Server Base System Architecture

Document number:	ARM-DEN-0029 Version 2.3
Date of Issue:	27th March 2014
Author:	Architecture and Technology Group
Confidentiality:	Non-Confidential

Proprietary Notice

Contents

1 ABOUT THIS DOCUMENT

1.1 References

This document refers to the following documents.

Ref	Doc No	Author(s)	Title
[1]	ARM DDI 0406		ARM Architecture Reference Manual Issue C
[2]	ARM IHI 0048		ARM Generic Interrupt Controller Specification
[3]	ARM DDI 0183		PrimeCell UART (PL011) TRM
[4]	ARM IHI 0067		ARM System Memory Management Unit Architecture Specification, 64KB Translation Granule Supplement
[5]	ARM IHI 0062B		ARM System Memory Management Unit Architecture Specification
[6]	ARM DDI 0487		Architecture Reference Manual ARMv8, for the ARMv8-A architecture profile

1.2 Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
Base Server System	A system compliant with the Server Base System Architecture
SBSA	Server Base System Architecture
GIC	Generic Interrupt Controller
VM	Virtual Machine
PMU	Performance Monitor Unit
IO Coherent	A device is IO Coherent with the CPU caches if its transactions snoop the CPU caches for cacheable regions of memory. The CPU does not snoop the device’s cache.
SGI	Software Generated Interrupt
SPI	Shared Peripheral Interrupt
PPI	Private Peripheral Interrupt
LPI	Locality-specific Peripheral Interrupt (GICv3)
SRE	System Register interface Enable (GICv3)
ARE	Affinity Routing Enable (GICv3)
System firmware data	System description data structures such as ACPI or FDT

1.3 Feedback

ARM welcomes feedback on its documentation.

1.3.1 Feedback on this manual

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

- The title.
- The document and version number, ARM-DEN-0029 v2.1.
- 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 of the needs of the markets.

The ARM architecture profiles – **A**pplication, **R**eal-time and **M**icrocontroller – exist in part to help segment the solutions produced by ARM and to describe the characteristics of particular target markets. The differences between products targeted at different profiles are typically substantial due to the very different functional requirements of their market segments.

However, even within an architectural profile, the wide range of uses of a product means there are frequently requests for features to be removed to save silicon area. This is particularly the case for products targeted at cost-sensitive markets, where the cost of customizing the software to allow for the lack of a feature is small compared to the overall cost saving of removing features.

In other markets, such as those which require an open platform where the software to be run is very complex, the savings from removing a feature in the hardware 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 in whether new features are widely deployed can act as 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 Floating-point, Advanced SIMD, and TrustZone, to tackle an increasing range of problems, while allowing features to be removed from implementations that do not need them and where the silicon area, and cost savings involved are compelling.

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 CPU features and certain key aspects of the system result in significant cost to software system development and associated quality risk.

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

3 INTRODUCTION

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

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

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

This specification embeds the notion of levels of functionality. Level 0 is the first level; level 1 adds functionality on top of level 0 and so on. Unless explicitly stated all specification items belonging to level N apply to levels greater than N.

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 specific to the functionality of that system. 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*.

This specification uses the phrase *software consistent with the Server Base System Architecture* to indicate software that is designed to be fully portable between different implementations that are consistent with the Server Base System Architecture. Software consistent with the Server Base System Architecture shall not depend on the presence of hardware features not mandated in this specification.

4 SBSA

4.1 Level 0

4.1.1 CPU Architecture

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

CPUs in the base server system shall be compliant with ARMv8 where the following is true:

- The number of CPUs in the system shall not exceed eight.

  Note: This restriction is due to GICv2 limitations and will be lifted in a future level.
- CPUs shall implement Advanced SIMD extensions.
- It is implementation defined as to whether the Instruction Caches are implemented as VIPT or PIPT.

  Note: Not all CPUs are required to support the same Instruction Cache addressing scheme
- CPUs shall implement 16-bit ASID support.
- CPUs shall support 4KB and 64KB translation granules at stage1 and stage2
- All CPUs will be coherent and in the same inner shareable domain.
- Where export restrictions allow, CPUs should implement cryptography extensions.
- CPUs shall implement little-endian support.
- CPUs shall implement EL2.
- CPUs shall implement AArch64 at all exception levels.
- The PMU overflow signal from each CPU must be wired to a unique PPI or SPI interrupt with no intervening logic.
- Each CPU shall implement a minimum of four programmable PMU counters.
- Each CPU shall implement a minimum of four synchronous watchpoints.
- Each CPU shall implement a minimum of four breakpoints, two of which must be able to match virtual address, contextID or VMID.
- All CPUs shall be architecturally symmetric except for the permissible exceptions laid out in APPENDIX C: *Permitted Architectural Difference between CPUs*.

Note: It is consistent with this specification to implement CPUs with EL3 and with support for AArch32.

4.1.2 Interrupt Controller

The base server system shall implement a GICv2 interrupt controller.

The system shall implement at least eight non-secure Software Generated Interrupts, assigned to interrupt IDs 0-7.

