

Arm[®] Server Base System Architecture 5.0
Platform Design Document
Non-confidential



Contents

Release information	4
Non-Confidential Proprietary Notice	5
1 About this document	6
1.1 Terms and abbreviations	6
1.2 References	7
1.3 Feedback	7
2 Background	8
3 Introduction	9
4 SBSA	10
4.1 Level 3	10
4.1.1 PE Architecture	10
4.1.2 Expected usage of Secure state	11
4.1.3 Memory Map	11
4.1.4 Interrupt Controller	12
4.1.5 PPI assignments	12
4.1.6 I/O Virtualization	13
4.1.7 Clock and Timer Subsystem	14
4.1.7.1 Summary of the required registers of the CNTControlBase frame	15
4.1.7.2 Summary of the required registers of the CNTReadBase frame	15
4.1.7.3 Summary of the required registers of the CNTCTLBase frame	16
4.1.7.4 Summary of the required registers of the CNTBaseN frame	16
4.1.8 Wakeup semantics	16
4.1.9 Power State Semantics	18
4.1.10 Watchdogs	21
4.1.11 Peripheral Subsystems	21
4.2 Level 3 – firmware	22
4.2.1 Memory Map	22
4.2.2 Clock and Timer Subsystem	22
4.2.3 Watchdogs	23
4.2.4 Peripheral Subsystems	23
4.3 Level 4	24
4.3.1 PE Architecture	24
4.3.2 I/O Virtualization	24
4.3.3 Peripheral Subsystems	24
4.4 Level 5	24
4.4.1 PE Architecture	24
4.4.2 Interrupt Controller	25
4.4.3 I/O Virtualization	25
4.4.4 Clock and Timer Subsystem	26
4.4.5 PPI Assignments	26
A GENERIC WATCHDOG	27
A.1 About	27
A.2 Watchdog Operation	27
A.3 Register summary	29
A.4 Register descriptions	30
A.4.1 Watchdog Control and Status Register	30
A.4.2 Watchdog Interface Identification Register	30
B GENERIC UART	31

B.1	About	31
B.2	Generic UART register frame	31
B.3	Interrupts	33
B.4	Control and setup	33
B.5	Operation	33
C	PERMITTED ARCHITECTURAL DIFFERENCE BETWEEN PES	35
D	PCI EXPRESS INTEGRATION	36
D.1	Configuration space	36
D.2	PCI Express Memory Space	36
D.3	PCI Express device view of memory	37
D.4	Message signaled interrupts	37
D.5	GICv3 support for MSI(-X)	37
D.6	Legacy interrupts	38
D.7	I/O Virtualization	38
D.8	I/O Coherency	38
	D.8.1 PCI Express I/O Coherency without System MMU	39
	D.8.2 PCI Express I/O Coherency with System MMU	39
D.9	Legacy I/O	39
D.10	Integrated end points	40
D.11	Peer-to-peer	40
D.12	PASID support	41
D.13	PCIe Precision Time Measurement	41
E	SMMUV3 INTEGRATION	42
F	DEVICEID GENERATION AND ITS GROUPS	43
F.1	ITS groups	43
	F.1.1 Introduction	43
	F.1.2 Rules	43
	F.1.3 Examples of ITS groups	44
F.2	Generation of DeviceID values	44
	F.2.1 Introduction	44
	F.2.2 Rules	45
F.3	System description of DeviceID and ITS groups from ACPI tables	46
F.4	DeviceIDs from hot-plugged devices	47

Copyright © 2016, 2017, 2018 Arm Limited. All rights reserved.

Release information

Version 5.0 - 30 May 2018

- RAS requirements (5).
- ROP and JOB requirements for SBSA (6).
- SBSA and SVE (7).
- Nested virtualization and SBSA (8).
- MPAM requirements for SBSA (9).
- Invisible caches (ban type 2 caches) AKA Forced WB (10).
- Add Activity Monitor Requirements to SBSA (11).
- Crypto Requirements to SBSA (12).
- Add support for 48-bit operating systems booting on Armv8.2+ systems with 52-bit PA (13).
- Ban non-standard interrupt controller (14).
- Base frequency standardisation (15).
- Assign PPIs for new timers in v8.4 (16).
- Secure EL2 not required (17).
- TLBI range homogeneity (18).
- SVE heterogeneity (19).
- PCIe clarifications (21).
- UART clarifications (22).
- PCIe requirements for assignable devices (23).
- PTM for SBSA-based systems (24).
- PCIe deadlocks (25).
- ACS and Peer-to-peer (26).
- TPM guidance for SBSA (27).
- Deprecate Levels 0, 1, and 2 (28).
- Errata Remove references to Prince Algorithm (29).
- Errata UART Trigger Levels (30).
- Heterogeneity in Server (31).
- Clean to point of persistence support (33).
- Watchdog scaling in SBSA (36).
- Clarify that PEs must be able to access all Non-secure address space (37).
- Appendix I needs to be clearer about RID spaces do not strictly need to supply all 16 bits of width (38).
- Address space input into an SMMU from a device is entirely contiguous, no holes (39).
- Clarify the uniqueness requirements of SPIs that are used to handle legacy PCIe interrupts (A/B/C/D) (40).
- SBSA Armv8.4 and SMMUv3.2 rules on break before make for page tables (41).
- Add reservation for trace buffer overflow PPI (42).
- FP16 support in SBSA (43).
- Correction on stating devices describe their ability to be virtualised through FW (44).
- Clarify IO BAR usage (45).
- Generalise SMMU requirements (46).
- Relaxations in future GIC PPI reservation for Level 5 (79).

Version 3.0 - 01 Feb 2016

- Initial Release. Non-confidential.

Version 3.1 - 01 Feb 2016

- Change of Proprietary Notice. Addition of release history. Non-confidential.

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm's trademark usage guidelines at <http://www.arm.com/company/policies/trademarks>.

Copyright © 2016, 2017, 2018 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

1 About this document

1.1 Terms and abbreviations

Term	Meaning
ACS	Access control services – A set of features intended for ensuring that uncontrolled peer-to-peer transaction cannot occur. See PCIe specification [1] for more details.
AER	Advanced Error Reporting. This is a PCIe feature which enables software to isolate and analyse errors with fine granularity. See PCIe specification [1] for more details.
ARE	Affinity Routing Enable (GICv3 [2]).
Arm ARM	Arm Architecture Reference Manual; see [3].
Base Server System	A system compliant with the Server Base System Architecture.
GIC	Generic Interrupt Controller.
I/O Coherent	A device is I/O Coherent with the PE caches if its transactions snoop the PE caches for cacheable regions of memory. The PE does not snoop the device's cache.
LPI	Locality-specific Peripheral Interrupt (GICv3 [2]).
P2P or Peer-to-peer	Traffic that is sent directly between two PCIe endpoints. See PCIe specification [1] for more details.
PCI Host Bridge (PHB)	A <i>PCI Host Bridge</i> (PHB) is a host-to-PCI bridge; this provides a design-time fixed range of bus and memory/I/O resource ranges to the system. A PHB connects a set of root ports to the host.
PE	Processing Element, as defined in the Arm ARM. Typically a single hardware thread of a PE.
PMU	Performance Monitor Unit.
PPI	Private Peripheral Interrupt.
PTM	Precision Time Measurement. PCIe standard specified method for finding the relationship between a PTM master clock and another clock in a device in the same hierarchy.
Root Complex (RC)	A <i>Root Complex</i> (RC) is a collection of PHBs and therefore RPs. A RC might be conceptually composed of one or more PCIe Host Bridge (PHB). A RC provides a PCI segment. All PHBs in a RC are on the same segment.
Root Port (RP)	A <i>Root Port</i> (RP) is associated with a physical port and appears in PCIe config space as a PCI bridge with a type 1 header, A RP is the root of a PCIe hierarchy.
SBBR	Server Base Boot Requirements [4].
SBSA	Server Base System Architecture.
SGI	Software Generated Interrupt.
SPI	Shared Peripheral Interrupt.
SR-IOV	Single Root IO virtualization. This is a method for a PCIe device to be virtualized. See PCIe specification [1] for more details.
SRE	System Register interface Enable (GICv3 [2]).
System firmware data	System description data structures such as ACPI or Flattened Device Tree.
TCG	Trusted Computing Group.
TPM	Trusted Platform Module (TPM). TPM is a technology, typically implemented through a secure microcontroller, that can securely store artifacts used to authenticate a platform, or platform components. The TPM technical specification is maintained by the TCG consortium.
VM	Virtual Machine

1.2 References

This section lists publications by Arm and by third parties.

See Arm Infocenter (<http://infocenter.arm.com>) for access to Arm documentation.

- [1] *PCI Express Base Specification Revision 4.0*. PCI-SIG.
- [2] *IHI 0069 Arm® Architecture Specification, GIC architecture version 3.0 and version 4.0*. ARM Ltd.
- [3] *DDI 0487 Arm® Architecture Reference Manual ARMv8, for the ARMv8-A architecture profile*. ARM Ltd.
- [4] *DEN 0044 Server Base Boot Requirements, System Software on Arm® Platforms*. ARM Ltd.
- [5] *11889:2015 standard publication. Information technology — Trusted platform module library*. ISO/IEC.
- [6] *Enhanced Allocation (EA) for Memory and I/O Resources*. PCI-SIG.
- [7] *DEN 0068 CoreSight Base System Architecture*. ARM Ltd.
- [8] *DDI 0183 Arm® PrimeCell® UART (PL011) Technical Reference Manual*. ARM Ltd.
- [9] *DEN 0029A Server Base System Architecture Version 3.1*. ARM Ltd.
- [10] *ARM IHI 0062 Arm® System Memory Management Unit Architecture Specification*. ARM Ltd.
- [11] *IHI 0067 Arm® System Memory Management Unit Architecture Specification, 64KB Translation Granule Supplement*. ARM Ltd.
- [12] *IHI 0048 Arm® Architecture Specification, GIC architecture version 2.0*. ARM Ltd.
- [13] *IHI 0070 Arm® System Memory Management Unit Architecture Specification, SMMU architecture version 3.0 and version 3.1*. ARM Ltd.
- [14] *PCI-To-PCI Bridge Architecture Specification 1.2*. PCI-SIG.

1.3 Feedback

Arm welcomes feedback on its documentation.

If you have comments on the content of this manual, send an e-mail to errata@arm.com. Give:

- The title (Server Base System Architecture).
- The document ID and version, DEN0029B 5.0.
- The page numbers to which your comments apply.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

2 Background

ARM processors are used in a wide variety of system-on-chip products in many diverse markets. The constraints on products in these markets are inevitably very different, and it is impossible to produce a single product that meets all of the needs of the various markets.

The ARM architecture profiles, **A**pplication, **R**eal-time, and **M**icrocontroller, segment the solutions produced by ARM and align with the varying functional requirements of particular target markets. The differences between products targeted at different profiles are substantial due to the diverse functional requirements of the market segments.

However, even within an architectural profile, the wide-ranging use of a product means that there are frequent requests for features to be removed to save silicon area. This is relevant for products targeted at cost-sensitive markets, where the cost of customizing software to accommodate the loss of a feature is small compared to the overall cost saving of removing the feature itself.

In other markets, such as those which require an open platform with complex software, the savings gained from removing a hardware feature are outweighed by the cost of software development to support the different variants. In addition, software development is often performed by third parties, and the uncertainty about whether new features are widely deployed can be a substantial brake to the adoption of those features.

