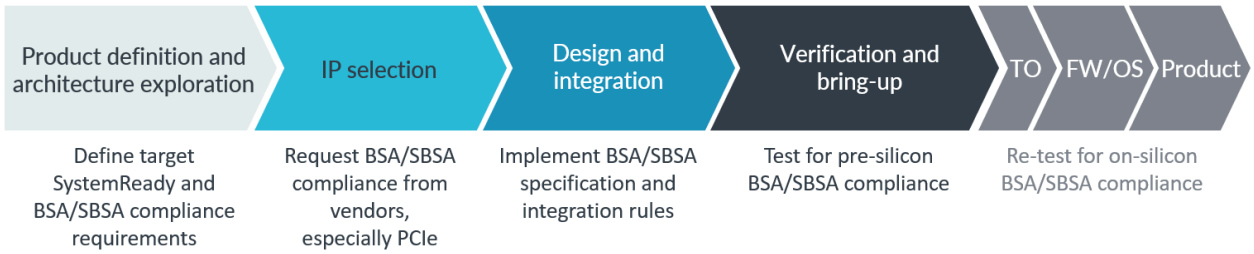
SystemReady pre-silicon compliance testing is an enablement program that helps silicon vendors achieve BSA compliance before taping out. It provides a well-defined and low-risk path to SystemReady. SystemReady pre-silicon compliance testing provides the following:

- A framework with specific steps for silicon vendors to take to be BSA compliant
- Guidance, including this document

- Tools, including the pre-silicon BSA/SBSA compliance tests

The following diagram shows the key steps involved in the silicon journey towards SystemReady:

Figure 1-3: Key steps in the silicon journey towards SystemReady



These steps are as follows:

1. Product definition and architecture exploration:
   - Define the standard system architecture specification for the target SoC
   - List the requirements for the chosen compliance according to the specification
2. IP selection:
   - Choose Arm processor IP and system IP components that comply with the BSA/SBSA specification chosen in step 1
   - Ensure that IP components sourced from vendors or third-parties are BSA/SBSA compliant

   It is always recommended to use the latest compliant version of IP components.
3. Design and integration:
   - Implement the BSA/SBSA guidelines and integration rules for the SoC design according to the system architecture specification
4. Verification and bring-up:
   - Run the pre-silicon BSA compliance tests on the RTL

Arm strongly recommends that silicon vendors run the tests on bare-metal RTL using the pre-silicon compliance solution. Pre-silicon BSA/SBSA compliance solutions are developed collaboratively by Arm and the EDA partners. Compliance solutions from EDA partners integrate compliance tests, drivers, and the exerciser to give a seamless out-of-the-box experience.

The compliance tests detect architectural bugs during the early phases of the design process, allowing BSA non-compliant issues to be fixed in the RTL before tape-out. Compliance testing is not a substitute for design verification, but rather complements it.

1.3 About this guide

This guide outlines the IP and system integration steps needed to develop a SoC that is BSA compliant and suitable for SystemReady compliance. It focuses on common compliance issues, and

the most common PCIe-BSA integration issues. In particular, this version of the document focuses on high-speed peripherals such as PCIe.

This guide also provides information about the following:

- Guidance on how hardware designers and verification engineers can address and prevent compliance issues
- An overview of the BSA specification, describing its rules and why they are important
- An overview of the compliance testing that is performed during design and verification and before tapeout

This guide focuses on the architecture, and not on any specific micro-architecture or implementation. This guide is meant as a starting point, to be complemented by other integration guides from Arm and other IP vendors which focus on specific IP that implements the BSA requirements and functionality.

This guide builds on the lessons learned by Arm and its partners throughout the years developing BSA and SBSA compliant systems, with the goal of helping silicon vendors to have a faster and lower risk path to develop silicon for SystemReady.

Note

BSA, in this guide, is used as generic term to refer to the BSA specification itself and its supplements. BSA/SBSA is used when both BSA and SBSA are applicable.

2. BSA and SBSA overview

The Arm Base System Architecture (BSA) specification defines a hardware system architecture, based on Arm 64-bit architecture, that system software can rely on. System software includes operating systems, hypervisors, and firmware. The requirements and runtime features required by the BSA ensure that users can install, boot, and run generic operating systems and hypervisors.

The Server Base System Architecture (SBSA) supplement defines a hardware system architecture, based on Arm 64-bit architecture, that server system software can rely on. Server system software includes operating systems, hypervisors, and firmware. SBSA provides a consistent target for software developers, driving the alignment of server market capabilities forward over time.

The primary goal of the BSA and SBSA specifications is to standardize system architecture so that a single suitably built OS image can run on all compliant hardware. A driver-based model for advanced platform capabilities beyond basic system configuration and boot is required. While this driver model is outside the scope of BSA and SBSA, having fully discoverable and describable peripherals aids in the implementation of such a model.

Arm BSA and SBSA require that some of the system hardware is abstracted, provisioned, and described to software using firmware, Hardware Abstraction Layers (HALs), or board support packages (BSPs). The Arm BBR specification describes boot requirements for operating systems that require the use of UEFI, ACPI, Device tree and SMBIOS. The BBR requirements are complementary to the BSA and SBSA requirements, to achieve a standard OS and hypervisor boot behavior.

2.1 BSA requirements summary

The Arm BSA specification requirements are structured for three software views of a system:

- Operating system. This view describes a base set of functionalities that an operating system, either running on bare-metal or under virtualization, can rely on.
- Hypervisor. This view describes a base set of functionalities that a Normal world hypervisor can rely on.
- Platform security functionality. This view describes a base set of functionality that standard platform secure firmware can rely on. Specifically, it provides requirements to support software running in Secure state.

This approach reflects the fact that software for Arm BSA compliant devices is composed from components that rely on these different software views, and that are supplied by different providers in the ecosystem.

The BSA specification groups the requirements into several sections. Each section describes the requirements for specific system components and features. The following is a summary of the requirements groups and examples of the requirements in each group. For a complete list of the requirements, please refer to the BSA specification.

Section 3.14 of the BSA specification offers a comprehensive checklist of minimum hardware requirements to install, boot, and run an operating system on bare-metal or within a virtualization environment.

2.1.1 PE architecture

The Processing Element (PE) requirements cover a range of Armv8-A and Armv9-A PE features, including:

- Details of PE features that are required, conditionally required, optional, or recommended. Examples include Advanced SIMD and floating-point support, cryptography extensions, PMU extension, CRC32 instructions, SVE/SVE2, Pointer Authentication, and Memory Tagging Extension
- Requirements for PE symmetry across the system, with some exceptions and limitations for systems with big.LITTLE architecture outlined in Section A (Heterogenous Systems)
- Requirements on the number of PEs in the system, with relation to the GIC controller used
- Requirements of PE security features, such as side-channel speculation restrictions and barriers
- Other PE requirements to support AArch64 operating systems and hypervisors, including little-endian support, support for 4KB translation granules, and requirements around EL0/EL1/EL2/EL3 Exception Level support

2.1.2 Memory map

The memory map requirements focus on how the system configures the memory map for operating system use. These requirements include the following:

- Requirements for memory address space population, access, and access errors
- Requirements around DMA requests and requesters, and their relation to the SMMU
- Requirements specific to Non-secure and Secure world memory address spaces and accesses, and their relation to the SMMU

2.1.3 Interrupt controller

The BSA requirements for interrupt controllers include:

- Requirements of standard interrupt controllers, including GICv2, GICv2m, GICv3, and GICv3+ITS, depending on the number of PEs present in the system, and whether the system supports PCIe devices or not
- Additional requirements for MSI X, ITS, LPI, and SPI for systems with GICv3 and PCIe support
- GICv3 security states requirements
- Additional GIC rules and information are listed in Section I (GICv2m Architecture) and Section J (GICv2m compatibility in a GICv3 system)

2.1.4 PPI assignments

These are requirements for specific interrupts, and recommended interrupt IDs, that a compliant base system must implement. The PPI requirements are described for operating systems, hypervisors, and platform security functionality views.