4.1.3 Memory Map

This specification does not mandate a standard memory map. It is expected that the system memory map will be 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 shall ensure that all memory and peripherals can be mapped using 64KB stage2 pages and as such must not require the use of 4KB pages at stage2. 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, shall respond within finite time, so as to avoid the possibility of system deadlock. Where a memory access is to an unpopulated part of the addressable memory space, accesses must be terminated in a manner that is either presented to the CPU as either a precise data abort, or causes a system error interrupt, or causes an SPI interrupt to be delivered to the GIC.

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

Note: The ARM implementation of GICv2, the GIC-400 product, needs special address bus wiring to make it work in a 64KB translation granule system. See Appendix F: *GIC-400 and 64KB Translation Granule*.

4.1.4 IO Virtualisation

It is implementation-specific as to whether any given device in a base server architecture system supports the ability to be hardware virtualised. It is expected that those devices that can be hardware virtualised have that property expressed by system firmware data.

If a device is virtualised and passed through to an operating system under a hypervisor then the device's memory transactions must be subject to stage2 translation, allocation of memory attributes and application of permission checks, under the control of the hypervisor. This specification will collectively refer to this translation, attribution and permission checking as policing. The act of this policing will be referred to as stage2 system MMU functionality.

This stage2 system MMU functionality must be provided by a system MMU compatible with the ARM SMMUv1 with support for a 64KB translation granule specification, where:

- Support for stage1 policing is not required.

Note: Support for broadcast TLB maintenance operations is not required, and as such compliant software must maintain the MMU TLBs using the software interface.

SMMUv1 with support for a 64KB translation granule does not have support for PCI Express ATS, thus support for PCI Express ATS will be system specific.

The base server system may 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 are to police.

Note: This is consistent with the ARM MMU-401 implementation. Software can either program stage2 system MMUs to use the same page tables as the CPU or build shadow page tables. Standard PCI Express ATS support is included in SMMUv3 which is introduced in a later level of this specification.

4.1.5 Clock and Timer Subsystem

The base server system shall include a Generic Timer as specified in the ARM ARM. A virtual timer is not required.

The System Counter (of the Generic Timer) shall run at a minimum frequency of 10MHz and maximum of 400MHz.

The architecture of the counter mandates that it shall be at least 56 bits in size, and at most 64 bits.

Note: The counter shall be sized and programmed to ensure that rollover never occurs in practical situations; the timers and watchdogs that use the counter as a timebase rely on the counter not rolling over.

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

The local CPU timers have a programmable count value that when expires generates a Private Peripheral Interrupt for the associated CPU.

The base server system shall implement a system-specific system wake-up timer that can be used when CPU Timers are powered down. On timer expiry the system wake up timer shall generate a level interrupt that shall be wired to the GIC as an SPI. Additionally the system wake up timer can be used to wake up CPUs (see 4.1.6)

4.1.6 Wake up 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 CPUs 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 may need to reprogram the power controller. Hardware-specific software is required to save and restore system state when entering and exiting 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 CPUs as wakeup interrupts. A wakeup interrupt is any interrupt that is any one of:

- An SPI that directly targets a CPU.
- An SGI.
- A PPI.

In addition, in order for an interrupt to be a wakeup interrupt it shall be enabled in the distributor. A CPU shall wake in response to a wakeup interrupt independent of the state of its CPSR interrupt mask bits (the A,I and F bits) and of the wake up interrupt priority.

Note: One would typically expect a wakeup signal to be exported from the GIC to the power controller to initiate the CPU wakeup.

Note: There are some power states where a CPU will not wake on an interrupt. It is the responsibility of system software to ensure there are no wakeup interrupts targeting a CPU entering these states.

The local CPU timers are an important source of interrupts and can be used to wake the CPU. However the local CPU timer may become powered down in some low-power states (because it may be in the same power domain as the CPU). In low-power states where the local CPU timer is powered down, system software can use an SGI from other running CPUs to wake the CPU, or can configure the system wakeup timer to send a wakeup interrupt to the CPU to wake it.

In some very deep low-power states the GIC will be powered down. To wake from these states this specification introduces another class of wakeup methods: always-on power domain wake events.

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

There may be other IMPLEMENTATION DEFINED always-on power domain wakeup events that can wake the CPUs from these deep low-power states, such as PCI Express wakeup events and Wake-on-LAN.
See 4.1.7 for a description of the power state semantics that the system must comply with.

4.1.7 Power State Semantics

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

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 the OS or hypervisor, or both, to be able to reason about wake up events and to know which timers will be available to wake the CPU, all CPUs must be in a state that is consistent with one of the semantics described in Table 1: CPU Power States and Table 2: Power State Semantics. Note that all CPUs 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.

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

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 want to support very low-power states where most system logic is powered down and system memory put into 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 shall go straight to the system power controller and not require use of the GIC to wake the CPUs. The system power controller is system-specific. When in a power state of semantic E, the system power controller shall wake an IMPLEMENTATION DEFINED CPU (or set of CPUs) when the system wakeup timer expires. Other system-specific events may also cause wakeup from this state, such as a PCI Express wakeup event. The events that will cause wake up from this state are expected to be discoverable from system firmware data.

