POWER STATE COORDINATION INTERFACE (PSCI)
System Software on ARM® Systems

ARM®

Power State Coordination Interface (PSCI)
System Software on ARM specification

Copyright © 2012, 2013, 2015 ARM Limited. All rights reserved.

Release information

Table 1 below lists the changes made to this document.

Table 1 Change history

Date	Issue	Confidentiality	Change
13 August 2012	A	Non-Confidential	First release
24 June 2013	B	Non-Confidential	Second release, PSCI version 0.2
25 Jun 2013	B.b	Non-Confidential	Removed inappropriate watermark, PSCI version 0.2
30 January 2015	C	Non-Confidential	PSCI 1.0 release, and errata fix for PSCI 0.2

Table of Contents

1   Introduction

This document defines a standard interface for power management that can be used by OS vendors for supervisory software working at different levels of privilege on an ARM device. Rich operating systems like Linux and Windows, hypervisors, secure firmware, and Trusted OS implementations must interoperate when power is being managed. The aim of this standard is to ease the integration between supervisory software from different vendors working at different privilege levels.

The interface is aimed at the generalization of code in the following power management scenarios:

- Core idle management.
- Dynamic addition and removal of cores, and secondary core boot.
- big.LITTLE migration.
- System shutdown and reset.

The interface does not cover *Dynamic Voltage and Frequency Scaling* (DVFS) or device power management (for example, management of peripherals such as GPUs).

The interface is designed so that it can work in conjunction with hardware discovery technologies such as *Advanced Configuration and Power Interface* (ACPI) and *Flattened Device Tree* (FDT). It is not a replacement for ACPI or FDT.

This document describes PSCI versions 1.0 and 0.2. For PSCI 0.2 the document provides an errata fix update. The PSCI 0.2 erratum applies to a single function and is described in section 5.1.7.

The document is arranged into the following sections:

- Section 1 provides this introduction and references.
- Section 2-4 provide background materials including:
    - Intended uses of PSCI.
    - Background definitions of power state terms.
    - Methods by which PSCI requests are made.
    - ARM architecture background.
- Section 5 provides the main description of the PSCI functions.
- Section 6 provides additional implementation details.
- Sections 7 and 8 provide a revision history of the PSCI specification and a glossary.

Readers already familiar with PSCI can skip straight to section 5.

1.1   Additional reading

This section lists publications by ARM and by third parties.

See Infocenter, http://infocenter.arm.com, for access to ARM documentation.

1.1.1   ARM publications

The following documents contain information relevant to this document:

[1] *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition* (ARM DDI 0406).

[2] *Embedded Trace Macrocell Architecture Specification* (ARM IHI 0014).

[3] *Program Flow Trace Architecture Specification* (ARM IHI 0035).

[4] *SMC Calling Conventions* (ARM DEN 0028).

[5] *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile* (ARM DDI 0487).

[6] *Advanced Configuration and Power Interface Specification*. See http://www.uefi.org/sites/default/files/resources/ACPI_5_1release.pdf.

[7] https://www.kernel.org/doc/Documentation/devicetree/bindings/arm/psci.txt.

[8] *ARM Trusted Firmware.* See https://github.com/ARM-software/arm-trusted-firmware.

1.1 Feedback

ARM welcomes feedback on its documentation.

1.1.1 Feedback on this manual

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

- The title.

- The number, ARM DEN 0022C.

- 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

Power management aware operating systems dynamically change the power states of cores, balancing the available compute capacity to match the current workload, while striving to use the minimum amount of power. Some of these techniques dynamically switch cores on and off, or place them in quiescent states, where they no longer perform computation. This means they consume very little power. The main examples of these techniques are:

Idle Management: When the kernel in an OS has no threads to schedule on a core, it places that core into a clock-gated, retention, or even fully power-gated state. However, the core remains available to the OS.

Hotplug: Cores are physically switched off when compute demand is low, and then brought back online when it increases. The OS migrates all interrupts and threads away from the cores that are taken offline, and rebalances the load when they are brought back online.

big.LITTLE™ technology provides powerful options for matching compute capacity to workload by appropriately distributing the load between big and LITTLE processors. The use of processors with different compute capacities, but the same Instruction Set Architecture (ISA), provides a wider dynamic range of power and performance than is possible using designs with only one type of processor. Performance management of a big.LITTLE system intersects directly with power management. In particular, *Operating System Power Management* (OSPM) might need to turn cores and clusters on and off as it transfers computation from a big core to a LITTLE one, or from a LITTLE core to a big one.

Although it would be simpler to consider the software of an embedded system to be provided by a single vendor, in most situations this is not the case, even when the end device is delivered with fixed functionality. The ARM architecture defines a set of Exception levels [5], that support the required partitioning of the software stack used on a device. Table 2 shows this partitioning and indicates the typical vendor of each level of the stack:

Table 2 Typical partitioning of software on a system that complies with the ARM architecture

AArch32 state	AArch64 state	Stack and typical vendor
Non-secure EL0 (PL0)	Non-secure EL0	Unprivileged applications, such as applications downloaded from an App Store
Non-secure EL1 (PL1)	Non-secure EL1	Rich OS kernels from, for example, Linux, Microsoft Windows, iOS
Non-secure EL2 (PL2)	Non-secure EL2	Hypervisors, from vendors such as Citrix, VMWare or OK-Labs
Secure EL0 (PL0)	Secure EL0	Trusted OS applications
Secure EL3 (PL1)	Secure EL1	Trusted OS kernels from Trusted OS vendors such as Trustonic
Secure EL3 (PL1)	Secure EL3	Secure Monitor, executing secure platform firmware provided by Silicon vendors and OEMs ARM Trusted Firmware

Note: AArch32 state is the 32-bit ARMv8 execution state, and the execution state used by all ARM processors before ARMv8. In an ARMv7 processor, the Exception levels are implicit, and not identified by the processor documentation. The Virtualization Extensions provide the EL2 functionality, and the Security Extensions provide the EL3 functionality, and within a Security state the *Privilege level* (PL) identifies the Exception level hierarchy. For more information, see section 3.2.

As various operating systems from various different vendors can be present in an ARM system, performing power control requires a method of collaboration. Considering operation in Non-secure state, if a supervisory system that is managing power, whether it is executing at the OS level (EL1) or at hypervisor level (EL2), wants to enter an idle state, power up or power down a core, perform a big.LITTLE migration, or reset or shutdown the system, supervisory systems at other exception levels will need to react to the power state change request.

Equally, if the power state of a core is changed by a wakeup event, it might be necessary for supervisory systems running at exception levels to perform actions such as restoring context. PSCI provides a standard interface definition to support this interoperation and integration across the various supervisory systems. This document defines such an interface, the Power State Coordination Interface, and describes its use for idle, Hotplug, big.LITTLE management, shutdown, and reset.

3 Assumptions and recommendations

This document defines an API that can be used to coordinate power control among supervisory systems concurrently running on a device. As the following sections explain, the API allows a supervisory system to request cores to be powered up or down, and to request context transfer where necessary for big.LITTLE systems. Throughout the description the document generally assumes that EL2 and EL3 are both implemented, but also covers other cases.

3.1 PSCI intended use

PSCI has the following intended uses:

- Provides a generic interface that supervisory software can use to manage power in the following situations:

  - Core idle management.

  - Dynamic addition of cores to and removal of cores from the system, often referred to as hotplug.

  - Secondary core boot.

  - big.LITTLE migration models, described later in this document.

  - System shutdown and reset.

- Provides an interface that supervisory software can use in conjunction with Firmware Table (FDT and ACPI) descriptions to support the generalization of power management code.

PSCI does not cover:

- Peripheral idle management. PSCI only applies to idle management of the cores used by the central scheduler of supervisory software.

- Dynamic Voltage and Frequency Scaling. PSCI does not provide interfaces for the management of core clock frequencies on a device.

PSCI does not provide power state representations to supervisory software. However, it is designed so that it can also be used with hardware description technologies such as ACPI or FDT.

3.2 Exception levels, the ARMv7 privilege levels, and highest privilege

ARMv8 introduces explicit Exception levels that also define the software execution privilege hierarchy within a Security state. An increase in Exception level, for example from EL0 to EL1, corresponds to an increase in execution privilege.

In ARMv7, the Exception level hierarchy is implicit in the architecture:

- The Virtualization Extensions provide the EL2 functionality. This is present only in Non-secure state.

- The Security Extensions provide the EL3 functionality, including the support for two Security states. The control features of this functionality are provided by a Monitor mode that is present only in Secure state.

The ARMv7 architecture [1] uses *Privilege levels* (PLs) to describe the software execution privilege hierarchy. Because Monitor mode was defined as a peer of the other Secure state privileged processor modes, this means the ARMv7 privilege levels are asymmetrical between Non-secure state and Secure state, as follows:

- In Non-secure state the privilege level hierarchy is:

  - PL0, unprivileged. Applies to User mode.

  - PL1, OS-level privilege. Applies to System, FIQ, IRQ, Supervisor, Abort, and Undefined modes.

  - PL2, hypervisor privilege. Applies to Hyp mode.

- In Secure state the privilege level hierarchy is:

  - Secure PL0, unprivileged. Applies only to User mode.

  - Secure PL1, Secure OS and Monitor level privilege. Applies to System, FIQ, IRQ, Supervisor, Abort, Undefined, and Monitor modes.

In the AArch32 description of execution privilege [5], which applies to both ARMv7 and ARMv8 implementations, the processor modes map to the Exception levels as follows:

- In Non-secure state the processor modes implemented at each Exception level are:

  - EL0: User mode.

  - EL1: System, FIQ, IRQ, Supervisor, Abort, and Undefined modes.

  - EL2: Hyp mode.

- In Secure state the processor modes implemented at each Exception level are:

  - Secure EL0: User mode.

  - EL3: System, FIQ, IRQ, Supervisor, Abort, Undefined, and Monitor modes.

Note: In an ARMv8 implementation, this mapping of the Secure modes applies only when EL3 is using AArch32. When EL3 is using AArch64, Monitor mode is not implemented, and if Secure EL1 is using AArch32 then Secure System, FIQ, IRQ, Supervisor, Abort, and Undefined modes are implemented as part of Secure EL1. For more information see [5].

This document generally uses Exception level terminology. In the document:

- References to EL1 and EL0 mean Non-secure EL1 and EL0, unless they explicitly indicate otherwise.

- *Highest privilege* refers to the first implemented Exception level in the following sequence, that runs from highest to lowest, EL3, Secure EL1, EL2, Non-secure EL1.

3.3 Software stacks on ARM based systems

On a given ARM device there can be a number of supervisory software kernels or privileged software components.
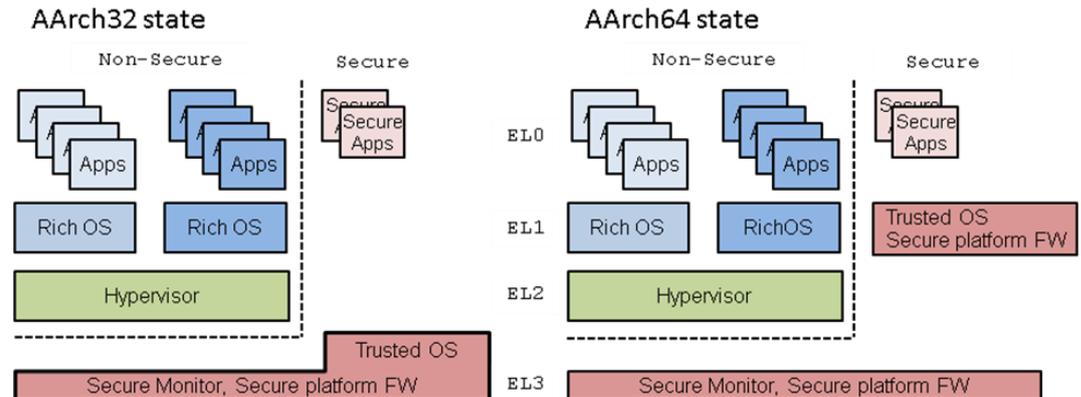
Figure 1 Software Layers of an ARM system

Figure 1 illustrates these various layers. The Normal (Non-secure) world has the following privileged components:

Rich OS kernels: such as Linux or Windows running in Non-secure EL1. When running under a hypervisor, the Rich OS kernels can be running as a guest or host depending on the hypervisor model.

Hypervisor: This component runs at EL2, which is always Non-secure. This component, when present and enabled, provides virtualization services to guests running Rich OS kernels.

The Secure world has the following privileged components:

Secure Platform Firmware (SPF): Owned by the silicon vendor and OEM. On an application processor, this firmware layer must be the first thing that runs at boot time. It provides a number of services, including platform initialization, the installation of the Trusted OS, and routing of Secure Monitor Calls. Some calls target the SPF and some target the Trusted OS. SPF can run in EL3 and secure EL1 on ARMv8 systems using AArch64 in EL3. ARM strongly recommends using EL3 for SPF. For ARMv7 systems, or ARMv8 systems using AArch32 at EL3, SPF executes in EL3. The implementation that acts on power management requests issued by the Power State Coordination Interface must reside in the SPF. A reference implementation for secure platform FW is provided by the ARM Trusted Firmware open source project [8].

Trusted OS: This provides secure services to the Normal world, and provides a runtime environment for executing secure applications. In AArch32 state Trusted OS software executes in Secure EL3, and in AArch64 state it primarily executes in Secure EL1.

Note: The ARM Architecture Reference Manuals [1,5] define two Security states, Secure and Non-secure, and ARM processor documentation uses these state names. ARM software documentation often refers to these states as Secure and Normal World respectively. This reflects the fact that software normally executes in Non-secure state.

The PSCI specification focuses on the interface between Secure and Normal worlds for power management. It provides a method for issuing power management requests. To deal with the requests, the SPF must include a PSCI implementation. The ARM Trusted Firmware project [8] provides a reference implementation of the PSCI specification.

The PSCI specification requires communication between the SPF and a Trusted OS. Currently, how this communication is handled is specific to individual vendors. Therefore this communication is IMPLEMENTATION DEFINED.

Although the PSCI specification focuses on power management requests between Secure and Normal worlds, the interface can also be reused easily at the junction between Rich OS kernels and hypervisors.

3.4   Conduits

The PSCI interface must support interaction at all levels of execution implemented on the device, where multiple levels of supervisory software might be executing. For the caller operating in the Normal world, the interface must forward a message to the Secure world. In a system that implements EL2, it must be possible to trap interface calls made by the EL1 kernel context to the hypervisor (EL2). If the hypervisor determines that a change of physical power state is required, it must then be able to use the same interface to inform the Secure world.

The conduits available to transfer a message from one Exception level to another depend on the implemented Exception levels.

The Secure Monitor Call instruction, SMC, provides a method for a Non-secure Exception level to call into the Secure world. In addition, SMC execution at EL1 can be trapped by the hypervisor to EL2. This means that SMC provides the flexibility required to support implementations with and without a hypervisor, making it the ideal conduit for PSCI power management requests. However, use of SMC requires the implementation of EL3 [1,5]. If EL3 is not implemented the SMC instruction is UNDEFINED and generates an exception in the calling OS.

The Hypervisor Call instruction, HVC, allows a Rich OS executing at EL1 to communicate with a hypervisor at EL2. This means that a hypervisor can provide a PSCI interface, using HVC as the calling conduit. If EL2 is implemented and EL3 is not implemented, then Hypervisor Call (HVC) is the only conduit available.

Note: In ARMv7, implementing the Virtualization Extensions (EL2 equivalent) requires that the Security Extensions (EL3 equivalent) are also implemented [1].

If an implementation does not include either EL2 or EL3, then no coordination with other supervisory software is required, as there can only be one OS, which must execute at EL1. Power control in this case must use board support adaptation code implemented specifically for that OS.

Table 3 describes the different conduits available.

Table 3 Dependence of available conduits on implemented Exception levels

EL3 Implemented	EL2 Implemented	Conduit	Notes
Yes	Yes	SMC,HVC	-
Yes	No	SMC	-
No	Yes	HVC	EL combination not permitted in ARMv7
No	No	Platform specific code	No conduit required

Discoverability methods for PSCI, and the conduit that applies, are described in section Figure 10 **Reference source not found.**.

3.5   Secure world software and power management

Most current Trusted OS implementations are not SMP-capable. When running on SMP devices, they are tied to a single core. Secure Monitor Calls destined for the Trusted OS are only expected to come from that core. The lack of SMP support in the OS helps to keep Trusted code simple and small, which in turn aids certification. Trusted OS services are invoked from the Normal world through Rich OS drivers or daemons that are provided with the Trusted OS implementations. The threads associated with these drivers and daemons are normally affinitized to the core used by the Trusted OS.

ARM systems generally include a power controller, or control logic, that can manage core power. This normally provides interfaces that support a number of power management functions. Often these include support for transitioning cores, clusters, or a superset into low-power states. In the low-power state, the cores are either fully switched off or in quiescent states where they are not executing code. **ARM strongly recommends that the Secure world is responsible for the control of these states**. Otherwise, the OSPM could enter a low-power mode without informing the Trusted OS. Even if such an arrangement could be made robust, it is unlikely to perform as well. In particular, for states where the core is fully power-gated, a longer boot sequence is required on wakeup, because the Secure world requires full initialization. This is because the secure components effectively boot from cold every time. On a system where the Secure world is responsible for this power control, Secure components can save their state before powerdown, providing a faster resumption on powerup. In addition, the Secure world might need to manage peripherals as part of a power transition.

Other forms of power management, such as dynamic performance management through voltage and frequency scaling, are not covered by this interface. **ARM strongly recommends that all policy in power and performance management is performed in the Normal world**. The Normal world has greater visibility of the current use and purpose of a given device. Where the Secure world has performance requirements, it is recommended that IMPLEMENTATION DEFINED mechanisms are used to communicate those requirements to the Normal world.

3.6   Virtualization and core power policy

Hypervisors are broadly split into two basic types:

Type 1: Sometimes described as native or bare metal. Type 1 hypervisors execute directly on the hardware. Any application guest OS sees a virtualized view of this hardware.

Type 2: Sometimes described as hosted. Type 2 hypervisors run within a host OS. The host OS has a physical view of the hardware it runs on, but guest operating systems see a virtualized view.

This is a very broad categorization. In reality, there are variations on the above. The ARMv8 architecture allows running a type 2 hypervisor in EL1 or EL2. This blurs the differences between type 1 and type 2. From a power management perspective, and for the purposes in this document, a type 2 hypervisor or host running in EL2, is analogous to a type 1 hypervisor. However, in general terms these two abstractions capture the forms of power management required by the hypervisors.

From the point of view of power management and virtualization there are two types of OSPM:

Physical OSPM: This comprises the software components that select the physical power states.

Virtual OSPM: This is an OSPM present in a guest OS running a virtual machine, which selects virtual, rather than physical power states.

With type 2 hypervisors, the physical OSPM resides in the host. Actual power policy is controlled from a Rich OS typically running at EL1. The physical OSPM is contained in this Rich OS. This layer has a physical view of the cores. An example of this is shown below in Figure 2 on the left hand side.
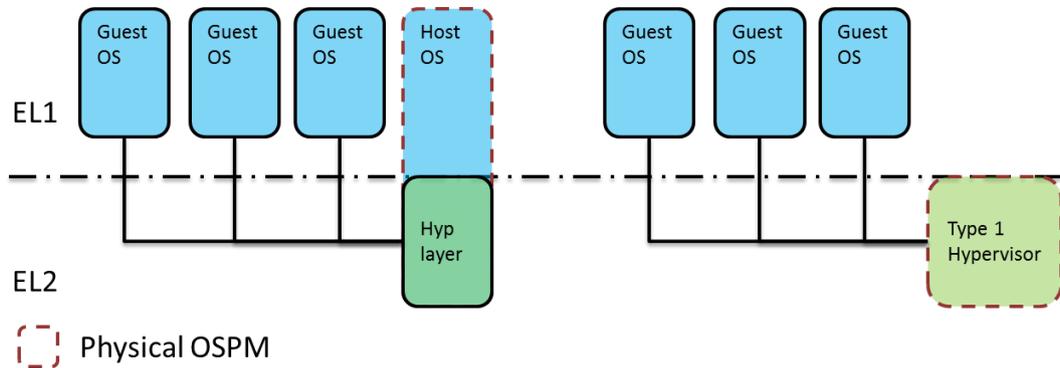
Figure 2 Typical Power management models in virtualization

For the power management functions covered in this document, type 2 hypervisor behavior depends on the calling OS.