2.1.5 System MMU and device assignment

This section covers a detailed list of System MMUs (SMMUs) related requirements. Additional SMMU requirements are listed in Section D (SMMUv3 Integration). The SMMU requirements include the following:

- SMMUv2, SMMUv3, and SMMUv3.2 specific conditional requirements, depending on supported system features such as PCIe and Secure-EL2
- SMMU feature requirements that are conditional on PE features, such as extended virtual addresses, range invalidation, 52-bit output size, and 16-bit ASID
- MPAM requirements for systems with SMMUv3.2 or higher
- Requirements for hypervisors and device assignments for virtualization, including around Secure-EL2, MPAM, 16-bit VMID, and other areas
- Additional SMMU requirements for platform security functionality

2.1.6 Clock and timer subsystem

The BSA defines requirements for system counters using the Arm Generic Timer, including its minimum frequency, roll-over period, and counter bits length.

There are also requirements for a system wakeup timer that can be used when the PE timers are powered down.

The requirements include specific generic counter and timer register frame and memory mappings.

2.1.7 Wakeup semantics

This section covers requirements around low-power states, PE wakeup methods, interrupts, and events. These requirements ensure that operating systems and hypervisors can understand the relationship between the different power domains and states in the system, and be able to control and respond to these state transitions.

2.1.8 Power state semantics

The BSA specification does not require a particular hierarchy of power domains, but there are some rules and semantics that must be followed. These rules and requirements allow the operating

system or hypervisor to be able to reason about the wakeup events, and to know which timers are available to wake the PE.

To help illustrate these rules, this section of the specification provides examples of possible hierarchy of power domains that can conform to BSA.

2.1.9 Watchdogs

Watchdogs are optional for BSA compliant systems. However, if they are implemented, this section lists several rules that the watchdog implementations must comply with. For example, the BSA specification defines an Arm Generic Watchdog (in Section C) with specific rules to ensure that the watchdog can be supported in generic operating systems and hypervisors.

2.1.10 Peripheral subsystems

This section includes conditional requirements for peripherals such as USB 2.0/3.0 and SATA, if those devices are used to install or boot the operating system.

This section also includes requirements for the UART to enable user access to the operating system's main and debug consoles. The UART requirements allow either an Arm Generic UART (defined in BSA specification, section B), or a fully compliant 16550 UART. The Arm Generic UART is a subset of PL011 Arm IP.

Finally, this section includes PCIe conditional requirements and points to detailed PCIe integration information in Section E (PCIe Integration).

2.1.11 Presenting an on-chip peripheral as a PCIe device

This section describes options and rules for presenting on-chip peripherals as PCIe devices. The options include using a Root Complex Integrated Endpoint (RCiEP), with rules defined in Section F, and using an Integrated Endpoint (i-EP), with rules defined in Section G.

2.1.12 PCIe integration

This section describes detailed rules and requirements for devices that are presented to software as PCIe devices. It covers the following areas:

- PCIe Configuration Space, including the standard PCIe Enhanced Configuration Access Mechanism (ECAM).
- PCIe Memory Space
- PCIe device view of memory
- PCIe Message Signaled Interrupts MSI(-X)
- GICv3 support for MSI(-X)

- Legacy interrupts
- System MMU and Device Assignment
- I/O Coherency, including memory type and attribute assignment
- Legacy I/O
- Integrated end points
- PCIe Peer-to-Peer traffic
- PCIe PASID support
- PCIe Precision Time Measurement

2.2 SBSA requirements summary

The SBSA specification uses the notion of levels of functionality. Each level adds functionality over and above previous levels. Unless explicitly stated, all specification items that belong to level N apply to levels greater than N. Currently, the SBSA specification defines Levels 3-7.

Table 2-1: SBSA Levels and requirements

Level	PE:A Profile	SMMU	GIC
3	v8.0	v2 or v3	v3.0
4	v8.3	v3.0	v3.0
5	v8.4	v3.2	v3.0
6	v8.5 or v9.0	v3.2	v3.0
7	v8.6 or v9.1	v3.2	v3.0

The SBBR and LBBR recipes in the Arm BBR specification describe firmware requirements for an Arm server system, and complement the SBSA hardware system architecture requirements. SBSA compliance is a requirement for servers in the SystemReady compliance.

Section 1.9 of the SBSA specification offers comprehensive checklists for minimum hardware requirements that are required to install, boot, and run a server operating system on bare-metal or within a virtualization environment. Each SBSA Level has a corresponding checklist.

Each SBSA Level describes the requirements for specific system components and features. The following is a summary of the requirements groups and examples of the requirements in each group. For a complete list of the requirements, please refer to the SBSA specification.

2.2.1 SBSA Level 3

SBSA Level 3 refers to many of the BSA requirements, and adds additional requirements that are specific for server systems in the following areas:

PE architecture

Support for 4KB and 64KB translation granules, 16-bit ASID, and all AArch64 Exception Levels.

Memory map

Requirements to enable Non-secure EL2 hypervisors, and peripherals assignments to virtual machines.

Interrupt controller

Requires for GICv3 or higher.

PPI assignments

Upgrading BSA recommendations to requirements.

SMMU

Conditional requirements for FEAT_TLBIRANGE.

Watchdog

Requires the Non-secure Generic Watchdog defined in Arm BSA specification.

2.2.2 SBSA Level 4

SBSA Level 4 adds additional server requirements in the following areas:

PE architecture

Armv8.2 RAS extensions, DC CVAP support, and FEAT_VIMD and FEAT_VHE support.

SMMU

SMMUv3 requirements and additional integration rules for stage 1 and 2 system MMU.

Peripheral subsystems

Requires that all peripherals that are intended for assignment to a virtual machine or a user space device driver are based on PCI Express.

2.2.3 SBSA Level 5

SBSA Level 5 adds additional server requirements in the following areas:

PE architecture

Requirements around FEAT_BBM, FEAT_S2FWB, FEAT_PAuth, CoreSight BSA, AMU, enhanced nested virtualization, and cryptography support for SHA3 and SHA512. There are also conditional requirements for MPAM extension.

Interrupt controller

Only GICv3 or higher is allowed to be exposed to the operating system.

SMMU

SMMUv3.2 requirements, including support for MPAM extension.

Clock and timer subsystem

Requires the generic counter to use 1GHz frequency.

Watchdog

Recommendations for using revision 1 of the generic watchdog.

PPI assignments

Additional PPI assignments reserved by SBSA.

2.2.4 SBSA Level 6

SBSA Level 6 adds additional server requirements in the following areas:

PE Architecture

Additional PE security and enhanced virtualization requirements.

SMMU

Additional requirements for coherent cache access, hardware translation table update (HTTU), and MSIs.

Watchdog

Requires revision 1 of the generic watchdog.

Armv8 RAS extension requirements

Requirements specific to the Armv8 RAS Extensions.

2.2.5 SBSA Level 7

SBSA Level 7 adds additional server requirements in the following areas:

PE Architecture

Requirements around grained tap support, enhanced counter virtualization, enhancements to AMU v1, Advanced SIMD Int8 matrix multiply, BFLOAT16, enhanced PAC2 and FPAC.

There are also conditional requirements for SVE Int8 matrix multiply, when SVE is implemented, as well as recommendations for WFE and enhanced PAN feature.

Some additional RAS PE requirements.

Conditional requirements for transactional memory extension (TME).

MPAM

Requirements for the PE MPAM support, including minimum number of partition IDs and monitors, among others.

Entropy Source

Requirements for the hardware entropy source.

SMMU

Additional SMMU requirements, including a requirement that all DMA capable requesters should be behind the SMMU, and SMMUs must implement SMMUv3 performance monitor extensions.

Performance Monitoring Unit

PMU requirements introduced for this level in Section B.

System RAS

System RAS requirements introduced for this level in Section C. There are also additional requirements for Scrubbing and Poison.

PCIe

Level 7 adds new PCIe Integration rules, as well as rules for iEP and PCIe error handling.