The ARM Application profile must balance these two competing business pressures. It offers a wide range of features, such as Advanced SIMD and floating point support, and TrustZone system security technology, to tackle an increasing range of problems. It also provides the flexibility to reduce silicon space by removing hardware features in cost-sensitive implementations.

ARM processors are built into a large variety of systems. Aspects of this system functionality are crucial to the fundamental function of system software.

Variability in PE features and certain key aspects of the system impact on the cost of software system development and the associated quality risks.

Base System Architecture specifications are part of ARM's strategy of addressing this variability.

3 Introduction

This document specifies a hardware system architecture, based on ARM 64-bit architecture, that server system software, such as operating systems, hypervisors and firmware can rely on. It addresses PE features and key aspects of system architecture.

The primary goal is to ensure sufficient standard system architecture to enable a suitably-built single OS image to run on all hardware compliant with this specification. A driver-based model for advanced platform capabilities beyond basic system configuration and boot is required, however this is outside the scope of this document. Fully discoverable and describable peripherals aid the implementation of such a driver model.

This specification also specifies features that firmware can rely on, allowing for some commonality in firmware implementation across platforms.

ARM does not mandate compliance to this specification but anticipates that OEMs and software providers will require compliance to maximize out of box software compatibility and reliability.

The Server Base System Architecture embeds the notion of levels of functionality. Each level adds functionality over and above a previous level. Unless explicitly stated, all specification items belonging to level N apply to levels greater than N. Previous versions of the SBSA included levels 0,1,2,3 and 3 Firmware. This version removes levels 0 to 2, and commences with level 3. All requirements from previous levels that are still valid have been folded into Level 3. This document also introduces levels 4 and 5.

An implementation is consistent with a level of the Server Base System Architecture if it implements all of the functionality of that level at performance levels appropriate for the target uses of that level. This means that all functionality of a level can be exploited by software without unexpectedly poor performance.

Note

This is intended to avoid approaches such as software emulation of functionality that is critical to the performance of software using the SBSA. It is not intended to act as a restriction of legitimate exploration of the power, performance, or area tradeoffs that characterize different products, nor to restrict the use of trapping within a virtualization system.

Implementations that are consistent with a level of the Server Base System Architecture can include additional features that are not included in the definition of that level. However, software written for a specific level must run, unaltered, on implementations that include such additional functionality.

Software running on a system including an ARM core inevitably includes code that is system-specific. Such code is typically partitioned from the rest of the system software in the form of Firmware, Hardware Abstraction Layers, Board Support Packages, Drivers and similar constructs. This document refers to such constructs as *Hardware Specific Software*. The ARM Server Base Boot Requirements (SBBR) specification [4] describes firmware requirements for an ARM server system. Where this specification refers to system firmware data, it refers to firmware specified in the SBBR.

This specification uses the phrase *software consistent with the Server Base System Architecture* to indicate software that is designed to be portable between different implementations that are consistent with the Server Base System Architecture. Software that is consistent with the Server Base System Architecture does not depend on the presence of hardware features that are not mandated in this specification. However, software might use features that are not included in this specification, after checking that the platform supports the features, for example by using hardware ID registers or system firmware data.

4 SBSA

4.1 Level 3

4.1.1 PE Architecture

The PEs referred to in this specification are those that are running the operating system or hypervisor, not PEs that are acting as devices.

PEs in the base server system are compliant with Armv8 and the following is true:

- The number of PEs in the system must not exceed 2^{28} . This reflects the maximum number of PEs GICv3 can support.
- PEs implement Advanced SIMD extensions.
- Whether the Instruction Caches are implemented as VIPT or PIPT is IMPLEMENTATION DEFINED.

Note

Not all PEs are required to support the same Instruction Cache addressing scheme.

- PEs must implement 16-bit ASID support.
- PEs must support 4KB and 64KB translation granules at stage 1 and stage 2.
- All PEs are coherent and in the same Inner Shareable domain.
- Where export restrictions allow, PEs must implement cryptography extensions.
- PEs must implement little-endian support.
- PEs must implement EL2 and EL3.

Note

For systems that implement Armv8.4 PEs, secure EL2 is optional.

- PEs must implement AArch64 at all Exception levels.
- The PMU overflow signal from each PE must be wired to a unique PPI interrupt with no intervening logic.
- Each PE must implement a minimum of six programmable PMU counters.
- Each PE must implement a minimum of six breakpoints, two of which must be able to match virtual address, contextID or VMID.
- Each PE implements a minimum of four synchronous watchpoints.
- All PEs are architecturally symmetric except for the permitted exceptions laid out in Appendix Section C.
- PEs must implement the CRC32 instructions.
- Where PEs implement the scalar vector extension, the vector length maximum must be at least 256 bits.

Note

- Typical operating systems will not be able to take advantage of differences in maximum vector length among PEs.
- It is consistent with this specification to implement PEs with support for the AArch32 Execution state.

Level 3 systems can be implemented using the Armv8.0 architecture revision or higher. Armv8.2 LPA [3], large *Physical Address* (PA) and *Intermediate Physical Address* (IPA) support, expands the maximum physical address width from 48 to 52 bits. Using 52-bit PA requires 64KB translation granule. Server base systems

that make use of Armv8.2 LPA must provide a mode where the memory map is wholly contained inside 2^{48} bytes (256TB). This is required to support operating systems that do not make use of the 64KB granule.

4.1.2 Expected usage of Secure state

The Level 3 base server system is expected to use the PE EL3 and Secure state as a place to implement platform-specific firmware. The system might choose to implement further functionality in the Secure state, but this is outside the scope of SBSA Level 3.

SBSA Level 3 does not expect PCI express to be present in the Secure state. This is reflected in the GICv3 architecture, which does not support Secure LPI.

4.1.3 Memory Map

This specification does not mandate a standard memory map. It is expected that the system memory map is described to system software by system firmware data.

To enable EL2 hypervisors to use a 64KB translation granule at stage 2 MMU translation, the base server system must ensure that all memory and peripherals can be mapped using 64KB stage 2 pages and must not require the use of 4KB pages at stage 2. It is expected therefore that peripherals that are to be assigned to different virtual machines will be situated within different 64KB regions of memory.

Systems will not necessarily fully populate all of the addressable memory space. All memory accesses, whether they access memory space that is populated or not, must respond within finite time, so as to avoid the possibility of system deadlock.

Note

Compliant software must not make any assumptions about the memory map that might prejudice compliant hardware. For example, the full physical address space must be supported. There must be no dependence on memory or peripherals being located at certain physical locations.

Where a memory access is to an unpopulated part of the addressable memory space, accesses must be terminated in a manner that is presented to the PE as either a precise Data Abort, or as a system error interrupt, or an SPI, or LPI interrupt to be delivered to the GIC.

All Non-secure on-chip masters in a base server system that are expected to be under the control of the operating system or hypervisor must be capable of addressing all of the Non-secure address space. If the master goes through a SMMU then the master must be capable of addressing all of the Non-secure address space when the SMMU is turned off. Equally, all PEs must be able to access all of the Non-secure address space.

Non-secure off-chip devices that cannot directly address all of the Non-secure address space must be placed behind a stage 1 System MMU compatible with the Arm SMMUv2 or SMMUv3 specification, that has an output address size large enough to address all of the Non-secure address space. See Section 4.1.6.

Where it is possible for the forward progress of a memory transaction to depend on a second memory access, the system must avoid deadlock if the memory access gets ordered behind the original transaction. This can occur in PCI memory or IO physical address space. For example, in the case of PCI, without an appropriate mitigation, a deadlock could arise if the memory access was also a PCI transaction, and therefore might be ordered behind the original PCI transaction. A system is permitted to resolve this dependency by terminating the memory accesses. A transaction that is terminated in this case might return any value, have any written data ignored, or be terminated with an error.

Note

- For example, in line with the PCI ordering rules, the completion for a read of an SMMU or GIC table might be blocked behind an earlier inbound PCI transaction which the SMMU or GIC is blocking, until the table access completes.
- Because a system is permitted to avoid deadlock by terminating transactions in this way, system software must not allocate any structures relating to a SMMU or GIC in PCIe address space or IO address space.

4.1.4 Interrupt Controller

The GICv3 specification introduces support for systems with more than eight PEs, as well as improved support for larger numbers of interrupts.

A level 3 base server system must implement a GICv3 interrupt controller. The GICv3 interrupt controller must support two Security states.

If the base server system includes PCI Express then the GICv3 interrupt controller must implement ITS and LPI.

Note

- It is expected that MSI and MSI-X are mapped to LPI interrupts.
- It is permissible to build a system with no support for SPI, however Arm expects that the peripheral eco-system will continue to rely on wired level interrupts and expects most systems to support SPI as well as LPI interrupts. PCI legacy INTA-INTD interrupts are wired interrupts, see Section D.6 for more details.
- The ARM PL011 UART requires a level interrupt, as does a PCIe root complex, for legacy interrupt support.

4.1.5 PPI assignments

A level 3 base server system must comply with the PPI mapping laid out in Table 2.

Table 2: PPI assignments

Interrupt ID	Interrupt	Description
30	Overflow interrupt from CNTP	Non-secure physical timer interrupt.
29	Overflow interrupt from CNTPS	Secure Physical timer interrupt.
28	Overflow interrupt from CNTHV	EL2 virtual timer (for PEs from at least the Armv8.1 architecture).
27	Overflow interrupt from CNTV	Virtual timer interrupt.
26	Overflow interrupt from CNTHP	Hypervisor timer interrupt.
25	GIC Maintenance interrupt	The virtual PE interface list register overflow interrupt.

Interrupt ID	Interrupt	Description
24	CTIIRQ	Cross Trigger Interface (CTI) interrupt.
23	Performance Monitors Interrupt	Indicates an overflow condition in the performance monitors unit.
22	COMMIRQ	Debug Communication Channel (DCC) interrupt.
21	PMBIRQ	Statistical Profiling Interrupt, if Statistical Profiling Extensions are implemented.
20	CNTHPS	Secure EL2 physical timer interrupt (if PEs are v8.4 or greater).
19	CNTHVS	Secure EL2 virtual timer interrupt (if PEs are v8.4 or greater).

Note

An implementation might need to reserve a PPI for trace buffer overflow.

4.1.6 I/O Virtualization

It is IMPLEMENTATION-SPECIFIC whether any given device in a base server architecture system supports the ability to be hardware virtualized. It is expected that devices that can be hardware virtualized have that property expressed either by system firmware data, or through hardware discoverability.

If a device is virtualized and passed through to an operating system under a hypervisor, then the memory transactions of the device must be subject to stage 2 translation, allocation of memory attributes, and application of permission checks, under the control of the hypervisor. This specification collectively refers to this translation, attribution, and permission checking as *policing*. The act of policing is referred to as stage 2 System MMU functionality.

Stage 2 System MMU functionality must be provided by a System MMU compatible with the ARM SMMUv2 specification, where:

- Support for stage 1 policing is not required.
- Each context bank must present a unique physical interrupt to the GIC.

Note

This behavior is consistent with ARM's MMU-500 implementation. Software can either program stage 2 System MMUs to use the same page tables as the PE or build shadow page tables. Standard PCI Express ATS support, is included in SMMUv3.

Or the Stage 2 System MMU functionality must be provided by a System MMU compatible with the ARM SMMUv3 spec where:

- Support for stage 1 policing is not required.
- The integration of the System MMUs is compliant with the specification in Section E.