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

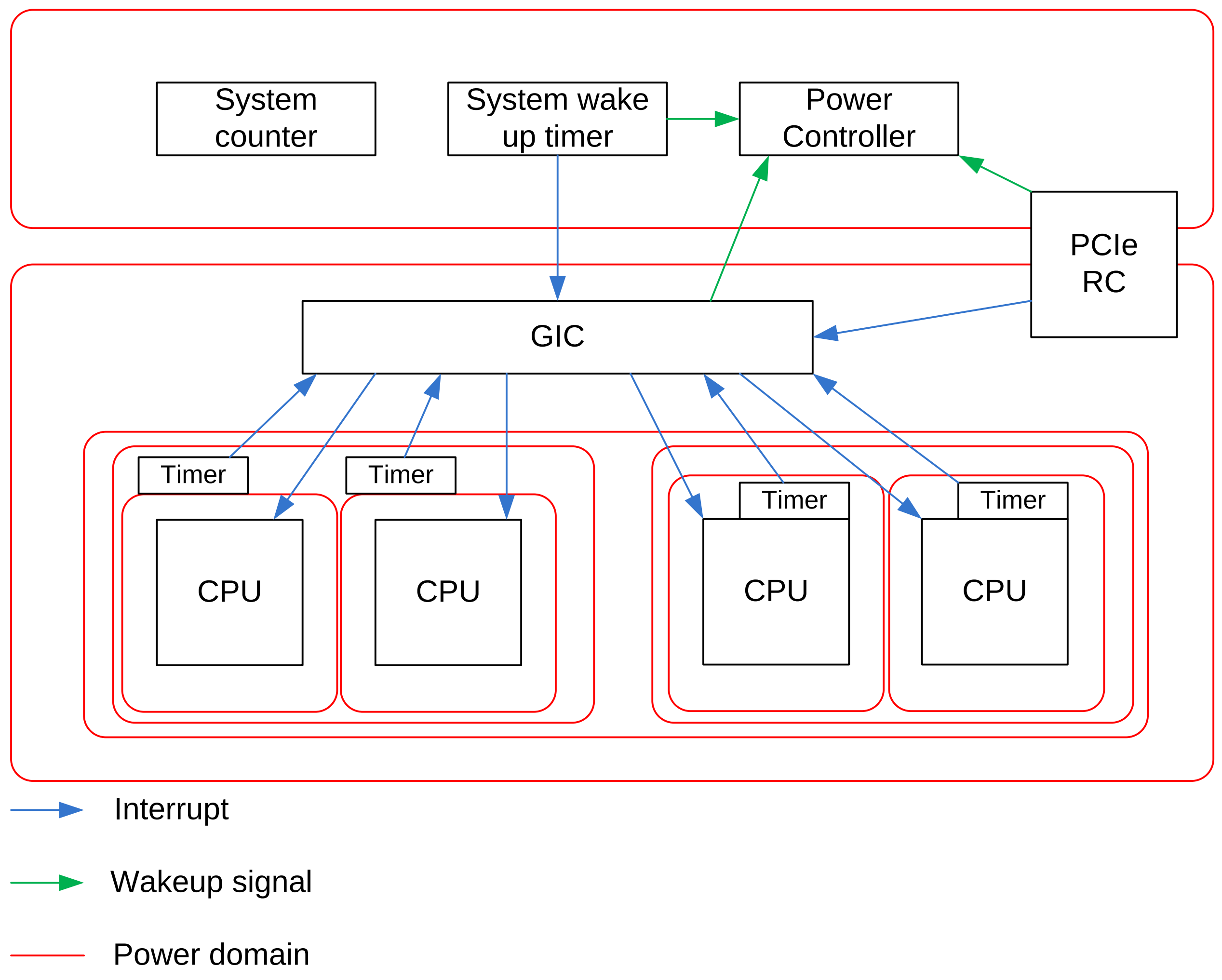

Figure 1: Example system block diagram showing power domains and timer hierarchy

CPU State	Description
Run	CPU is powered on and running code.
Idle_standby	CPU is in STANDBYWFI state, but remains powered. Full state retention, no state saving or restoration required. Execution automatically resumes after any interrupt or external debug request (EDBGRQ). Debug registers are accessible.
Idle_retention	CPU is in STANDBYWFI state, but remains powered. Full state retention, no state saving or restoration required. Execution automatically resumes after any interrupt or external debug request (EDBGRQ). Debug registers are not accessible.
Sleep	CPU is powered off but hardware will wake the CPU autonomously, e.g. on receiving a wake up interrupt. No CPU state retained. State must be explicitly saved. Woken CPU starts at the reset vector, and then hardware specific software will restore state.
Off	CPU is powered off and is not required to be woken by interrupts. The only way to wake the CPU is by explicitly requesting the power controller, for example from system software running on another CPU, or an external source such as a power_on_reset. This state can be used when the system software explicitly decides to remove the CPU from active service, giving the hardware opportunity for more aggressive power saving. No CPU state retained.

Table 1: CPU Power States

Semantic	CPU	CPU timers	GIC	System wake up timers and system counter	Note
A	Run	On	On	On	-
B	Idle	On	On	On	CPU will resume execution on receipt of any interrupt.
C	Sleep	On	On	On	CPU will wake on receipt of a wake up interrupt.
D	Sleep	Off	On	On	CPU will wake on receipt of a wake up interrupt, but local timer is off.
E	Sleep	Off	Off	On	CPU will wake from system timer wake up event or other system specific events.
F	Off	Off	On	On	Some, but not all CPUs are in Off state.
G	Off	Off	Off	Off	All CPUs in Off state.