3. Design and integration guidance

This section of the guide examines some of the common mistakes Arm has seen with SoC implementations. The focus is on high-speed I/O peripherals like PCIe.

This section is organized by providing a description of the common mistake, followed by the impact seen on the overall system, and guidance on how an integrator could avoid these mistakes to build a system that is compliant with the Base System Architecture. References are provided to rules from the following system architecture documents:

- Arm Base System Architecture Specification
- Arm Server Base System Architecture Specification

The following are common mistakes seen when implementing and integrating high-speed peripherals:

- Issue 1 - Non-contiguous configuration space as the ECAM region for endpoints or switches under the Root Port hierarchy
- Issue 2 - Non-compliance with completion handling of configuration reads
- Issue 3 - PCIe Root Port configuration space supports only 32-bit R/W access
- Issue 4 - Root Port exposes ATS and PRI capabilities
- Issue 5 - Enumeration code reports valid entries for device 1 to device 31 on bus 0
- Issue 6 - ATC invalidate all command sent by SMMU does not automatically translate the address field
- Issue 7 - Alternate Routing ID (ARI) mode handling
- Issue 8 - Incorrect type (Type 0 or Type 1) conversion for downstream configuration requests
- Issue 9 - Support for message-based interrupts
- Issue 10 - Completion timeout response type
- Issue 11 - PCIe response of CpI-SC and not CplD-SC for bad ATS Translation Requests
- Issue 12 - PCIe Root Port supports only 32-bit aligned R/W access
- Issue 13 - Non-compliant UART

3.1 Issue 1 - Non-contiguous configuration space as the ECAM region for endpoints or switches under the Root Port hierarchy

With some PCIe controllers, the Root Port's configuration space is not in the same contiguous memory space as the ECAM region for endpoints and switches under the Root Port hierarchy.

Impact

Non-compliance means that standard enumeration software cannot run without modifications. Customizations such as this come with significant software cost in terms of development and maintenance.

Guidance

To ensure that the PCIe enumeration process works correctly, a Root Port's configuration space should be in the same ECAM region as the configuration space of the endpoints and switches in the hierarchy that originates at that Root Port. Arm also recommends third-party IP to implement the entire Root Port's PCIe hierarchy as accessible using the same Advanced Microcontroller Bus Architecture (AMBA) subordinate interface of the controller. The system architecture should allocate 256MB of contiguous ECAM memory for each Segment.

If a PCIe controller is implemented with an integrated PCIe RP and a Host Bridge, the expectation is that the Host Bridge implements a decoder that either routes the request to the local configuration space, or sends the configuration downstream to endpoints and switches as a configuration access. To convert an incoming AMBA Advanced eXtensible Interface (AXI) read or write request to a PCIe configuration read or write request, it is expected that the controller detects whether the access is to the ECAM space, and then maps the AXI request address bits to bus, device, function, and register numbers as described in the following table:

Table 3-1: Enhanced Configuration Address Mapping

AXI address	PCIe configuration space
AxADDR[63:28]	256MB aligned base address of ECAM space in the memory map. This part of the AXI address is checked with the value programmed in the ECAM Base Address register in the controller.
AxADDR[27:20]	Bus Number
AxADDR[19:15]	Device Number
AxADDR[14:12]	Function Number
AxADDR[11:2]	Register Number
AxADDR[1:0]	Used to generate Byte Enables, in conjunction with the size of the access.

The following table provides a summary of the ECAM region decoding and response handling requirements. It is assumed that these requirements are met by the controller and no additional logic is required externally in the SoC.

Table 3-2: ECAM Region Decoding Requirements

Bus range	Bus range usage	Description
Bus number > Subordinate bus number	Unimplemented on-Chip function.	All 1's must be returned for PE read requests to this region.
Bus number > Secondary bus number and <= Subordinate bus number	Downstream device/Switch function	If unpopulated, an Unsupported Request (UR) response would be returned for configuration read requests. The Root Port (RP) must convert the UR response to an all 1s response back to the requestor PE. Similarly, the RP must return an all 1s response back to the requestor PE for Completer Abort (CA) responses or if the request times out and a Complete Time Out (CTO) occurs. The RP must return an all 1s response back to the requestor PE for Configuration Request Retry Status (CRS) responses if the target register was not the Vendor ID register or if CRS software visibility is not enabled. Note: this forwarding must not consider device number because Alternate Routing ID (ARI) being enabled or disabled is not relevant for this bus range.
Bus Number == Secondary bus number	Downstream device/Switch function	The Root Port must convert Type 1 to Type 0 for Secondary bus number. If ARI mode is not enabled and device number > 0, then an all 1s response must be returned to the requestor PE. If ARI mode is enabled, then it is not necessary to consider the device number while forwarding downstream. If unpopulated, a UR response would be returned for configuration read requests. The RP must convert the UR response to an all 1s response back to the requestor PE. RP must return an all 1s response back to the requestor PE for CA responses or if the request times out and a CTO occurs. The RP must return an all 1s response back to the requestor PE for CRS responses if the target register was not the Vendor ID register or if CRS software visibility is not enabled. Note: If Secondary bus number is != Root Bus number +1 , then all configuration reads that target a Bus number between Secondary Bus Number and Root Bus number must return all 1s to the requestor PE.
Bus Number = Root Bus	Unused	This is the rest of the 1MB configuration region for Bus number = Root Bus. All 1s must be returned for PE read requests to this region.
Bus Number = Root Bus	Root Port Configuration space (4K)	This is the Root Port's 4K configuration space. The Root Port must sink all accesses to this 4K region correctly.

Relevant BSA/SBSA Rules

The following BSA/SBSA rules are relevant to this issue:

- $R_{PCI\_IN\_01}$
- $R_{PCI\_IN\_03}$
- $R_{PCI\_IN\_04}$
- $R_{PCI\_IN\_06}$

3.2 Issue 2 - Non-compliance with completion handling of configuration reads

Some PCIe controllers are non-compliant with completion handling of configuration reads. To ensure that the PCIe enumeration process works correctly, a combination of PCIe Host Bridge

(PHB) and PCIe Root Port must follow the completion handling rules as described in the PCIe specification.

Impact

Non-compliance means that standard enumeration software cannot run without modifications. Customizations such as this come with significant software cost in terms of development and maintenance.

Guidance

If a PCIe controller is implemented with an integrated PCIe RP and Host Bridge, the Host Bridge is expected to return a read-data value of all 1s when a Configuration Read Request is terminated as an Unsupported Request (UR) or Completer Abort (CA).

Relevant BSA/SBSA Rules

The following BSA/SBSA rules are relevant to this issue:

- $R_{PCI\_IN\_07}$
- $R_{PCI\_IN\_08}$
- $R_{PCI\_IN\_09}$
- $R_{PCI\_IN\_14}$
- $R_{PCI\_IN\_15}$
- $R_{PCI\_IN\_16}$

3.3 Issue 3 - PCIe Root Port configuration space supports only 32-bit R/W access

Some PCIe Root Port implementations only support 4-byte (32-bit) access for configuration reads and writes.

Impact

Standard UEFI and Linux drivers may perform 8-bit and 16-bit reads and writes to the configuration space, giving an erroneous result. Non-compliance means that standard enumeration software cannot run without modifications. Customizations such as this come with significant software cost in terms of development and maintenance.

Guidance

1-byte, 2-byte, and 4-byte configuration reads and writes must be supported and forwarded properly by the root complex as required by the PCIe specification. This must work for both Root Port configuration registers and configuration registers in the downstream hierarchy. If a PCIe controller is implemented with an integrated PCIe RP and a Host Bridge, the expectation is that when the Host Bridge receives a Type 1 configuration request on the AXI subordinate interface, it directs the transaction to the Root Port configuration space or forwards it downstream with the Byte Enables.