All the System MMUs in the system must be compliant with the same architecture version.

Note

- System MMUv3 is not backwards compatible with System MMUv2 and as such any system implementing System MMUv3 MMUs is not strictly backwards compatible with level 2, which requires System MMUv2.
 - Support for broadcast TLB maintenance operations is not required.
 - For systems that implement SMMU architecture revision 3.2, and Armv8.4 PEs, if the PEs do not implement secure EL2, the SMMUs are not required to implement secure EL2.
-

The base server system might instance an IMPLEMENTATION DEFINED number of SMMU components. It is expected that these components will be described by system firmware data along with a description of how to associate them with the devices they police.

Armv8.4 introduces TLB Invalidation instructions which apply to a range of input addresses rather than just to a single address. If PEs used by the server base system support TLB range instructions, then all OS visible masters that contain a TLB must support range invalidates.

4.1.7 Clock and Timer Subsystem

The base server system must include the system counter of the Generic Timer as specified in the Arm ARM [3].

The system counter of the Generic Timer must run at a minimum frequency of 10MHz. The counter must not roll over inside a 10 year period.

Note

Previous versions of the SBSA imposed an upper limit of 400Mhz. This is instead replaced by an upper limit on the roll over period. This permits implementations to target high frequencies for markets where this is an advantage.

The architecture of the counter mandates that it must be at least 56 bits, and at most 64 bits. From Armv8.4, for systems that implement counter scaling, the minimum becomes 64 bits.

The Generic Timer system counter also exports its count value, or an equivalent encoded value, through the system to the timers in the PEs as part of the Generic Timer subsystem. This count must be available to the PE timers when they are active, which is when the PEs are in power states where the PE timer is required to be on.

The local PE timers have a programmable count value. When the value expires it generates a Private Peripheral Interrupt for the associated PE.

The local PE timers can be built so that they are always on. This property is described in the system firmware data.

Unless all of the local PE timers are always on, the base server system must implement a system wakeup timer that can be used when PE timers are powered down. The system wakeup timer must in the form of the memory mapped timer described in the ARMv8 ARM [3]. On timer expiry, the system wakeup timer must generate an interrupt that must be wired to the GIC as an SPI or LPI. Additionally, the system wakeup timer can be used to wake up PEs. See Section 4.1.8.

It is recognized that in large system a shared resource like the system wakeup timer can create a system bottleneck, as access to it must be arbitrated through a system-wide lock. It is anticipated that this will be dealt with by the platform by having the firmware tables describe the PE timers as always on and remove

the need for the system timer. The platform will either implement hardware always on PE timers or use the platform firmware to save and restore the PE timers in a performance scalable fashion.

Note

Systems compliant with Level 3 – firmware will have standard hardware that the firmware can use, see Section 4.1.7

The wakeup timer does not require a virtual timer to be implemented and it is permissible for the virtual offset register to read as zero. Writes to the virtual offset register in CNTCTLBase frame are ignored. The timer is not required to have a CNTELOBase frame.

If the system includes a system wakeup timer, this memory-mapped timer must be mapped on to Non-secure address space. This is referred to as the *Non-secure system wakeup timer*. Table 3 summarizes which address space the register frames should be mapped on to.

Table 3: Generic counter and timer memory mappings

Register Frame	
CNTControlBase	Secure
CNTReadBase	Not required
CNTCTLBase	Non-secure and Secure
CNTBaseN	Non-secure

4.1.7.1 Summary of the required registers of the CNTControlBase frame

Offset	Name	Type	Description
0x000	CNTCR	RW	Counter Control Register
0x004	CNTSR	RO	Counter Status Register
0x008	CNTCV[31:0]	RW	Counter Count Value Register
0x00C	CNTCV[63:32]	RW	Counter Count Value Register
0x010-0x01C	-	RES0	Reserved
0x020	CNTFID0	RO OR RW	Frequency modes table, and end marker. CNTFID0 is the base frequency, and each CNTFID n is an alternative frequency. For more information see the ARM ARM.
0x020+4n	CNTFID n	RO OR RW	Frequency modes table, and end marker. CNTFID0 is the base frequency, and each CNTFID n is an alternative frequency. For more information see the ARM ARM.
(0x024+4n)-0x0BC	-	RES0	Reserved
0x0C0-0X0FC	-	IMPLEMENTATION DEFINED	Reserved for IMPLEMENTATION DEFINED registers
0x100-0xFCC	-	RES0	Reserved
0xFD0-0xFFC	CounterID<n>	RO	Counter ID registers 0-11

4.1.7.2 Summary of the required registers of the CNTReadBase frame

Offset	Name	Type	Description
0x000	CNTCV[31:0]	RO	Counter Count Value Register
0x004	CNTCV[63:32]	RO	Counter Count Value Register
0x008-0xFFC	-	RES0	Reserved
0xFD0-0xFFC	CounterID<n>	RO	Counter ID registers 0-11

4.1.7.3 Summary of the required registers of the CNTCTLBase frame

Offset	Name	Type	Security	Description
0x000	CNTFRQ	RW	Secure	Counter Frequency register
0x004	CNTNSAR	RW	Secure	Counter Non-secure Access register
0x008	CNTTIDR	RO	Both	Counter Timer ID register
0x00C-0x03F	-	RES0	Both	Reserved
0x040+4N	CNTACR<N>	RW	Configurable	Counter Access Control register N
0x060-0x07F	-	RES0	Both	Reserved
0x0C0-0x0FC	-	UNK/SBZP	Both	Reserved
0x100-0x7FC	-	-	Both	IMPLEMENTATION DEFINED
0x800-0xFBC	-	UNK/SBZP	Both	Reserved
0xFC0-0xFCF	-	-	Both	IMPLEMENTATION DEFINED
0xFD0-0xFFC	CounterID<n>	RO	Both	Counter ID registers 0-11

4.1.7.4 Summary of the required registers of the CNTBaseN frame

Offset	Name	Type	Description
0x000	CNTPCT[31:0]	RO	Physical Count register
0x004	CNTPCT[63:32]	RO	Physical Count register
0x010	CNTFRQ	RO	Counter Frequency register
0x020	CNTP_CVAL[31:0]	RW	Physical Timer Compare Value register
0x024	CNTP_CVAL[63:32]	RW	Physical Timer Compare Value register
0x028	CNTP_TVAL	RW	Physical Timer Value register
0x02C	CNTP_CTL	RW	Physical Timer Control register
0x040-0xFCF	-	RES0	Reserved
0xFD0-0xFFC	CounterID<n>	RO	Counter ID registers 0-11

4.1.8 Wakeup semantics

Systems implement many different power domains and power states. It is important for the OS or hypervisor, or both, to understand the relationship between these power domains and the facilities it has for waking PEs from various low power states.

A key component in controlling the entry to and exit from low-power states is the IMPLEMENTATION DEFINED

power controller. The power controller controls the application of power to the various power domains. On entry to low-power states hardware-specific software will program the power controller to take the correct action. On exit from a low-power state, hardware-specific software might need to reprogram the power controller. Hardware-specific software is required to save and restore system state when entering and exiting some low-power states.

This specification defines two classes of wakeup methods: interrupts, and always-on power domain wake events.

The first class of wakeup methods are interrupts. This specification defines interrupts that wake PEs as wakeup interrupts.

A wakeup interrupt is any interrupt that is any one of the following:

- An SPI that directly targets a PE.
- An SGI.
- A PPI.
- An LPI

In addition, for an interrupt to be a wakeup interrupt, it must be enabled in the distributor. A PE must wake in response to a wakeup interrupt, independent of the state of its CPSR interrupt mask bits, which are the A, I, and F bits, and of the wakeup interrupt priority.

Note

- Typically, a wakeup signal is exported from the GIC to the power controller to initiate the PE wakeup.
 - There are some power states where a PE will not wake on an interrupt. It is the responsibility of system software to ensure there are no wakeup interrupts targeting a PE entering these states.
-

The local PE timers are an important source of interrupts that can wake the PE. However the local PE timer might be powered down in some low-power states, as it might be in the same power domain as the PE. In low-power states where the local PE timer is powered down, system software can use an SGI from other running PEs to wake the PE, or it can configure the system wakeup timer to send a wakeup interrupt to the PE to wake it.

In some very deep low-power states the GIC will be powered down. To wake from these states, there is another class of wakeup methods that can be used; always-on power domain wake events.

If the system supports a low-power state where the GIC is powered down, then there must be an IMPLEMENTATION DEFINED way to program the power controller to wake a PE on expiry of the system wakeup timer or the generic watchdog. In this scenario, the system wakeup timer or generic watchdog is still required to send its interrupt.

There might be other IMPLEMENTATION DEFINED always-on power domain wakeup events that can wake PEs from deep low-power states, such as PCI Express wakeup events and Wake-on-LAN.

See Section 4.1.9 for a description of the power state semantics that the system must comply with.

Whenever a PE is woken from a sleep or off state the OS or Hypervisor must be presented with an interrupt so that it can determine which device requested the wakeup. The interrupt must be pending in the GIC at the point that control is handed back to the OS or Hypervisor from the system-specific software performing the state restore.

This interrupt must behave like any other: a device sends an interrupt to the GIC, and the GIC sends the interrupt to the OS or Hypervisor. The OS or Hypervisor is not required to communicate with a system-specific interrupt controller.

Note

If the wakeup event is an edge then the system must ensure that this edge is not lost. The system must ensure that the edge wakes the system and is subsequently delivered to the GIC without losing the edge.

An example of an expected chain of events would be:

1. Wakeup event occurs e.g. GPIO or wake-on-LAN.
2. The power controller responds by powering on the necessary resources that include the PE and the GIC.
3. The PE comes out of reset and hardware-specific software restores state, including the GIC.
4. An interrupt is presented to the GIC representing the wakeup event. In many situations this might be exactly the same signal as the wakeup event.
5. The system must ensure that, by the time the hardware-specific restore software has delegated to the OS or Hypervisor, the interrupt is pending in the GIC.
6. The OS or Hypervisor can respond to the interrupt.

4.1.9 Power State Semantics

This specification does not mandate a given hierarchy of power domains, but there are some rules and semantics that must be followed.

Figure 1 is an example block diagram showing a possible hierarchy of power domains. Note that there are other examples that conform to this specification that are not subsets of the system in the diagram.

In order for either the OS or hypervisor, or both, to be able to reason about wakeup events and to know which timers will be available to wake the PE, all PEs must be in a state that is consistent with one of the semantics described in Table 8 and Table 9. Note that all PEs do not need to be in the same state. It is expected that the semantics of the power states that a system supports will be described by system firmware data. Table 10 describes the power state semantics in a set of component-specific rules.

System MMUs and GICv3 make use of tables in memory in the power states where GIC is 'On'. For such states, system memory must be available and will respond to requests without requiring intervention from software running on the PEs.

Hardware-specific software is required to save and restore system state when entering and exiting low-power states.

It is highly likely that many systems will support very low-power states where most system logic is powered down and the system memory is in self-refresh, but the OS retains control over future wakeup. This is reflected in power state semantic E. In this state, the GIC can be powered off after system software has saved its state. In this state, wakeup signals go straight to the system power controller and do not require use of the GIC to wake the PEs. The system power controller is system-specific. When in a power state of semantic E, the system power controller wakes an IMPLEMENTATION DEFINED PE, or set of PEs, when the system wakeup timer expires. Other system-specific events might also cause wakeup from this state, such as a PCI Express wakeup event. The events that will cause wakeup from this state are expected to be discoverable from system firmware data.