If the caller is the host, the hypervisor complies with the power request or allows the call to pass straight through to the secure platform firmware. The hypervisor typically only performs any necessary operations resulting from the call, for example, saving state on a powerdown if needed. From this point onwards, the hypervisor calls through to the secure platform firmware using the parameters supplied by the caller. If no special operations are required, the hypervisor does not even trap calls from the host, and instead routes them directly to the secure platform firmware.

When a type 2 hypervisor is implemented, guests use a virtual OSPM. They can also issue power requests through the PSCI APIs, but the requests are issued in relation to virtual cores and virtual power states. These requests are trapped by the hypervisor which issues them back to the physical OSPM. This component can then determine whether physical power management is required. For these guests, the power calls effectively terminate at the hypervisor.

With a type 1 hypervisor, power policy is typically owned entirely by the hypervisor. This is shown on the right hand side in Figure 2. The physical OSPM is implemented in the hypervisor. In this case all guests have a virtualized view of the cores. The hypervisor determines from the virtual power states of the guests whether physical power control is required, and if so uses the PSCI API to coordinate with the secure platform firmware. Guests can also use this API to communicate virtual power requirements to the hypervisor. For these guests the calls effectively terminate at the hypervisor.

In some cases, type 1 hypervisors delegate power management to a privileged guest. In these cases, the physical OSPM is implemented in that privileged guest. This power management approach is equivalent to the model described above for type 2 hypervisors, shown in Figure 2.

4   PSCI Use Cases and Use Case Requirements

4.1   Idle management

When a core is idle the OSPM transitions it into a low-power state. Typically, a choice of states is available, with different entry and exit latencies, and different levels of power consumption associated with each state. The state that is used typically depends on how quickly the core will be needed again. The power states that can be used at any one time might also depend on the activity of other components in an SoC, in addition to the cores. Each state is defined by the set of components that are clock-gated or power-gated when the state is entered. States are sometimes described as being shallow or deep. Typically, a state X is said to be deeper than a state Y if:

- The set of components that are powered down in state X subsumes and is larger than the corresponding set for state Y.

- The set of components is the same, but various power modes are supported, and the modes used in state X save more power than those used in state Y.

The time required to move from a low-power state to a running state is known as the wakeup latency. Generally, deeper power states have longer wakeup latencies, but this is not necessarily always the case.

Although idle power management is driven by thread behavior on a core, the OSPM can place the platform into states that affect many other components beyond the core itself. For example, if the last core in a SoC goes idle, the OSPM can target power states that affect the whole SoC. The choice is also driven by the use of other components in the system, and therefore might require coordination among multiple agents. A typical example is placing the system into a state where memory is in self-refresh when all cores, and any other bus masters, are idle. The OSPM has to provide the necessary power management software infrastructure to determine the correct choice of state.

In idle management, once a core has been placed into a low-power state, it can be reactivated at any time by a wakeup event. That is, an event that may wake up a core from a low-power state, such as an interrupt. No explicit command is required by the OSPM to bring the core or cluster back into operation. The OSPM considers the affected core or cores to be available at all times even if they are currently in a low-power state.

An ARM core can be in any of the following power states:

Run: The core is powered up and operational.

Standby: The core is powered up, but measures are employed to reduce energy consumption. In a typical implementation, the core enters standby by executing a `WFI` or `WFE` instruction and exits on a corresponding wakeup event. The core preserves all core state. Changing from standby to running operation does not require a reset of the core. In standby, all core context is maintained, and can be directly accessed on wakeup. No special actions are required by the OS to ensure context is maintained. An external debugger can access debug registers in the core power domain [1,5].

Retention: The core state, including the debug settings, is preserved in low-power structures, allowing the core to be at least partially turned off. Changing from low-power retention to running operation does not require a reset of the core. The saved core state is restored on changing from low-power retention state to running operation. From an OS point of view there is no difference between a retention state and standby state, other than method of entry, latency, and usage-related constraints. However, from an external debugger point of view the states differ as External Debug Request debug events stay pending and debug registers in the core power domain cannot be accessed [1,5].

Power-down: In this state the core is powered off. Software on the device needs to save all core state, so that it can be preserved over the powerdown. Changing from powerdown to running operation must include:

- A reset of the core, after the power has been restored.
- Restoring the saved core state.

The defining characteristic of powerdown states is that they are destructive of context. This affects all the components that are switched off in a given state, including the core, and in deeper states other components of the system such as the GIC or platform-specific IP. Depending on how debug and trace power domains are organized, in some power-down states one or both of debug and trace context might be lost. Mechanisms must be provided to enable the OS to perform the relevant context saving and restoring for each given state. Resumption of execution starts at the reset vector, after which each OS must restore its context.

To an OS that is managing power, a standby state is mostly indistinguishable from a retention state. The difference is evident to an external debugger, and in hardware implementation, but not evident to the idle management subsystem of an OS. Consequently, unless otherwise stated, this document uses the term standby to refer to both standby and retention states.

An interface is required so that the OSPM can place a core into a low-power state when it has no work for it. The messages sent through this interface must be received by all relevant levels of execution. That is, if EL2 and EL3 are implemented, a message sent by a Rich OS should be received by a hypervisor and the Secure world. Within the Secure world the message needs to be seen by a Trusted OS, if present, and by any secure platform firmware. This means that each level of supervisory software can determine whether it must perform context saving.

ARM expects that the Exception level with the highest privilege, as defined in section 3.2, to be the component that can program the power controller to enter an idle state. Typically, this is the secure platform firmware. PSCI provides a mechanism for the OSPM to pass the desired idle state to the next Exception level.

For standby states that do not require any explicit programming of the power controller, no specific interface is required. The OSPM can use `WFI` or `WFE` instructions directly. However, for deeper standby or retention states that require programming a power controller, PSCI provides an interface that can hide the platform-specific code that accesses the power controller.

Power-down states require an interface so that each level of execution can save and restore its context appropriately. For power-down states, the interface requires a return address. This is the address at which the calling OS expects resumption of execution on wakeup at its Exception level. From a powered-down state, the core restarts at the reset vector, in Secure state if the implementation includes EL3. After initializing, the Secure world must re-start execution of the OS that called the power down interface, at the required return address. PSCI provide a method of specifying return addresses, and cookies that can be used for pointers to saved context.

4.2   Power state system topologies and coordination

Multiprocessor systems can have a number of different power domains to power different elements of the system. Each power domain might contain a combination of one or more processing elements (such as cores, co-processors, or GPUs), memories (caches, DRAMs), and fabric (for example inter and intra cluster coherency fabric).

Each component in a power domain has a set of power states that affect the components in the domain. Although physically the power domains are not necessarily built in a hierarchical fashion, from a software control point of view, they are arranged in a logical hierarchy. The hierarchy arises out of ordering dependencies that are required when

placing the power domains into different power states. For example, consider a power domain that encompasses a shared cache, and power domains for the cores that use it. In such a system the core power domains must be powered down before the shared cache domain, in order to guarantee correct operation.
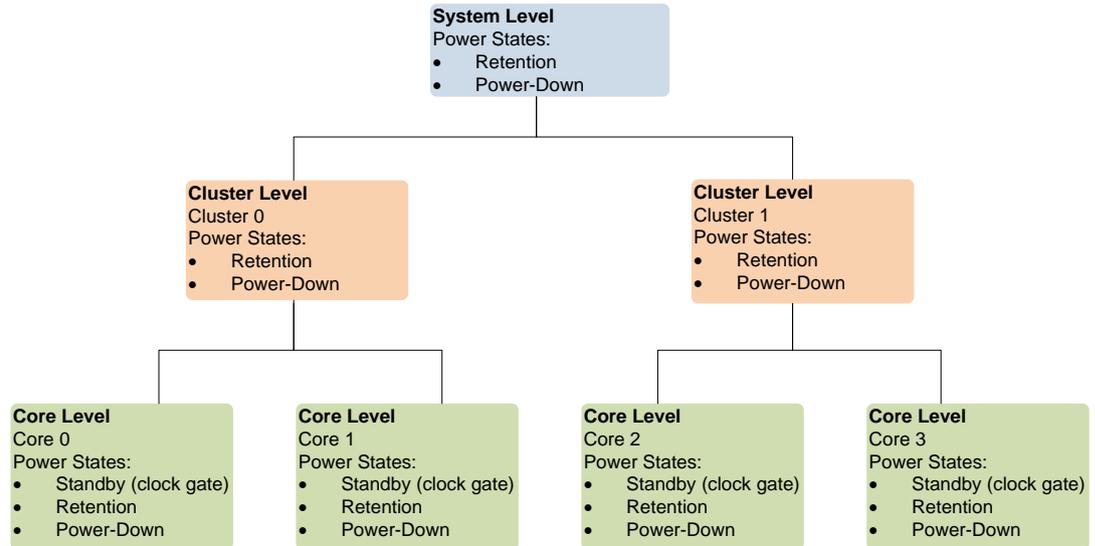
Figure 3 Example power domain topology

The diagram above provides the power domain topology of an example system. The example shows a system level power domain that supports two power states. This power domain has two children power domains, each of which includes a cluster and supports a set of cluster power states. Each cluster power domain has a couple of children core level power domains. Each of the core level power domains includes a core and supports additional power states.

From a hardware perspective, a system is divided into multiple exclusive or shared power domains. Each power domain can be represented as a node in a power domain topology tree. Sibling power domains are mutually exclusive. Parent power domains are shared by the children. The various levels in the tree (core, cluster, and system in the example) are referred to as power levels. Higher levels are closer to the root of the tree (system) and lower levels are closer to the leaves (the cores).

4.2.1 Local power states and composite power states

Individual nodes in a topology have their own specific power states, which are referred to here as local power states. When an OS, running on an idle core, requests a power state, it might need to express not just a local power state for the core, but also for any parent nodes. For instance taking our example system from Figure 3, if Core 1 was the last to go idle in Cluster 0 it would make sense for the OS to request a local power state for the cluster as well as the core. We refer to the combined power state that an OS might request as a composite power state. Not every combination of local states is possible. When a node at a power level is in a certain local power state, parent nodes at higher power levels cannot be in a deeper local state. In other words:

1. A power-down in a higher-level node can only be combined with power-down states in lower level children.

2. A retention state in a higher level node might only be combined with retention or power-down states in lower level children.

3. A standby (clock gated) state in a higher level node allows lower level children to be in standby, retention, or power-down states.

4. A running state in a higher level node allows lower level children to be in any state.

Note: Note that the PSCI specification is only concerned with power states that require explicit software control. Hardware power modes, which are not exposed to software, are not relevant to PSCI.

Taking these rules into account, Table 4 illustrates the valid low-power composite states for the example system described in Figure 3.

Table 4 Valid local state combinations for composite power states in the example system

System Level State	Cluster Level State	Core Level State
Run	Run	Standby
Run	Run	Retention
Run	Run	Power-down
Run	Retention	Retention
Run	Retention	Power-down
Run	Power-down	Power-down
Retention	Retention	Retention
Retention	Retention	Power-down
Retention	Power-down	Power-down
Power-down	Power-down	Power-down

4.2.2 Affinity hierarchy

ARM systems can have multiple cores or even multiple clusters of cores. This specification uses the term affinity hierarchy to describe the hierarchical arrangement of cores. This often, but not always, maps directly to the processor power topology of the system.

This specification uses the term affinity level to describe a level in the affinity hierarchy, for example, the level of cores or clusters. Finally the term affinity instance is used to refer to an individual entry in the affinity hierarchy, for example, an individual core or cluster.

4.2.3 Power state coordination

Entry into local power states for high level nodes in a power topology (e.g. clusters or system) requires coordinating children nodes. For example, entry into a cluster power-down state is only possible when all cores in the cluster are powered down. To achieve this, every core but the last one has to be placed into a power-down state, and the last one places itself and the cluster into a power-down state.

PSCI supports two modes of power state coordination, platform coordinated mode and OS initiated mode.

4.2.3.1 Platform coordinated mode

This is the default mode of coordination. In this mode the PSCI implementation is responsible for coordinating power states. When a core has no more work to do, the

OSPM requests the deepest state it can tolerate for that core and its parent nodes. For power state requests that affect a topology node above the core level, the implementation chooses the deepest power state that can be tolerated by all the cores in the node. In effect the power state request expresses the following two constraints:

1. The caller allows entry to states of this depth, but no deeper.

2. The caller cannot tolerate a higher wakeup latency than that associated with the requested state.

The PSCI implementation then determines the deepest state that satisfies the constraints expressed by each core in a given node. Table 5 builds on the example system of Figure 3, and shows how composite power state requests are arbitrated for that system. In the example, the abbreviations Ret and PD denote retention and power-down respectively. The example is not exhaustive, and assumes that Cluster 1 has already been powered down. The example shows requests from Core0 and Core1, and the resulting power state for the clusters and the system.

Table 5 Example platform coordination of power state requests.

Composite state requested						Power state granted		
Core 0			Core 1			Cluster 0	Cluster 1 (we assume this has already powered down)	System
Core	Cluster	System	Core	Cluster	System			
Ret	Run	Run	Ret	Run	Run	Run	Power-down	Run
Ret	Ret	Run	Ret	Run	Run	Run	Power-down	Run
Ret	Ret	Run	Ret	Ret	Run	Retention	Power-down	Run
Ret	Ret	Ret	Ret	Ret	Run	Retention	Power-down	Run
Ret	Ret	Ret	Ret	Ret	Ret	Retention	Power-down	Retention
PD	Ret	Ret	Ret	Ret	Ret	Retention	Power-down	Retention
PD	PD	Ret	PD	Ret	Ret	Retention	Power-down	Retention
PD	PD	Ret	PD	PD	Ret	Power-down	Power-down	Retention
PD	PD	PD	PD	PD	Ret	Power-down	Power-down	Retention
PD	PD	PD	PD	PD	PD	Power-down	Power-down	Power-down

As a general rule, wakeup latency increases with depth of state, so in the example above we are assuming the retention (for core, cluster, or system) has smaller wakeup latency than power-down. However this is not necessarily the case. There may be systems where this ordering breaks. As stated by the second constraint above, the state chosen by the implementation has to satisfy the wakeup time associated with every core's request. Consider a dual core system that has three system-level states, ordered by increasing depth, stateA, stateB, and stateC, but where wakeup latency ordering is state A < state C < state B. If Core 0 chooses State B and Core 1 chooses State C, the system will need to enter state A. State C is not possible as Core 0 does not want to go that deep, and State B is not possible as Core 1 cannot tolerate its longer wakeup latency.

Versions of PSCI prior to 1.0 support only the platform coordinated mode.

4.2.3.2 OS initiated mode

Introduced in PSCI1.0, OS Initiated Mode places the responsibility for coordination on the calling OS. In the OS Initiated coordination scheme, OSPM only requests an idle state for a particular topology node when the last underlying core goes idle.

When a core goes idle it always selects an idle state for itself, but idle states for higher level nodes such as clusters are only selected when the last running core in the node goes idle. In addition, the implementation only considers the most recent request for a particular node when deciding on its idle state. Using the above example of a dual cluster, dual core per cluster system, the following table illustrates the steps involved in taking Cluster0 into a power-down state. In the example, abbreviations R, Ret, and PD, denote run, retention and power-down respectively. In addition, states where OS view and implementation view differ are marked in red, and states that change during a step are underlined.

Table 6 Example flow: Cluster power-down entry

Step		OS View			PSCI View		
		Core0	Core1	Cluster0	Core0	Core1	Cluster0
1: OS on Core0 requests Core0 power-down	before	R	R	R	R	R	R
	after	PD	R	R	R	R	R
2: PSCI Implementation observes request and places Core0 into power-down	before	PD	R	R	R	R	R
	after	PD	R	R	PD	R	R
3: OS on Core1 requests Core1 power-down and, knowing it is last in the cluster, Cluster0 power-down	before	PD	R	R	PD	R	R
	after	PD	PD	PD	PD	R	R
4: PSCI Implementation observes requests for Core1 and Cluster0 and processes them	before	PD	PD	PD	PD	R	R
	after	PD	PD	PD	PD	PD	PD

As the table illustrates, there are periods (marked in red) where the OS view of core state and the implementation view of core state do not match. This might happen after the OS requests a state, but before the implementation has processed the request. This can also happen when a core powers up, as the implementation sees the core before the OS. In order to implement OS Initiated mode, it is necessary to deal with the races that arise due to the differing views of core state. Solving the races gives rise to the following requirements:

- The implementation must deny any requests from the calling OS that are inconsistent with its view of core state.

- The calling OS must indicate when the calling core is the last running core at a particular power hierarchy level. It must also specify which power hierarchy level the core is last in, for example, whether it is the last core in the cluster or the last core in the system.

A full discussion of the reasoning behind these requirements, and races that arise, is described in sections 6.2 and 6.3.

4.3 CPU hotplug and secondary CPU boot

CPU hotplug is a technique that can dynamically switch cores on or off. Hotplug can be used by an OSPM to change available compute capacity based on current compute requirements. Hotplug is also sometimes used for reliability reasons. There are a number of differences between using hotplug and using a power-down state in idle management:

1) When a core is hot unplugged, the supervisory software stops all use of that core in interrupt and thread processing. The calling supervisory software regards the core as no longer available.

2) The supervisory software has to issue an explicit command to bring a core back online, that is, to hotplug a core. The appropriate supervisory software will only start using that core in thread scheduling, or interrupt service routines, after this command.

3) With hotplug, wakeup events that could restart a powered-down core are not expected on the cores that have been hot unplugged.

Operating systems typically perform much of the kernel boot process on one primary core, bringing secondary cores online at a later stage. For systems that support hotplug, the operations involved in booting a core for secondary boot or hotplug, are the same. Therefore they can be provided by a single interface. PSCI provides an interface with the following properties:

1) The supervisory software can request that a core be powered up. The supervisory software must provide an appropriate start address for the Non-secure Exception Level where it will resume operation once it exits the secure platform firmware. The provision of a start address means the caller can shortcut any bootloader related code when onlining a core, by providing an entry point directly in its own OS address space. Different addresses can be provided to handle different startup reasons. Alternatively, the supervisory software can use internal per-core data structures for this purpose.

2) The supervisory software can request powering down the core, and inform higher Exception levels that it is doing so.

4.4 big.LITTLE

Three scheduling models have been used with big.LITTLE systems:

Cluster migration: In Cluster migration, the big.LITTLE hardware platform is typically defined by an SoC that has the same number of big and LITTLE cores. Only one cluster, the big or the LITTLE, is active at any one time. The OSPM considers the overall performance loading on the currently-active cluster. Typically, this is the load of the core that is busiest in the cluster. If the load warrants a change from big to LITTLE or from LITTLE to big, the OSPM synchronizes all the cores and then transfers all compute context to the other cluster. As part of this process every OS, on every core, has to save its state whilst still executing on the old (outbound) cluster. When the new (inbound) cluster boots up, the operating systems need to restore their state on each core. Once the inbound cluster has restored context, the outbound cluster is switched off. In this scheme, operating systems maintain a logical view of the compute cluster and the cores it contains, where the logical cluster can at any one time be instantiated as the big cluster or the LITTLE cluster.

CPU migration: In CPU migration, each core on the LITTLE cluster is paired with a core on the big cluster. The OSPM monitors the load on each core. High load causes the execution context to be moved to the big core, and equally when the load is low, the execution is moved to the LITTLE core. Only one core in the pairing is active at any time. When the load is transitioned from an outbound core to an inbound core the former is switched off. This model allows a mix of big and LITTLE cores to be active at any one time. As with cluster migration, in this scheme

operating systems maintain a logical view of cores, where the logical core can at any one time be instantiated as a big core or a LITTLE core.

Global Task Scheduling (GTS): The OS is topology aware and operates across all cores in all clusters. The OS is aware of the compute capacity differences between big and LITTLE cores. The scheduler in the OS assigns tasks to the right type of core based on the compute requirements of each task. OSPM idle management and hotplug powers off unused or under-utilized cores.