Relevant BSA/SBSA Rules

The following BSA/SBSA rules are relevant to this issue:

- $R_{PCI\_IN\_18}$

3.4 Issue 4 - Root Port exposes ATS and PRI capabilities

Some implementations expose Address Translation Service (ATS) and Page Request Interface (PRI) capabilities when configured in Root Port mode.

Impact

If system software and hardware incorrectly identifies the Root Port with ATS/PRI capabilities, it issues table maintenance operations like ATS invalidate that would require the Root Port to respond back with the completion indication. Software will need to make custom changes to workaround this issue, with a cost in terms of development and maintenance.

Guidance

Arm recommends that ATS and PRI capabilities are disabled when the controller is configured in Root Port mode. ATS capability should only be enabled only when endpoint mode is selected, and the controller has a software visible cache for address translations. PRI capability should only be enabled when endpoint mode is selected, and the controller requires memory pages dynamically.

Relevant BSA/SBSA Rules

The following BSA/SBSA rules are relevant to this issue:

- $R_{RE\_SMU\_2}$
- $R_{RE\_SMU\_3}$
- $R_{IE\_SMU\_1}$
- $R_{IE\_SMU\_2}$

3.5 Issue 5 - Enumeration code reports valid entries for device 1 to device 31 on bus 0

In some implementations, while enumeration code is reading the configuration space for the Root Port, the Root Port reports valid entries for non-existent devices (from device 1 to device 31) on bus 0.

Impact

For the example given in the description above, enumeration software needs to have a custom workaround built where it is restricted from reading device 1 to device 31 on bus 0. Any standard enumeration software might see 32 Root Ports instead of 1 Root Port. Customizations such as this come with significant software cost in terms of development and maintenance.

Guidance

Similar to the guidance provided for Issue 1, Arm recommends that PCIe controllers report valid entries only for devices that exist. Configuration access to non-existent devices on bus 0 should be returned with read data of all 1s.

Relevant BSA/SBSA Rules

The following BSA/SBSA rules are relevant to this issue:

- $R_{PCI\_IN\_07}$

3.6 Issue 6 - ATC invalidate all command sent by SMMU does not automatically translate the address field

Some implementations of the PCIe controller with Address Translation Services (ATS) capabilities do not detect the special case to invalidate all translations in the Address Translation Cache (ATC). When software needs to invalidate all translations in the ATC, it sends the `CMD_ATC_INV` command to the SMMU with the size field set to 52. The DTI-ATS manager is expected to detect this special case and fill the address field with the PCI ATS specification-compliant value of `0x7FFFFFFFFFFFF000`.

Impact

Because some PCIe controllers with DTI-ATS manager capabilities do not detect this special case, the impact is that the system software's request to invalidate all translations in the ATC fails. This would require a custom workaround. Customizations such as this come with significant software cost in terms of development and maintenance.

Guidance

Arm recommends that PCIe controllers with DTI-ATS capabilities should detect this special case, and translate the address field in the request appropriately.

Relevant BSA/SBSA Rules

The following BSA/SBSA rules are relevant to this issue:

- (not a requirement from BSA or SBSA)

3.7 Issue 7 - Alternate Routing ID (ARI) mode handling

In some implementations, when ARI forwarding is disabled, the Root Port forwards configuration accesses with a nonzero device number to its secondary or subordinate buses. The PCI Express specification requires that if ARI forwarding is disabled, the RP must not forward any configuration access targeting its secondary bus if the request has a nonzero device number.

Impact

If ARI forwarding is disabled, and the Root Port forwards the configuration access downstream for a nonzero device number, it is possible that a non-ARI device can respond to the configuration space access under what it interprets as different device numbers, and its functions can be aliased under multiple device numbers, leading to undesired behavior.

Guidance

If ARI forwarding is disabled and the target device number is a nonzero value, then the access should be terminated, and a read-data value of all 1s should be returned to the system software. Conversely when ARI forwarding is enabled, the Root Port must forward requests targeting its secondary bus regardless of the device number. This guidance works in conjunction with the guidance provided for Issue 1.

Relevant BSA/SBSA Rules

The following BSA/SBSA rules are relevant to this issue:

- $R_{PCI\_IN\_17}$

3.8 Issue 8 - Incorrect type (Type 0 or Type 1) conversion for downstream configuration requests

In some implementations, the PCIe controller expects the external Host Bridge to convert the configuration access to have the correct type (Type 0 or Type 1) depending on the intended target and the topology.

Impact

Non-compliance means that standard enumeration software cannot run without modifications. Customizations such as this come with significant software cost in terms of development and maintenance.

Guidance

The PCIe root complex is required to send a configuration access with the correct type (Type 0 or Type 1) depending on the intended target. Type 0 is for a secondary bus, and Type 1 is for a subordinate bus range of the Root Port). Arm recommends that PCIe controllers should implement the conversion in the PCIe Host Bridge (PHB) in conjunction with the Root Port (RP). This guidance works in conjunction with the guidance provided for Issue 1.

Relevant BSA/SBSA Rules

The following BSA/SBSA rules are relevant to this issue:

- $R_{PCI\_IN\_10}$
- $R_{PCI\_IN\_11}$

3.9 Issue 9 - Support for message-based interrupts

Some Root Port implementations do not support message-based interrupts (MSIs) for local events and errors. These are reported as wire-based interrupts.

Impact

Typically there is a limit on the number of wire-based interrupts, such as SPIs, that are supported. Supporting wire-based interrupts comes with both an area cost and a routing cost.

Guidance

If a PCIe controller is integrated with Host Bridge, Arm recommends that that Host Bridge converts local interrupts as MSIs. If the Host Bridge uses an AXI interface, Arm recommendeds that a reserved AXI manager ID is used for these MSI requests. Other AXI memory writes should not be blocked by the MSI writes on the AXI interface.

Relevant BSA/SBSA Rules

The following BSA/SBSA rules are relevant to this issue:

- $R_{PCI\_MSI\_1}$
- $R_{PCI\_MSI\_2}$
- $R_{PCI\_ER\_02}$
- $R_{PCI\_ER\_03}$

3.10 Issue 10 - Completion timeout response type

Some Root Port implementations return a `SLVERR` response on the AXI manager interface when they receive a completion timeout (CTO).

Impact

The `SLVERR` response could cause the system software or kernel to abort.

Guidance

Arm recommends that PCIe controller implementations provide an option to downgrade the error condition by sending a read-data value of all 1s to the requester for an MMIO read that suffers a completion timeout (CTO).

Relevant BSA/SBSA Rules

There are no relevant BSA/SBSA rules for this issue.

3.11 Issue 11 - PCIe response of Cpl-SC and not CplD-SC for bad ATS Translation Requests

Some implementations of the PCIe controller return a Cpl-SC (Completion without data, success) to the endpoint on a bad address translation request. When an endpoint sends a translation request to the RP, this request is forwarded to the SMMU over the ATS interface. The SMMU returns a `DTI_ATS_TRANS_FAULT` response on a bad address translation request, which should be forwarded as a CplD-SC (Completion with Data and Success) with the data R,W bits set to zero, indicating an invalid translation to the endpoint that initiated the address translation.

Impact

Receiving an unexpected response could lead to the endpoint hanging. This would require customizations in software to resolve the hanging condition.

Guidance

Arm recommends that the DTI block implemented within the PCIe controller, on receiving a `DTI_ATS_TRANS_FAULT` response from the SMMU, returns a CplD-SC (Completion with data and success) to the endpoint. The DTI block should set the data R,W bits to zero indicating an invalid address translation request.

Relevant BSA/SBSA Rules

There are no relevant BSA/SBSA rules for this issue.

3.12 Issue 12 - PCIe Root Port supports only 32-bit aligned R/W access

Some implementations of the PCIe controller are not able to properly translate unaligned CPU requests or combinations of byte enables on the Host AXI interface to appropriate TLP level address and byte enables.

Impact

Standard UEFI and Linux drivers may perform unaligned accesses or 8-bit or 16-bit reads and writes, giving an erroneous result. This may require complicated workarounds in the software that could come at a significant development and maintenance cost.