When the system is in a state where the GIC is powered down devices must not send messaged interrupts to the GIC.

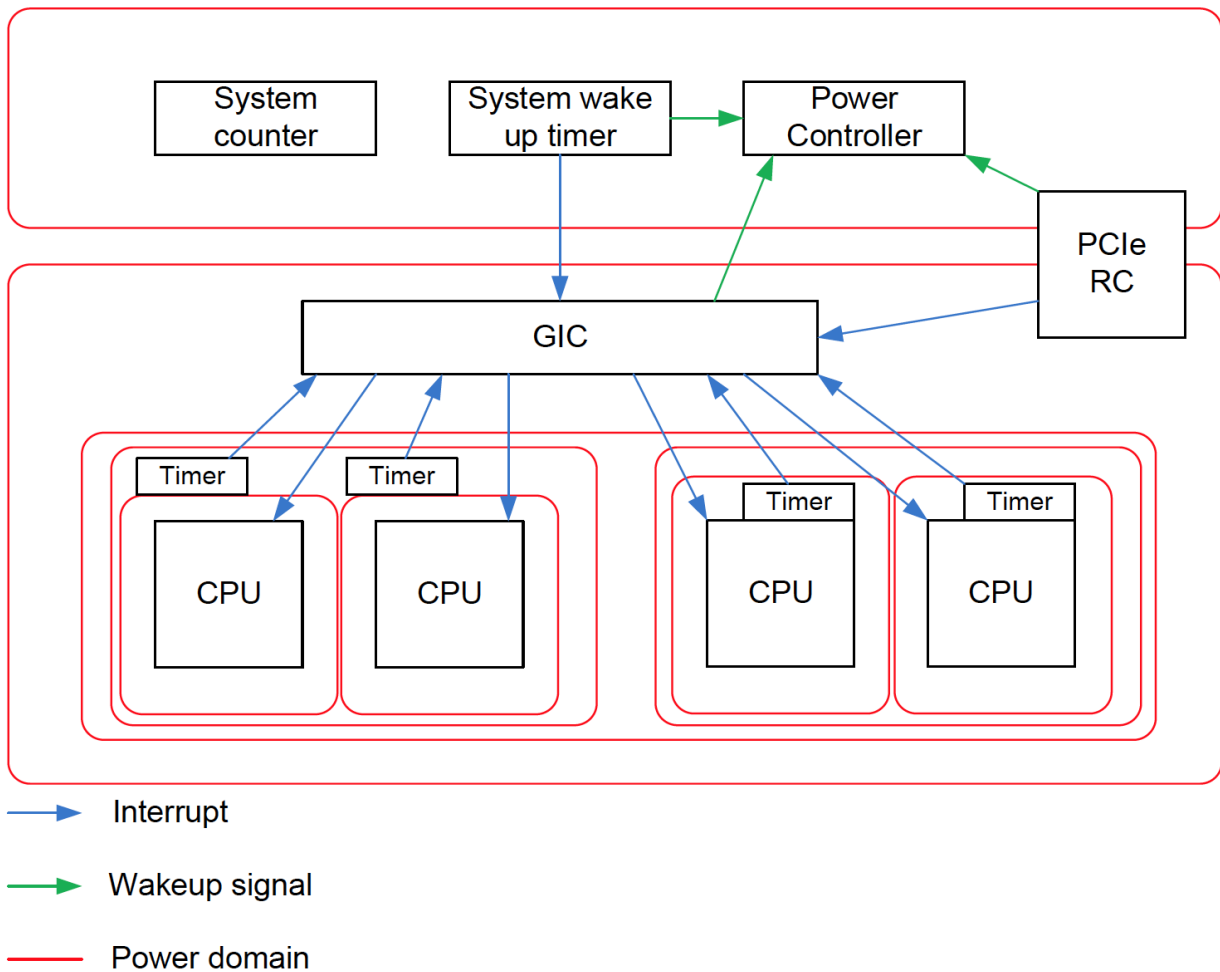


Figure 1: Example system block diagram showing power domains and timer hierarchy

Table 8: PE Power States

PE State	Description
Run	The PE is powered up and running code.
Idle_standby	The PE is in STANDBYWFI state, but remains powered up. There is full state retention, and no state saving or restoration are required. Execution automatically resumes after any interrupt or external debug request (EDBGRQ). Debug registers are accessible.
Idle_retention	The PE is in STANDBYWFI state, but remains powered up. There is full state retention, and no state saving or restoration are required. Execution automatically resumes after any interrupt or external debug request (EDBGRQ). Debug registers are not accessible.
Sleep	The PE is powered down but hardware will wake the PE autonomously, for example, on receiving a wakeup interrupt. No PE state is retained. State must be explicitly saved. The woken PE starts execution at the reset vector, and then hardware-specific software restores state.

PE State	Description
Off	The PE is powered down and is not required to be woken by interrupts. The only way to wake the PE is by explicitly requesting the power controller, for example, from system software running on another PE, or an external source such as a <code>poweron_reset</code> . This state can be used when the system software explicitly decides to remove the PE from active service, giving the hardware opportunity for more aggressive power saving. No PE state is retained.

Table 9: Power State Semantics

Semantic	PE and GIC PE Interface	PE timers	GIC Distributor	System wake timers, system counter generic watchdog	Note
A	Run	On	On	On	-
B	Idle	On	On	On	PE will resume execution on receipt of any interrupt.
C	Sleep	On	On	On	PE will wake on receipt of a wakeup interrupt.
D	Sleep	Off	On	On	PE will wake on receipt of a wakeup interrupt, but local timer is off.
E	Sleep	Off	Off	On	PE will wake from system timer wakeup event or other system specific events.
F	Off	Off	On	On	Some, but not all, PEs are in Off state.
G	Off	Off	Off	Off	All PEs in Off state.
H	Sleep	On	Off	On	PE will wake from PE timer, system timer wakeup event or other system specific events.
I	Idle	Off	On	On	PE will resume execution on receipt of any interrupt, but the local timer is off.

Table 10: Component Power State Semantics

PE and GIC PE Interface	Individual PEs and their associated GIC PE interface can be in Run, Idle, Sleep or Off state.
PE timers	Must be On if the associated PE is in the Run state. Might be On or Off if the PE is in Idle or Sleep state. Must be Off if the PE is in the Off state.

GIC Distributor	Must be On if any PE is in the Run or Idle state. Might be On or Off if all PEs are in either the Sleep or Off state, with at least one PE in the Sleep state. Must be Off if all PEs are in the Off state.
System wakeup timers and system counter and generic watchdog	Must be On if any PE is not in the Off state. Must be Off if all PEs are in the Off state.

4.1.10 Watchdogs

The base server system implements a Generic Watchdog as specified in Section A. Watchdog Signal 0 is routed as an SPI or an LPI to the GIC and it is expected this will be configured as an EL2 interrupt, directly targeting a single PE.

Watchdog Signal 1 must be routed to the platform. In this context, *platform* means any entity that is more privileged than the code running at EL2. Examples of the platform component that services Watchdog Signal 1 are: EL3 system firmware, or a system control processor, or dedicated reset control hardware.

The action taken on the raising of Watchdog Signal 1 is platform-specific.

Note

Only directly-targeted SPI are required to wake a PE; see Section 4.1.8 for further information. Programming the watchdog SPI to be directly targeted ensures delivery of the interrupt independent of PE power states. However it is possible to use a 1 of N SPI to deliver the interrupt as long as one of the target PEs is running.

The watchdog must have both its register frames mapped on to Non-secure address space; and is referred to as the *Non-secure watchdog*.

4.1.11 Peripheral Subsystems

If the system has a USB2.0 host controller peripheral it must conform to EHCI v1.1 or later.

If the system has a USB3.0 host controller peripheral it must conform to XHCI v1.0 or later.

If the system has a SATA host controller peripheral it must conform to AHCI v1.3 or later.

If a TCG TPM based security model is supported, the server base system needs to provide a TPM implementation that is compliant to TPM Library Specification, Family 2.0 [5].

Peripheral subsystems which do not conform to the above are permitted, provided that they are not required to boot and install an OS.

For the purpose of system development and bring up, the base server system must include a Generic UART. The Generic UART is specified in Section B. The UARTINTR interrupt output is connected to the GIC as an SPI or an LPI.

The Generic UART required by level 3 and above must be mapped on to Non-secure address space. This is referred to as the *Non-secure Generic UART*.

If the system has a PCI Express root complex then it must comply with the rules in Section D.

For systems that include PCI express, the PCI express integration appendix introduces an additional rule applicable to a level 3 system. See Section D.

The memory attributes of DMA traffic must be one of the following:

- Inner writeback, outer writeback, Inner Shareable.
- Inner Non-cacheable, outer Non-cacheable.
- A device type.

I/O Coherent DMA traffic must have the attribute “Inner writeback, outer writeback, Inner Shareable”.

4.2 Level 3 – firmware

Level 3 – firmware is an optional additional set of requirements for a level 3 system. It is designed to give a base set of functionality that standard platform firmware can rely on. A system that is compliant with level 3 and not compliant with level 3 – firmware is still a fully compliant level 3 system. It has all the features required by the operating systems and hypervisors.

4.2.1 Memory Map

The system must provide some memory mapped in the Secure address space. The memory must not be aliased in the Non-secure address space. The amount of Secure memory provided is platform-specific as the intended use of the memory is for platform-specific firmware.

All Non-secure on-chip masters in a base server system that are expected to be used by the platform firmware must be capable of addressing all of the Non-secure address space. If the master goes through a SMMU then the master must be capable of addressing all of the Non-secure address space even when the SMMU is off.

4.2.2 Clock and Timer Subsystem

A system compatible with level 3- firmware must also include a Secure wakeup timer in the form of the memory mapped timer described in the ARMv8 ARM [3] This timer must be mapped into the Secure address space, and the timer expiry interrupt must be presented to the GIC as an SPI. This timer is referred to as the Secure system wakeup timer.

The Secure wakeup timer does not require a virtual timer to be implemented and it is permissible for the virtual offset register to read as zero, where writes to the virtual offset register in CNTCTLBase frame are ignored. The timer is not required to have a CNTELOBase frame.

The following table summarizes which address space the register frames related to the Secure wakeup timer should be mapped on to.

Register Frame	
CNTControlBase	Secure
CNTReadBase	Not required
CNTCTLBase	Secure
CNTBaseN	Secure

CNTCTLBase might be shared amongst multiple timers, including various Secure and Non-secure timers. The SBSA specification does not require this.

Note

- GICv3 does not support Secure LPI; therefore the Secure system timer interrupt must not be delivered

as LPI.

- It is recognized that in a large system, a shared resource like the system wakeup timer can create a system bottleneck, as access to it must be arbitrated through a system-wide lock. Level 3-firmware requires just a single timer so that standard firmware implementations have a guaranteed timer resource across platforms. It is anticipated that large PE systems will implement a more scalable solution such as one timer per PE.

4.2.3 Watchdogs

The required behavior of watchdog signal 1 of the Non-secure watchdog is modified in level 3- firmware and is required to be routed as an SPI to the GIC. It is expected that this SPI be configured as an EL3 interrupt, directly targeting a single PE.

A system compatible with level 3- firmware must implement a second watchdog, and is referred to as the Secure watchdog. It must have both its register frames mapped in the Secure memory address space and must not be aliased to the Non-secure address space.