Table 2: Power State Semantics

4.1.8 Peripheral Subsystems

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

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

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

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

4.2 Level 1

4.2.1 CPU Architecture

In addition to the level 0 requirements the following shall be true of the CPUs in the base server system:

- Each CPU shall implement a minimum of six programmable PMU counters.
- Each CPU shall implement a minimum of four synchronous watchpoints.
- Each CPU shall implement a minimum of six breakpoints, two of which must be able to match virtual address, contextID or VMID.

4.2.2 Interrupt Controller

If the base server system includes PCI Express then the base server system shall implement a GICv2m interrupt controller. The system shall implement at least one non-secure MSI frame with a minimum of 32 SPIs. There is no requirement to support a secure MSI frame.

Note that a GICv2m interrupt controller is a GICv2 interrupt controller with additional register frames specified in the GICv2m specification (APPENDIX E: GICv2m Architecture) for standardised support of PCI Express MSI and MSI-X.

If the base server system does not include a PCI Express root complex then the base server system shall implement a GICv2 interrupt controller.

4.2.3 Clock and Timer Subsystem

Level0 of the SBSA requires a system specific system timer. Level1 of the SBSA supersedes that requirement and requires that there is a system wake up timer in the form of the memory mapped timer described in the ARMv8 ARM [6].

4.2.4 Watchdogs

The base server system shall implement a Generic Watchdog as specified in APPENDIX A: Generic Watchdog. Watchdog Signal 0 shall be routed as an SPI to the GIC and it is expected this will be configured as an EL2 interrupt, directly targeting a single CPU. The rising edge of the assertion of Watchdog Signal 1 shall cause all CPUs to warm reset at EL2. The watchdog state shall survive this reset event so that software can interrogate the watchdog as it comes out of reset.

Note: Only directly targeted SPI are required to wake a CPU (see section 4.2.5) so programming the watchdog SPI to be directly targeted ensures delivery of the interrupt independent of CPU power states. However it is possible to use a 1 of N SPI to deliver the interrupt as long as one of the target CPUs are guaranteed to be running.

Note: CPU debug logic is not reset on a CPU warm reset.

4.2.5 Requirements on power state semantics

The power state semantic table, Table 3, is extended to include the Generic Watchdog.

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

Table 3: Power State Semantics

4.2.6 Peripheral Subsystems

For the purpose of system development and bring up the base server system shall include a standard UART. The standard UART is specified in Appendix B.

If the system has a PCI Express root complex then it must comply with APPENDIX D: PCI Express Integration.

4.3 Level 2

4.3.1 CPU Architecture

The maximum number of CPUs is raised to 2^{28}. This reflects the maximum number of CPUs GICv3 can support.

1. The PMU overflow signal from each CPU must be wired to a unique PPI interrupt with no intervening logic

4.3.2 Interrupt Controller

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

A level2 base server system shall implement a GICv3 interrupt controller.

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

Note: It is expected that MSI and MIS-X will be mapped to LPI interrupts.

Note: 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 as such expects most systems to support SPI as well as LPI interrupts.

Note: The ARM PL011 UART requires a level interrupt as does a PCIe root complex for legacy interrupt support.

It is optional for the interrupt controller to include GICv2 and GICv2m backward compatibility (that is ARE may be implemented as RAO/WI for both security states and SRE as RAO/WI for all exception levels). Any system that doesn't include this compatibility will not be able to run level 0- or level 1-compliant software.

See Appendix G: GICv2m compatibility in a GICv3 system on page 41 for how backwards compatibility with GICv2m can be achieved in a GICv3 system.

Support for compatibility with level 0 and level 1 of this spec (and hence GICv2 and GICv2m) is deprecated.

Note: A level2 system running in backwards-compatible mode will only be able to use a maximum of 8 CPUs.

4.3.2.1 PPI assignments

A level2 base server system shall comply with the PPI mapping laid out in: Table 4 - 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.
27	Overflow interrupt from CNTV	Virtual timer interrupt.
26	Overflow interrupt from CNTHP	Hypervisor timer interrupt.
25	GIC Maintenance interrupt	The virtual CPU interface list register overflow interrupt.
24	CTIIRQ	CTI (Cross Trigger Interface) interrupt.
23	Performance Monitors Interrupt	Indicates an overflow condition in the performance monitors unit.
22	COMMIRQ	DCC (comms channel) interrupt.

Table 4 - PPI assignments

4.3.3 Memory Map

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 CPU as either a precise data abort or causes a system error interrupt or causes an SPI or LPI interrupt to be delivered to the GIC.

4.3.4 Requirements on power state semantics

GICv3 introduces a new class of interrupt: LPI. A CPU receiving any of the following types of interrupt shall wake up:

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

4.3.5 IO Virtualisation

Stage2 system MMU functionality must be provided by a system MMU compatible with the ARM SMMUv2 spec where:

- Support for stage1 policing is not required.

Note: Support for broadcast TLB maintenance operations is not required, and as such compliant software must maintain the MMU TLBs using the software interface.

Note: This behaviour is consistent with ARMs MMU-500 implementation.

4.3.6 Clock and Timer Subsystem

The timer expiry interrupt shall be presented to the GIC as either an SPI or LPI.