Guidance

Similar to the guidance in issue 3, Arm recommends that if a PCIe controller is implemented with an integrated RP and Host Bridge, the Host Bridge takes these unaligned accesses with the byte enables, and converts them to an appropriate TLP with the appropriate request address and byte enables.

Relevant BSA/SBSA Rules

The following BSA/SBSA rules are relevant to this issue:

- $R_{PCI\_IN\_19}$
- $R_{S\_PCIe\_01}$
- $R_{S\_PCIe\_02}$
- $R_{S\_PCIe\_03}$
- $R_{S\_PCIe\_04}$

3.13 Issue 13 - Non-compliant UART

Some implementations of the SoC do not feature a 16550 fully-compliant UART.

Impact

This leads to issues in bring-up for the OS console and debug, and may require customizations in the software implementation.

Guidance

Please confirm with the IP vendor whether the UART is 16550 fully-compliant UART. You can also refer to the BSA specification for reference.

Relevant BSA/SBSA Rules

The following BSA/SBSA rules are relevant to this issue:

- $R_{B\_PER\_05}$

4. Pre-silicon BSA compliance testing

This chapter examines pre-silicon BSA compliance testing, its benefits and usage. The figure below highlights the various stages in the journey to BSA/SBSA compliance and pre-silicon testing.

Figure 4-1: BSA/SBSA compliance journey

The pre-silicon BSA compliance tests are required to be run on the RTL during the SoC design verification stage.

4.1 Why is pre-silicon BSA compliance testing needed?

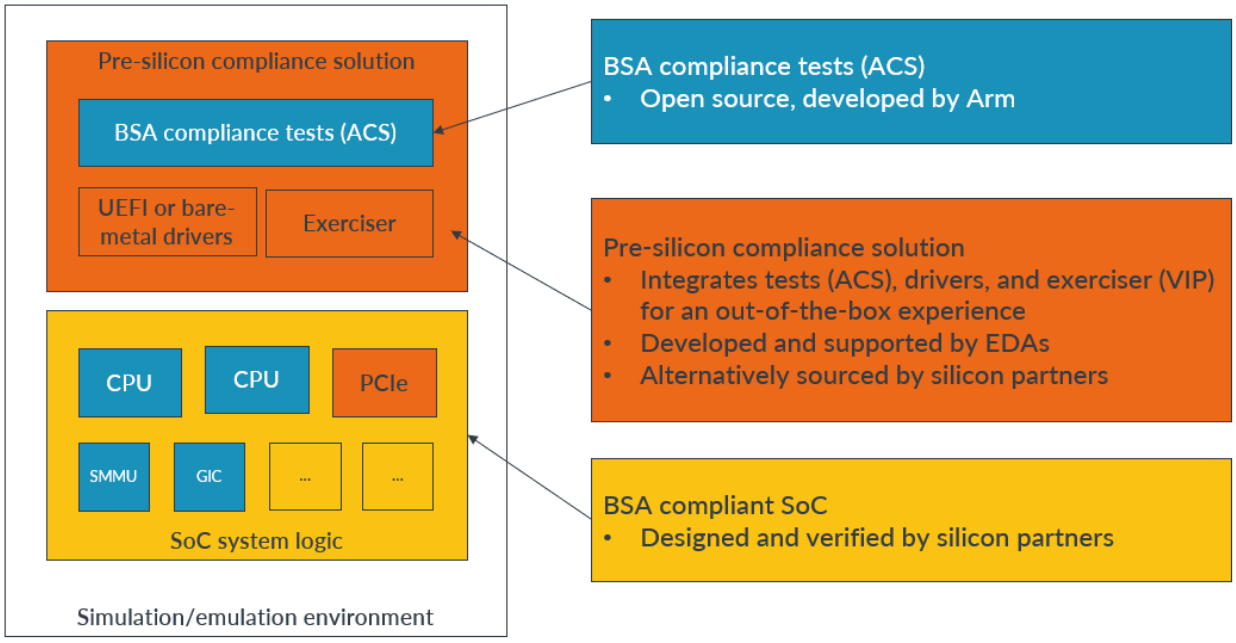
The BSA specification and its supplements such as SBSA define rules that need to be implemented and followed during the design and integration of a SoC that is to be compliant. After integrating the system according to the architecture specification and before tape-out, silicon designers need to run the BSA compliance tests to ensure the hardware is compliant with the specification.

The compliance tests confirm the design matches the architecture intent and checks that the architecture rules are understood and not missed. The focus is on system-level and software-visible architectural behavior, and not on microarchitecture implementation and correctness. Therefore, the compliance tests are not a substitute for design verification.

4.2 Overview of the pre-silicon BSA compliance solution

The following figure provides an overview of the pre-silicon compliance solution.

Figure 4-2: The pre-silicon BSA compliance solution overview



Testing for pre-silicon BSA compliance involves running the tests from the BSA/SBSA Architecture Compliance Suites (ACS) on a pre-silicon simulation or emulation environment. Each ACS is open source and developed by Arm.

Silicon designers can run each ACS on top of UEFI or directly on bare-metal. By running on UEFI, silicon designers can leverage existing firmware development work for faster porting and less integration effort. Alternatively, running on bare-metal means that tests run earlier in the design cycle, with shorter simulation and emulation cycles, and faster root cause analysis.

Exerciser (also known as a Transactor, Verification IP, or VIP) is needed to achieve complete BSA coverage. Exerciser generates custom stimuli, making the pre-silicon tests a super-set of the on-silicon tests. Exerciser gives greater controllability and observability, and helps to identify deeper integration issues, particularly for PCIe.

Arm collaborates with EDA partners to provide silicon designers with a complete solution for pre-silicon BSA compliance testing. These solutions integrate the ACS tests, exerciser, and bare-metal drivers for their pre-silicon simulation or emulation environments for an out-of-the-box experience. The following section provides more details on the ACS, exerciser, and its usage.

4.3 What is ACS?

The tests within each Arm Compliance Suite (ACS) are example sof the invariant behaviors defined by a system specification. Each Suite can be used to validate that these behaviors are implemented correctly.

4.4 What are BSA/SBSA compliance tests?

BSA/SBSA compliance tests are self-checking, portable C-based tests with directed stimulus that are packaged as BSA-ACS and SBSA-ACS respectively. The BSA/SBSA compliance tests confirm the design matches the BSA/SBSA specification and that no rules have been missed.

The following table describes the BSA/SBSA compliance test components.

Table 4-1: Compliance Test Components

Component	Description
PE	Verifies PE compliance.
GIC	Verifies GIC compliance.
Timer	Verifies PE timer and system timer compliance.
Watchdog	Verifies watchdog timer compliance.
PCIe	Verifies PCIe sub-system compliance.
Peripherals	Verifies USB, SATA, and UART compliance.
Power states	Verifies system power states compliance.
SMMU	Verifies SMMU sub-system compliance.
Exerciser	Verifies PCIe sub-system with a custom stimulus generator.
NIST	Verifies the suitability of a generator for a cryptographic application