Migration approaches require explicit power control of cores as part of the migration process. When the workload is transferred from the outbound core or cluster to its inbound sibling, the former can be powered down. With GTS, power-down only occurs through hotplug or idle management. Migration adds another decision point in the OS that can power down cores, in addition to idle and hotplug decisions.

PSCI supports big.LITTLE by providing interfaces for turning cores on and off, and an interface to aid migration of Trusted OS context.

4.5   System shutdown, reset and suspend

PSCI provides an interface to allow an OS to request system shutdown, system reset, and system suspend (suspend to RAM). This allows a silicon vendor to provide a common implementation of these functions that is independent of the supervisory software running on the device. No explicit function is provided for suspend to disk, as this is a special case of system shutdown.

The usage of the term *system* in the PSCI function definitions, refer to the machine view that is available to the calling OS. If the caller is a guest running in a virtual machine system, shutdown, reset, and suspend operations affect the virtual machine and might not result in any physical power changes. However, if a hypervisor is not present, or the caller is a hypervisor, the result is physical changes in power. Even if the caller is running on a physical machine, the term system may not mean the entire physical machine. For example, consider an advanced server system consisting of multiple boards, each with a board management controller (BMC), and each containing multiple SoCs. Such a system could run an OS instance per SoC. In this example, a PSCI command to shut down the system applies to a single SoC, while powering down the entire board requires access to the BMC through an administration interface that is beyond the reach of the calling OS or a PSCI implementation. In this document, the term system refers only to the machine view that is visible to an OS. In the example above this maps to a single SoC.

5 Functions

This section introduces the functions for Power State Coordination.

The APIs are described here without reference to the underlying conduit (SMC or HVC). However, the functions adhere to the SMC Calling Conventions [4]. In an implementation that includes EL2 but not EL3, a hypervisor providing support to a PSCI-compliant EL1 Rich OS can use HVC as the conduit. In the HVC case, the format of the call, in terms of immediate value, and register usage, is the same as in the SMC case. The only chance is actual instruction, HVC instead of SMC. PSCI functions can only be called from the Normal world (EL1 or EL2).

5.1 Function prototypes

5.1.1 PSCI_VERSION

Description

Return the version of PSCI implemented. See section 5.3 for more details.

Parameters	
Declaration	**Value**
uint32 Function ID	0x8400 0000
Return	
Declaration	**Value**
uint32	• Bits [31:16] Major Version • Bits [15:0] Minor Version

5.1.2 CPU_SUSPEND

Description

Suspend execution on a core or higher level topology node. This call is intended for use in idle subsystems where the core is expected to return to execution through a wakeup event. See section 5.4 for more details.

Parameters	
Declaration	**Value**
uint32 Function ID	• 0x8400 0001 SMC32 version • 0xC400 0001 SMC64 version
uint32 power_state	This parameter is described in section 5.4.2

SMC32 uint32 SMC64 uint64 entry_point_address	Address at which the core must resume execution, when it enters the return Non-secure exception level, following wakeup from a power-down state. This parameter is only valid if the target power state is a power-down state. For standby states the value passed in is ignored. For the SMC64 version, this parameter is a 64-bit entry point *Physical Address* (PA) or *Intermediate Physical Address* (IPA). For the SMC32 version this parameter is a 32-bit entry point PA or IPA. **Note**: An ARMv7 system must use the SMC32 version of the function. This limits the entry point addresses to 32 bits, even if the implementation includes the *Large Physical Address Extension* (LPAE), as the MMU will be disabled when the platform re-enters at this address.
SMC32 uint32 SMC64 uint64 context_id	This parameter is only valid if the target power state is a Power-down state. For Standby states the value is ignored. For the SMC64 version this parameter is a 64-bit value. Following wakeup from a Power-down state, when the calling core first enters the return non-secure exception level, this value must be present in X0. For the SMC32 version this parameter is a 32-bit value. Following wakeup from a Power-down state, when the calling core first enters the return non-secure exception level, this value must be present in R0.

Return

Declaration	Value
int32	• SUCCESS • INVALID_PARAMETERS • INVALID_ADDRESS • DENIED – Only in OS Initiated Mode. See sections 4.2.3.2 and 5.4. See section 5.2.2 for integer values associated with each return code.

5.1.3 CPU_OFF

Description

Power down the calling core. This call is intended for use in hotplug. A core that is powered down by CPU_OFF can only be powered up again in response to a CPU_ON. See section 5.5 for more details.

Parameters

Declaration	Value
uint32 Function ID	0x8400 0002

Return	
Declaration	**Value**
int32	• The call does not return when successful. • Otherwise, it returns: • DENIED See section 5.2.2 for integer values associated with each return code.

5.1.4 CPU_ON

Description

Power up a core. This call is used to power up cores that either:

• Have not yet been booted into the calling supervisory software.

• Have been previously powered down with a CPU_OFF call.

See section 5.6 for more details.

Parameters	
Declaration	**Value**
uint32 Function ID	• 0x8400 0003 SMC32 version • 0xC400 0003 SMC64 version
SMC32 uint32 SMC64 uint64 target_cpu	This parameter contains a copy of the affinity fields of the MPIDR register (See [1,5]). If the calling Exception level is using AArch32, the format is: • Bits [24:31]: Must be zero. • Bits [16:23] Aff2 : Match Aff2 of target core MPIDR • Bits [8:15] Aff1 : Match Aff1 of target core MPIDR • Bits [0:7] Aff0 : Match Aff0 of target core MPIDR If the calling Exception level is using AArch64, the format is: • Bits [40:63]: Must be zero • Bits [32:39] Aff3 : Match Aff3 of target core MPIDR • Bits [24:31] Must be zero • Bits [16:23] Aff2 : Match Aff2 of target core MPIDR • Bits [8:15] Aff1 : Match Aff1 of target core MPIDR • Bits [0:7] Aff0 : Match Aff0 of target core MPIDR
SMC32 uint32 SMC64 uint64 entry_point_address	For the SMC64 version, this parameter is a 64-bit entry point physical address (PA) or intermediate physical address (IPA). For the SMC32 version this parameter is a 32-bit entry point physical address (or IPA). Address at which the core must commence execution, when it enters the return Non-secure exception level.

SMC32 uint32 SMC64 uint64 context_id	For the SMC64 version this parameter is a 64-bit value. When the core identified by `target_cpu` first enters the return non-secure exception level, this value must be present in X0. For the SMC32 version this parameter is a 32-bit value. When the core identified by `target_cpu` first enters the return non-secure exception level, this value must be present in R0.

Return

Declaration	Value
int32	- SUCCESS - INVALID_PARAMETERS - INVALID_ADDRESS - ALREADY_ON - ON_PENDING - INTERNAL_FAILURE See section 5.2.2 for integer values associated with each return code.

5.1.5   AFFINITY_INFO

Description

Enable the caller to request status of an affinity instance (as defined in section 4.2.2). See section 5.7 for more details.

Parameters

Declaration	Value
uint32 Function ID:	0x8400 0004 SMC32 version 0xC400 0004 SMC64 version
SMC32 uint32 SMC64 uint64 target_affinity	In SMC32 version this is a uint32 value. In SMC64 version this is a uint64 value. This follows the same format as the `target_cpu` parameter of a `CPU_ON` call (see section 5.1.4).

uint32 lowest_affinity_level	Denotes the lowest affinity level field that is valid in `target_affinity` parameter. This argument allows the caller of `AFFINITY_INFO` to request information about affinity levels higher than 0.
	Possible values are:
	0: All affinity level fields in `target_affinity` are valid. In a system where processors are not hardware threaded, the `target_affinity` parameter would represent an individual core.
	1: Aff0 field should be ignored. The `target_affinity` parameter denotes an affinity level 1 processing unit.
	2: Aff0 and Aff1 fields should be ignored. The `target_affinity` parameter denotes an affinity level 2 processing unit.
	3: Aff0, Aff1 and Aff2 fields should be ignored. The `target_affinity` parameter denotes an affinity level 3 processing unit.
	From version 1.0 of PSCI onwards, `AFFINITY_INFO` is no longer required to support affinity levels higher than 0. For more details on return values when this parameter is non-zero, see section 5.7.1.

Return

Declaration	Value
int32	• 2 `ON_PENDING`: the affinity instance is transitioning to an `ON` state • 1 `OFF` • 0 `ON`: At least one core in the affinity instance is `ON` • `INVALID_PARAMETERS` • `DISABLED` See section 5.2.2 for integer values associated with each return code.

5.1.6   MIGRATE

Description

Optional. This is used to ask a uniprocessor Trusted OS to migrate its context to a specific core. See section 5.8 for more details.

Parameters

Declaration	Value
uint32 Function ID:	• 0x8400 0005 SMC32 version • 0xC400 0005 SMC64 version
uint32/uint64 target_cpu	This field follows the same format as the `target_cpu` parameter of a `CPU_ON` call (see section 5.1.4)

Return

Declaration	Value

int32	• SUCCESS
	• NOT_SUPPORTED
	• INVALID_PARAMETERS
	• DENIED
	• INTERNAL_FAILURE
	• NOT_PRESENT

See section 5.2.2 for integer values associated with each return code.

5.1.7 MIGRATE_INFO_TYPE

Description

Optional. This function allows a caller to identify the level of multicore support present in the Trusted OS. See section 5.9 for more details.

Parameters

Declaration	Value
uint32 Function ID:	0x8400 0006

Return

Declaration	Value
int32	• 0 Uniprocessor migrate capable Trusted OS. The Trusted OS will only run on one core. The Trusted OS supports the MIGRATE function and can be migrated to any core that has not been turned off with CPU_OFF. Attempts to call CPU_OFF on the core where the Trusted OS is resident, will fail with DENIED.
	• 1 Uniprocessor not migrate capable Trusted OS. The Trusted OS will only run one core. The Trusted OS does not support the MIGRATE function . Calls to MIGRATE will fail with DENIED. Attempts to call CPU_OFF on the core where the Trusted OS is resident will fail with DENIED.
	• 2 Trusted OS is either not present or does not require migration. A system of this type does not require the caller to use the MIGRATE function. MIGRATE function calls return NOT_SUPPORTED.
	• NOT_SUPPORTED can be treated by the calling OS as an indication that MIGRATE is not required, so that the behavior is the same as for a return value of 2. This effectively makes this function optional, as for SMC32 a return value of NOT_SUPPORTED must be returned if the function is not implemented.
	Note that for issue B.b of this document MIGRATE_INFO_TYPE was compulsory, and NOT_SUPPORTED was not a valid return value. This was fixed in issue C of this document, which acts a release of PSCI1.0, and errata fix release for PSCI 0.2. Therefore this change applies to all versions of PSCI from 0.2 onwards.

5.1.8 MIGRATE_INFO_UP_CPU

Description

Optional. For a uniprocessor Trusted OS, this function returns the current resident core. See section 5.9 for more details.

Parameters

Declaration	Value
uint32 Function ID:	• 0x8400 0007 SMC32 version • 0xC400 0007 SMC64 version

Return

Declaration	Value
SMC32 uint32 SMC64 uint64	• UNDEFINED if `MIGRATE_INFO_TYPE` returns 2 or `NOT_SUPPORTED` • MPIDR based value of core where the Trusted OS is resident if return value of `MIGRATE_INFO_TYPE` is 0 or 1. This field follows the same format as the `target_cpu` parameter of a `CPU_ON` call. See section 5.1.4 for further details

5.1.9 SYSTEM_OFF

Description

Shutdown the system. See section 5.10 for more details.

Parameters

Declaration	Value
uint32 Function ID:	0x8400 0008

Return

Declaration	Value
void	Does not return

5.1.10 SYSTEM_RESET

Description

Reset the system. See section 5.11 for more details.

Parameters

Declaration	Value
uint32 Function ID:	0x8400 0009

Return

Declaration	Value

void	Does not return

5.1.11 PSCI_FEATURES

Description

Introduced in PSCI v1.0.

Query API that allows discovering whether a specific PSCI function is implemented and its features. See section 5.12.1 for more details.

Parameters

Declaration	Value
uint32 Function ID:	0x8400 000A
uint32 psci_func_id	Function ID for a PSCI Function

Return

Declaration	Value
Int32	• NOT_SUPPORTED if the function identified by psci_func_id is not implemented or is an invalid function ID. • A set of feature flags associated with the implemented function indicated by psci_func_id is implemented. Feature flags are specific to each function. In all cases the format is: • Bit[31] is zero • Bits[0:30] represent the feature flags. See section 5.12.2 for the defined flags. See section 5.2.2 for integer values associated with each return code.

5.1.12 CPU_FREEZE

Description

Introduced in PSCI v1.0.

Optional.

Places the core into an IMPLEMENTATION DEFINED low-power state. Unlike CPU_OFF it is still valid for interrupts to be targeted to the core. However, the core must remain in the low-power state until it a CPU_ON command is issued for it. See section 5.13 for more details.

Parameters

Declaration	Value
uint32 Function ID:	0x8400 000B

Return

Declaration	Value

	• This call does not return when successful.
	• Otherwise, the return values are:
Int32	• `NOT_SUPPORTED`
	• `DENIED`
	See section 5.2.2 for integer values associated with each return code.

5.1.13 CPU_DEFAULT_SUSPEND

Description

Introduced in PSCI v1.0.

Optional.

Will place a core into an IMPLEMENTATION DEFINED low-power state. Unlike CPU_SUSPEND the caller need not specify a `power_state` parameter. See section 5.14 for more details.

Parameters

Declaration	Value
uint32 Function ID:	• 0x8400 000C SMC32 version • 0xC400 000C SMC64 version
SMC32 uint32 SMC64 uint64 entry_point_address	As described for the CPU_SUSPEND function in section 5.1.2.
SMC32 uint32 SMC64 uint64 context_id	As described for the CPU_SUSPEND function in section 5.1.2.

Return

Declaration	Value
Int32	• `SUCCESS` • `INVALID_ADDRESS` See section 5.2.2 for integer values associated with each return code.

5.1.14 NODE_HW_STATE

Description

Introduced in PSCI v1.0.

Optional.

This API is intended to return the true HW state of a node in the power domain topology of the system. See section 5.15 more details.

Parameters

Declaration	Value

uint32 Function ID:	• 0x8400 000D SMC32 version
	• 0xC400 000D SMC64 version
SMC32 uint32 SMC64 uint64 target_cpu	This field follows the same format as the target_cpu parameter of a CPU_ON call (see section 5.1.4).
uint32 power_level	This parameter describes the power domain level for the node. The power level parameter format is IMPLEMENTATION DEFINED, however the value of 0 is reserved to represent cores (or for SMT systems, hardware threads).

Return	
Declaration	**Value**
int32	• 2 HW_STANDBY: The power control logic indicates that the node is in a standby or retention power state ,as defined in section 4.1
	• 1 HW_OFF: The power control logic indicates that the node is fully powered down, meaning that it is in a power-down state described in section 4.1
	• 0 HW_ON: The power control logic indicates that the node is operational and not in a low-power state (in the run state described in section 4.1)
	• NOT_SUPPORTED
	• INVALID_PARAMETERS
	See section 5.2.2 for integer values associated with each return code.

5.1.15 SYSTEM_SUSPEND

Description

Introduced in PSCI v1.0.

Optional.

Used to implement suspend to RAM. The semantics are equivalent to a CPU_SUSPEND to the deepest low-power state. Further detail can be found in section 5.16.

Parameters	
Declaration	**Value**
uint32 Function ID:	• 0x8400 000E SMC32 version
	• 0xC400 000E SMC64 version
SMC32 uint32 SMC64 uint64 entry_point_address	As described for the CPU_SUSPEND function in section 5.4.3

SMC32 uint32 SMC64 uint64 context_id	As described for the CPU_SUSPEND function in section 5.4.4.
Return	
Declaration	Value

Int32	• This call does not return when successful
	
	• Otherwise, the return values are:
	
	    • NOT_SUPPORTED
	
	    • INVALID_ADDRESS
	
	    • ALREADY_ON
	
	See section 5.2.2 for the integer values associated with each return code.

5.1.16 PSCI_SET_ SUSPEND_MODE

Description

Introduced in PSCI v1.0.

Optional.

This API allows setting the mode used by CPU_SUSPEND to coordinate power states. See section 4.2 for a description of coordination modes and section 5.17 for a full description of this function.

Parameters	
Declaration	Value
uint32 Function ID:	0x8400 000F
uint32 mode	• 0: platform coordinated mode • 1: OS Initiated mode
Return	
Declaration	Value
Int32	• SUCCESS • NOT_SUPPORTED • INVALID_PARAMETERS • DENIED See section 5.2.2 for integer values associated with each return code.

5.1.17 PSCI_STAT_RESIDENCY

Description

Introduced in PSCI v1.0.

Optional.

Returns the amount of time the platform has spent in the given power state since cold boot. See section 5.18 for more details.

Parameters

Declaration	Value
uint32 Function ID:	• 0x8400 0010 SMC32 version • 0xC400 0010 SMC64 version
SMC32 uint32 SMC64 uint64 target_cpu	This parameter follows the same format as the `target_cpu` parameter of a `CPU_ON` call (see section 5.1.4).
uint32 `power_state`	As described for the CPU_SUSPEND function in section 5.4.4.

Return

Declaration	Value
SMC32 uint32 SMC64 uint64 residency	Returns the amount of time, in microseconds, spent in a local state identified by `power_state` by the topology node represented by the `target_cpu` and the highest power level of `power_state`.

5.1.18 PSCI_STAT_COUNT

Description

Introduced in PSCI v1.0.

Optional.

Return the number of times the platform has used the given power state since cold boot. See section 5.18 for more details.

Parameters

Declaration	Value
uint32 Function ID:	• 0x8400 0011 SMC32 version • 0xC400 0011 SMC64 version
SMC32 uint32 SMC64 uint64 target_cpu	This parameter follows the same format as `target_cpu` parameter of a `CPU_ON` call (see section 5.1.4).
uint32 `power_state`	As described for the `CPU_SUSPEND` function in section 5.4.4.

Return

Declaration	Value
SMC32 uint32 SMC64 uint64 count	Returns the number of times a local state, identified by `power_state,` has been leveraged since cold boot by the topology node represented by the `target_cpu` and the highest power level of `power_state`.

5.2 Arguments and return values in PSCI

5.2.1 Register usage in arguments and return values

The SMC Calling Conventions [4] provide support for calls using only 32-bit parameters (SMC32), and for calls using 32 and 64-bit parameters (SMC64). Some of the PSCI functions defined above use only SMC32, some of the functions have both an SMC32 and an SMC64 version.

For PSCI functions that use only 32-bit parameters, the arguments are passed in R0 to R3 (AArch32) or W0 to W3 (AArch64), with return values in R0 or W0.

For versions using 64-bit parameters, the arguments are passed in X0 to X3, with return values in X0 or W0, depending on the return parameter size.

In line with the SMC Calling Conventions, the immediate value used with an SMC (or HVC) instruction must be 0.

Adherence to the SMC Calling Conventions implies that any AArch32 caller of an SMC64 function will get a return code of 0xFFFFFFFF (int32). This matches the NOT_SUPPORTED error code used in PSCI. As some PSCI functions have SMC32 and SMC64 versions, it is also possible for an AArch64 client to call an SMC32 function. In these cases it is assumed that the caller understands the limitations of using SMC32, and limits itself to 32-bit parameters, and the implementation is not required to provide any specific error returns.

Note: PSCI 0.2 instead used INVALID_PARAMETERS as the return code when an AArch32 caller used an SMC64 API, or when an AArch64 caller used an SMC32 API. This requirement has been dropped since PSC1.0 to improve alignment with the SMC Calling Conventions [4].

PSCI calls between a guest and hypervisor may use the HVC conduit instead of SMC. In this case the rules described above apply. Where this document uses SMC32, the reader can assume HVC32 if the conduit is HVC. Equally, SMC64 maps to HVC64.

5.2.2 Return error codes

Table 7 defines the values for error codes used with PSCI functions. All errors are considered to be 32-bit signed integers. Therefore, when using the SMC64 calling convention, the upper word will be zero.