4.3.7 Wake up semantics

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

This interrupt must look 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 shall not be 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:

- Wake up event occurs (such as GPIO or wake-on-LAN).
- The power controller responds by powering on the necessary resources such as CPU, GIC and so on.
- CPU comes out of reset and system specific software restores state (including GIC).
- An interrupt is presented to the GIC representing the wakeup event (in many situations this might be exactly the same signal as the wake up event).
- The system must ensure that, by the time the system-specific restore software has handed control back to the OS or Hypervisor, the interrupt is pending in the GIC.

- The OS or Hypervisor can respond to the interrupt.

4.3.8 Watchdogs

The watchdog signal WS0 shall be presented to the GIC as SPI or an LPI interrupt.

5 APPENDIX A: GENERIC WATCHDOG

5.1 About

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

The Generic Watchdog has two register frames, one that contains the refresh register and one for control of the watchdog. This permits mapping the control frame to higher-privileged software than that required to refresh the watchdog.

5.2 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 may be. This is to facilitate the CPUs in the system going through a reset sequence while the watchdog retains its state so it can be examined once the CPUs 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 started.

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 count value as the timebase against which the decision to trigger an interrupt is made.

The Watchdog is based on a 64-bit compare value and comparator. Once 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.

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

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 as to 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 then WS0 is asserted.

If WS0 is asserted and a timeout refresh occurs then 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 allowed behaviour.

```
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[31:0])
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 shall be disabled when the System Counter is being updated, or the results are UNPREDICTABLE.

Note: the watchdog offset register is 32 bits wide. This gives a maximum watch period of around 10s at a system counter frequency of 400MHz. If a larger watch period is required then the compare value can be programmed directly into the compare value register.

5.3 Register summary

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

All registers have 32 bits. There are two register frames, one for a refresh register and the other containing the status and setup registers.

Table 5 shows the 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_IIDR	See Watchdog Interface Identification Register on page 25.
`0xFD0 – 0xFFC`	Identification registers	Read-only identification registers, see Identification registers on page 25.

Table 5 Refresh Frame

Table 6 shows the 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	Watchdog offset register. A Read/Write register containing the unsigned 32 bit watchdog countdown timer value.
`0x00C – 0x00F`	-	Reserved.
`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 Watchdog Interface Identification Register on page 25.
`0xFD0 – 0xFFC`	Identification registers	Read-only identification registers, see Identification registers on page 25.

Table 6 Watchdog Control Frame

5.4 Register descriptions

5.4.1 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] reflecting the status of WS1 and bit[1] reflecting 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 a 0 on watchdog cold reset.

5.4.2 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:

- 0x0 for Generic Watchdog v0

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.

5.4.3 Identification registers

This architecture specification defines the offsets `0xFD0` to `0xFFC` in the watchdog register maps as a read-only identification register space. Table 2-2 shows the architecturally-required implementation of the identification register space.

Offset	Name	Description
`0xFD0 – 0xFE4`	-	IMPLEMENTATION DEFINED registers
`0xFE8`	W_PIDR2	Peripheral ID2 Register
`0xFEC – 0xFFC`	-	IMPLEMENTATION DEFINED registers

Table 5-7 Watchdog identification registers

The assignment of this register space, and naming of registers in this space, is consistent with the ARM identification scheme for CoreLink and CoreSight components.

5.4.3.1 Peripheral ID2 Register

W_PIDR2 is a 32-bit read-only register.

The format of the register is:

Bits [31:8]

IMPLEMENTATION DEFINED.

ArchRev, bits [7:4]

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

- 0x0 for Generic Watchdog v0

Bits [3:0]

IMPLEMENTATION DEFINED.

6 APPENDIX B: GENERIC UART

6.1 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 behaviour 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, this is system specific.

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

The generic UART supports 32-entry separate transmit and receive byte FIFOs and 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 8 on page 28 identifies the minimum register set used for SW management of the UART.

Offset	Name	Description
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)
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)
0x018 – 0x01c	UARTFR – Flag Register	A 32-bit Read-only register. Bits[2:0] Reserved. Bits[7:3] Bits used indicate state of UART and FIFOs, with operation as PL011. Bits[15:8] Reserved. (Ref Section 3.3.3 – PL011TRM)
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)
0x038 – 0x03b	UARTIMSC – Interrupt Mask Set/Clear Register	A 32-bit Read/Write register showing the current mask status. Bits[3:0] Write as Ones. Bits[10:4] Bits used to set(1=mask) or clear(0=unmask) the mask bits assigned to the corresponding interrupts. Bits[31:11] Reserved, preserve value. (Ref Section 3.3.10 – PL011TRM)
0x044 – 0x047	UARTICR – Interrupt Clear Register	A 32-bit Read-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)

Table 8 Base UART Register Set

6.2 Interrupts

The UARTINTR interrupt output shall be connected as an SPI to the GIC.

6.3 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
```

6.4 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 Pl011 TRM. Operations of the IrDA SIR, modem, hardware flow control, and DMA are not supported.

7 APPENDIX C: PERMITTED ARCHITECTURAL DIFFERENCE BETWEEN CPUS

Table 9 shows the permitted differences in architected registers between CPUs in a single base server system. The permitted differences column calls out the bit fields for a register that may vary from CPU to CPU, where a bit field is not called out the value must be the same across all CPUs in the system.