4.5 Why is exerciser needed?

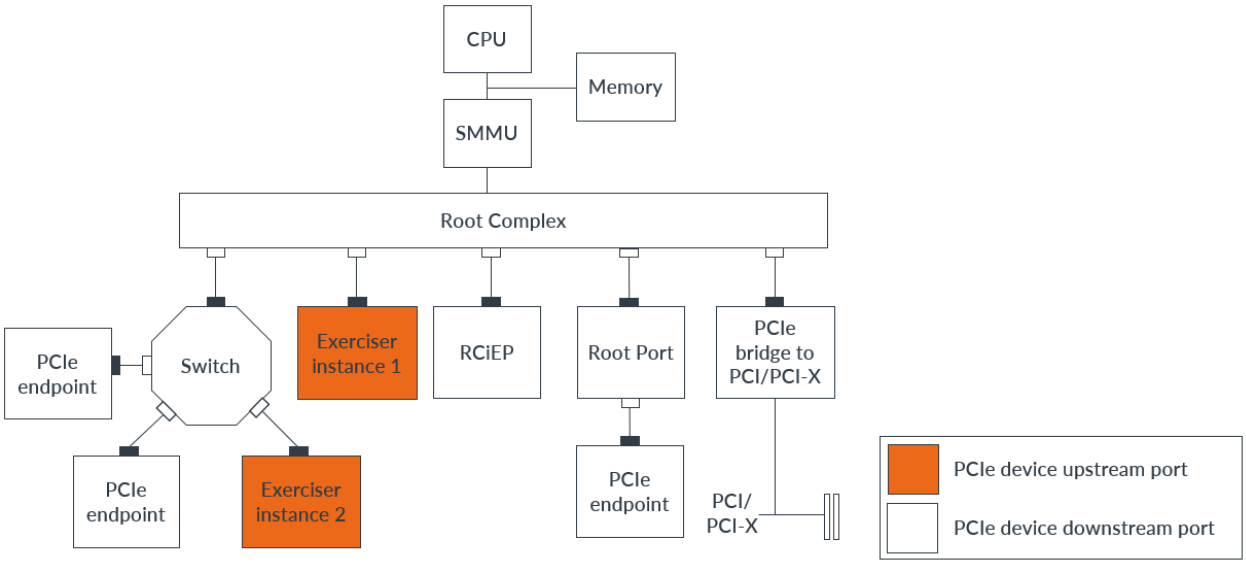
Validating the compliance of certain PCIe rules defined in the BSA/SBSA specifications requires the PCIe endpoint to generate specific stimuli during the runtime of the test. Examples of such stimuli are P2P, PASID, and ATC. The tests that require these stimuli are grouped together in the exerciser module. The exerciser layer is an abstraction layer that enables the integration of hardware capable of generating such stimuli to the test framework.

4.6 What is exerciser?

Exerciser is a PCIe endpoint device that can be programmed to generate custom stimuli for verifying the BSA/SBSA compliance of PCIe IP integration into an Arm SoC. The stimuli are used to verify the compliance of PCIe functionality including I/O coherency, snoop behavior, address translation, PASID transactions, DMA transactions, MSI, and legacy interrupt behavior.

The following diagram shows a PCIe hierarchy consisting of various endpoints, switches, and bridges:

Figure 4-3: Block diagram showing PCIe integration in an Arm SoC



This block diagram shows two instances of exerciser that are present in the system:

- Exerciser instance 1 is connected directly to the Root Complex as an RCiEP
- Exerciser instance 2 is connected to the downstream port of a switch as a PCIe endpoint device

Root Complex integrated EndPoint (RCiEP) and Root Complex Event Collector (RCEC) are endpoints connected directly to Root Complex.

PCIe endpoints are connected either to the Root Port or to downstream ports.

Bridges connect PCI devices into the PCIe hierarchy while switches connect multiple PCIe devices to a single downstream port.

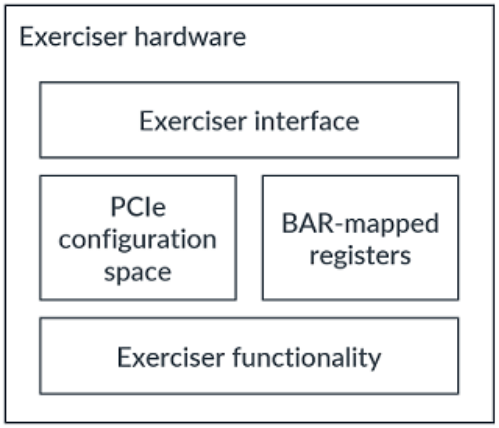
PCIe devices access the GIC, memory, and the PE through the Root Complex, also called the Host Bridge.

Note

The number of exercisers instantiated is platform-specific. To achieve higher coverage, Arm recommends that multiple exercisers are presented to the ACS. To generate custom stimuli, exerciser must provide functionality to configure interrupt and DMA attributes, trigger them, and know the status of these operations, the details of which are **IMPLEMENTATION DEFINED**. This can be done by providing a set of BAR-mapped registers and writing specific values to trigger the necessary operations.

The following diagram shows the reference implementation of exerciser hardware:

Figure 4-4: Exerciser hardware



4.7 Compliance test software stack for exerciser with a UEFI shell application

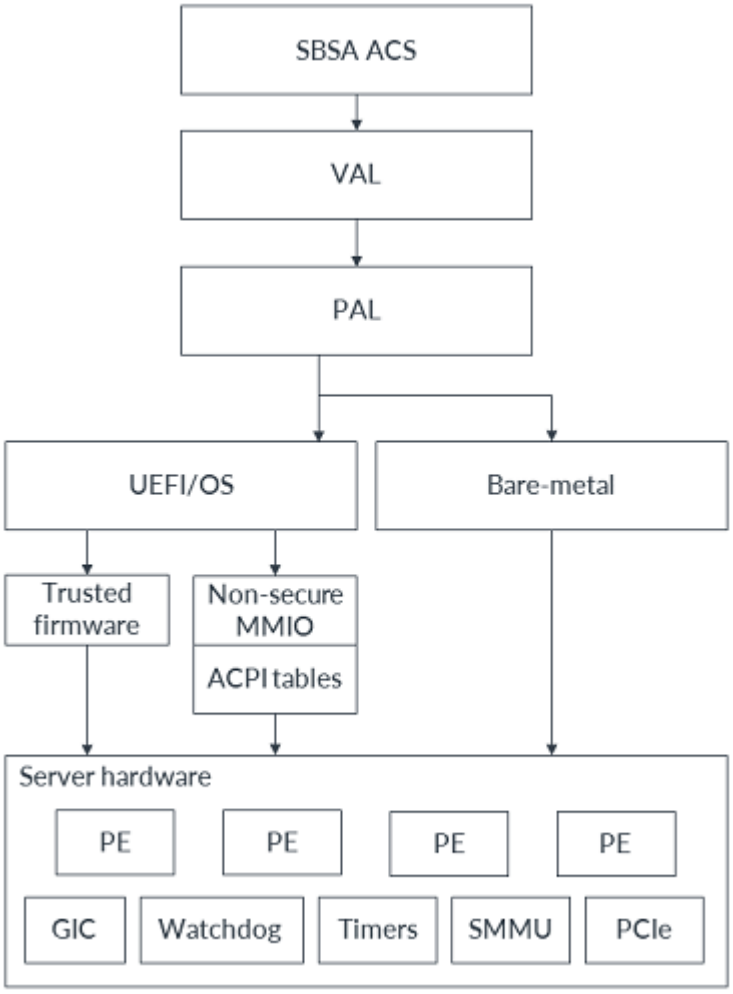
Exerciser tests validate device interrupts (legacy interrupt and MSI-X interrupt), DMA (address translation and memory access), and coherency behavior. Exerciser PCIe configuration space is accessed using UEFI or MMIO APIs and exerciser functionality like interrupt generation and DMA transactions can be accessed using exerciser APIs.

The following diagram shows the compliance test software stack for exerciser with UEFI shell application.

Figure 4-5: Software stack for exerciser with a UEFI shell application

The following diagram shows the BSA/SBSA test abstraction layer.

Figure 4-6: BSA/SBSA test abstraction layer



4.8 How to access pre-silicon BSA compliance tests

The SBSA Architecture Compliance Suite (ACS) is a super-set of pre-silicon BSA compliance tests, an open source collection of self-checking, portable C-based tests developed by Arm. It is available for download from the following Github repository:

- SBSA Architecture Compliance Suite

A pre-silicon PAL for RD N1 reference implementation is available with the SBSA ACS:

- Pre-silicon PAL for RD N1 reference implementation

The BSA ACS is platform-agnostic, and is portable to different verification, simulation, and emulation environments. However, Arm recommends that silicon vendors source the pre-silicon BSA compliance solution from the EDA vendor as it is already integrated and ported by the EDA partner in collaboration with Arm.

5. Related information

The following resources are related to material in this guide.