Table 7 Return error codes

SUCCESS	0
NOT_SUPPORTED	-1
INVALID_PARAMETERS	-2
DENIED	-3

ALREADY_ON	-4
ON_PENDING	-5
INTERNAL_FAILURE	-6
NOT_PRESENT	-7
DISABLED	-8
INVALID_ADDRESS	-9

5.3 PSCI_VERSION

5.3.1 Intended use

A caller can use the PSCI_VERSION function to ascertain the current version of the interface. The version number is a 32-bit unsigned integer, with the upper 16 bits denoting the major revision, and the lower 16 bits denoting the minor revision.

The following rules apply to the version numbering:

- Different major revision values indicate possibly incompatible functions.

- For two revisions, A and B, for which the major revision values are identical, if the minor revision value of revision B is greater than the minor revision value of revision A, then every function in revision A must work in a compatible way with revision B. However, it is possible for revision B to have a higher function count than revision A.

For the revision history of the PSCI specification see section 7.

5.3.2 Implementation responsibilities

An implementation conforming to the PSCI specification described in this document must implement and support all of the functions described, unless it is explicitly stated that a function is optional. This does not mean that supervisory software using the implementation is required to use all of the functions implemented. Any function that is not implemented has to return NOT_SUPPORTED in accordance with the SMC Calling Conventions [4]. In addition, PSCI_FEATURES should also return NOT_SUPPORTED for any non-implemented function. Table 18 in section 6.10 lists all the compulsory and optional functions for a given PSCI version.

Note: Any implementation conforming to issue C of this specification must return a minor version number of 0 and major version number of 1. An implementation conforming to issue B.b of this specification document must return a minor version of 2 and major version of 0.

5.4 CPU_SUSPEND

5.4.1 Intended use

The CPU_SUSPEND API is used to move a topology node into a low-power state. The function is called from a specific core in that topology node, and indicates that the caller intends to make use of the core in the future, but that it has no current work for it. The CPU_SUSPEND API is called by an OSPM as part of idle management.

The following sections describe the parameters of CPU_SUSPEND in detail as well as caller and implementation responsibilities.

A description of the call flow can be found in section 6.1.1.

5.4.2   CPU_SUSPEND parameters: `power_state`

From PSCI1.0 two formats are supported for the power_state parameter.

5.4.2.1 Original format

This is the only format that is supported by versions of PSCI prior to 1.0. When this format is in use, bit 1 of the flags field returned by PSCI_FEATURES with a CPU_SUSPEND function ID is set to 0.

In this format the power_state parameter is broken into the following fields:

Table 8 `power_state` parameter bit fields in Original format

Bit field	Description
31:26	Reserved. Must be zero.
25:24	PowerLevel
23:17	Reserved. Must be zero.
16	StateType
15:0	StateID

The fields are described below

PowerLevel:

Describes the power level, as defined in section 4.2, to be powered down, such as core, cluster, or group of clusters. Note that it is only possible to call CPU_SUSPEND from the current core, that is, it is not possible to request suspension of another core.

Note: Versions of PSCI prior to 1.0 called this field AffinityLevel. This name has been changed in acknowledgement of the fact that MPIDR based levels of affinity do not necessarily map to power domain implementations

---

How the power levels are numbered is IMPLEMENTATION DEFINED, however, as described in section 4.2, it is expected that lower numbers are closer to the core and higher levels to the system. For the example system depicted in Figure 3, a logical numbering scheme is:

- Level 0: for cores
- Level 1: for clusters
- Level 2: for system

StateType:

A value of 0 indicates a standby or retention state as defined in section 4.1.

A value of 1 indicates a power-down state as defined in section 4.1. This also indicates that the entry_point_address and context_id contain valid data.

The state type is always considered to be power-down if any core context is lost, even if at power levels higher than a core, there is no loss of context. For example, if the state implies powering down a core but keeping a cluster in retention, the StateType field would denote power-down.

StateID:

Field to express a platform-specific state ID. Contents are IMPLEMENTATION DEFINED.

This field can be used to distinguish which combination of local states is being requested.

Through the use of StateID and the other fields in the `power_state` parameter, it must be possible to uniquely describe every composite power state that the calling OS can use. In addition, in OS Initiated mode, the StateID encoding must allow expressing the power level in which the calling core is the last to go idle. The method by which this is done must be expressed to the OSPM via firmware tables (FDT or ACPI), so that it can add this information to the StateID when requesting a power state.

A recommended encoding for StateID is discussed in section 6.5.

5.4.2.2 Extended StateID format

A given platform will support a fixed set of states for each core, cluster, or overall system. These give rise to a set of valid `power_state` values. ARM recommends that these states are expressed through firmware tables, for example, ACPI or FDT, to OSPMs. This aids generalization of OSPM code.

PSCI 1.0 introduces a new extended StateID format. This provides higher flexibility for the implementers of PSCI, and acknowledges improvements in ACPI and FDT power state descriptions, which make some of original format fields redundant.

When this format is in use, bit 1 of the flags field returned by `PSCI_FEATURES` with a `CPU_SUSPEND` function ID, is set to 1.

Note that it is not possible for an implementation to mix formats. It either uses the original format or the extended ID format but not both. This is a static design time decision and cannot be changed at runtime.

The bit fields in the `power_state` parameter take the following format in this case:

Table 9 `power_state` parameter bit fields in Extended StateID format

Bit field	Description
31	Reserved. Must be zero.
30	StateType
29:28	Reserved. Must be zero.
27:0	StateID

The fields retain the same definitions as in the original format.

A recommended encoding for StateID is discussed in section 6.5.

5.4.3   CPU_SUSPEND parameters: `entry_point_address`

The `entry_point_address` parameter is used by the caller to specify where code execution needs to resume at wakeup time. The parameter must be a Physical Address (PA), or, for a guest OS in a virtualized platform, an Intermediate Physical Address (IPA).

In this case, the hypervisor must trap the call. Further details can be found in sections 6.4.1 and 6.4.2.

5.4.4 CPU_SUSPEND parameters: `context_id`

The `context_id` parameter is only meaningful to the caller. The PSCI implementation must preserve a copy of the value passed in this parameter. Following wakeup from a power-down state, the PSCI implementation must place this value in R0, W0, or X0, when it first enters the Trusted OS. The context identifier can be used by the caller to point to the saved context that must be restored on a core, when it starts up at the return Exception level (see section 6.4.1). The caller can use other methods to implement this functionality.

5.4.5 Caller responsibilities

Before a CPU_SUSPEND call, the Normal world must observe the following:

- For a power down request, the calling supervisory software must have saved all the state it requires to enable resumption of operation on reset. The context saved must reflect all caller-visible state that is lost if the power level indicated by the power state parameter is powered down. For more information on the required state following the application of power after a CPU_SUSPEND or CPU_ON call, see section 6.4.2.

- For a power down request, the caller is not required to perform any cache or coherency management. This management must be performed by the PSCI implementation.

- The caller must not assume that a power-down request will return using the specified entry point address. The power down request may not complete due, for example, to pending interrupts. It is also possible that, because of coordination with other cores, the actual state entered is shallower than the one requested. Because of this it is possible for an implementation to downgrade the power-down state request to a standby state. In the case of a downgrade to standby the implementation returns at the instruction following the PSCI call, at the Exception level of the caller, instead of returning via the specified entry point address. The return code in this case is SUCCESS. In the case of an early return due to a pending wakeup event, the implementation can return at the next instruction, with a return code of SUCCESS, or resume at the specified entry point address.

- The caller must ensure that appropriate wakeup events are enabled to allow resumption from that state.

- The entry point address provided by the CPU_SUSPEND call must be a physical address from the point of view of the caller.

- The context identifier is meaningful only to the caller. The value is preserved by the implementation and presented to the core when it is started up at the return Exception level.

The caller must be able to handle potential return error codes:

- INVALID_PARAMETERS is returned if any of the following is true:

  - The power state parameter supplied is invalid. It is expected that the valid lists of power states are supplied by platform firmware tables, for example, ACPI or Flattened Device Tree.

  - In OS Initiated mode, this error is also returned when the following two conditions are true:

    - A low-power state is requested for a higher than core level topology node.

    - At least one of the children in that node is in a local low-power state that is incompatible with the request.

> For example a core makes a request to place the system level node into a power-down state, and another core in the system node is in retention.

- INVALID_ADDRESS is returned when the entry point address is known by the implementation to be invalid, because it is in a range that is known not to be available to the caller.

  Note: Versions of PSCI prior to 1.0 use INVALID_PARAMETERS where INVALID_ADDRESS applies.

- In OS Initiated mode a return value of DENIED is returned if the following two conditions are true:

  - A low-power state is requested for a higher than core level topology node.

  - All the cores that are in an incompatible state with the request are running, as opposed to being in a low-power state.

  In OS Initiated mode INVALID_PARAMETERS can also be returned if the system is in a state that is inconsistent with the caller's request. The differences are:

  - In the DENIED case the inconsistent cores must be running. This may arise though errors in the calling OS or races due to the different views of node state between the calling OS and the implementation. See section 6.2 for more details.

  - In the INVALID_PARAMETERS case, inconsistent nodes must be in a low-power state, and the inconsistency can only arise through errors in the calling OS.

5.4.6 Implementation responsibilities: State coordination

In platform coordinated mode, the semantic expressed by the caller through the power state parameter is not a mandatory requirement to enter the specific state. Instead, the power state parameter indicates the deepest state the caller can tolerate. The PSCI implementation is expected to coordinate requests coming from all idle cores to determine the deepest state that the cores can tolerate. For the purposes of this coordination, a core that has not been powered up through a call to CPU_ON, or that has been powered down through a call to CPU_OFF, is assumed to have requested the deepest platform state available. More details on platform coordination can be found in section 4.2.3.1.

In OS Initiated mode, the caller is making an explicit request for a specific power state, as opposed to expressing a vote. The implementation must comply with the request, unless it is not consistent with the implementation's view of the current state of the system. In that case, the call should return immediately with DENIED or INVALID_PARAMETERS as described above. More information on OS Initiated mode can be found in section 4.2.3.2

5.4.7 Implementation responsibilities: Interaction with a Trusted OS

When a caller requests a power state, the PSCI implementation in the secure platform firmware might need to communicate with a Trusted OS. The method for interfacing between SPF and a Trusted OS is IMPLEMENTATION DEFINED. However, the specification requires the Trusted OS to always comply with CPU_SUSPEND requests. The SPF can inform the Trusted OS that a power state needs to be entered, and the Trusted OS can use this information to take any preparatory actions. For example, it might have to save its context. However, this communication must not allow the Trusted OS to modify or prevent the power state requested.

A possible mechanism for this communication is for the SPF to offer a notification-based scheme to the Trusted OS. The notification scheme might allow the execution of Trusted

---

OS code when the SPF receives a `CPU_SUSPEND` call, on cores where the Trusted OS is resident.

The Trusted OS might not be able to tolerate a particular state, for latency or other reasons. In this case, ARM recommends that the Trusted OS uses an IMPLEMENTATION DEFINED mechanism to communicate with the Normal world to ensure its constraints are considered in the power requests originating from the Normal world.

5.4.8 Implementation responsibilities: Cache and coherency management

Power-down states generally require a cache clean. Prior to powering down a topology node, the PSCI implementation must perform a cache clean operation for all of the caches present in that node, and the last children of that node that are being powered down. The implementation must also perform any required coherency management. In addition, the PSCI implementation needs to perform invalidation of caches on booting, unless this is automatically supported by the hardware, and to manage coherency. Sequences to be observed when powering cores up or down can be found in the Technical Reference Manuals for the relevant processors and interconnect IP.

5.4.9 Implementation responsibilities: State upon return

When returning from a standby state, the caller should observe no change in core state, other than any timer changes expected because of the time spent in the state, and changes in CPU interface because of the wakeup reason. To the core, a standby state is indistinguishable from the use of a `WFI` instruction. The only exceptions are the registers used in making the SMC call that follow the SMC Calling Conventions [4]. The return value expected in R0 or W0 is the return error code. For standby states, `SUCCESS` must be returned on success. Power-down states do not return on success because restart is through the entry point address at wakeup. If not successful, the error code indicates the reason.

The core state when it exists the Secure world following wake up from a power-down state, or a CPU_ON call, is described in described in section 6.4.3.

5.5 CPU_OFF

5.5.1 Intended use

`CPU_OFF` is designed for dynamic removal of the calling core from the system. When SPF receives a `CPU_OFF` call, it must power down the calling core.

Unlike `CPU_SUSPEND`, the call is not expected to return. With this function the calling supervisory software is explicitly stating that it will no longer use a core. This situation is only reversed when a `CPU_ON` call is used to bring the core back online.

`CPU_OFF` calls can only originate from the Normal world. If the Secure world needs to manage the number of active cores, it needs to use Trusted OS specific IMPLEMENTATION DEFINED communication with the Normal world to achieve its aims. It must not be possible for Secure world to allow a core to be seen as powered up and available by the Trusted OS, while the Normal world sees the core as being powered down.

A description of the call flow can be found in section 6.1.2.

5.5.2 Caller responsibilities

Before a `CPU_OFF` call the following must be observed:

- The calling OS must have migrated all threads and interrupts away from the core that is being powered down. Asynchronous wakeups on a core that has been switched off through a PSCI `CPU_OFF` call result in an erroneous state. When this erroneous state

is observed, it is IMPLEMENTATION DEFINED how the PSCI implementation reacts. Possible actions are:

- The PSCI implementation ignores the wakeup and keeps the core powered down.

- The PSCI implementation resets the system.

- For a power down request, the caller is not required to perform any cache or coherency management. This management must be performed by the PSCI implementation.

- CPU_OFF can have varying latency in the time it takes to complete, depending on the power domain level that is being powered down. The caller must not make any assumptions about latency.

- A core can only power itself down.

- CPU_OFF calls are always platform coordinated with each other, regardless of the suspend mode of operation (platform coordinated or OS initiated). That is, if all cores in a topology node call CPU_OFF, the last core will power the node down.

  - In an OS Initiated mode, CPU_OFF calls are considered to platform coordinate between themselves. If only a subset of cores has called CPU_OFF, an OS initiated OS can assume they have powered down to the highest power level possible. Referring to the example system in Figure 3, if all cores except Core 3 have called CPU_OFF, the OS can assume that Cluster 0 and Core 2 are in the power-down state and that Core 3 is the last man in the system (and cluster 1). Therefore in this mode the OS could power down the system through a CPU_SUSPEND call from Core 3. However, this coordination only works for calls to CPU_SUSPEND that take place after calls to CPU_OFF. Coordination in the opposite order cannot be assumed. If in our example Core 3 was the penultimate core and this called CPU_SUSPEND, and Core 2 was the last one, and it called CPU_OFF, Cluster 1 (and the system) would not be able to power down. This is because Core 3 would not be a last man, at either the cluster or the system level, and consequently it could not request a power state at that level.

  - In platform coordinate mode, a call to CPU_OFF is consistent with a call to CPU_SUSPEND for power down at maximum power level. If only a subset of the cores in a node call CPU_OFF, the node must only be powered down if the remaining cores have called CPU_SUSPEND, requesting a power-down state at that power level, or a power state at a higher power level.

CPU_OFF can return the following errors:

- DENIED, if called on a core where the Trusted OS is resident, see section 5.9.1.

5.5.3 Implementation responsibilities: Cache and coherency management

As with entry into a power-down state, the PSCI implementation must perform a cache clean operation for all of the caches present in the nodes being powered down. PSCI must also perform any required coherency management. In a multicluster implementation, powering down a core typically requires:

1. Disabling the data cache to prevent data cache allocation.

2. Cleaning and invalidating any caches that are private to the core. This prevents any new data cache snoops or data cache maintenance operations from other cores being issued to the core that is powering down.

3. If the core is the last in a cluster, and the cluster is being powered down, cleaning and invalidating any shared caches that are private to that cluster.

4. Taking the core out of coherency within that cluster. This is typically controlled by an IMPLEMENTATION DEFINED method, for example ACTLR.SMP.

5. If the core is the last in a cluster, and the cluster is being powered down, taking the cluster out of coherency.

The operations required depend on the specific cores and topology, as well as any cache coherent interconnects that may be in use. For more information see the *Technical Reference Manual*s (TRMs) of the components.

5.6   CPU_ON

5.6.1   Intended use

CPU_ON is intended for dynamic addition of cores, for use in secondary boot, hotplug, or big.LITTLE migration. When an OS at an Exception level requires another core, it calls the CPU_ON function, providing an entry point address and a context identifier.

The PSCI implementation provides the necessary platform-specific code to program the power controller, as required to turn on a core, and then restart its execution at the entry point address. The context identifier value must be present in R0, W0, or X0 when the core first enters the return Exception level. Section 6.4.1 describes the return Exception level, and section 6.4.3 describes the expected state for a core when it accesses the entry point address. Both CPU_ON and CPU_OFF calls can only originate from the Normal world. If a Trusted OS needs to manage the number of cores, it needs to use a Trusted OS specific IMPLEMENTATION DEFINED communication with the Normal world to achieve its aims.

A description of the call flow can be found in section 6.1.3.

5.6.2   Caller responsibilities

Before a CPU_ON call the following must observed:

- CPU_ON must be implemented asynchronously. The mechanism by which supervisory software can detect that the core is finally powered up is IMPLEMENTATION DEFINED. However the provision of caller-specific entry point and context identifier parameters allows the calling supervisory software to implement such a mechanism.

- The entry point address provided needs to be physical from the point of view of the caller.

- The context identifier is meaningful only to the caller. The value is preserved by the implementation and presented to the core when it starts up at the return Exception level (section 6.4.1).

CPU_ON can return the following errors:

- INVALID_PARAMETERS is returned if target_cpu describes an invalid MPIDR.

- INVALID_ADDRESS is returned when the entry point address is known by the implementation to be invalid, because it is in a range that is known not to be available to the caller.

    Note: Versions of PSCI prior to 1.0 use INVALID_PARAMETERS where INVALID_ADDRESS applies.

- ALREADY_ON is returned if in the PSCI implementation the core is already in an ON state (see section 6.6).

- ON_PENDING is returned if a call to CPU_ON on the target core has already been made, and the core is not yet ON in the PSCI implementation (see section 6.6)

- INTERNAL_FAILURE can be returned if a core cannot be powered up for physical reasons. Examples include lack of power, thermal constraints, manufacturing faults, or reliability reasons.

5.6.3 Implementation responsibilities: Cache management

For CPU_ON, the PSCI implementation must:

- Perform invalidation of caches on boot, unless this invalidation is automatically performed by the hardware.

- Manage coherency.

5.7 AFFINITY_INFO

5.7.1 Intended use

The valid states that can be returned by AFFINITY_INFO are:

ON: At least one core in the affinity instance, as defined in section 4.2.2, meets both of the following conditions:

- The core has been enabled with a call to CPU_ON, or is the cold boot primary core.

- The core has not called CPU_OFF.

Idling cores that have called CPU_SUSPEND are considered to be in the ON state. Therefore a core in this state will be running or in a low power mode.

OFF: All of the cores in the affinity instance have called CPU_OFF and each of these calls has been processed by the PSCI implementation.

ON_PENDING: At least one core in the affinity instance is in the ON_PENDING state, as described in section 5.5. All other cores in the affinity instance are OFF.

It is also possible for AFFINITY_INFO to return the DISABLED code when the processing unit described by the input parameters is valid, but is disabled for physical reasons, for example, because a core is faulty.

For versions of PSCI prior to 1.0, an implementation must track the states of all the cores in an affinity instance to produce the overall return value. From version 1.0 of PSCI, it is no longer required for AFFINITY_INFO to support affinity levels greater than 0. If a caller requests the state of an affinity instance above level 0, the implementation can respond in one of two ways:

- Provide a valid return value, behaving in a compatible fashion with a version of PSCI prior to 1.0.

- Return INVALID_PARAMETERS.

5.7.2 Caller responsibilities

Callers must be aware that the state returned by the AFFINITY_INFO call is affected by calls to CPU_ON and CPU_OFF, both of which could be in flight at the same time.