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 may vary.
Cache level ID register	CLIDR_EL1	All bits, each CPU can have a unique cache hierarchy.
Cache Size ID Register	CCSIDR_EL1	Sets [27:13], Data cache associativity [12:3]. Caches on different CPUs can be different sizes.
Auxiliary Control Register	ACTLR_EL{1,2,3}	Specific to implementation, all bits may vary.
Auxiliary Fault Status Registers	AFSR{0,1}_EL{1,2,3}	Specific to implementation, all bits may vary.

Table 9 Permitted architectural differences

8 APPENDIX D: PCI EXPRESS INTEGRATION

8.1 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 3.0*.

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 10.

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.

Table 10 Enhanced Configuration Address Mapping

The system may 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 if a system supports non-CPU agents accessing ECAM regions.

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

8.2 PCI Express Memory Space

It is system-specific as to 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.

8.3 Message signalled interrupts

Support for Message Signalled 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 shall implement an interrupt controller compliant with the ARM Generic Interrupt Controller – GICv2m, GICv3 or later.

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

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

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

8.3.1 GICv2m support for MSI(-X)

GICv2m has the MSI_SETSPI_NS register to support MSI(-X). If IO virtualisation of PCI Express drivers is to be supported in a GICv2m system then multiple versions of the register can exist in different memory pages, allowing different virtual machines to see different registers. Each register targets a unique set of SPIs.

8.3.2 GICv3 support for MSI(-X)

GICv2m is limited in scalability in terms of CPU count, but also in the number of MSI(-X) that can be supported. The architectural maximum number of SPI is 1023.

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

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

In GICv3 SPI are still limited in scale, but an implementation may 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 IO virtualisation.

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 IO virtualisation, and is backed by memory-based tables to support flexible retargeting of interrupts.

8.4 Legacy interrupts

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

- Each of the 4 legacy interrupt lines 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

No implementation defined registers shall be required to deliver these messages, only registers defined in the PCI Express specification and the ARM GIC specification.

8.5 IO Virtualization

Hardware support for IO Virtualization is optional, but if required shall use system MMU compliant with the ARM System MMU specification.

Each function (or virtual function) that requires hardware IO virtualization shall be 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't have a SMMUv3 or later SMMU.

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

8.6 IO Coherency

PCI Express transactions not marked as No_snoop accessing memory that the CPU page tables attribute as cacheable and shared shall be IO Coherent with the CPUs.

The PCI Express root complex shall be in the same inner shareable domain as the CPUs.

IO Coherency fundamentally means that no software coherency management is needed on the CPUs in order for the PCI Express root complex, and therefore devices, to get a coherent view of the CPU memory.

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

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

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

In the case where there is not a System MMU translating transactions from the root complex then 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; that is 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 this and how coherency will be managed. If a memory page is marked as non-cached in the CPU page tables then all PCI Express transactions accessing that memory must be marked as No_snoop. Failure to do so may result in loss of coherency.

CPU 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

Table 11 : PCI Express transaction types and IO coherency

In the case where the system has an SMMU translating and attributing the transactions from the root complex, the PCI Express transactions must keep the memory attributes assigned by the SMMU. It is IMPLEMENTATION DEFINED if No_snoop transactions replace the SMMU-assigned attributes with non-cached.

8.7 Legacy IO

The specification does not specify a standard mechanism for supporting legacy IO transactions. As such software consistent with the Server Base System Architecture shall not support legacy IO.

8.8 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.

8.9 Peer-to-peer

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

9 APPENDIX E: GICV2M ARCHITECTURE

9.1 Introduction

ARM is working on standardizing how PCI Express MSI and MSI-X interrupts are handled. The key part of this standardization is ensuring that each individual MSI(-X) message is presented to the operating system (OS) as a unique interrupt ID.

To facilitate this with GICv2 interrupt controllers, ARM is offering this specification as a way of converting MSI(-X) writes to unique interrupts. It is designed to be used alongside an existing GICv2 implementation.

The standard abstraction that is expected to be given to the OS is the address of the MSI_SETSPI register and the set of SPIs that it can generate. It is expected that this abstraction will be represented by system firmware data.

9.2 About the GICv2m architecture

GICv2m provides an extension to GICv2 Generic Interrupt Controller Architecture that enables MSIs to set GICv2 Shared Peripheral Interrupts (SPIs) to pending. This provides a similar mechanism to the message-based interrupt features added in GICv3.

The additional registers provided by GICv2m are specified as an additional memory-mapped Non-Secure MSI register frame, described in *Non-Secure MSI register summary* on page 37. This allows a GICv2m implementation to be built by adding a component that implements the additional registers to an existing GICv2-compatible interrupt controller. The additional component is connected to a subset of the SPI inputs to the GICv2 interrupt controller. When the additional component receives an MSI it generates an edge on the corresponding SPI input.

9.3 Security

GICv2m can optionally include Security Extensions to include support for secure MSI or MSI-X. The GICv2m Security Extensions are optional even when the GIC Security Extensions are included. However, if the GICv2m Security Extensions are included the GIC Security Extensions are mandatory.

GIC Security Extensions	GICv2m Security Extensions	Description
Not included	Not included	No support for secure interrupts.
Included	Not included	MSI are non-secure. Other interrupts can be secure or non-secure.
Not included	Included	Not supported.
Included	Included	All interrupts can be secure and non-secure.