Watchdog Signal 0 of the Secure watchdog must be routed as an SPI to the GIC and it is expected this will be configured as an EL3 interrupt, directly targeting a single PE.

Note

- GICv3 does not support Secure LPI. The Secure watchdog interrupts must not be delivered as LPI.
- Only directly targeted SPI are required to wake a PE. Programming the watchdog SPI to be directly targeted ensures delivery of the interrupt independent of PE power states. However it is possible to use a 1 of N SPI to deliver the interrupt provided that one of the target PEs is running. See Section 4.1.9 for information about SPI waking a PE.

Watchdog Signal 1 of the Secure watchdog must be routed to the platform. In this context, platform means any entity that is more privileged than the code running at EL3. Examples of the *platform* component that services Watchdog Signal 1 are a system control processor, or dedicated reset control hardware.

The action taken on the raising of Watchdog Signal 1 of the Secure watchdog is platform-specific.

4.2.4 Peripheral Subsystems

A system compatible with level 3-firmware must provide a second generic UART, referred to as the Secure Generic UART, that can be configured to exist in the Secure memory address space. It must not be aliased in the Non-secure address space. The UARTINTR output of the Secure Generic UART must be connected to the GIC as an SPI.

Note

GICv3 does not support Secure LPI. The Secure Generic UART interrupt must not be delivered as LPI.

Systems that integrate PCI express should note that PCI express integration appendix introduces an additional rule applicable to a level 3 system, see Section D.

4.3 Level 4

4.3.1 PE Architecture

In addition to the level 3 requirements, the following must be true of the PEs in the base server system:

- All PEs must implement the RAS extension introduced in Armv8.2.
- All PEs must implement support for 16-bit VMID.
- All PEs must implement virtual host extensions.
- If PEs implement Armv8.3 pointer signing, the PEs must provide the standard algorithm defined by the ARM architecture [3], identified by ID_AA64ISAR1_EL1[7:4]==0001.
- If the system contains persistent memory that is exposed to the OS, all PEs must support the clean to point of persistence instruction (DC CVAP). The instruction must be able to perform a clean to the point of persistence for all memory that is exposed as persistent memory to the OS.

4.3.2 I/O Virtualization

Stage 1 and 2 System MMU functionality must be provided by a System MMU compatible with the ARM SMMUv3, or higher, architecture revision where the integration of the System MMUs is compliant with the specification in Section E.

All the System MMUs in the system must be compliant with the same architecture version.

All addresses output from a device to an SMMU must lie in a continuous space with no holes. All address in said space will be treated equally by the SMMU. There should be no areas within the address space that receive exceptional treatment, such as bypassing the SMMU.

Note

MSIs from a device are translated in the same way as any other writes from that device.

4.3.3 Peripheral Subsystems

All peripherals intended for assignment to a virtual machine or a user space device driver must be based on PCI Express.

PCI Express integration is covered appendix Section D. Device assignment requirements are covered in Section D.7

There must be no OS observable use of PCIe Enhanced Allocation [6].

4.4 Level 5

4.4.1 PE Architecture

In addition to the level 4 requirements, the following must be true of the PEs in the base server system:

- All PEs must provide support for Stage2 control of memory types and cacheability, as introduced by the Armv8.4 extensions.
- All PEs must implement enhanced nested virtualization, that is provide HCR_EL2.NV2 and the VNCR_EL2 register.

- All PEs must support changing of page table mapping size using the level 1 or level 2 solution proposed in the Armv8.4 extension. Level 2 is recommended. See Section 4.4.3 for the equivalent requirements for the SMMU.
- All PEs must implement pointer signing using the standard algorithm defined by the ARM architecture [3], as indicated by ID_AA64ISAR1_EL1[7:4]==0001.
- PEs based on Armv8.4 must implement the requirements of the CS-BSA combination C [7].
- All PEs must implement the Activity Monitors Extension.
- Where export control allows, all PEs must implement cryptography support for SHA3 and SHA512.

Note

ARM recommends that SM3 and SM4 is also supported in hardware aimed at the Chinese market.

Implementation of the MPAM extension is optional, however if implemented, the following minimal requirements must be met:

- The implementation must provide a minimum of 16 physical partition IDs.
-

Note

ARM strongly recommends that the number of partition IDs scales with the number of PEs.

- The implementation must provide virtualization support with a minimal of 8 virtual partition IDs.
 - The implementation must provide a minimal of 2 performance monitor groups.
 - The implementation must provide cache portion partitioning in the last level cache.
-

Note

Here last level cache refers to an on SoC cache, as opposed to an off chip DRAM cache.

4.4.2 Interrupt Controller

A level 4 base server system must implement an interrupt controller compatible with the GICv3 or higher architecture [2]. Only this interrupt controller will be visible to hypervisor or operating system software. Other forms of interrupt controller, such as interrupt combining or forwarding engines, are not permissible if they require a platform specific kernel driver.

4.4.3 I/O Virtualization

SMMU implementations must be compliant with the ARM SMMUv3.2 architecture revision or higher.

SMMU implementations must provide level 1 or level 2 support for page table resizing. ARM recommends the SMMU implements level 2. If the SMMU implementation provides level 2, then ARM recommended that the PE also provides level 2.

Note

MPAM architecture requires that all masters that can access an MPAM controlled resource, must support passing MPAM ID information. Therefore, an SMMUv3.2 implementation must support the MPAM extension if the requests it serves access MPAM controlled resources.

4.4.4 Clock and Timer Subsystem

A system compatible with level 5 will implement counter scaling as described in the Armv8.4 architecture.

4.4.5 PPI Assignments

In addition to PPI assignment in Table 2 the following PPIs are reserved by the SBSA specification:

Table 12: PPI assignments

Interrupt ID	Interrupt	Description
1056-1071	Reserved	Reserved for future SBSA usage.
1088-1103	Reserved	Reserved for future SBSA usage.

A GENERIC WATCHDOG

A.1 About

The Generic Watchdog aids the detection of errant system behavior. If the Generic Watchdog is not refreshed periodically, it will raise a signal, which is typically wired to an interrupt. If this watchdog remains un-refreshed, it will raise a second signal which can be used to interrupt higher-privileged software or cause a PE reset.

The Generic Watchdog has two register frames, one that contains the refresh register and one for control of the watchdog.

A.2 Watchdog Operation

The Generic Watchdog has the concept of a Cold reset and a Warm reset. On a Cold reset, certain register values are reset to a known state. Watchdog Cold reset must only occur as part of the watchdog powering-up sequence. On a Warm reset, the architectural state of the watchdog is not reset, but other logic such as the bus interface might be. This is to facilitate the PEs in the system going through a reset sequence, while the watchdog retains its state so it can be examined when the PEs are running.

The basic function of the Generic Watchdog is to count for a fixed period of time, during which it expects to be refreshed by the system indicating normal operation. If a refresh occurs within the watch period, the period is refreshed to the start. If the refresh does not occur then the watch period expires, and a signal is raised and a second watch period is begun.

The initial signal is typically wired to an interrupt and alerts the system. The system can attempt to take corrective action that includes refreshing the watchdog within the second watch period. If the refresh is successful, the system returns to the previous normal operation. If it fails, then the second watch period expires and a second signal is generated. The signal is fed to a higher agent as an interrupt or reset for it to take executive action.

The Watchdog uses the Generic Timer system counter as the timebase against which the decision to trigger an interrupt is made.

Note

The Arm ARM states that the system counter measures the passing of real-time. This counter is sometimes referred to as the physical counter.

The Watchdog is based on a 64-bit compare value and comparator. When the generic timer system count value is greater than the compare value, a timeout refresh is triggered.

The compare value can either be loaded directly or indirectly on an explicit refresh or timeout refresh.

When the watchdog is refreshed explicitly, the compare value is loaded with the sum of the zero-extended watchdog offset register and the current generic timer system count value.

The SBSA 5.0 release introduces revision 1 for the generic watchdog. Revision 1 increases the length the watchdog offset register to 48 bit. The operation of the watchdog remains the same. Software can determine which version of the watchdog is implemented through the watchdog interface identification register (W_IID).

When the watchdog is refreshed through a timeout, the compare value is loaded with the sum of the zero-extended watchdog offset register and the current generic timer system count value. See below for exceptions.

An explicit watchdog refresh occurs when one of a number of different events occur:

- The Watchdog Refresh Register is written.
- The Watchdog Offset Register is written.
- The Watchdog Control and Status register is written.

In the case of an explicit refresh, the Watchdog Signals are cleared. A timeout refresh does not clear the Watchdog Signals.

The watchdog has the following output signals:

- Watchdog Signal 0 (WS0).
- Watchdog Signal 1 (WS1).

If WS0 is asserted and a timeout refresh occurs, then the following must occur:

- If the system is compliant to SBSA level 0 or level 1, then it is IMPLEMENTATION DEFINED whether the compare value is loaded with the sum of the zero-extended watchdog offset register and the current generic timer system count value, or whether it retains its current value.
- If the system is compliant to SBSA level 2 or higher, the compare value must retain its current value. This means that the compare value records the time that WS1 is asserted.

If both watchdog signals are deasserted and a timeout refresh occurs, WS0 is asserted.

If WS0 is asserted and a timeout refresh occurs, WS1 is asserted.

WS0 and WS1 remain asserted until an explicit refresh or watchdog Cold reset occurs.

WS0 and WS1 are deasserted when the watchdog is disabled.

The status of WS0 and WS1 can be read in the Watchdog Control and Status Register.

Note

The following pseudocode assumes that the compare value is not updated on a timeout refresh when WS0 == 1 and does not show the other permitted behavior.

```

TimeoutRefresh = (SystemCounter[63:0] > CompareValue[63:0])
If WatchdogColdReset
    WatchdogEnable = DISABLED
Endif
If LoadNewCompareValue
    CompareValue = new_value
ElseIf ExplicitRefresh == TRUE or (TimeoutRefresh == TRUE and WS0 == FALSE)
    CompareValue = SystemCounter[63:0] + ZeroExtend(WatchdogOffsetValue[47:0])
Endif
If WatchdogEnable == DISABLED
    WS0 = FALSE
    WS1 = FALSE
ElseIf ExplicitRefresh == TRUE
    WS0 = FALSE
    WS1 = FALSE
ElseIf TimeoutRefresh == TRUE
    If WS0 == FALSE
        WS0 = TRUE
    Else
        WS1 = TRUE
    Endif
Endif

```

The Generic Watchdog must be disabled when the System Counter is being updated, or the results are UNPREDICTABLE.

A.3 Register summary

This section gives a summary of the registers, relative to the base address of the relevant frames.

All registers are 32 bits in size and should be accessed using 32-bit reads and writes. If an access size other than 32 bits is used then the results are IMPLEMENTATION DEFINED. There are two register frames, one for a refresh register, and the other containing the status and setup registers.

The Generic Watchdog is little-endian. Table 13 shows the refresh frame.

Table 13: Refresh Frame

Offset	Name	Description
0x000-0x003	WRR	Watchdog refresh register. A write to this location causes the watchdog to refresh and start a new watch period. A read has no effect and returns 0.
0x004-0xFCB	-	Reserved.
0xFCC-0xFCF	W_IID	See Section A.4.2
0xFD0-0xFFF	-	IMPLEMENTATION DEFINED.

Table 14 shows the watchdog control frame.