Specifications:

- Arm Base System Architecture Specification (BSA)
- Arm Server Base System Architecture (SBSA) Specification
- Arm SystemReady Requirements Specification

Repositories:

- BSA ACS Repository
- SBSA ACS Repository

User Guides:

- SystemReady FAQ

SystemReady Pages:

- Arm SystemReady Pre-Silicon Page
- Arm SystemReady Compliance Program
- Arm Community - SystemReady Forum

Other Resources:

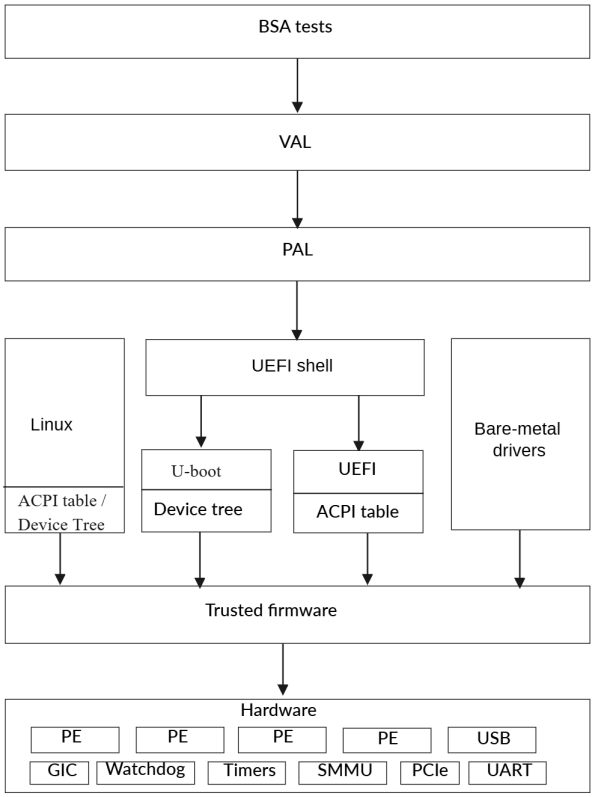
- PCI Express Base Specification Revision 5.0, Version 1.0. PCI-SIG

Appendix A Running BSA ACS tests on the Arm Neoverse N2 reference design (RD-N2) Fixed Virtual Platform (FVP) model

This section provides a step-by-step example of how to run the BSA-ACS on bare metal and UEFI against a Fixed Virtual Platform (FVP) model.

The Arm Neoverse N2 reference design (RD-N2) Fixed Virtual Platform (FVP) models much of the Arm IP. The FVP enables partners to develop ahead of hardware availability and to explore the design from a software perspective.

Figure A-1: Test platform abstraction



For pre-silicon hardware design, we anticipate these tests would be run on RTL, either through simulation, emulation, or FPGA implementation.

The example steps provided in this section allows for early familiarization with the software stack and how the ACS is executed prior to any RTL test runs. It outlines every step, from setting up the

RDN2 to executing the ACS and gathering essential logs. Additionally, the example highlights the integration of the Exerciser, including how the PAL APIs are utilized, providing a comprehensive understanding of the software flow and interaction with hardware components.

For more detailed guidance please refer to the PCIe Exerciser bare-metal readme, which serves as a comprehensive guide. This readme directs us to download the pre-built model for the RD-N2 FVP and construct the software stack using source code. It also highlights how to activate the exerciser for comprehensive testing.

For this example we use the RD-N2 Cfg3 Platform.

Using the FVP model allows for early software development and testing, enabling teams to catch potential issues and refine the software stack even before physical silicon becomes available.

While the example focuses on BSA ACS, the same approach, logic, steps, and principles are equally applicable to running the SBSA ACS, ensuring a consistent methodology for compliance testing and validation across different standards.

A.1 Setting up the RD-N2 FVP

This section outlines how to set up the RD-N2 FVP.

A.1.1 Setting up the Workspace and Software Stack

Use the repo sync command to download and sync the required reference design.

1. Create a folder and change directory to it, e.g.

```
mkdir <workspace>

cd <workspace>
```

2. Initialize and sync the source code, specifying the release tag

```
repo init -u https://git.gitlab.arm.com/infra-solutions/reference-design/infra-refdesign-manifests.git -m <manifest-file-name> -b refs/tags/<RELEASE_TAG> --depth=1

repo sync -c -j $(nproc) --fetch-submodules --force-sync --no-clone-bundle
```

If you encounter network errors or timeouts, retry repo sync with a smaller -j value (e.g., -j 4).

For detailed current guidance please refer to the Getting Started Guide for the Neoverse software stack.

For more information about RD-N2 FVP getting started, please refer to the Get started with the Neoverse Reference Design software stack learning path.

A.1.2 Setup the Host Based Build Environment

Set up the build environment before building the software stack by executing the script that installs all the build dependencies.

```
sudo ./build-scripts/rdinfra/install_prerequisites.sh
```

A.1.3 Download the Reference Design Model

1. Download the required version of RD-N2 FVP from Arm Ecosystem FVPs and set the MODEL PATH. This involves selecting the Neoverse Infrastructure section and acquiring the Linux package for the Neoverse N2 FVP.

```
wget https://developer.arm.com/-/media/Arm%20Developer%20Community/Downloads/OSS/
FVP/Neoverse-N2/FVP_RD_N2_11.25_23_Linux64.tgz
```

If the toolchain download fails, verify internet connection or try the latest version from the Arm Developer page.

2. Upon downloading the model package, we extract it and execute the installation script, making the RDN2 model readily available for utilization.

```
export MODEL=model_path_to_Linux64_GCC-9.3/FVP_RD_N2
```

The MODEL environment variable specifies the path to the downloaded FVP model binary, enabling other scripts and commands to locate and use this model during setup and testing.

Returning to software stack installation, set the environment variable MODEL to point to the model binary.

A.1.4 Modify and Build the Software Stack

Subsequent adjustments might be necessary to enhance certain features, such as increasing the max interrupt count. Finally, we initiate the platform build using the build command.

1. Change directory

```
cd ~/<workspace>/rd-infra
```

2. Modify the following in

```
uefi/edk2/ArmPkg/Drivers/ArmGic/GicV3/ArmGicV3Dxe.c
   - mGicNumInterrupts = ArmGicGetMaxNumInterrupts (mGicDistributorBase);
   + mGicNumInterrupts = 16384;
```

3. Build the software stack

```
./build-scripts/rdinfra/build-test-acs.sh -p rdn2 all
```

A.1.5 Setting the PCIe Hierarchy

RD-N2 offers the flexibility to customize the PCIe hierarchy to accommodate various device and port configurations. A pre-customized file is available in the ACS GitHub, simplifying the integration process. Copy the downloaded hierarchy JSON file to the RDN2 software stack and modify the run_model.sh file accordingly to incorporate our preferred PCIe hierarchy.

1. Download the pre-customized PCIe hierarchy file

```
wget https://github.com/ARM-software/bsa-acs/blob/main/docs/PCIe_Exerciser/
example_pcie_hierarchy_0.json
```

2. Copy the PCIe hierarchy file to the model script folder of RD N2 software stack

```
cp example_pcie_hierarchy_0.json ~/<workspace>/rd-infra/model-scripts/rdinfra/
platforms/rdn2/
```

3. To run the BSA exerciser tests make the following change in `~/<workspace>/rd-infra /model-scripts/rdinfra/platforms/rdn2/run_model.sh`

```
- -C pcie_group_0.pciex16.hierarchy_file_name=<default> \
- -C pcie_group_1.pciex16.hierarchy_file_name="example_pcie_hierarchy_2.json" \
- -C pcie_group_2.pciex16.hierarchy_file_name="example_pcie_hierarchy_3.json" \
- -C pcie_group_3.pciex16.hierarchy_file_name="example_pcie_hierarchy_4.json" \
+ -C pcie_group_0.pciex16.hierarchy_file_name="example_pcie_hierarchy_0.json" \
```

A.2 Baremetal BSA-ACS Run

This section demonstrates how to run the Baremetal ACS on RD-N2.