Table 9-1 GIC and GICv2m Security Extensions

The inclusion of the GICv2m Security Extensions adds a further memory-mapped Secure MSI register frame, described in Secure MSI register summary on page 37.

9.4 Virtualization

To support virtualization, GICv2m supports the inclusion of an IMPLEMENTATION DEFINED number of instances of the Non-Secure MSI register frame.

A hypervisor can allocate one or more instance to each guest operating system. Stage 2 page tables ensure each PCI Express function only has visibility of the Non-Secure MSI register frames allocated to the operating system that is controlling the device. This ensures that a guest operating system is not able to program a PCI Express device to trigger MSI interrupts allocated to another guest operating system.

9.5 SPI allocation

GICv2m allows the allocation of SPIs to each of the register frames defined by the architecture.

Each instance of the Non-Secure MSI register frame is allocated an IMPLEMENTATION DEFINED number of contiguous SPIs. For details of Non-Secure MSI register frame instances, see *Virtualization* on page 36.

When GICv2m includes the Security Extensions an additional IMPLEMENTATION DEFINED number of contiguous SPIs are allocated for secure MSI.

The secure MSI SPI range and the non-secure MSI SPI range must not overlap, and are not required to be adjacent.

SPIs that are allocated to MSIs must only be controllable by the GICv2m MSI registers. This means that other interrupt sources must not share SPIs that are allocated as MSIs.

9.6 GICv2 programming

MSIs have edge-triggered properties. All SPIs that are allocated to MSIs must be programmed as edge-triggered in the appropriate GICv2 GICD_ICFGRn registers. For details of the GICD_ICFGRn registers see the *ARM Generic Interrupt Controller v2 Architecture Specification*.

In implementations that include the GICv2m Security Extensions, secure system software must program the GIC so that:

- SPIs that are allocated to secure MSI can be defined as secure or non-secure interrupts.
- SPIs that are allocated to non-secure MSI must be defined as non-secure interrupts, unless the GIC has been configured to permit non-secure software to create and manage the interrupt.

When used with a processor that includes the ARM Security Extensions, this means that SPIs allocated to secure MSI must be included in Group 0, and SPIs allocated to non-secure MSI must be included in Group 1. This is achieved using the GICv2 GICD_IGROUPRn registers. Additionally, non-secure software can be permitted to manage a Group 0 interrupt using the GICv2 GICD_NSACRn registers. For details of the GICD_IGROUPRn and GICD_NSACRn registers see the *ARM Generic Interrupt Controller v2 Architecture Specification*.

When using GICv2m it is a programming error to incorrectly define the security of SPIs mapped to non-secure MSI interrupts in GICv2 and this will adversely affect the ability to port GICv2m-compatible software to GICv3.

9.7 Non-Secure MSI register summary

This section gives a summary of the Non-Secure MSI registers, relative to a base memory address. This register frame is present in all GICv2m implementations.

This register frame is 4KB in size, and all registers are 32-bits wide. It must be accessible using non-secure accesses. All registers have similar behaviour to equivalent registers in the GICv3 distributor.

Offset	Name	Description
`0x000 – 0x007`	-	Reserved.
`0x008`	MSI_TYPER	See *MSI Type Register* on page 38.
`0x00C – 0x03C`	-	RESERVED.
`0x040`	MSI_SETSPI_NS	See *Set SPI Register* on page 38.
`0x044 – 0xFC8`	-	Reserved.
`0xFCC`	MSI_IIDR	See *MSI Interface Identification Register* on page 38
`0xFD0 – 0xFFC`	Identification registers	Read-only identification registers, see *Identification registers* on page 39.

Table 9-2 GICv2m Non-Secure MSI register summary

9.8 Secure MSI register summary

This section gives a summary of the optional Secure MSI registers, relative to a base memory address. This register frame is only included in GICv2m implementations that include the optional GICv2m Security Extensions.

This register frame is 4KB in size, and all registers are 32 bits wide. It must only be accessible using secure accesses. All registers have similar behaviour to equivalent registers in the GICv3 distributor.

Offset	Name	Description
`0x000 – 0x007`	-	Reserved.
`0x008`	MSI_TYPER	See *MSI Type Register* on page 38.
`0x00C – 0x03C`	-	Reserved.
`0x040`	MSI_SETSPI_S	See *Set SPI Register* on page 38.
`0x044 – 0xFC8`	-	Reserved.
`0xFCC`	MSI_IIDR	See *MSI Interface Identification Register* on page 38.
`0xFD0 – 0xFFC`	Identification registers	Read-only identification registers, see *Identification registers* on page 39.

Table 9-3 GICv2m Secure MSI register summary

9.9 Register descriptions

All registers must support 32-bit word accesses. The MSI_SETSPI_S and MSI_SETSPI_NS registers must also support 16-bit writes to bits [15:0]. Whether other access sizes are permitted is IMPLEMENTATION DEFINED.

9.9.1 MSI Type Register

MSI_TYPER is a 32-bit read-only register that provides information about the SPIs that are assigned to the MSI frame. For information about how SPIs are assigned to each frame, see *SPI allocation* on page 36.

The format of the register is:

Bits [31:26]

Reserved, RES0.