The caller must deal with any errors arising from the call:

- INVALID_PARAMETERS is returned if the lowest_affinity_level and target_affinity parameters describe an affinity instance that is not present in the platform.

- DISABLED is returned if the parameters describe an affinity instance that is disabled for physical reasons, for example, because the core is faulty.

5.8 MIGRATE

5.8.1 Intended use

The MIGRATE function supports powering down a core where a uniprocessor (UP) migrate-capable Trusted OS is resident. The function should be used to move the Trusted OS to another core, and thus enable the original core to call CPU_OFF to power down. This function can also be used to support big.LITTLE systems that use a migration scheme. A description of how MIGRATE can be used to support such a scheme is provided in section 5.8.

5.8.2 Caller responsibilities

The following must be observed by the caller:

- The migration target core must be powered up before any call to MIGRATE.

- The target core must not make requests for Trusted OS services until after the migration has completed.

- MIGRATE must only be called from the core on which the Trusted OS is present. If called from another core, the implementation must return a NOT_PRESENT error.

MIGRATE can return the following errors:

- NOT_SUPPORTED if the function is not supported, or if migration is not required (see section 5.8).

- INVALID_PARAMETERS if target_cpu describes an invalid MPIDR. This value is also returned if the register width of the caller does not match the register width of the function ID used in the call.

- DENIED can be returned if the Trusted OS is UP but not migrate-capable, see section 5.8.

- INTERNAL_FAILURE is returned for an IMPLEMENTATION DEFINED reason why the MIGRATE was not possible. For example, the target core was not awake or did not respond within an acceptable time.

- NOT_PRESENT if called on a core where the Trusted OS is not currently resident.

5.8.3 Implementation responsibilities

Implementation of the MIGRATE function is optional.

The mechanism by which the secure platform firmware coordinates with a Trusted OS to migrate context is IMPLEMENTATION DEFINED. However ARM suggests that a callback registration scheme is provided by the secure platform firmware to the Trusted OS vendor. The secure platform firmware can use this to notify the Trusted OS when key PSCI functions like MIGRATE are received.

A UP Trusted OS that is migrate-capable can register for the callbacks. The secure platform firmware must track the resident core of a UP Trusted OS. This way it can directly respond to a number of error conditions for a MIGRATE call without having to call into the Trusted OS. In particular, it is possible to handle the following errors directly from the secure platform firmware:

- NOT_SUPPORTED

- INVALID_PARAMETERS

- DENIED

- NOT_PRESENT

5.9 MIGRATE_INFO_TYPE and MIGRATE_INFO_UP_CPU

5.9.1 Intended use

Trusted operating systems must be rigorously audited to ensure they meet their security requirements. This means that they must be kept simple. Often this leads to simple uniprocessor kernels, to avoid the inherent complexities of multicore programming. This has implications for when it is possible to hot unplug the core where the Trusted OS is resident.

The MIGRATE function provides a method to request a uniprocessor Trusted OS to move to another core. However this might not be supported by the Trusted OS vendor. The Normal world can use the MIGRATE_INFO functions to determine whether it can use CPU_OFF and MIGRATE for a given Trusted OS. MIGRATE_INFO_TYPE takes no parameters and returns:

- **Uniprocessor (UP) and migrate capable 0:** This indicates a Trusted OS that can run on only one core. The core it runs on can be changed dynamically through use of the MIGRATE function. Calls to CPU_OFF on the core where the Trusted OS is currently residing return a DENIED error.

- **Uniprocessor (UP) not migrate capable 1:** This indicates a Trusted OS that does not support migration. Calls to hot-unplug the core on which the Trusted OS is running return a DENIED error. Calls to MIGRATE return the same error.

- **Multiprocessor (MP) capable or not present 2**: This indicates that the Trusted OS is fully MP-aware and has no special requirements for migration, or a system that does not use a Trusted OS. Any core can be hot unplugged without first asking the Trusted OS to migrate to another core. In this case, calls to MIGRATE are not relevant. They return NOT_SUPPORTED and have no other effect.

In addition, MIGRATE_INFO_UP_CPU takes no parameters and returns an MPIDR-based value to indicate where the Trusted OS currently resides. The value returned uses the same format as the target_cpu parameters of CPU_ON and MIGRATE, see section 5.1.4. The returned value is only valid if MIGRATE_INFO_TYPE returns 0 or 1. The returned value of MIGRATE_INFO_UP_CPU is UNDEFINED if MIGRATE_INFO_TYPE returns 2.

For each possible return value of MIGRATE_INFO_TYPE, Table 10 shows the expected return values for calls to MIGRATE and CPU_OFF, when called on the CPU where the Trusted OS currently resides. The table assumes that valid parameters are passed, that is, that valid target_cpu expressions are passed.

Table 10 Return values for calls to MIGRATE and CPU_OFF (on resident core)

MIGRATE_INFO_TYPE	Return value of MIGRATE	Return value of CPU_OFF
MP or not present 2 or NOT_SUPPORTED	NOT_SUPPORTED	does not return
UP not migrate capable (1)	DENIED	DENIED
UP migrate capable (0)	SUCCESS	DENIED

The information returned by MIGRATE_INFO_TYPE in must be constant and not change on subsequent calls.

5.9.2 Caller responsibilities

The caller can target any core with the `MIGRATE_INFO` functions. The caller can also assume that the value returned by `MIGRATE_INFO_TYPE` is constant, and will not change from one invocation to another.

The caller must avoid races between `MIGRATE` and `MIGRATE_INFO_UP_CPU`. ARM expects that the latter function will be used once by a caller, upon initial boot. From then on, a migrating OS can make use of the success or failure of `MIGRATE` calls to track Trusted OS residency.

5.9.3 Implementation responsibilities

`MIGRATE_INFO` functions can be called from any core. How the Secure world obtains and tracks the information provided through these functions is IMPLEMENTATION DEFINED. However, ARM expects that, at cold boot, the secure platform firmware optionally installs a Trusted OS. As part of the installation process, an API can be defined between the secure platform firmware and the Trusted OS that, in turn, can be used to provide the required return values of `MIGRATE_INFO_TYPE`. The secure platform firmware is aware of which core the Trusted OS is being installed on. By using this information, and tracking the success or failure of `MIGRATE` calls, the secure platform firmware can directly respond to `MIGRATE_INFO` calls from any core.

5.10 SYSTEM_OFF

5.10.1 Intended use

`SYSTEM_OFF` provides a system shutdown API.

As detailed in section 4.5, the shutdown command applies to the machine view of the calling OS.

The caller must place all cores in a known state prior to the call. Like all other APIs presented here, the calls can only originate in the Normal world. On a `SYSTEM_OFF` call the implementation completely removes power from highest power level.

The following start must be a cold boot from the point of view of the caller.

5.10.2 Implementation responsibilities

- The implementation must support `SYSTEM_OFF` calls from every core in the system. If a UP Trusted OS is present, and the call does not come from the core on which it is resident, then the implementation has two options:
    - If the Trusted OS requires it, provide an IMPLEMENTATION DEFINED mechanism to inform the Trusted OS of the impending shutdown.
    - Use a Trusted OS that can cope with the shutdown. In particular, in mobile applications Trusted Operating Systems must be able to deal with a sudden loss of power.

- The implementation must ensure that any data it requires is saved to non-volatile storage.

5.10.3 Caller responsibilities

The caller must perform any necessary operations to ensure a clean shutdown in its OS:

- Ensure cores are in a known state and that any necessary data has been saved to non-volatile storage.
- It is up to the calling supervisory software to store any information it requires on restart in storage that will survive the shutdown.

One way in which cores can be placed into a known state is to use calls to CPU_OFF on all online cores except for the last one, which instead uses SYSTEM_OFF. If a UP Trusted OS is present, this method only works if the core that calls SYSTEM_OFF is the one where the Trusted OS is resident, as calls to CPU_OFF on this core return a DENIED error. Any core can call SYSTEM_OFF.

5.11 SYSTEM_RESET

5.11.1 Intended use

This function provides a method for performing a platform cold reset. To the caller the behavior is equivalent to a hardware power-cycle sequence. As detailed in section 4.5, the reset command applies to the machine view of the calling OS.

An external debugger can use the **DBGNOPWRDWN[1,5]** signal to prevent the reset of the debug logic. This changes the effect of SYSTEM_RESET into a warm reset, as defined by the ARM debug architecture [1,5].

5.11.2 Caller responsibilities

Where possible, cores should be placed in a known state. No specific coordination is required between cores. Apart from the case when the response to the call is virtualized, when one core calls SYSTEM_RESET, the system power cycles.

5.11.3 Implementation responsibilities

See section 5.11.1.

5.12 PSCI_FEATURES

Introduced in PSCI1.0, this function is mandatory.

5.12.1 Intended use

This function allows the caller to detect which PSCI functions have been implemented and their properties.

PSCI functions generally return NOT_SUPPORTED when they are not implemented. However it is not always convenient to use that mechanism for discovery. Optional APIs such as CPU_DEFAULT_SUSPEND or CPU_FREEZE can cause entry of a CPU into a low-power state. Consequently, calling the functions is not an appropriate mechanism for discovering their presence. The PSCI_FEATURES API identifies whether a function is present or not without needing to call the specific function.

The caller passes the function ID in the psci_func_id parameter. The implementation returns:

- If the function is implemented, a set of feature flag bits. In this case, bit 31 is always zero. Thus, positive 32-bit integers indicate implemented functions.

- NOT_SUPPORTED, if the function is not implemented.

Table 18 in section 6.10 lists all the compulsory and optional functions for a given PSCI version.

5.12.2 Implementation responsibilities

If a function described by psci_func_id is not implemented, the implementation must return NOT_SUPPORTED. For functions that are implemented, Table 11 lists possible return values:

Table 11 Return values if a function is implemented

psci_func_id parameter	Feature Flags

Function ID for CPU_SUSPEND (0x8400 0001 SMC32 version 0xC400 0001 SMC 64 version)	Bits [31:2] : Reserved, must be zero
	 Bits[1:1] : Parameter Format bit
	      0 if the implementation uses Original Format (PSCI0.2) for the power_state parameter
	      1 if the implementation uses the new extended StateID Format for the power_state parameter
	 Bits[0:0]: OS Initiated Mode
	      0 if the platform does not support OS Initiated mode
	      1 if the platform supports OS Initiated mode

Function ID for any function other than CPU_SUSPEND	Bits [31:0]: Reserved, must be zero

5.13 CPU_FREEZE

Introduced in PSCI1.0, this function is optional.

5.13.1 Intended use

This API places a core in an IMPLEMENTATION DEFINED low-power state. It differs from other APIs that place cores into low-power states in the following ways:

- Unlike CPU_SUSPEND or CPU_DEFAULT_SUSPEND, an interrupt does not return the core back into a running state. Instead, the interrupt must remain pending.

- Unlike CPU_OFF, the calling OS is not required to migrate interrupts away from the core prior to calling CPU_FREEZE.

The effect of interrupts on a core that is powered down with a PSCI API is described in the table below:

Table 12 Effect of wake interrupts

PSCI Function	Interrupt Effect
CPU_SUSPEND	Wake up
CPU_DEFAULT_SUSPEND	Wake up
CPU_OFF	Error condition, effect is IMPLEMENTATION DEFINED

`CPU_FREEZE`	core must remain in power down, and interrupt must pend

The function must only be implemented if the power controller or power controlling logic of the platform allows wakeup interrupts for a given core to be ignored.

This API is meant for debug cases where an OS wants to allow powering down of any core that is not running a debugger.

5.13.2 Implementation responsibilities

The power state into which the platform places the calling core is IMPLEMENTATION DEFINED. The level of coordination that is performed in this power state is also IMPLEMENTATION DEFINED. For example, if all cores in a cluster call `CPU_FREEZE`, the implementation can choose to power down the cluster. Although the choice of state and coordination is up to the implementation, it must not be possible for the platform to enter a thermally critical condition if all except one core have called `CPU_FREEZE`.

The implementation must be able to ensure that wakeup interrupts will not bring the core back into execution. If this is not possible, `CPU_FREEZE` must not be provided.

5.13.3 Caller responsibilities

The caller must follow a standard power down sequence appropriate to the core. Return to execution is performed through a call to `CPU_ON`. Therefore, the caller must observe the following conditions:

- The caller must save all state required to enable resumption of execution through the call to `CPU_ON`.

- The caller is not required to perform any cache or coherency management. This management must be performed by the PSCI implementation.

- As the power state entered by a `CPU_FREEZE` is IMPLEMENTATION DEFINED, the caller cannot make assumptions about what power state the core is in. Equally the level of coordination with power states in other cores in the topology is IMPLEMENTATION DEFINED and therefore the calling OS cannot make any assumptions about it. For example if all cores in a cluster call `CPU_FREEZE` it is not safe to assume that any core or the cluster is fully power gated. For a given topology node, it is only safe to make assumptions about its power state when all children cores been resumed from `CPU_FREEZE`.

When using OS Initiated Idle, a simple scheme to prevent errors in idling when cores have called CPU_FREEZE is for the OS to consider those as running cores in its last man tracking algorithm.

`CPU_FREEZE` does not return when it is successful, but might return the following error codes:

- `NOT_SUPPORTED` if the API is not implemented.

- `DENIED` in the same cases as those where `CPU_OFF` also returns `DENIED,` as described in section 5.9.1

5.14 CPU_DEFAULT_SUSPEND

Introduced in PSCI1.0, this function is optional.

5.14.1 Intended use

From the point of view of the caller, this API is equivalent to calling `CPU_SUSPEND` with a `power_state` parameter reflecting a core power-down state, as defined in section 4.1. Unlike `CPU_SUSPEND`, however, the `power_state` parameter is omitted. The actual power state entered is IMPLEMENTATION DEFINED.

In every other way, the semantics are like a `CPU_SUSPEND` and are therefore primarily described in section 5.4.

The `CPU_DEFAULT_SUSPEND` function is optional. The function is meant for systems where all the information required to use `CPU_SUSPEND`, that is the power states and their properties, is available in the later phases of boot. In these systems, `CPU_DEFAULT_SUSPEND` provides an alternative to WFI that can be used during early boot.

5.14.2 Implementation responsibilities

The power state into which the platform places the calling core is IMPLEMENTATION DEFINED. The level of coordination that is performed in this power state is also IMPLEMENTATION DEFINED. For example, if all cores in a cluster call `CPU_DEFAULT_SUSPEND`, the implementation can choose to power down the cluster. Although the choice of state and coordination are up to the implementation, it must not be possible for the platform to enter a thermally critical condition if all cores have entered a low-power state as a result of calling `CPU_DEFAULT_SUSPEND`

5.14.3 Caller responsibilities

Before a `CPU_DEFAULT_SUSPEND` call, the Normal world must observe the following conditions:

- The calling supervisory software must have saved all the state it requires to enable resumption of operation on reset. The context saved must reflect all caller-visible state that would be lost by the core. For more information on the required state following the application of power after a `CPU_SUSPEND`, `CPU_DEFAULT_SUSPEND`, or `CPU_ON` call, see section 6.4.3.

- The caller is not required to perform any cache or coherency management. This management must be performed by the PSCI implementation.

- The caller must not assume that a power-down state is entered. The actual state used is IMPLEMENTATION DEFINED, and can be a standby state. In addition, the request might not complete due, for example, to pending interrupts. Therefore the function might return at the next instruction, or return at the specified entry point address. If returning at the next instruction due to pending interrupts, or because the implemented state is a standby state, the return code is `SUCCESS`.

- The caller must be able to handle potential return error codes:

    - `INVALID_ADDRESS` is returned if the entry point address is known by the implementation to be invalid because it is in a range that is known not to be available the caller.

- The caller must ensure that appropriate wakeup events are enabled to allow resumption from that state.

- The entry point address provided by the `CPU_DEFAULT_SUSPEND` call must be a physical address from the point of view of the caller.

- The context identifier is meaningful only to the caller. The value is preserved by the implementation and presented to the core when it is started up at the return Exception level (section 6.4.1).

- The power state entered by a CPU_DEFAULT_SUSPEND is IMPLEMENTATION DEFINED. The caller cannot make assumptions about what the power state might be. Equally, the level of coordination with power states in other cores in the topology is IMPLEMENTATION DEFINED and therefore the calling OS cannot make any assumptions about it. For example if all cores in a cluster call CPU_DEFAULT_SUSPEND it is not safe to assume that any core or the cluster will be fully power gated. For a given topology node, it is only safe to make assumptions about its power state when all children cores been resumed from CPU_DEFAULT_SUSPEND. When using OS Initiated Idle, a simple scheme to prevent errors in idling when cores have called CPU_DEFAULT_SUSPEND is for the OS to consider those as running cores in its last man tracking algorithm.

5.15 NODE_HW_STATE

Introduced in PSCI1.0, this function is optional.

5.15.1 Intended use

This API allows the power state of a node to be determined directly from the power controller or power controlling logic. Unlike AFFINITY_INFO, which returns the view of state of the implementation, this API returns the physical view of power state.

5.15.2 Caller responsibilities

The node is uniquely represented using two parameters:

target_cpu:

Derived from the MPIDR, this parameter follows the same format as the target_cpu parameter of CPU_ON.

power_level:

This parameter denotes the power domain level for the node. The format of this parameter is IMPLEMENTATION DEFINED, however 0 is always reserved to represent a core, or HW thread, in an SMT system.

> **Note**: It is acknowledged that an SMT system is highly unlikely to have a power domain around hardware thread. However, it is useful to use 0 to uniquely identify the level that aligns to an operating system's view of a core.

Together, both parameters allow indexing any individual node in the power node topology of the system. Power level values to be used for this function should be described alongside the power domain topology using firmware table approaches such as FDT or ACPI. More details can be found in section 6.7.

The calling OS must recognize that the returned value is a snapshot of the state of the node at a point in time, and might therefore be stale by the time it is interpreted. In addition the call my return the following errors:

- NOT_SUPPORTED if not implemented.

- INVALID_PARAMETERS if the target_cpu and power_level values describe an invalid node.

5.15.3 Implementation responsibilities

This API must only be implemented if the supporting power control logic allows detection of the power state of nodes in the system. The values returned by this function must reflect the view of the power controlling logic.

If successful, the values returned by NODE_HW_STATE are described below:

- HW_STANDBY(2) if the node is seen by the power control logic as being in a retention of standby state (as defined by the semantics described in Section 4.1).

- HW_OFF(1) if the node is seen by the power control logic as being in a power-down state (as defined by the semantics described in Section 4.1).

- HW_ON(0) if the node is seen by the power control logic as being in a run state (as defined by the semantics described in Section 4.1).

5.16   SYSTEM_SUSPEND

Introduced in PSCI1.0, this function is optional.

5.16.1 Intended use

In a typical implementation, the semantics are equivalent to a CPU_SUSPEND to the deepest low-power state. However it is possible that an implementation might reserve a deeper state for SYSTEM_SUSPEND than those used with CPU_SUSPEND. This function might be used to implement a system suspend to RAM, as described by the S2 and S3 states in the ACPI specification [6]. Note that entering the system into S2 or S3 carries with it a number of preconditions. For example, all devices in the system must be in a state that is compatible with entry into the system state. These preconditions are beyond the scope of this specification and are therefore not described here. The SYSTEM_SUSPEND function limits itself to providing the mechanism by which the calling OS can request entry into S2 or S3 having met all necessary preconditions. While ACPI distinguishes between two system level suspend to RAM states, S2 and S3, PSCI provides a single API. In practice, operating systems use only one suspend to RAM state, so this is not seen as a limitation. Note that this specification is using the state definitions of S2 and S3 provided by ACPI, but does not limit use of this PSCI function to ACPI based systems. Semantics such as those described for sleep states S2 and S3 are used in non-ACPI based systems.

As is the case with SYSTEM_SHUTDOWN and SYSTEM_RESET, this function applies to the machine view that is available to the calling OS. See section 4.5 for more details.

To use this API, a calling OS must power down all but one core through calls to CPU_OFF. From this point on, the remaining core can call SYSTEM_SUSPEND, passing the necessary entry_point_address and context_id parameters to enable resumption on wakeup. A calling OS can use the AFFINITY_INFO function to ensure that all cores are OFF prior to calling SYSTEM_SUSPEND.