Table 14: Watchdog Control Frame

Offset	Name	Description
0x000-0x003	WCS	Watchdog control and status register. A read/write register containing a watchdog enable bit, and bits indicating the current status of the watchdog signals.
0x004-0x007	-	Reserved.
0x008-0x00B	WOR[31:0]	Watchdog offset register. A read/write register containing the lower 32 bits of the unsigned watchdog countdown timer value.
0x00C-0x00F	WOR[63:32]	Watchdog offset register: Bits [31:16] Reserved. Read all zeros, write has no effect. Bits [15:0] Read/write upper 16 bits of the watchdog countdown timer.
0x010-0x013	WCV[31:0]	Watchdog compare value. Read/write registers containing the current value in the watchdog compare register.
0x014-0x017	WCV[63:32]	
0x018-0xFCB	-	Reserved.
0xFCC-0xFCF	W_IIDR	See Section A.4.2
0xFD0-0xFFF	-	IMPLEMENTATION DEFINED

Note

In the previous revision of the SBSA generic watchdog, revision 0, the Watchdog offset register was 32 bit, and offset 0x00C-0x00F was Reserved.

A.4 Register descriptions

A.4.1 Watchdog Control and Status Register

The format of the Watchdog Control and Status Register is:

Bits [31:3]

Reserved. Read all zeros, write has no effect.

Bits [2:1] – Watchdog Signal Status bits

A read of these bits indicates the current state of the watchdog signals; bit [2] reflects the status of WS1 and bit [1] reflects the status of WS0.

A write to these bits has no effect.

Bit [0] – Watchdog Enable bit

A write of 1 to this bit enables the Watchdog, a 0 disables the Watchdog.

A read of these bits indicates the current state of the Watchdog enable.

The watchdog enable bit resets to 0 on watchdog Cold reset.

A.4.2 Watchdog Interface Identification Register

W_IIDR is a 32-bit read-only register. The format of the register is:

ProductID, bits [31:20]

An IMPLEMENTATION DEFINED product identifier.

Architecture version, bits [19:16]

Revision field for the Generic Watchdog architecture. The value of this field depends on the Generic Watchdog architecture version:

- 0x1 for Generic Watchdog v1.

Revision, bits [15:12]

An IMPLEMENTATION DEFINED revision number for the component.

Implementer, bits [11:0]

Contains the JEP106 code of the company that implemented the Generic Watchdog:

Bits [11:8] The JEP106 continuation code of the implementer.

Bit [7] Always 0.

Bits [6:0] The JEP106 identity code of the implementer.

B GENERIC UART

B.1 About

This specification of the ARM generic UART is designed to offer a basic facility for software bring up and as such specifies the registers and behavior required for system software to use the UART to receive and transmit data. This specification does not cover registers needed to configure the UART as these are considered hardware-specific and will be set up by hardware-specific software. This specification does not cover the physical interface of the UART to the outside world, as this is system specific.

The registers specified in this specification are a subset of the ARM PL011 r1p5 UART. An instance of the PL011 r1p5 UART will be compliant with this specification.

An implementation of the Generic UART must provide a transmit FIFO and a receive FIFO. Both FIFOs must have the same number of entries, and this number must be at least 32. The generic UART does not support DMA Features, Modem control features, Hardware flow control features, or IrDA SIR features.

The generic UART uses 8-bit words, equivalent to `UARTLCR_H.WLEN == b11`.

The basic use model for the FIFO allows software polling to manage flow, but this specification also requires an interrupt from the UART to allow for interrupt-driven use of the UART.

Table 15 identifies the minimum register set used for SW management of the UART.

B.2 Generic UART register frame

The Generic UART is specified as a set of 32-bit registers. However it is required that implementations support accesses to these registers using read and writes accesses of various sizes. The required access sizes are included in Table 15. The base address of each access, independent of access size, must be the same as the base address of the register being accessed.

If an access size not listed in the table is used, the results are IMPLEMENTATION DEFINED.

The Generic UART is little-endian.

Table 15: Base UART Register Set

Offset	Name	Description	Permitted access sizes/bits
0x000-0x003	UARTDR Data Register	A 32-bit read/write register. Bits [7:0] An 8-bit data register used to access the Tx and Rx FIFOs. Bits [11:8] 4 bits of error status used to detect frame errors. Read-only. Bits [31:12] - Reserved. (Ref Section 3.3.1 - PL011TRM [8])	Read: 16,32 Write: 8,16,32

Offset	Name	Description	Permitted access sizes/bits
0x004-0x007	UARTRSR/UARTECR Receive status and error clear register	A 32-bit read/write register – a write clears the bits Bits [3:0] Four bits of error status, used to detect frame errors as in the UARTDR register, except it allows clearing of these bits. Bits [31:4] - Reserved. (Ref Section 3.3.2 - PL011TRM [8])	Read: 16,32 Write: 8,16,32
0x018-0x01c	UARTFR Flag Register	A 32-bit read-only register. Bits [2:0] - Reserved Bits [7:3] Bits indicate state of UART and FIFOs, with operation as PL011. Bits [15:8] - Reserved. (Ref Section 3.3.3 - PL011TRM [8])	Read: 8, 16, 32
0x03c-0x03f	UARTRIS Raw Interrupt Status Register	A 32 bit read-only register. Bits [3:0] - Reserved. Bits [10:4] Bits used indicate state of Interrupts. Bits [31:11] - Reserved. (Ref Section 3.3.11 - PL011TRM [8])	Read: 16, 32
0x040-0x043	UARTMIS Masked Interrupt Status Register	A 32 bit read-only register. Bits [3:0] - Reserved. Bits [10:4] Bits used indicate state of Interrupts. Bits [31:11] - Reserved. (Ref Section 3.3.12 - PL011TRM [8])	Read: 16, 32

Offset	Name	Description	Permitted access sizes/bits
0x038-0x03b	UARTIMSC Interrupt the current mask status	A 32-bit read/write register showing the current mask status. Bits [3:0] Write as Ones. Bits[10:4] Bits used to set or clear the mask bits assigned to the corresponding interrupts: 1 = mask 0 = unmask. Bits [31:11] Reserved, preserve value. (Ref Section 3.3.10 - PL011TRM [8])	Read: 16, 32 Write: 16, 32
0x044-0x047	UARTICR Interrupt Clear Register	A 32-bit write-only register. Bits[3:0] Reserved Bits [10:4] Bits used to clear the interrupts whose status is indicated in UARTRIS. Bits[31:11] - Reserved (Ref Section 3.3.13 - PL011TRM [8])	Write: 16, 32

B.3 Interrupts

The UARTINTR interrupt output must be connected to the GIC.

B.4 Control and setup

Hardware-specific software is required to set up the UART into a state where the above specification can be met and the UART can be used.

This setup is equivalent to the following PL011 state:

```

UARTLCR_H.WLEN == b11 // 8-bit word
UARTLCR_H.FEN == b1 // FIFO enabled
UARTCR.RXE == b1 // receive enabled
UARTCR.TXE == b1 // transmit enabled
UARTCR.UARTEN == b1 // UART enabled

```

B.5 Operation

The base UART operation complies with the subset of features implemented of the PL011 PrimeCell UART, the operation of which can be found in sections 2.4.1, 2.4.2, 2.4.3, and 2.4.5 of the ARM® PrimeCell® UART

(PL011) Technical Reference Manual [8]. Operations of the IrDA SIR, modem, hardware flow control, and DMA are not supported.

C PERMITTED ARCHITECTURAL DIFFERENCE BETWEEN PEs

Table 16 shows the permitted differences in architected registers between PEs in a single base server system. The permitted differences column lists the bit fields for a register that can vary from PE to PE. Where a bit field is not listed, the value must be the same across all PEs in the system.

Table 16: Permitted architectural differences

Description	Short-Form	Permitted Differences
AArch64 Memory Features Register	ID_AA64MMFR0_EL1	Bits [3:0] describing the supported physical address range.
Main ID Register	MIDR_EL1	Part number [15:4], Revision [3:0], Variant [23:20].
Virtualization Processor ID Register	VPIDR_EL2	Same fields as MIDR_EL1, writable by hypervisor.
Multiprocessor ID Register	MPIDR_EL1	Bits [39:32] and Bits [24:0]. Affinity fields and MT bit.
Virtualization Multiprocessor ID Register	VMPIDR_EL2	Same fields as MPIDR, writable by hypervisor.
Cache type register	CTR_EL0	Bits [15:14] Level 1 Instruction Cache Policy.
Revision ID Register	REVIDR_EL1	Specific to implementation indicates implementation specific Revisions/ECOs. All bits can vary.
Cache level ID register	CLIDR_EL1	All bits, each PE can have a unique cache hierarchy.
Cache Size ID Register	CCSIDR_EL1	Sets [27:13], Data cache associativity [12:3]. Caches on different PEs can be different sizes.
Auxiliary Control Register	ACTLR_EL{1,2,3}	Specific to implementation, all bits can vary.
Auxiliary Fault Status Registers	AFSR{0,1}_EL{1,2,3}	Specific to implementation, all bits can vary.

Note

Not all SBSA compliant operating software supports heterogeneous PEs. SoC designers should factor the level of support in operating software for a SoC's target market into their design decisions.

D PCI EXPRESS INTEGRATION

D.1 Configuration space

Systems must map memory space to PCI Express configuration space, using the PCI Express Enhanced Configuration Access Mechanism (ECAM). For more information about ECAM, see *PCI Express Base Specification Revision 3.1*[1].

The ECAM maps configuration space to a contiguous region of memory address space, using bit slices of the memory address to map on to the PCI Express configuration space address fields. This mapping is shown in Table 17.

Table 17: Enhanced Configuration Address Mapping

Memory Address bits	PCI Express Configuration Space address field
(20 + n - 1):20	Bus Number 1 ≤ n ≤ 8.
19:15	Device Number.
14:12	Function Number.
11:8	Extended Register Number.
7:2	Register Number.
1:0	Byte.

The system can implement multiple ECAM regions.

The base address of each ECAM region within the system memory map is IMPLEMENTATION DEFINED and is expected to be discoverable from system firmware data.

It is system-specific whether a system supports non-PE agents accessing ECAM regions.

Note

Alternative Routing-ID Interpretation (ARI) is permitted. For buses with an ARI device the ECAM field [19:12] is interpreted as the 8-bit function number.

D.2 PCI Express Memory Space

It is system-specific whether a system supports mapping PCI Express memory space as cacheable.

All systems must support mapping PCI Express memory space as either device memory or non-cacheable memory. When PCI Express memory space is mapped as normal memory, the system must support unaligned accesses to that region.

PCI Type 1 headers, used in PCI-to-PCI bridges, and therefore in root ports and switches, have to be programmed with the address space resources claimed by the given bridge. For *non-prefetchable* (NP) memory, Type 1 headers only support 32-bit addresses. This implies that endpoints on the other end of a PCI-to-PCI bridge only support 32-bit NP BARs. Systems compliant with SBSA Level 4 or above must support 32-bit programming of NP BARs on such endpoints. This can be achieved in two ways:

Method 1: PE physical address space can be reserved below 4G, whilst maintaining a one to one mapping between PE physical address space and NP memory address space.

Method 2: It is also possible to use a fixed offset translation scheme that creates a fixed offset indirection between PE physical address space, and PCI memory. This allows a window in PE physical address space that is above 4G to be mirrored in PCI memory space below 4G. This requires support in the PHB. Furthermore, firmware must program the PHB with the fixed offset, and to supply this information to the OS [4].