Base SPI number, bits [25:16]

Returns the IMPLEMENTATION DEFINED ID of the lowest SPI assigned to the frame. SPI ID values must be in the range 32 to 1020.

Bits [15:10]

Reserved, RES0.

Number of SPIs, bits [9:0]

Returns the IMPLEMENTATION DEFINED number of contiguous SPIs assigned to the frame.

9.9.2 Set SPI Register

MSI_SETSPI_NS and MSI_SETSPI_S are 32-bit write-only registers.

The format of the register is:

Bits [31:10]

Reserved, RES0.

SPI, bits [9:0]

On a write, an edge-triggered interrupt will be generated to the GICv2 generic interrupt controller for an SPI with the ID identified by the value of this field. If the resulting value does not identify an SPI that is allocated to this frame then the write has no effect.

9.9.3 MSI Interface Identification Register

MSI_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 GICv2m architecture. The value of this field depends on the GICv2m architecture version:

- 0x0 for GICv2m v0.

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 GICv2m:

Bits [11:8]	The JEP106 continuation code of the implementer.
Bit [7]	Always 0.
Bits [6:0]	The JEP106 identity code of the implementer.

9.9.4 Identification registers

This architecture specification defines the offsets `0xFD0` to `0xFFC` in the MSI register map as a read-only identification register space. Table 2-2 shows the architecturally-required implementation of the identification register space.

Offset	Name	Description
`0xFD0 – 0xFE4`	-	IMPLEMENTATION DEFINED registers.
`0xFE8`	MSI_PIDR2	Peripheral ID2 Register.
`0xFEC – 0xFFC`	-	IMPLEMENTATION DEFINED registers.

Table 9-4 GICv2m identification registers

The assignment of this register space, and naming of registers in this space, is consistent with the ARM identification scheme for CoreLink and CoreSight components.

9.9.4.1 Peripheral ID2 Register

MSI_PIDR2 is a 32-bit read-only register.

The format of the register is:

Bits [31:8]

IMPLEMENTATION DEFINED.

ArchRev, bits [7:4]

Revision field for the GICv2m architecture. The value of this field depends on the GICv2m architecture version:

- 0x0 for GICv2m v0.

Bits [3:0]

IMPLEMENTATION DEFINED.

10 APPENDIX F: GIC-400 AND 64KB TRANSLATION GRANULE

The GICC register frame as defined in the GICv2 specification must be spread across two MMU pages such that GICC_DIR is in a different page to the rest of GICC.

In a 64KB translation granule system this means that GICC needs to have its base at 4KB below a 64KB boundary.

The ARM implementation of GICv2, the GIC-400 product, aligns GICC to an 8KB boundary. As such an address wiring workaround is needed to use the GIC-400 in a 64KB translation granule system.

```
AddressGIC400[14:0] = {AddressSystem[18:16],AddressSystem[11:0]}
```

This has the effect of aliasing each 4KB of GIC registers 16 times in a 64KB page. System software can now be told that the last alias in a 64KB page is the GICC base, which conveniently runs into the first alias of the next 64KB page completing the GICC register frame.

11 APPENDIX G: GICV2M COMPATIBILITY IN A GICV3 SYSTEM

A key difference between GICv3 and GICv2 is that GICv3 can support more than 8 CPUs, which is the maximum supported by GICv2. In order to achieve this there is a change in some of the architectural concepts. GICv3 introduces a new type of interrupt, called LPI, which is designed to be far more scalable. It also changes the routing semantics of SPI to enable them to scale to more than 8 CPUs.

However, GICv3 supports full backwards compatibility with GICv2 when ARE==SRE==0.

GICv2m is an extension of the GICv2 architecture that adds register frames to support MSI(-X). This note explains how to achieve compatibility between a GICv3 hardware system and GICv2m software.

11.1 GICv2m based hypervisor (GICv2m guests) or GICv2m OS without hypervisor

The GICv3 must be configured to have SRE==ARE==0 and can therefore only be used with 8 CPUs or less, but is fully GICv2 compatible. In order to be GICv2m compatible the hardware system must implement GICv2m register frames for MSI support.

11.2 GICv3 based hypervisor with GICv2m guest OS

The hypervisor runs with ARE==1 so can address >8 CPUs.

The GICv2m guests run with EL1.SRE==EL1.ARE==0 which gives it GICv2 functionality. The guest must be restricted to eight CPUs or fewer.

The GICv2m register frames are not needed for the guests though as long as the OS is using a suitable abstraction for MSI support. The expected abstraction for the MSI targets is the tuple of (register address, interrupt ID set).

It is expected that the firmware interface of the OS will hand over a set of these MSI registers to the OS (which in this case will be supplied by the hypervisor).

In this compatibility case the hypervisor would hand over the address of GICR_TRANSLATER and a set of IDs (the ID set need to be in the valid SPI range of 32-1019). The hypervisor will have created a single interrupt translation table for all the devices that belong to the OS, and will have created translations for the IDs handed to the OS to unique LPIs.

Whenever a device sends an MSI the hypervisor receives the corresponding LPI. The hypervisor then posts the original ID to the guest. The target CPU is chosen by the hypervisor based on the routing information the GICv2m guest programs into the SPI route register (which is trapped by the hypervisor).