5.16.2 Caller responsibilities

The caller needs to ensure that every core apart from the one calling is in an OFF state, as returned by the AFFINITY_INFO function. The caller must be able to cope with the following error return codes:

- NOT_SUPPORTED if system suspend is not implemented.

- INVALID_ADDRESS if the entry point address is known by the implementation to be invalid because it is in a range that is known not to be available to the caller.

- DENIED if the implementation views a core, other than the one calling, as not being in the OFF state.

Prior to the call, the OS must disable all sources of wakeup other than those it wishes to support for its implementation of suspend to RAM. The call is equivalent to using the CPU_SUSPEND call for the deepest possible platform power-down state. Consequently the caller must observe all the rules described for CPU_SUSPEND. See section 5.4.

5.17 PSCI_SET_SUSPEND_MODE

Introduced in PSCI1.0, this function is optional.

5.17.1 Intended use

This function is used to switch between the two different power state coordination modes described in section 4.2 .

The platform will boot in platform coordinated mode. If the calling OS wishes to use OS Initiated, the API is should be called once during OS boot. CPU_DEFAULT_SUSPEND can be used before the suspend mode is set.

Some operating systems support rebooting a new kernel within their Exception level, and without resetting the system as a whole. In this case, the calling OS should place the PSCI implementation back into a platform coordinated mode before rebooting.

5.17.2 Implementation responsibilities

Switching from platform coordinated to OS initiated modes (see section 4.2) is only allowed if the following conditions are met:

- All cores are in one of the following states:

    - Running.

    - OFF through a call to CPU_OFF or simply not yet booted.

    - Suspended through a call to CPU_DEFAULT_SUSPEND.

- None of the processors has called CPU_SUSPEND since the last change of mode or boot.

A switch from OS Initiated mode to platform coordinated mode is only possible if all cores other than the calling one are OFF, either through a call to CPU_OFF or not booted.

If these conditions are not met the platform must return DENIED.

The default coordination mode is platform coordinated.

5.17.3 Caller responsibilities

The caller has to be able to cope with the following error return codes:

- NOT_SUPPORTED, in which case the calling OS can assume the platform will work in platform coordinated mode. In this case the PSCI_FEATURES feature flag 0 for CPU_SUSPEND will also be 0.

- INVALID_PARAMETERS if the parameter mode is not a valid value (0 or 1).

- DENIED:

    - If any core is not in in the correct state, as detailed in the previous section.

    - When switching to OS Initiated mode, if a core has made use of CPU_SUSPEND.

5.18 PSCI_STAT_RESIDENCY/COUNT

Introduced in PSCI1.0, these functions are optional.

5.18.1 Intended use

These APIs provides residency statistics for power states that have been used by the platform. The statistics must be counted regardless of the entry mechanism to state, and

thus, any of the PSCI calls that can result in a power state being entered can contribute to the statistics.

The functions take two input parameters:

`target_cpu`: This parameter follows the same format as the `target_cpu` parameter of a `CPU_ON` call. See section 5.1.4.

`power_state:` As described for the `CPU_SUSPEND` function in section 5.1.2.

Together both parameters can be used by the platform to index a unique state affecting a unique node of the processor power topology of the system, for example, a specific core or a specific cluster.

Only local low-power states, and not running states, are tracked by the statistic functions. However, the composite power state expressed through the `power_state` parameter, can reflect a combination of local states for the various power levels in the system. The functions return the appropriate count or residency time of the local state for the highest power level expressed in the composite `power_state`. In OS Initiated mode, the implementation should disregard the section of the `power_state` parameter that describes the power level in which the calling core is the last man. For example consider the following example of state encoding:

Core level state, bits [3:0]: 0 Run, 1 Standby, 2 Retention, 3 Power-Down.

Cluster level state, bits [7:4]: 0 Run, 2 Retention, 3 Power-Down.

System level state, bits [11:8]: 0 Run, 2 Retention, 3 Power-Down.

Core is last in Level, bits [15:12]: 0 Core, 1 Cluster, 2 System.

`power_state` parameters with StateIDs of 0x#223 and 0x#233, both return statistics for the system level Retention state. Encodings that permit indexing a local state directly are also allowed. In the example above, the local state for system level retention carries a state ID of 0x#200. A recommended encoding for the StateIDs is described in section 6.5.

`PSCI_STAT_COUNT` returns the number of times a `power_state` has been used by the node represented by the `target_cpu` and the highest power level of `power_state`. Note that an OS internal count of state usage (by tracking calls to PSCI) and the value returned by this API might not always agree. This can be for a variety of reasons, for example, the power state requested by the OS and the state entered by the implementation can differ.

`PSCI_STAT_RESIDENCY` returns the amount of time spent in `power_state` in microseconds, by the node represented by the `target_cpu` and the highest level of `power_state`.

Both functions should return zero if the node and `power_state` do not represent a valid combination. This can happen if the `target_cpu` represents a node that is not present or if the node does not support the state passed in `power_state`. Firmware tables, FDT or ACPI, describe the valid states that the calling OS can request from any given core.

5.18.2 Implementation responsibilities

The implementation needs to observe the following conditions:

- The statistics count time spent in a low-power state regardless of the entry method. This includes power states entered through `CPU_OFF`, `CPU_FREEZE`, or `SYSTEM_SUSPEND.`

- Statistics only track local power states. This does not include the time spent in a run state by a node.

- Statistics are set to zero at cold boot. SYSTEM_RESET and SYSTEM_SHUTDOWN have the same effect, because from the perspective of the caller the next boot will be a cold boot.

- It is IMPLEMENTATION DEFINED whether the PSCI_STAT_COUNT updates on entry or exit to a power state.

- Though these functions are optional, both must be implemented together if statistics are supported

5.18.3 Caller responsibilities

The functions use the full resolution of the returned parameters (count or residency). Therefore the functions do not support error codes for return values. If the functions are not implemented, the returned value is NOT_SUPPORTED, but the OS cannot rely on this value to determine whether the functions are implemented. The calling OS must discover if the functions are present by using the PSCI_FEATURES API (see section 5.1.11).

The calling OS must be aware that the returned count or residency values wrap when they overflow the available resolution. This is not a problem for the SMC64 version of the call, but for the SMC32 PSCI_STAT_RESIDENCY function, a wrap is possible in just over 1hr, 11 minutes, and 34seconds.

6   Additional Implementation Details

6.1   PSCI call flows

6.1.1 `CPU_SUSPEND`, `CPU_DEFAULT_SUSPEND`, and `SYSTEM_SUSPEND` call flow

Figure 4 shows how a `CPU_SUSPEND` call (or other suspend call) for a power-down state can flow through the Exception levels and terminate in secure platform firmware. The diagram also shows how the state is saved and restored at each stage. In the model above the hypervisor traps Non Secure-EL1 power requests. This allows a hypervisor at EL2 to save its context, the entry point, and the context ID of the caller, before making its own power state request to the Secure world.
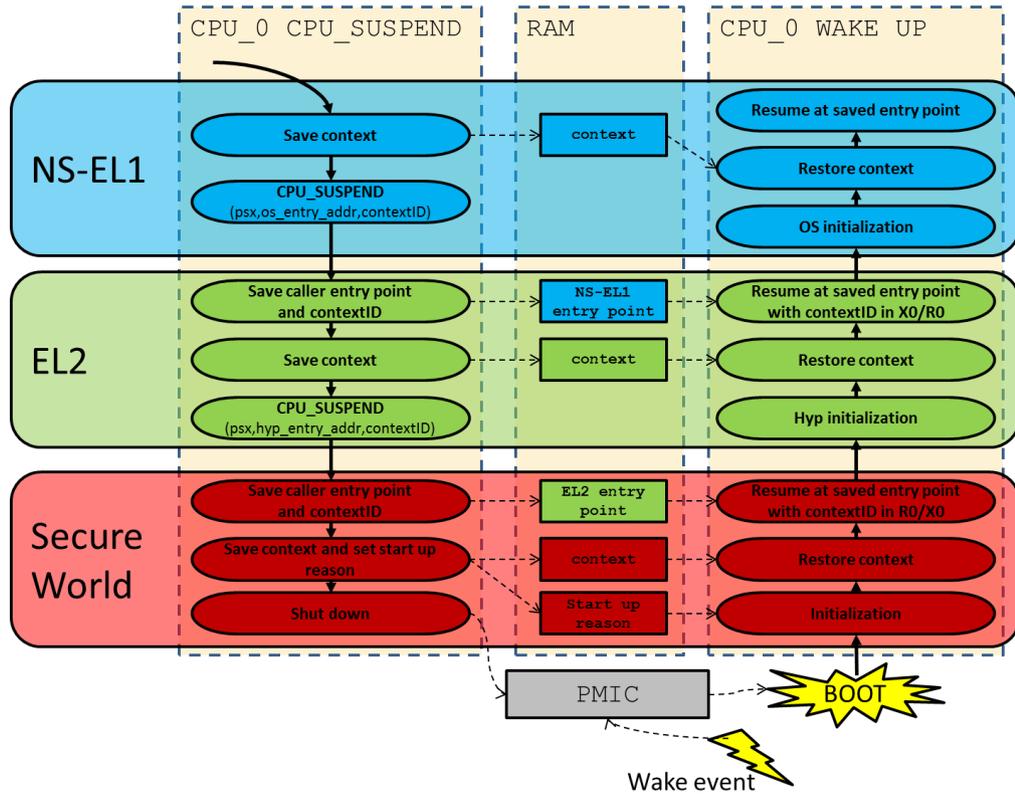
Figure 4 Example call flow for a CPU_SUSPEND call

The Secure world works in a similar way, saving its state, the entry point, and the context ID of its caller, and setting a startup reason, before shutting down the core. When a wakeup event occurs, the power controller boots the core. On initializing, the Secure world sees the startup reason, restores context, and then re-starts the caller process at the saved entry point, supplying the saved context id. The hypervisor then can do the same with the Non-secure EL1 process.

Figure 4 is an example of how the call flow might work when all Exception levels are implemented. As described in section Figure 4, the return Exception level that the Secure world restores to is the highest Non-secure Exception level implemented. This rule allows for the model described above.

In systems that implement EL2 and EL3, this rule also permits bypassing EL2, by allowing a call to go directly from Non-secure EL1 to the Secure world. However, the return path must go through EL2. How this is coordinated is determined by the hypervisor implementation. An example of this approach could be a type 2 system, which enables

Non-secure EL1 calls to go directly to the Secure world on the host, but traps at EL2 for guests.

Systems that do not implement EL3, and therefore do not access the Secure world, can still provide a PSCI interface to Non-secure EL1 rich operating systems that are PSCI compliant by using an HVC conduit.

6.1.2   `CPU_OFF` call flow
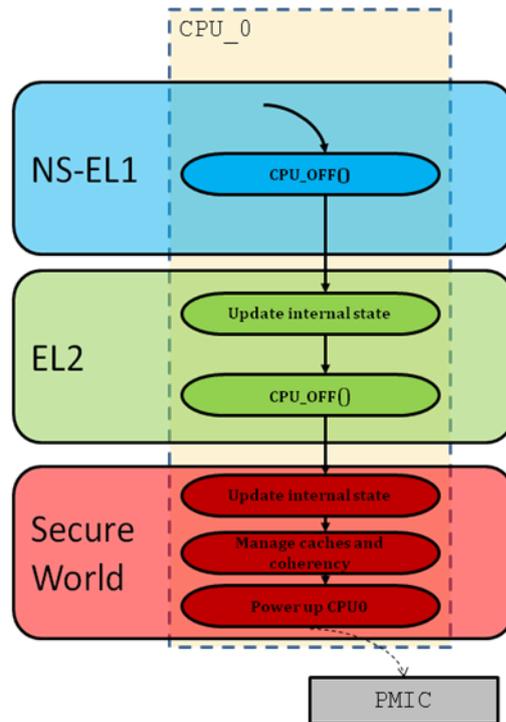
Figure 5 Example call flow for a CPU_OFF call

Figure 5 shows how a `CPU_OFF` call can be propagated through the Exception levels in a system that implements EL2 and EL3. Whether the call is trapped at EL2 or not is defined by the hypervisor implementation.

6.1.3   `CPU_ON` call flow

As is the case with a call to `CPU_SUSPEND`, a call to `CPU_ON` can go through the various Exception levels. Figure 6 below shows an example of a system where a hypervisor traps at EL2.
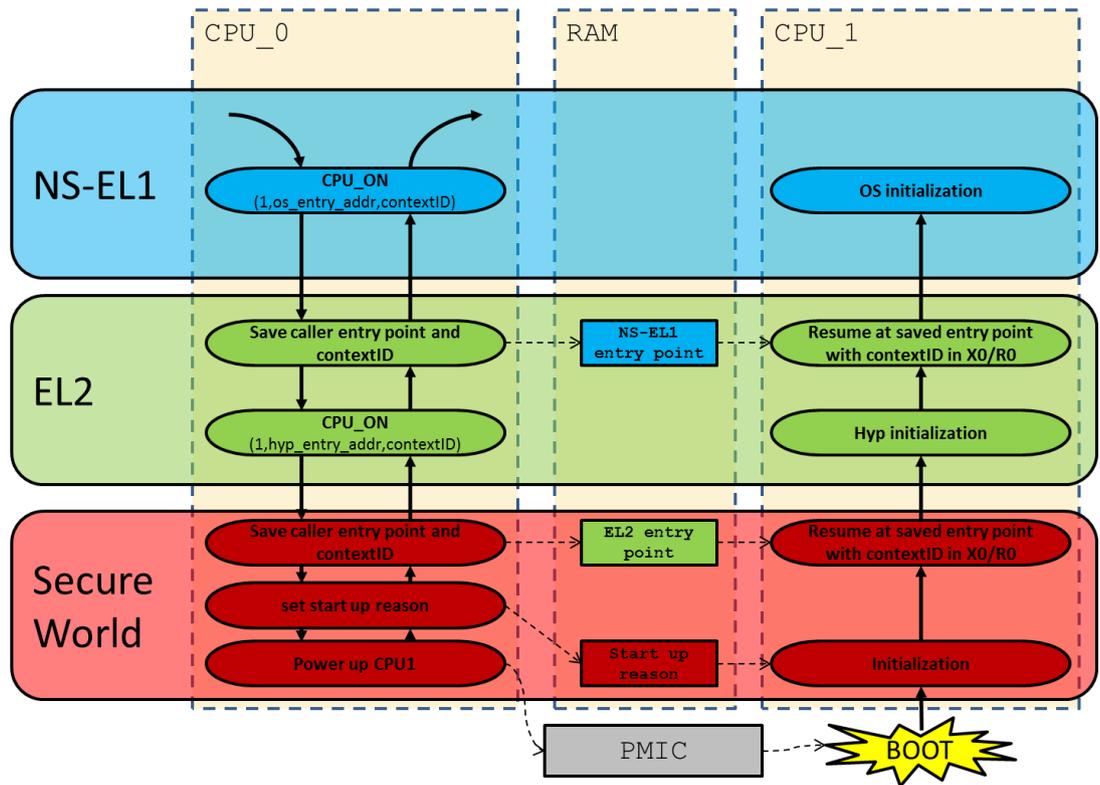
Figure 6 Example call flow for a CPU_ON call

Figure 6 is an example of how the call flow might work when all Exception levels are implemented. In the example, CPU_0 is requesting a power up of CPU_1. The flow follows a similar model to the CPU_SUSPEND flow, described above. CPU_0 requests powering up of CPU_1 and supplies an entry point address and a context identifier for NS-EL1. In the example, the hypervisor at EL2 traps the call, saving the requisite entry point address and context ID, before making the same call to secure platform firmware and supplying an EL2 entry point address and context ID. The Secure platform FW then starts up CPU1, initializes it and resumes it at EL2 using the supplied entry point address and context ID. Finally the hypervisor, having initialized on CPU1, resumes the core at NS-EL1 using the originally supplied entry point address and context identifier.

As described in section Figure 6 the return Exception level for a `CPU_ON` call is the highest Non secure Exception level implemented. This rule allows for the model described above, where the hypervisor traps the call.

In systems that implement EL2 and EL3, this rule also allows bypassing of EL2, that is, it enables a call to go directly from Non-secure EL1 to the Secure world. However the start-up path has to go through EL2. This requires IMPLEMENTATION DEFINED mechanisms for the OS at NS-EL1 to supply a valid EL2 entry point address and context identifier to the secure platform firmware, and for the NS-EL1 return address and context identifier. This is discussed further in section 6.4.

6.2 OS and Implementation View of Cores State

One of the challenges in power state coordination is that the OS and the implementation have individual views of the current state of a node. This can lead to periods where the view of current core state differs between the implementation and the calling OS. This is illustrated in Figure 7.

ARM DEN 0022C
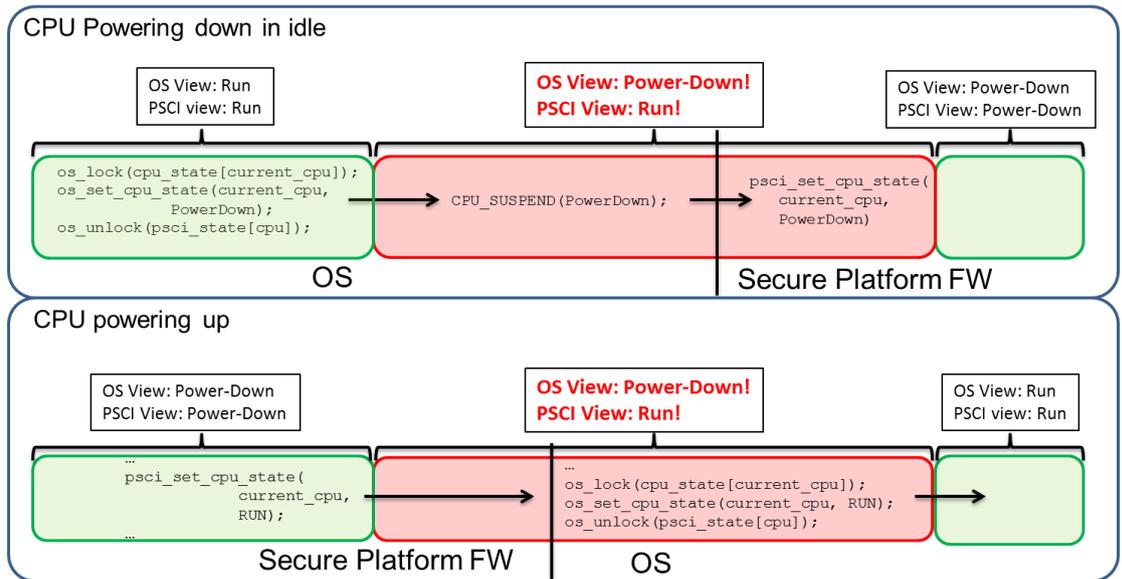
Figure 7 Periods where OS view of Core state differs from implementation view

When an OS powers down the core, it needs to update its internal view of core state before calling into the implementation to actually effect the power down. This gives rise to a period during which the OS sees the core as being powered down, whereas the implementation sees it as running, as it has not yet processed the command. This is illustrated in the upper half of Figure 7. Equally, when a core powers up it is seen by the PSCI implementation before it is seen by the calling OS. This situation results in a window where the implementation sees the core as being in a running state, while the OS considers it to be powered down. This is illustrated in the bottom half of Figure 7.

6.3   Races in OS Initiated Mode

With OS initiated mode the implementation is still responsible for ensuring functional correctness. As described above, there are race windows where the OS view and implementation view of core state differ. Therefore it is possible for the OS to make reasonable requests, from its point of view, which cannot be satisfied by the implementation given its view of core state. For example, the OS might request a power down state for a node from one core, while at the same time the implementation observes that another core in the node is powering up. This section provides a number of example flows to illustrate how these races are solved. In the examples, abbreviations R, Ret, and PD, denote run, retention and power-down respectively. In addition, states where OS view and implementation view differ are marked in red, and states that change during a step in the example are underlined.