A.2.1 Compile and build the BSA-ACS

First clone the ACS repo. Ensure all the tools have the required version to support the ACS. If not, download the required version of the toolchain.

1. Download the BSA-ACS

```
cd ~/<workspace>
```

```
mkdir bm_build

cd bm_build

git clone https://github.com/ARM-software/bsa-acs.git
```

2. Download the Arm GNU Toolchain - AArch64 bare-metal target (aarch64-none-elf)

```
cd ~/<workspace>

wget https://developer.arm.com/-/media/Files/downloads/gnu/13.2.rel1/binrel/arm-gnu-toolchain-13.2.rel1-x86_64-aarch64-none-elf.tar.xz

tar -xf arm-gnu-toolchain-13.2.rel1-x86_64-aarch64-none-elf.tar.xz
```

3. Once the toolchain is setup, set the environment variable CROSS_COMPILE to point to the toolchain

```
cd bm_build/bsa-acs/

export CROSS_COMPILE=~/<workspace>/arm-gnu-toolchain-13.2.Rel1-x86_64-aarch64-none-elf/bin aarch64-none-elf-
```

4. The next step is to create build directory and run the build command.

```
mkdir build

cd build/

cmake ../ -G"Unix Makefiles" -DCROSS_COMPILE=$CROSS_COMPILE -DTARGET="RDN2”

make
```

On a successful build, the ACS `*.bin`, `*.elf`, .img and debug binaries are generated at build/output directory

A.2.2 Run the BSA-ACS

By default, the RD-N2 software stack boots to the UEFI shell. To transition to Baremetal ACS, modifications in the software stack are essential. This involves redirecting the system to execute the ACS binary post-basic system initialization.

1. Repackage the FIP image with `bsa.bin`

```
cp output/bsa.bin ~/<workspace>/rd-infra/output/rdn2/components/css-common/acs_latest.bin
cd ~/<workspace>/rd-infra/
```

2. In `build-scripts/build-target-bins.sh`, replace `uefi.bin` with `acs_latest.bin`:

```
    if [ "${!tfa_tbbr_enabled}" == "1" ]; then
```

```
          $TOP_DIR/$TF_A_PATH/tools/cert_create/cert_create  \
          ${cert_tool_param} \
  -       ${bl33_param_id} ${OUTDIR}/${!uefi_out}/uefi.bin
  +       ${bl33_param_id} ${OUTDIR}/${!uefi_out}/acs_latest.bin
  fi
  ${fip_tool} update \
  ${fip_param} \
- ${bl33_param_id} ${OUTDIR}/${!uefi_out}/uefi.bin \
+ ${bl33_param_id} ${OUTDIR}/${!uefi_out}/acs_latest.bin \
  ${PLATDIR}/${!target_name}/fip-uefi.bin
```

Once that is done, it is time to repackage the software stack.

3. Repackage the FIP image with `bsa.bin`

```
./build-scripts/rdinfra/build-test-acs.sh -p rdn2 package
```

The next step to run the model which by default would start to execute the ACS tests.

4. Run the BSA ACS

```
cd model-scripts/rdinfra/platforms/rdn2

export MODEL=~/<workspace>/FVP_RD_N2/models/Linux64_GCC-9.3/FVP_RD_N2

./run_model.sh
```

Once the run has completed, the logs can be found in `~/<workspace>/rd-infra/model-scripts/rdinfra/platforms/rdn2/rdn2/`.

A.3 UEFI BSA-ACS RUN

The previous section walked through how to run the Baremetal ACS on RD-N2. The other option is to run as an app on UEFI shell, which is summarized next.

A.3.1 EDK2 Setup and BSA-ACS Compilation

Download the edk2 from the mentioned stable tag and the library required for `edk2`.

1. Download EDK2 and EDK2 port of `libc`

```
cd ~/<workspace>

mkdir uefi_build

cd uefi_build

git clone –recursive –branch edk2-stable202402 https://github.com/tianocore/edk2.git

git clone https://github.com/tianocore/edk2-libc.git edk2/edk2-libc
```

Ensure that you have the required tool chain. This step can be skipped if you have already installed the toolchain.

2. Download the Arm GNU Toolchain - AArch64 GNU/Linux target (aarch64-none-linux-gnu)

```
cd ~/<workspace>

wget https://developer.arm.com/-/media/Files/downloads/gnu/13.2.rel1/binrel/arm-gnu-toolchain-13.2.rel1-x86_64-aarch64-none-linux-gnu.tar.xz

tar -xf arm-gnu-toolchain-13.2.rel1-x86_64-aarch64-none-linux-gnu.tar.xz
```

3. Download the ACS from GitHub

```
cd uefi_build/edk2

git submodule update --init --recursive

git clone https://github.com/ARM-software/bsa-acs.git ShellPkg/Application/bsa-acs
```

Make the changes to add the ACS to build it as an application which then can be run on UEFI shell.

4. Add the following to the [LibraryClasses.common] section in `ShellPkg/ShellPkg.dsc`

```
BsaValLib|ShellPkg/Application/bsa-acs/val/BsaValLib.inf

BsaPalLib|ShellPkg/Application/bsa-acs/pal/uefi_acpi/BsaPalLib.inf
```

5. Add the following in the [Components] section of `ShellPkg/ShellPkg.dsc`

```
ShellPkg/Application/bsa-acs/uefi_app/BsaAcs.inf
```

6. Add the GCC49_AARCH64_PREFIX variable to point to the GCC toolchain.

```
export GCC49_AARCH64_PREFIX=~/<workspace>/arm-gnu-toolchain-13.2.

Rel1-x86_64-aarch64-none-linux-gnu/bin/aarch64-none-linux-gnu-
```

7. Update the PACKAGE_PATH variable to point to the edk2-libc library

```
export PACKAGES_PATH=~/<workspace>/uefi_build/edk2/edk2-libc/
```

8. Then compile the edk2 and build the ACS.

```
source edksetup.sh

make -C BaseTools/Source/C

source ShellPkg/Application/bsa-acs/tools/scripts/acsbuild.sh
```

The EFI executable file is generated at `Build/Shell/DEBUG_GCC49/AARCH64/Bsa.efi`

9. Create an image file which contains the `'Bsa.efi'` file

```
cd ~/<workspace>/uefi_build
mkfs.vfat -C -n HD0 hda.img 2097152
sudo mkdir /mnt/bsa
sudo mount hda.img /mnt/bsa
```

10. Once the image is created, copy the BSA.efi binary to the image

```
sudo cp edk2/Build/Shell/DEBUG_GCC49/AARCH64/Bsa.efi /mnt/bsa/
sudo umount /mnt/bsa
```

A.3.2 Run the BSA-ACS

The next step is to run the model with the image.

1. Run the BSA-ACS

```
cd ~/<workspace>/rd-infra/
Recover build-scripts/build-target-bins.sh - replace acs_latest.bin with uefi.bin
./build-scripts/rdinfra/build-test-acs.sh -p rdn2 package
export MODEL=~/<workspace>/FVP_RD_N2/models/Linux64_GCC-9.3/FVP_RD_N2
cd model-scripts/rdinfra/platforms/rdn2
./run_model.sh -v ~/<workspace>/uefi_build/hda.img
```

2. Once the model boots up, stop at the UEFI shell, go to the directory containing the ACS and the run the ACS.

- Select ‘Boot Manager’
- Select ’UEFI Shell’
- In the UEFI shell, input ’FS2:’
- Input ‘Bsa.efi –f acs.txt’

3. Get the ACS result.

```
sudo mount ~/<workspace>/uefi_build/hda.img /mnt/bsa
```

- The result is allocated at /mnt/bsa/acs.txt

To achieve SBSA compliance, it is necessary to run both BSA ACS and SBSA ACS tests.

Appendix B Arm SystemReady Pre-Silicon Compliance Checklist

For the compliance checklist see Arm SystemReady Pre-Silicon Compliance Checklist.