Arm recommends Method 1 as it eases peer-to-peer support in NP memory.

D.3 PCI Express device view of memory

Transactions originating from a PCI express device will either directly address the memory system of the base server system or be presented to a SMMU for optional address translation and permission policing.

For accesses from a PCIe endpoint to the host memory system, in systems compatible with SBSA Level 3 or above, the following must be true:

- The addresses sent by PCI express devices must be presented to the memory system or SMMU unmodified.
- In a system where the PCI express does not use an SMMU, the PCI express devices have the same view of physical memory as the PEs. In a system with a SMMU for PCI express there are no transformations to addresses being sent by PCI express devices before they are presented as an input address to the SMMU.

Note

For accesses from a PE to the PCI memory space in a PCIe endpoint, offset based translation is permissible, as described in Method 2 of section Section [D.2](#)

D.4 Message signaled interrupts

Support for Message Signaled Interrupts (MSI/MSI-X) is required for PCI Express devices. MSI and MSI-X are edge-triggered interrupts that are delivered as a memory write transaction.

The system must implement an interrupt controller compliant with the ARM Generic Interrupt Controller, each SBSA level specifies which version should be used.

Note

ARM introduced standard support for MSI(-X) in the GICv2m architecture [9], this support is extended in GICv3.

The intended use model is that each unique MSI(-X) must trigger an interrupt with a unique ID and the MSI(-X) must target GIC registers requiring no hardware specific software to service the interrupt.

D.5 GICv3 support for MSI(-X)

GICv3 adds a new class of interrupt, LPI, to address this. LPI can be targeted to a single PE.

In GICv3, SPI can be targeted at a single PE or can be “1 of N”, where the interrupt will be delivered to any one of the PEs in the system currently powered up.

In GICv3, SPI are still limited in scale, but an implementation can support thousands of LPis.

In GICv3, MSI(-X) can target SPI or LPI.

A single GICD_SETSPI_NSR register is supported for MSI targeting SPI. This is a compatibility break with GICv2m, and does not support I/O virtualization.

GICv3 provides the GITS_TRANSLATER register for MSI targeting LPI. This register uses a Device_ID to uniquely identify the originating device to fully support I/O virtualization, and is backed by memory-based tables to support flexible retargeting of interrupts.

D.6 Legacy interrupts

PCI Express legacy Interrupt messages must be converted to an SPI:

- Each of the four legacy interrupt lines, of all PCI root ports within a PCI host bridge , must be allocated a unique SPI ID.
- The exact SPI IDs that are allocated are IMPLEMENTATION DEFINED.
- Each legacy interrupt SPI must be programmed as level-sensitive in the appropriate GIC_ICFGR.

IMPLEMENTATION DEFINED registers must not be used to deliver these messages, only registers defined in the PCI Express specification and the ARM GIC specification.

D.7 I/O Virtualization

Hardware support for I/O Virtualization is optional, but if required must use a System MMU compliant with the ARM System MMU specification [10].

Each function, or virtual function, that requires hardware I/O virtualization is associated with a SMMU context. The programming of this association is IMPLEMENTATION DEFINED and is expected to be described by system firmware data.

SMMU does not support PCI Express ATS until SMMUv3, and as such ATS support is system-specific in systems that don not have a SMMUv3 or later.

Note

The Server Base System Architecture requires certain versions of the SMMU to be used at particular levels of the specification.

Functions intended for VM assignment, or assignment to a user space driver must implement function level reset.

D.8 I/O Coherency

PCI Express transactions not marked as No_snoop accessing memory that the PE translation tables attribute as cacheable and shared are I/O Coherent with the PEs.

The PCI Express root complex is in the same Inner Shareable domain as the PEs.

I/O Coherency fundamentally means that no software coherency management is required on the PEs for the PCI Express root complex, and therefore devices, to get a coherent view of the PE memory.

This means that if a PCI Express device is accessing cached memory then the transactions from the PCI Express devices will snoop the PE caches.

PCI Express also allows PCI Express devices to mark transactions as No_snoop. The memory accessed by such transactions must have coherency managed by software.

The following sections summarize the attributes the transactions from the PCI Express root complex must have and how coherency is maintained.

D.8.1 PCI Express I/O Coherency without System MMU

In the case where there is not a System MMU translating transactions from the root complex, the system must be able to distinguish between addresses that are targeted at memory and devices. Transactions that are targeted at devices must be treated as device type accesses. They must be ordered, must not merge and must not allocate in caches. Transactions that are targeted at memory and that are marked No Snoop must be presented to the memory system as non-cached. Transactions that are targeted at memory and not marked as No_snoop must be presented as cached, shared.

The following table shows how coherency is managed for PCI Express transactions. If a memory page is marked as non-cached in the PE translation tables, all PCI Express transactions accessing that memory must be marked as No_snoop. Failure to do so can result in loss of coherency.

Table 18: PCI Express transaction types and I/O coherency

PE page table attribute	PCI Express transaction type	PCI Express transaction memory attributes	Coherency management
Cacheable, shared	Snoop	Cacheable, shared	Hardware
	No_snoop	Non-cached	Software
Cacheable, non-shared	Snoop	Cacheable, shared	Software
	No_snoop	Non-cached	Software
Non-cached	Snoop	Not allowed	Not allowed
	No_snoop	Non-cached	Hardware

D.8.2 PCI Express I/O Coherency with System MMU

In the case where the system has a System MMU translating and attributing the transactions from the root complex, the PCI Express transactions must keep the memory attributes assigned by the System MMU. If the System MMU-assigned attribute is cacheable then it is IMPLEMENTATION DEFINED if No_snoop transactions replace the attribute with non-cached.

D.9 Legacy I/O

The specification does not require a support for legacy I/O transactions. If an implementation supports legacy I/O, it is supported using a one to one mapping between legacy I/O space and a window in the host physical address space. However, such schemes must not require a kernel driver to be set up, any necessary initialization must be performed prior to OS boot.

D.10 Integrated end points

Feedback from OS vendors has indicated that they have seen many 'almost PCI Express' integrated endpoints. This leads to a bad experience and either no OS support for the endpoint or painful bespoke support.

Anything claiming to follow the PCI Express specification must follow all the specification that is software-visible to ensure standard, quality software support.

D.11 Peer-to-peer

It is system-specific whether peer-to-peer traffic through the system is supported.

Systems compatible with level 3 or above of the SBSA must not deadlock if PCI express devices attempt peer-to-peer transactions – even if the system does not support peer-to-peer traffic. This rule is needed to uphold the principle that a virtual machine and its assigned devices should not deadlock the system for other virtual machines or the hypervisor.

In a system where the PCIe hierarchy allows peer-to-peer transactions, the root ports in an ARM based SoC must implement PCIe access control service (ACS) features.

For Root ports this means that the following must be supported:

1. ACS Source Validation. (V)
2. ACS Translation Blocking. (B)
3. ACS P2P Request Redirect (R).
4. ACS P2P Completion Redirect (C).
5. ACS Upstream Forwarding (U).
6. The root port must support redirected request validation by querying an ARM architecture compliant SMMU to get the final target physical address and access permission information.
7. The root port must support ACS violation error detection, logging and reporting. Logging and reporting must be through the usage of AER mechanism.

ACS P2P egress control is optional.

If the Root port supports peer-to-peer traffic with other root ports, then it must support the following:

1. Validation of the peer-to-peer transactions prior to sending it to the destination root port using the same mechanism as ACS redirected request validation. Any ACS violation error generated because of the request validation should be reported using the standard ACS violation error detection, logging and reporting mechanism specified in PCIe specification.
2. If the root port supports Address Translation services and peer-to-peer traffic with other root ports, then it must support ACS direct translated P2P (T).

Since isolation between IO devices can be broken by the presence of any peer-to-peer capable entity in a PCIe hierarchy, ARM strongly recommends the following for an ARM based system:

- All PCIe switches should support the following features:
 1. ACS Source Validation (V).
 2. ACS Translation Blocking (B).
 3. ACS P2P Request Redirect (R).
 4. ACS P2P Completion Redirect (C).
 5. ACS Upstream Forwarding (U).
 6. ACS Direct Translated P2P (T).
 7. ACS violation error detection, logging and reporting as specified in PCIe specification for ACS.
 8. Use of AER capability for logging and reporting ACS violation errors
- All multi-function devices, SR-IOV and non-SR-IOV, that are capable of peer-to-peer traffic between different functions should support the following features:

1. ACS P2P Request Redirect (R).
2. ACS P2P Completion Redirect (C).
3. ACS Direct Translated P2P (T)
4. The device must support ACS violation error detection, logging and reporting as specified in PCIe specification for ACS.

D.12 PASID support

SMMUv3 included optional support for PCIe PASID. If the system supports PCIe PASID, then at least 16 bits of PASID must be supported. This support must be full system support, from the root complex through to the SMMUv3 and any end points for which PASID support is required.

D.13 PCIe Precision Time Measurement

Any system that implements PCIe *Precision Time Measurement* (PTM) [1] must use the ARM architecture defined System Counter [3] as PTM master time source at the PTM root(s).

E SMMUV3 INTEGRATION

This appendix details rules about the integration of a SMMUv3 SMMU into an SBSA system.

The system is permitted to include any number of SMMUs.

All SMMU translation table walks and all SMMU accesses to SMMU memory structures and queues are I/O coherent (SMMU_IDR0.COHAAC == 1).

SMMUv3 supports two distinct page table fault models: stall on fault, and terminate on fault. Care must be taken when designing a system to use the stall on fault model.

The system must be constructed so the act of the SMMU stalling on a fault from a device must not stall the progress of any other device or PE that is not under the control of the same operating system as the stalling device.

The SMMUv3 spec requires that PCIe root complex must not use the stall model due to potential deadlock.

See Section [D.12](#) for requirements on PCIe PASID support.

See Section [F](#) for requirements on how DeviceID and StreamID should be assigned and how ITS groups should be used.

F DEVICEID GENERATION AND ITS GROUPS

F.1 ITS groups

F.1.1 Introduction

Every ITS block in the system is a member of a logical ITS group. Devices that send MSIs are also associated with an ITS group. Devices are only programmed to send MSIs to an ITS in their group. In the simplest case, the system contains one ITS group which contains all devices and ITS blocks. Devices are assigned DeviceID values within each ITS group. See Section F.2.

Note

The concept of ITS grouping means the system does not have to support the use of any ITS block from any device, which can ease system design.

F.1.2 Rules

The system contains one or more ITS group(s).

- An ITS group can contain one or more ITS blocks.
- An ITS block is associated with one ITS group.
- A device that is expected to send an MSI is associated with one ITS group.
- Devices can be programmed to send MSIs to any ITS block within the group.
- If a device sends an MSI to an ITS block outside of its assigned group, the MSI write is illegal and does not trigger an interrupt that could appear to originate from a different device. See Section F.2.2 for permitted behavior of illegal MSI writes.
- The association of devices and ITS blocks to ITS groups is considered static by high-level software.
- An ITS group represents a DeviceID namespace independent of any other ITS group.
- All ITS blocks within an ITS group support a common DeviceID namespace size, a common input EventID namespace size and are capable of receiving an MSI from any device within the group.
- All ITS blocks within an ITS group observe the same DeviceID for any given device in the same ITS group.

Note

The two preceding rules, allow software to use ITS blocks sharing a common group interchangeably.