With OS Initiated coordination, the ordering of requests from different cores is critically important. If the implementation does not process requests in the order the calling OS intended, then it can put the implementation into the wrong state. Consider the following scenario based on our example system from Figure 3 (dual cluster, dual core per cluster):

Table 13 Example race

Step		OS View			PSCI View		
		Core0	Core1	Cluster0	Core0	Core1	Cluster0
0: Initial state. Core0 is in power-down,	before	PD	R	R	PD	R	R

		Core0	Core1	Cluster0	Core0	Core1	Cluster0
and Core1 is running.	after	PD	R	R	PD	R	R
1: Core1 goes idle. The OSPM requests that Core1 enter power-down and Cluster0 enter Retention state.	before	PD	R	R	PD	R	R
	after	PD	PD	Ret	PD	R	R
2: Core0 receives an interrupt and wakes up into the PSCI implementation in the secure platform firmware.	before	PD	PD	Ret	PD	R	R
	after	PD	PD	Ret	R	R	R
3: Core0 moves into OSPM and starts processing interrupts.	before	PD	PD	Ret	R	R	R
	after	R	PD	R	R	R	R
4: Core0 goes idle and OSPM requests Core0 power-down, Cluster0 power-down.	before	R	PD	R	R	R	R
	after	PD	PD	PD	R	R	R
5: Core0's idle request overtakes Core1's request. The implementation puts Core0 into power-down but ignores the cluster request since it sees Core1 is still running.	before	PD	PD	PD	R	R	R
	after	PD	PD	PD	PD	R	R
6: The implementation observes Core1's request and puts Core1 to power-down and Cluster0 to retention.	before	PD	PD	PD	PD	R	R
	after	PD	PD	PD	PD	PD	Ret

The race described above arises because the order in which the OSPM issues requests is not the order observed by the implementation. In rare cases, this might happen for a variety of potential reasons, for example, different frequencies on different cores, or different cache behavior. This sequence of events results in the implementation incorrectly acting on the stale Cluster0 request from Core1 rather than the latest request from Core0. The net result is that Cluster0 is left in the wrong state until the next wakeup.

To address such race conditions and ensure that the implementation and the calling OS have a consistent view of the request ordering, OS Initiated idle state requests include a dependency check. When the implementation receives a request, it must check whether the requesting core is really the last core to go idle in the requested domain, and reject the request if not. When performing this check, the implementation can start treating a particular core as being in a low-power state once it has observed the core's request. This allows it to be correctly ordered relative to other calling OS requests. The implementation must start treating a core as running before returning control to the calling OS upon wakeup from the idle state.

With this dependency check, the above example changes as follows:

Step		OS View			PSCI View		
		Core0	Core1	Cluster0	Core0	Core1	Cluster0
Resuming from Step 4 above	before	R	PD	R	R	R	R
	after	PD	PD	PD	R	R	R
5: Core0's idle request overtakes	before	PD	PD	PD	R	R	R

Step		OS View Core0	Core1	Cluster0	PSCI View Core0	Core1	Cluster0
Core1's request. **The implementation rejects Core0's request since it includes Cluster0 and Core1 is still awake.**	after	PD	PD	PD	R	R	R
6: The implementation observes Core1's request and rejects it **since it includes Cluster0 but Core0 is still awake.**	before	PD	PD	PD	R	R	R
	after	PD	PD	PD	R	R	R
7: The OS resumes on Core 0.	before	PD	PD	PD	R	R	R
	after	R	PD	R	R	R	R
8: The OS resumes on Core 1.	before	R	PD	R	R	R	R
	after	R	R	R	R	R	R

Once control is returned to the calling OS, it can handle the situation as it sees fit, probably by re-evaluating the idle state on both cores. When requests are received out of order, some overhead is introduced by rejecting the command and forcing the OS to re-evaluate, but this is expected to be rare. Requests sent by the calling OS are seen by the implementation in the same order the vast majority of the time, and in this case, the idle command proceeds as requested.

It is possible that the calling OS might choose to keep a node running, even if all CPUs underneath it are idle. This gives rise to another potential corner case as demonstrated below:

Step		OS View			PSCI View		
		Core0	Core1	Cluster0	Core0	Core1	Cluster0
0: Initial state Core0 in power-down, and Core1 is running	before	PD	R	R	PD	R	R
	after	PD	R	R	PD	R	R
1: Core1 goes idle – the OSPM requests Core1 power-down and Cluster0 retention	before	PD	R	R	PD	R	R
	after	PD	PD	Ret	PD	R	R
2: Core0 receives an interrupt and wakes up into the implementation	before	PD	PD	Ret	PD	R	R
	after	PD	PD	Ret	R	R	R
3: Core0 moves into OSPM and starts processing interrupts	before	PD	PD	Ret	R	R	R
	after	R	PD	R	R	R	R
4: Core0 goes idle and OSPM requests Core0 power-down, Cluster0 to stay Running	before	R	PD	R	R	R	R
	after	PD	PD	R	R	R	R
5: Core0's idle request overtakes	before	PD	PD	R	R	R	R

		OS View			PSCI View		
		Core0	Core1	Cluster0	Core0	Core1	Cluster0
Core1's request. The implementation puts Core0 to power-down. **Even though the OS made a request for the cluster to run, the implementation does not know to reject Core0's request, since it doesn't include a Cluster state**	after	PD	PD	R	PD	R	R
6: Core1's request is observed by the implementation. The implementation puts Core1 to Power Down and Cluster0 to retention	before	PD	PD	R	PD	R	R
	after	PD	PD	R	PD	PD	Ret

The problem here is that the implementation cannot deduce which level of hierarchy a power request applies to simply from the requested power states. In step 5 above, even though Core 0 wanted to keep the cluster running, because the request does not include an actual Cluster power state, the implementation cannot infer that the power request applies to the cluster. To mitigate against this, each idle state request must include a power level parameter specifying the highest power level for which the OS sees the calling core as the last running core (last man). Even if the OS does not want some higher level node to enter an idle state, it should indicate if the core is the last core to power down for that node. This allows the implementation to understand the OS's view of the state of the power topology and ensure ordering of requests.

This enhancement is illustrated in this example:

Step		OS View			PSCI View		
		Core0	Core1	Cluster0	Core0	Core1	Cluster0
0: Initial state Core0 in power-down, and Core1 is running	before	PD	R	R	PD	R	R
	after	PD	R	R	PD	R	R
1: Core1 goes idle – the OSPM requests Core1 power-down and Cluster0 retention and **identifies itself as last down in Cluster0**	before	PD	R	R	PD	R	R
	after	PD	PD	Ret	PD	R	R
2: Core0 receives an interrupt and wakes up into the implementation	before	PD	PD	Ret	PD	R	R
	after	PD	PD	Ret	R	R	R
3: Core0 moves into OSPM and starts processing interrupt	before	PD	PD	Ret	R	R	R
	after	R	PD	R	R	R	R
4: Core0 goes idle and OSPM	before	R	PD	R	R	R	R

---

requests Core0 power-down, Cluster0 to stay Running, and **identifies itself as last down in Cluster0**	after	PD	PD	R	R	R	R
5: Core0's idle request overtakes Core1's request. The implementation rejects it, as it is a cluster request and Core 1 is still running	before	PD	PD	R	R	R	R
	after	PD	PD	R	R	R	R
6: Core1's request is observed by the implementation. The implementation rejects it, as it is a cluster request and Core 0 is still running.	before	PD	PD	R	R	R	R
	after	PD	PD	R	R	R	R
7: OS resumes on Core 0	before	PD	PD	R	R	R	R
	after	R	PD	R	R	R	R
8: OS resumes on Core 1	before	R	PD	R	R	R	R
	after	R	R	R	R	R	R

As before, once control is returned to the calling OS, it can handle it as it sees fit, typically by re-requesting the idle state on both cores.

6.4 Initial state after CPU_ON, CPU_SUSPEND

The following sections describe the expected state of a core when it starts up, following a call to CPU_ON, or on wakeup from a power-down state, following a CPU_SUSPEND call. This section also applies to calls that share semantics with CPU_SUSPEND, that is, CPU_DEFAULT_SUSPEND, and SYSTEM_SUSPEND.

6.4.1 First Non-secure exception level

When returning from a CPU_SUSPEND call after a power-down, or following a CPU_ON call, the first Non-secure Exception level that the Secure world must move the core to is the highest implemented Non-secure Exception level. This is described here as EL_{NS}.

In a system that implements EL3 and EL2, it is not possible to use any services of a hypervisor until the Secure world has enabled use of the HVC instruction. This is controlled by the Secure configuration register HCE field, SCR.HCE or SCR_EL3.HCE. If the hypervisor has been enabled in this way, EL_{NS} is EL2, otherwise it is EL1. For the entry to EL_{NS}:

- If EL_{NS} is using AArch32, the initialization code must set the processor mode to be entered.

- If EL_{NS} is using AArch64, the initialization code must set the stack pointer to be used. With PSCI, the stack pointer must be the one associated with $EL_{NS,}$ that is, SP_EL_{NS}, denoted as EL2h or EL1h.

The following table shows this requirement:

Table 14 First Non-secure Exception level and state on entry

	Caller using AArch32	Caller using AArch64

	Caller	Callee	EL_{NS}	Processor mode	EL_{NS}	Stack pointer
Hypervisor enabled	EL1	SPF	EL2	Hyp	EL2	EL2h
	EL2	SPF	EL2	Hyp	EL2	EL2h
Hypervisor not enabled	EL1	SPF	EL1	Svc	EL1	EL1h

When control passes from EL2 to EL1 the startup state must be:

- Supervisor mode, if EL1 is using AArch32.

- The EL1h stack pointer state, if EL1 is using AArch64.

6.4.2 Entry point address

The entry point address parameter passed in calls to CPU_ON or CPU_SUSPEND must be a valid physical address at EL_{NS}. For systems using virtualization, this might mean that PSCI calls from Non-secure EL1 have to be trapped to EL2. This applies when the physical OSPM as defined in section 3.6 resides at EL1. There are two possible schemes:

1. Calls from EL1 bypass EL2 altogether, and are routed directly to the secure platform firmware. In this case, the OS at EL1 that is making the PSCI calls must either:

- Ensure that stage 2 address translation is disabled.

  Note: Stage 2 address translation is controlled from EL2, by HCR.VM or HCR_EL2.VM. The mechanism by which an OS at EL1 can determine whether Stage 2 translation is disabled is IMPLEMENTATION DEFINED.

- Use an IMPLEMENTATION DEFINED mechanism to obtain a valid physical address for the entry point. In this case, the hypervisor at EL2 and the OS at EL1 must have an IMPLEMENTATION DEFINED mechanism to determine the address at which EL1 resumes execution.

2. EL1 calls are trapped by EL2, either by setting the HCR.TSC or HCR_EL2.TSC bit to 1 or by enforcing use of an HVC conduit. EL2, knowing that the caller contains the physical OSPM, forwards the PSCI call to the secure platform firmware using an appropriate entry point address.

6.4.3 Initial core configuration

For an Exception level to start execution, higher Exception levels must perform appropriate initialization. From ARMv8, platform reset values are defined to provide the minimal functional set for the highest implemented Exception level to start execution. This Exception level then has to set up enough state for a lower Exception level to execute.

This section describes the items that must be correctly initialized when a core starts up in response to a power-down state wakeup for CPU_SUSPEND, or through a call to CPU_ON.

Execution state on ARMv8 systems

The ARMv8 architecture defines two execution states [5], AArch32 and AArch64. On an ARMv8 system, the Execution state entered when starting up a lower Exception level must match the Execution state in use by that Exception level when the PSCI call is made. **Table 15** shows this requirement.

Table 15 Required Execution state on Exception level startup

Implemented Exception levels	Caller	Callee	Transition	Required Execution state
EL1, EL2, and EL3	EL1	EL3	EL3->EL2	EL2 Execution state at time of call
			EL2->EL1	EL1 Execution state at time of call
	EL1	EL2	EL2->EL1	EL1 Execution state at time of call
EL1 and EL2	EL1	EL2	EL2->EL1	EL1 Execution state at time of call
EL1 and EL3	EL1	EL3	EL3->EL1	EL1 Execution state at time of call

Endianness

The caller endianness must be restored when starting up the core at the Exception level from which the call was made. This means, for the Execution level where execution is restarting:

• If starting in AArch32 state CPSR.E must be set to the appropriate value.

• If starting in AArch64 state SCTLR_ELx.EE must be set to the appropriate value. ELx is the Exception level being returned to.

Note that only the highest implemented Exception level has a defined reset value for SCTLR_ELx.EE.

Exceptions

The appropriate asynchronous exception masks must be set when starting up the core at the Exception level from which the call was made, or at EL$_{NS}$. Typically, this means that for the Exception level where execution is restarting:

• If starting in AArch32 state, the CPSR.{A,I,F} bits should be set to {1,1,1}.

> **Note**: On an ARMv7 implementation that includes EL3 (the Security Extensions) but does not include EL2 (the Virtualization Extensions), setting mask bits to 1 in the CPSR masks the interrupts for all Security states and processor modes, regardless of the values of SCR.{FIQ, IRQ, EA, FW AW}. For these implementations, if secure platform firmware, or the Trusted OS, requires FIQs and asynchronous aborts to be routed to the Secure world, the secure platform firmware must start Non-secure EL1 with CPSR.{A, F} set to {0, 0}.

• If starting in AArch64 state, the SPSR_ELx.{D,A,I,F} bits must be set to {1, 1, 1, 1}. ELx is the Exception level being returned to.

MMU, Cache and branch predictor enables

When starting up the core at the Exception level from which the call was made, execution must start with the stage 1 MMU and caches disabled. Also, if starting up in an Exception level that is using AArch32, Branch predictors must be disabled. This means:

• If starting in AArch32 state, the SCTLR.{I, C, M} bits must be set to {0, 0, 0}. In addition, the Z bit must be set in a way that is consistent with the reset behavior of the platform.

• If starting in AArch64 state, the SCTLR_ELx.{ I, C, M} bits must be set to {0, 0, 0}. ELx is the Exception level being returned to.

On start-up, from the point of view of the caller the caches must be clean and coherent with main memory.

T32 support

If the caller is using AArch32 state, and the entry point address specified in a call to `CPU_ON` or `CPU_SUSPEND` has bit[0] set to 1, the core must be started in T32 state at the return exception level. In this case the entry point address specified by the caller is equivalent to the address passed in the call, but with bit[0] set to 0.

Generic Timer

`CNTFREQ` holds the frequency of the Generic timer. This register must be initialized by the Secure world, to enable use of the Generic timer.

Context ID

When the core first enters the return Exception level, the context ID value supplied in the `CPU_SUSPEND` or `CPU_ON` call must be present in:

- X0 if the caller was using AArch64.

- R0 if the caller was using AArch32.

6.5 Recommended StateID Encoding

The contents of the `CPU_SUSPEND power_state` parameter StateID field are IMPLEMENTATION DEFINED. However, it must be set so that each composite power state possible has a unique value. The `power_state` parameter reflects a composite as opposed to a local power state (discussed in section 4.2.1). The StateID field of the parameter can be used to describe the specific combination of local power states being requested. A simple way of doing this is to reserve bit fields for the various power levels in the system. This allows a simple additive mechanism to be used when constructing a composite state from local state indices.

For OS initiated mode, it is also necessary to pass an index for the power level where the calling core is last to go idle. The format of this field matches the `power_level` parameter of the `NODE_HW_STATE` function (see section 5.15.1). If a bit field is reserved, the OS can use a simple additive scheme to place the information into a StateID field when it is making a request. Consider the example in Figure 3. Given the number of local states at each power level, a simple encoding could use one nibble of the StateID field as a local state index for each power level, and another nibble for the OS Initiated last level index. This would give rise to the following encoding:

Table 16 StateID sample encoding

Bit field	Description
15:12	Core is last in Power Level: - 0: Core Level - 1: Cluster Level - 2: System Level
11:8	System Level Local Power State - 0: Run - 2: Retention

	• 3: Power-down
7:4	Cluster Level Local Power State • 0: Run • 2: Retention • 3: Power-down
3:0	Core Level Local Power State • 0: Run • 1: Standby • 2: Retention • 3: Power-down

The above method uses 1-based indices for local states in a power level. Thus, at core level, the states standby, retention, and power-down are indexed 1, 2, 3 respectively. For consistency, cluster and system states are numbered 2 and 3 for retention and power-down respectively. The value of 0 is reserved to indicate a running state. Using a simple additive composition method, and taking into account only the local power states, this encoding results in the possible composite power state parameters identified in Table 17:

Table 17 StateID encodings for local and composite states in example system

Last in Level Bits [15:12]	System Local States Bits [11:8]	Cluster Local States Bits[7:4]	Core Local States Bits[3:0]	Composite States (Sum of Local States)
0x#000	Run 0x000	Run 0x000	Standby 0x001	Run|Run|Standby 0x#000+0x000+0x000+x001=0x#001
0x#000	Run 0x000	Run 0x000	Retention 0x002	Run|Run|Retention 0x#000+0x000+0x000+x002=0x#002
0x#000	Run 0x000	Run 0x000	Power-down 0x003	Run|Run|Power-down 0x#000+0x000+0x000+x003=0x#003
0x#000	Run 0x000	Retention 0x020	Retention 0x002	Run|Retention|Retention 0x#000+0x000+0x020+x002=0x#022
0x#000	Run 0x000	Retention 0x020	Power-down 0x003	Run|Retention|Power-down 0x#000+0x000+0x020+x003=0x#023

0x#000	Run 0x000	Power-down 0x030	Power-down 0x003	Run| Power-down|Power-down 0x#000+0x000+0x030+x003=0x#033
0x#000	Retention 0x200	Retention 0x020	Retention 0x002	Retention |Retention| Retention 0x#000+0x200+0x020+x002=0x#222
0x#000	Retention 0x200	Retention 0x020	Power-down 0x003	Retention |Retention| Power-down 0x#000+0x200+0x020+x003=0x#223
0x#000	Retention 0x200	Power-down 0x030	Power-down 0x003	Retention |Power-down| Power-down 0x#000+0x200+0x030+x003=0x#233
0x#000	Power-down 0x300	Power-down 0x030	Power-down 0x003	Power-down | Power-down| Power-down 0x#000+0x300+0x030+x003=0x#333

The last in level information is encoded in the fourth nibble. In a system like this, firmware tables describe the power levels as 0x0000, 0x1000, and 0x2000 for core, cluster, and system respectively. These values are used when adding the last in level value of the StateID portion of a power_state parameter, for a CPU_SUSPEND call in OS Initiated mode. These values are also used for the power_level parameter of the NODE_HW_STATE function (see section 5.15.1).

This method provides the required unique composite state power_state parameters. This scheme also provides the ability to uniquely refer to local states as well as composite states, which is useful, though not required, for the PSCI_STAT_COUNT and RESIDENCY functions. Furthermore, the scheme is additive, so composite state IDs can be obtained by simply adding the local states for each power level, and in the OS initiated case, also adding the last in level index.

Note that more compact additive schemes are also possible, by using the same approach by which multidimensional arrays are flattened to single dimensional arrays.

6.6 Implementation CPU_ON/CPU_OFF races

Implementation of CPU_ON and CPU_OFF requires special care to avoid races. Both the calling supervisory software and the PSCI implementation must use appropriate locks to maintain the correct state representation of a core.

The AFFINITY_INFO function defines a set of valid states for a core, see section 5.7.1. As seen from the PSCI implementation, a core can be in any of the following states:

ON: This core has at some point been enabled with a call to CPU_ON, or is the cold boot primary core, and has not called CPU_OFF. In practice, the core might be running or in a low power mode, either WFI or deeper because of a call to CPU_SUSPEND. The term ON reflects the fact that the core is considered to be available for computation.

OFF: This core has called CPU_OFF, and the call has been processed by the PSCI implementation, or the core has not been booted yet. The core is not available for computation.

ON_PENDING: `CPU_ON` has been called on the target core, but it is still in the process of booting and has not transitioned to an `ON` state.

The state diagram shown below in Figure 8, illustrates the valid transitions that an implementation must provide. Circles represent the states, and a state transition is represented as an arrow between two circles. The arrow label denotes the state transition such that:

1. The transition takes place while the original state is TRUE.

2. A transition can include a *post transition event*. This is indicated by the text that follows a forward slash character "/". This event happens when the destination state is TRUE.