- System firmware data, for example, firmware tables such as ACPI/FDT, describe the association of ITS blocks and devices with ITS groups to high-level software.

ARM recommends that the DeviceID namespace in each group is packed as densely as possible, and that it starts at 0.

Note

It is not required that all DeviceIDs be entirely contiguous but excessive fragmentation makes the software configuration of an ITS more difficult.

F.1.3 Examples of ITS groups

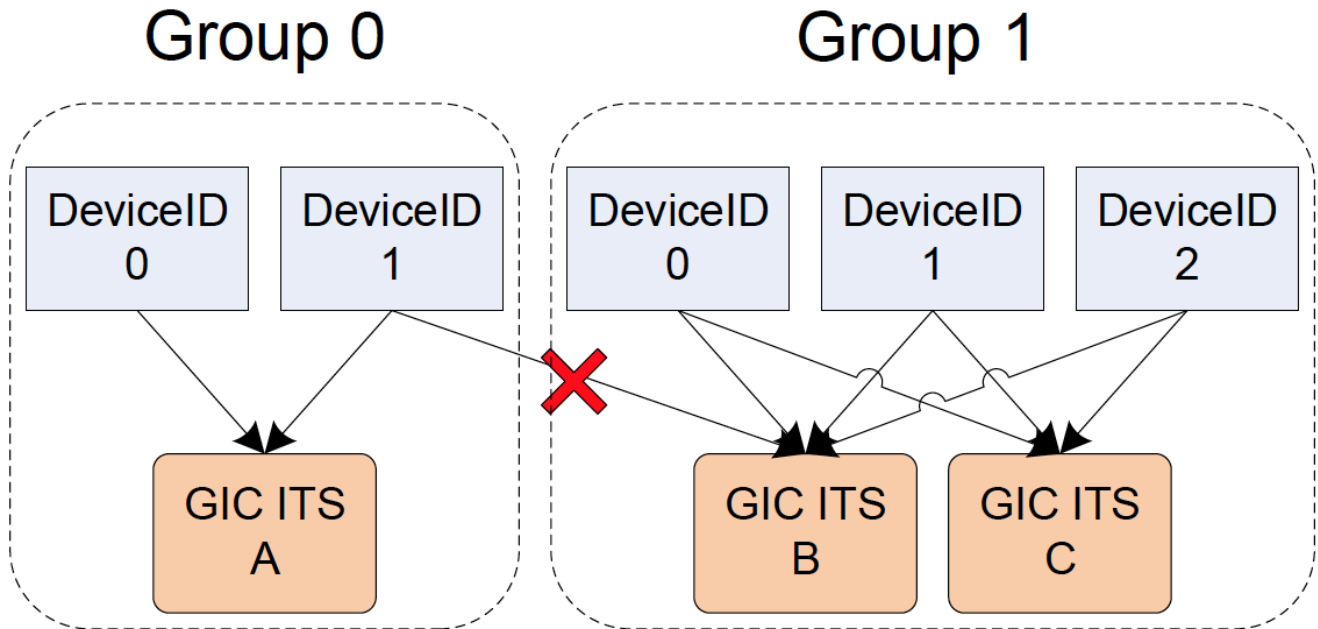


Figure 2: Device and ITS grouping

In Figure 2:

- ITS A serves two devices.
- ITS B and ITS C serve three devices; any of these devices can send an MSI to either of B or C.
- Two unrelated DeviceID namespaces exist. DeviceID 0 in Group 0 is different to DeviceID 0 in Group 1.
- A device in Group 0 can only trigger an MSI on its assigned ITS, A, and should not be configured to do otherwise. It cannot send an MSI to ITS B as it is in a different group to the device. If this is done, the MSI write might be ignored or aborted, but in any case does not cause an interrupt that might appear to be valid.

The properties that system-description structures convey to high-level software are:

- Identification of the two devices that are associated with Group 0, and the three associated with Group 1.
- ITS block A is in Group 0, B and C are in Group 1.
- For each MSI-capable master device, which DeviceID in the group's namespace the device has been assigned.

F.2 Generation of DeviceID values

F.2.1 Introduction

Every device master that is expected to send MSIs has a DeviceID associated with it. This DeviceID is used to program the interrupt properties of MSIs originating from each device. The term “device” is used in the context of a logical programming interface used by one body of software.

Where a device is a client of an SMMU, that is, behind an SMMU, a granularity of ‘source’ identification is typically chosen distinguish the client device’s traffic from other clients. This allows the device to be assigned

to a less-privileged software independent of other SMMU client devices. The system designer assigns a master-unique StreamID to device traffic input to the SMMU. The simplest way to achieve the same granularity of interrupt source differentiation and SMMU DMA differentiation is for the device's DeviceID to be generated from the device's SMMU StreamID. It is beneficial for high-level software and firmware system descriptions to ensure that this relationship is as simple as possible. DeviceID is derived from a StreamID 1:1 or with a simple linear offset.

When a device is not behind an SMMU, its DeviceID appears to high-level software as though it is assigned directly by the system designer.

If a master is a bridge from a different interconnect with an originator ID, such as a PCIe RequesterID, and devices on that interconnect might need to send MSIs, the originator ID is used to generate a DeviceID. The function to generate the DeviceID should be an identity or a simple offset.

The overall principle of DeviceID and StreamID mapping is that the relationship between one ID space, for example, a PCIe RequesterID namespace and a DeviceID, be easily described using linear span-and-offset operations.

When an SMMU is used to allow devices to be programmed by possibly malicious software that is not the most privileged part of the system, devices that are not designed to directly trigger MSIs could be misused to direct a DMA write transaction at an ITS MSI target register. The system must not allow this behavior to trigger an MSI that masquerades as originating from a different master. The system must anticipate that PEs also have the potential to be misused in this manner. Exposing an ITS to a VM for legitimate MSI purposes can mean that the untrusted VM software is able to write to the ITS MSI target register from a PE.

F.2.2 Rules

- Every device that is expected to originate MSIs is associated with a DeviceID.
- DeviceID arrangement and system design prevents any mechanism that any software that is not the most privileged in the system, for example VM, or application, can exploit to trigger interrupts associated with a different body of software, for example. a different VM, or OS driver.
 - A write to an ITS GITS_TRANSLATER from a PE, or from a device that is known at design time to not support genuine MSIs and is under control of software less privileged than the software controlling the ITS, is an illegal MSI write and must not be able to trigger an MSI appearing to have a DeviceID associated with a different device. See Section F.1.2, an MSI sent to an ITS in a group different to the originating device is also an illegal MSI write.
 - An illegal MSI write is permitted to complete with WI semantics, or be terminated with an abort, or trigger an MSI having a DeviceID that does not alias any DeviceID of a legitimate source.

Note

Devices that are known at design time to only be controlled by the most privileged software in the system, such as those without an MMU/MPU, can be trusted not to send malicious writes to the ITS. For these devices no special steps are required to prevent malicious MSI writes. Devices that have the potential to be controlled by a VM cannot be trusted. Devices that are clients of an SMMU fall into the latter category.

- If a device is a client of an SMMU, the associated DeviceID is derived from the SMMU's StreamID with an identity or simple offset function:
 - The SMMU component must output the input StreamID unmodified so it can be used to derive the DeviceID downstream of the SMMU.
 - If two devices have different StreamIDs, they must also have distinct DeviceIDs.
 - * It is not permitted for >1 StreamID to be associated with 1 DeviceID.
 - * It is not permitted for >1 DeviceID to be associated with 1 StreamID.
 - The generic StreamID to DeviceID relationship is:
 - * `DeviceID = zero_extend(SMMU_StreamID) + Constant_Offset_A`

- A PCIe Root Complex behind an SMMU generates a StreamID on that SMMU from its RequesterID with this relationship:
 - * $StreamID = zero_extend(RequesterID[N-1:0]) + (1 \ll N) * Constant_B$

Note

ARM expects N above to be 16 bits, but this is not mandatory.

- * This StreamID is then used post-SMMU, as above, to generate a DeviceID.
- DeviceIDs derived from other kinds of system IDs are also created from an identity or simple offset function.
 - For a Root Complex without an SMMU, the relationship is:
 - * $DeviceID = zero_extend(RequesterID[N-1:0]) + (1 \ll N) * Constant_C$
- The relationships between a device, its StreamID and its DeviceID are considered static by OS or hypervisor software. If the mapping is not fixed by hardware, the relationship between a StreamID and a DeviceID must not change after system initialization, and OS drivers must not be required to set it up.

ARM recommends that all devices expected to originate MSIs have a DeviceID unique to their ITS group, even if the devices are not connected to an SMMU.

Note

Providing separate DeviceIDs for different devices can improve the efficiency of structure allocation in GIC driver software.

F.3 System description of DeviceID and ITS groups from ACPI tables

The properties of the GIC distributor, Redistributors and ITS blocks such as base addresses will be described to high-level software by system firmware data. In addition, for any given device expected to send MSIs, system firmware data tables must ensure that:

- The device's DeviceID can be determined, either:
 - Directly: A device is labeled with a DeviceID value.
 - Hierarchically indirect: If a device has a known ID on a sub-interconnect, the transformation between that interconnect's ID and the DeviceID namespace is described in a manner that allows the DeviceID to be derived. This might comprise multiple transformations ascending a hierarchy, where a device is associated with intermediate IDs (such as a StreamID) which are ultimately used to generate a DeviceID.
 - * Example: A PCIe Root Complex without an SMMU is described in terms of the DeviceID range output for its RequesterID range. The DeviceID of an endpoint served by the Root Complex is not directly provided, but is derived from the endpoint's RequesterID given the described mapping.
 - * Example: A PCIe Root Complex with an SMMU is described in terms of the transformation of RequesterID range to SMMU input StreamID range and the transformation of StreamID range to DeviceID range.
- The device's association with an ITS group can be determined and the ITS blocks within the group can be enumerated.

Note

- More compact descriptions result by describing a range of DeviceIDs to allow DeviceIDs to be derived

from a formula instead of directly describing individual DeviceIDs. This is especially pertinent for interconnects such as PCIe.

- PEs and other masters that do not support MSIs are not described as being part of an ITS group; as they are not intended to invoke valid MSIs, there is no association to an ITS on which it is valid to invoke MSIs.

The DeviceID and ITS group associations are not expected to be discoverable through a programming interface of hardware components and a system is not required to provide such an interface.

F.4 DeviceIDs from hot-plugged devices

- If a device is not physically present at system initialization time, values in the DeviceID namespace appropriate to the potential physical location of future devices must be reserved and associated with the device when it later becomes present, in a system-specific manner.
- When a device is hot-plugged, it can be enumerated using an interconnect ID whose mapping to DeviceID was statically described in system description tables and its DeviceID derived from this existing mapping.
- If a new device's DeviceID cannot be derived from existing mappings in system description tables, the hot-plug mechanism, for example via firmware, must provide a means to determine the new device's DeviceID.

Note

- These points also apply to a new device's SMMU StreamID.
- In current systems, hot-plug device masters that are capable of sending MSIs are most likely to be PCIe endpoints. When a system and PCIe-specific mechanism makes a new endpoint present, the existing indirect description of the Root Complex's DeviceID span is used to calculate the new DeviceID from the new RequesterID.

ARM recommends that description of a sub-interconnect bridge, such as a PCIe Root Complex, includes all potential endpoints (on PCIe, up to 2^{16}) rather than limiting description to the endpoints present at boot time, if more client endpoints can later become present.