3. After the transition the destination state is TRUE.
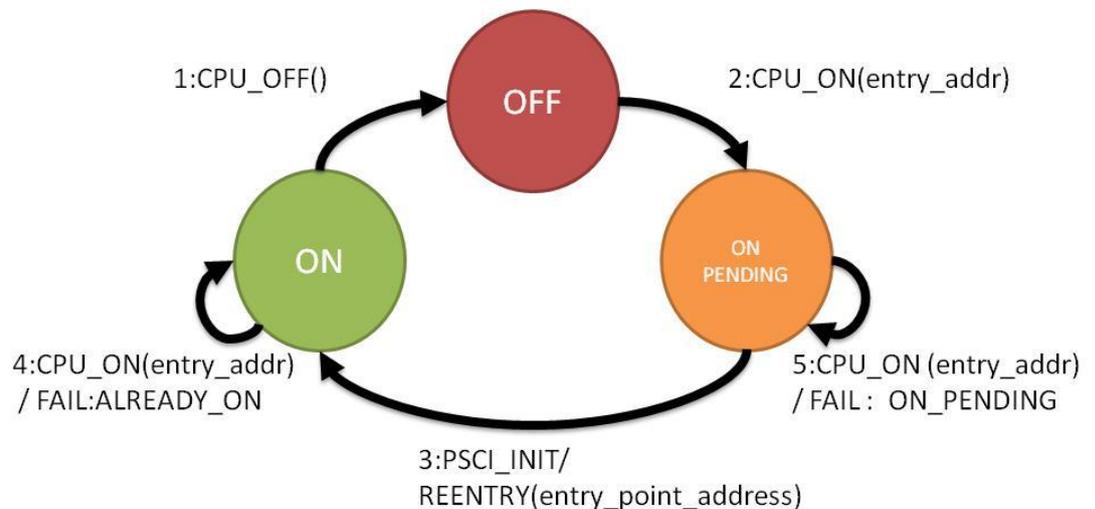
Figure 8 CPU_ON/CPU_OFF PSCI implementation state machine

Figure 8 shows the following transitions:

CPU_OFF (1): The implementation receives a `CPU_OFF` call from the client running on the core in question.

CPU_ON (2): The implementation receives a `CPU_ON` call for the core in question.

PSCI_INIT (3): The core in question initializes and starts executing PSCI implementation firmware as part of the boot sequence. After this transition, when the PSCI implementation sets the core state to `ON`, the core re-enters the return Exception level at the specified entry point address.

The diagram also describes failures to transition when invalid calls are observed. In particular:

CPU_ON (4): Fails with an `ALREADY_ON` error because the PSCI implementation sees the core as being `ON`.

CPU_ON (5): Fails with an `ON_PENDING` error because a valid `CPU_ON` call has already been made to the target core, but the core has not initialized yet, meaning it has not made transition 3 (`PSCI_INIT`).

An actual implementation can distinguish between further internal states. However, externally it must guarantee the states and transitions shown in Figure 8.

A caller of PSCI might maintain a similar set of states, and therefore a caller might consider any cores to be `ON`, `OFF`, or `ON_PENDING`. The following pseudocode shows an example implementation from the perspective of the caller to `CPU_ON` and `CPU_OFF`.

```
void os_cpu_off() {
      lock(psci_state[current_cpu]);
      assert(psci_state[cpu] == ON);
      psci_state[cpu] = OFF;
      unlock(psci_state[cpu]);
      /* race window starts here*/
      CPU_OFF();

      assert(0); /* CPU_OFF() failed */
      }


int os_cpu_on(cpu,contexID) {

      int r = 0;
      lock(psci_state[cpu]);
      if (psci_state[cpu]==ON) r = E ERROR_ALREADY_ON;
      if (psci_state[cpu]==ON_PENDING) r = ERROR_ON_PENDING;

      if (r) {
          unlock(psci_state[cpu]);
          return r;
      }

      do {
          r = CPU_ON(cpu,
                  physical_addr(os_entry_point),
                  contextID);
          } while (r == ERROR_ALREADY_ON);

      assert(!r);
      psci_state[cpu]=ON_PENDING;
      unlock(psci_state[cpu]);

      return r;
      }
```

This example implementation maintains a lock for each core, and uses the states described in this section, ON, OFF, ON_PENDING, but these states represent the view of the caller of the core status.

There is race opportunity, while a caller is transitioning a core to OFF (from the point of view of the caller), but before the PSCI implementation has set the state to OFF (from the point of view of the PSCI implementation). Red text signals the start of the race window. During this time, the caller might consider the core to be OFF, while the PSCI implementation sees the core as being ON. The race closes when the PSCI implementation performs the atomic update of the state of the core to OFF.

The race is solved by the use of the ALREADY_ON error. When a CPU_ON call on the same core arrives during this window the PSCI returns an ALREADY_ON error as the implementation sees that core as still being in the ON state. A caller can use this error code to identify this situation, and retry the call. The time spent retrying will depend on the following factors:

1. The time between a caller setting the core state to OFF and calling CPU_OFF.

2. The time it takes the PSCI implementation to transition its representation of the core state to OFF.

Both time windows should be as short as possible. Finally, when a core enters the calling exception level it can transition the state to ON:

```
void os_entry_point(){
      ...
      lock(psci_state[cpu]);
      psci_state[cpu]=ON;
      unlock(psci_state[cpu]);
      ...
      }
```

6.7   Discoverability

The functions defined here are independent of the conduit used to invoke them. The PSCI specification expects that, at boot time, a hypervisor and Rich OS can discover which conduit applies. The ARM architecture provides feature registers in the ARMv7 and ARMv8 System registers, that can be used to determine at runtime whether EL2 and EL3 are implemented. This, in theory, could be used to determine the conduit, SMC or HVC. However, this does not determine whether firmware support for PSCI is present. Therefore, an implementation must be accompanied by firmware tables, for example, FDT or ACPI, that describe whether the interface is present or not.

In addition, the tables must allow the caller to determine the conduit to use for PSCI (SMC, or HVC).

Presence of PSCI and its functions is described in the documentation on ACPI [6] and Flattened Device Tree [7].

ARM recommends that the firmware provides descriptions of the topology of the system that supervisory software can access to determine:

- The power domain topology.

- Composite and local states. The descriptions should contain information on the available power state parameters that can be used in CPU_SUSPEND calls.

6.8   Save and Restore

Use of power-down states in idle management, where context is lost, and use of big.LITTLE migration, both require the ability to resume execution on a previously powered down core. The return to execution must be transparent to the applications that were executing on the core before its shutdown. To achieve this, operating systems must save the context of execution for the core. When a core resumes, its context must be restored so that execution can restart from the same point with the same context.

To implement this, every OS must provide a software stack that can:

- Save the architectural context that is otherwise lost when power is removed.

- Save the non-architectural state that is otherwise lost when power is removed. There might be SoC implementation-specific components that have a state that needs to be saved.

- In a big.LITTLE migration, interrupts must be redirected from each outbound core to the corresponding inbound core.

- Provide a return path for execution that can perform basic initialization and the restoration of non-architectural and architectural state.

Figure 9 below, depicts the components involved in such a stack across all levels of supervisory software that might be executing on an ARM system. The PSCI specification focuses primarily on the interface between the Normal world and Secure world, shown by

the red arrow in Figure 9. PSCI also covers the interface between rich operating systems and hypervisors, shown by the orange arrow. This specification does not document the interface between the secure platform firmware and a Trusted OS.
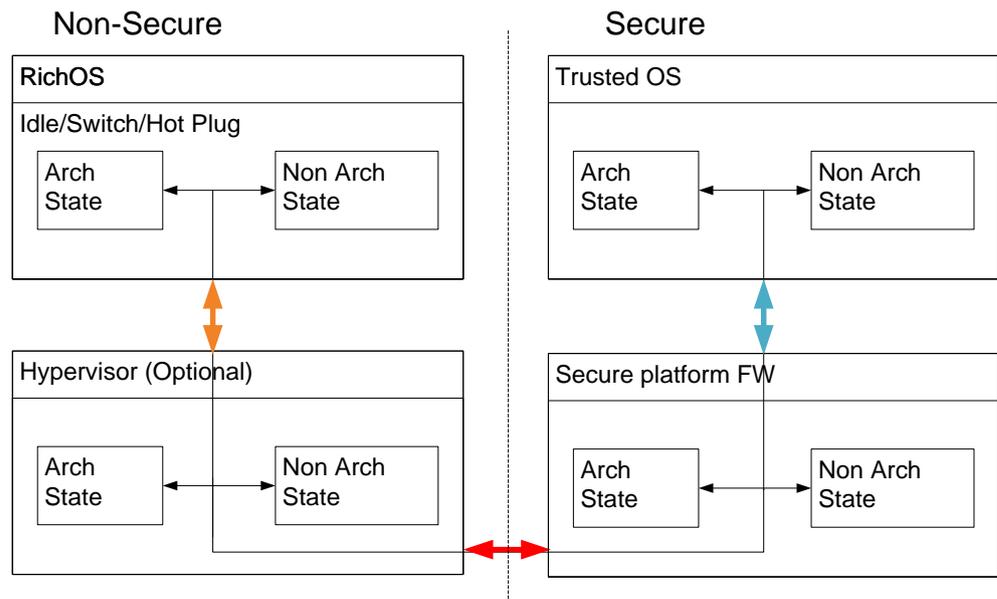

Figure 9 Save and restoring context across operating systems

Every OS must save and manage its architectural state and the GIC state. The former includes:

- The general-purpose register bank.

- The NEON and floating-point register bank.

- System control registers, for example, MMU and Generic Timer System control registers at the level appropriate to the OS.

- Debug context.

- Trace context.

In big.LITTLE migration models, for a GICv2 system, management of interrupts involves:

1) Saving the CPU interface registers (binary point, priority mask, and control register) for any outbound core.

2) Retargeting *shared peripheral interrupts* (SPIs) to the CPU interfaces of any inbound core, and away from the corresponding outbound core.

3) The OSPM must update its internal representations to ensure any future *Software Generated Interrupts* (SGIs) are sent to any inbound core instead of the corresponding outbound core. Any pending SGIs must be cleared on the outbound core and pended on the inbound core.

4) The OSPM must prevent any new private peripheral interrupts (PPIs) from arising on any inbound core before initiating the migration. Any pending PPIs must be cleared on any outbound core and pended on the corresponding inbound core. After the switch, PPI sources must be re-enabled.

5) Transfer state of distributor banked registers (group, set-enable or clear-enable for PPIs, priority, Non-secure access, and configuration if implementation requires it) from any outbound core to the corresponding inbound core.

In core hotplug OFF it is also necessary to manage the following GIC context:

1) The OSPM must remove the core from its representation of available cores.

2) This means the OSPM must ensure that no SGIs and no PPIs are pending on a core that is going to be hotplugged off.

3) The OSPM must retarget SPIs away from a core before hotplugging it off.

In core hotplug ON:

1) The OSPM can instruct the kernel to add the core to its representation of available cores.

2) If required, the OSPM can retarget SPIs onto the newly-available core.

In a typical system, before powering down a core as part of idle management or hot unplug, the CPU interface on that core must be placed in a quiescent state. This ensures that the final WFI instruction, that signals to a power controller that the core can be powered down, will not complete even if an interrupt is pending.

When powering down a core for idle management it might be necessary to save the CPU interface state and to save GIC distributor banked (per core) registers. This includes group, set-enable and clear-enable for PPIs, priority, Non-secure access, and configuration if the implementation requires it. In addition, if entering a system power-down state where the GIC context will be lost, the last core to power down must save shared GIC registers, meaning it must save the control, enable, target, configuration, group, and Non-secure access registers.

6.8.1 Debug and Trace save and restore

The ARM architecture provides support for external debug and for self-hosted debug. External debuggers provide hardware assisted debug that can modify core and memory data. The external debugger can also modify debug context to manage debug events such as breakpoints and watchpoints. Finally, an external debugger can also affect the power state of a core by acting as a source of wakeup events, or by preventing power downs. For more information see [1], [5].

Saving and restoring invasive debug context

Support for debug on multicore systems is complicated by the fact that system software can move the software threads being debugged from one core to another. Therefore, supporting debug requires a strategy that either prevents this migration or that appropriately duplicates debug context on every core. Duplicate programming provides a more realistic debug method from a scheduling point of view, as there is no need for pinning.

Regardless of the method chosen, when using an external debugger, the debugger might interact with a core that is powered down. Any attempts by the debugger to access a debug register when the core power domain is powered down, or in a low-power state where the core power domain registers cannot be accessed, returns an error. The debugger can retry the access. However, if the core power domain is regularly put into such a state, this might lead to unreliable debugger behavior. In a multicore system utilizing dynamic power control, it might be rare for all the cores to be powered up simultaneously, but this is necessary for duplicating programming across all cores. There are two options an external debugger can use in this case:

1. A debugger can request emulation of power down, through the use of **DBGNOPWRDWN** and **DBGPWRUPREQ,** as documented in [1,5]. This approach prevents use of debug over a power down.

2. If the supervisory software supports debug over power down, the debugger can allow full power cycling of cores. Under this scheme the debugger keeps a valid central copy of the debug context. When a user changes this context, the debugger must change the context of the cores to match. If any core is powered down, the debugger can use the OS Unlock Catch debug event to halt the core after it has woken up and completed its

debug context restore sequence. At this point the debugger can program the debug registers from Debug context. This method means the debugger can avoid using emulation of power down.

The second option requires a method the debugger can use to inform the supervisory software that it must save and restore debug registers during a power down transition. ARM recommends the use of CLAIM tag registers for this purpose. CLAIM tags can also be used to prevent conflicts between external debug and self-hosted debug use of debug registers. The following CLAIM tags bits should be used for controlling save and restore of debug context:

DBGCLAIM[0] should be set to 1 to indicate that debug is in use by the external debugger. If this bit is set to 1, it is expected that:

1. The supervisory software must perform a save or restore sequence of the debug context.

2. Debug context will not be overwritten by self-hosted debug software.

DBGCLAIM[1] should be set to 1 to indicate that debug is in use by a self-hosted debugger. This bit can be used to indicate to power management software that it must save and restore debug context when power cycling.

In an AArch32 implementation that is using v7 Debug, **DBGCLAIM[6]** should be set to 1 by the supervisory software as an acknowledgement for bit [0]. This step is only required for compatibility with v7 Debug.

These claim tags can be used to indicate that debug context requires a save/restore sequence.

Note: For AArch32 implementations that use v7 Debug, ARM recommends that **DBGCLAIM**[6] is reserved for the OS to acknowledge that it will perform save/restore. The external debugger can write to **DBGCLAIM** when the core is powered down, meaning it should either not clear **DBGPRCR**.**CORENPDRQ** or not program any volatile context until the power-down software acknowledges that it will save/restore the context by setting **DBGCLAIM**[6] to 1.

Debug CLAIM tags should be programmed by the external debugger or self-hosted debug agent using the following sequence

1. Check CLAIM tags. If ownership is clear (bit [0:1]), proceed, otherwise fail, or retry later.

2. Claim ownership by setting bit [0], for external debugger, or bit 1 for self-hosted debug agent.

3. Check CLAIM tags again:

    a. If only the bit programmed in step 2 is set, and the other bit is clear, then claim tags have been successfully programmed.

    b. If both bits are set, then clear the bit set in step 2 and fail, or retry later.

Self-hosted debug does not always require a save and restore sequence of the debug context used in invasive debug to support power management transitions. Breakpoint and watchpoint registers are associated with threads, and these can be migrated from one core to another through OS scheduling. Therefore, the OS might already implicitly save and restore breakpoint and watchpoint data with every context switch. However, generic events that are not associated with a specific thread must be saved and restored for power-down states, using the sequences described in [1], [5].

With self-hosted debug, big.LITTLE migration models must save and restore debug context as part of a normal context migration. If the migration always occurs in specific software threads, and there is no need to debug these threads, then only generic events, not associated with threads, and not saved and restored through scheduling, must be saved on the outbound core and restored on the inbound core. If the migration does not occur in

specific software threads, or support for debug of the migration threads is required, then all of the debug context used by the self-hosted debugger, must be saved and restored.

Saving and restoring Performance Monitoring Units

*Performance Monitoring Unit*s (PMUs) also require some management on MP systems. Once again, external and self-hosted debuggers must ensure that they do not conflict with the use of PMUs. In addition, the external debugger might require save and restore of PMU registers during a power transition. ARM recommends the use of CLAIM tag bits to indicate whether PMUs are in use by the external debugger or self-hosted debug as follows:

 DBGCLAIM[2] should be set to 1 to indicate that PMUs are in use by the external debugger. If this bit is set to 1, it is expected that:

1. The supervisory software will perform a save/restore sequence of the PMU context.

2. PMU context will not be overwritten by self-hosted debug software.

DBGCLAIM[3] should be set to 1 to indicate that PMUs are in use by a self-hosted debugger. This bit can be used to indicate to power management software that it must save and restore debug context over a power cycle.

In an AArch32 implementation that is using v7 Debug, **DBGCLAIM[7]** should be set to 1 by the supervisory software as an acknowledgement for bit [2]. For more information see the Note for **DBGCLAIM[1]** .

Supervisory software performing big.LITTLE migration can take different approaches to PMUs. For events that are common to big and LITTLE processors, it is possible to provide a logical view of cores. Under this scheme the user sees events for a single core. Although this core can at any time be instantiated as a big processor or a LITTLE processor, the event counting and event interrupts are treated as coming from the single logical core. This requires the migration stack to transfer PMU context from one core to another as it moves the rest of the context.

Alternatively, the supervisory software could just provide a physical view of cores. This does not require transferring any context, but the programmer needs to be aware that events are counted separately for the LITTLE and big cores. In this case the supervisory software must save the PMU context of each core when it is powered down, and restore it when powered up, without migrating data. A GTS solution, such as big.LITTLE MP, or any other multicore OS solution only needs to support physical views of core. When working with heterogeneous systems it is also worth noting that:

- There are events that are specific to each core type.

- The number of counters can differ between cores.

- Even events that are common to different types of core might require specific interpretation, due to the differences in microarchitecture.

ARM recommends always providing a physical view, as appropriate interpretation of the PMU requires understanding of the microarchitecture of the devices under test.

Saving and restoring trace context

Using a similar handshake to that proposed for debug context, ARM recommends the use of ETMCLAIM registers to indicate whether an external or a self-hosted debugger is using trace, and to indicate the need to save and restore:

ETMCLAIM[0] should be set to 1 to indicate that trace is in use by the external debugger. If this bit is set to 1, it is expected that:

1. The supervisory software will perform a save and restore sequence of the trace context.

2. Trace context will not be overwritten by self-hosted debug software.

ETMCLAIM [1] should be set to indicate that debug is in use by self-hosted debugger. This bit can be used to indicate to power management software that it must save and restore trace context when power cycling.

In an AArch32 implementation that is using ETMv3.4 or PFTv1.0, **ETMCLAIM [6]** should be set to 1 by the supervisory software as an acknowledgement for bit [0].

The save and restore sequence must be appropriate for the type of trace architecture implemented by the core, and this might differ depending on core type. For example, the Cortex®-A15 processor uses Program Flow Trace Architecture [3], whereas the Cortex-A7 processor uses the Embedded Trace Macrocell Architecture [2]. Both architectures support ETMCLAIM tags.

Saving and restoring debug context and Exception levels

Save and restore sequences for debug and trace context can be performed at any privileged Exception level that is making use of the services. It might be that virtualized access is being provided to debug services by higher Exception levels. In this case the higher Exception levels must trap accesses as appropriate.

In all cases the save sequence must use the CLAIM tags to decide whether a save and restore sequence is required, and whether it needs to include the subset of registers affected by self-hosted debug, or the set affected by external debug. For descriptions of the save and restore sequences see the ARMv7 ARM [1], or the ARMv8 ARM [5], as well as the Program Flow Trace Architecture Specification [3], or the Embedded Trace Macrocell Architecture Specification [2].

6.9   Migration for CPU or Cluster Migration Based big.LITTLE Systems

Figure 10 shows the proposed migration for a big.LITTLE system that uses the migration model:
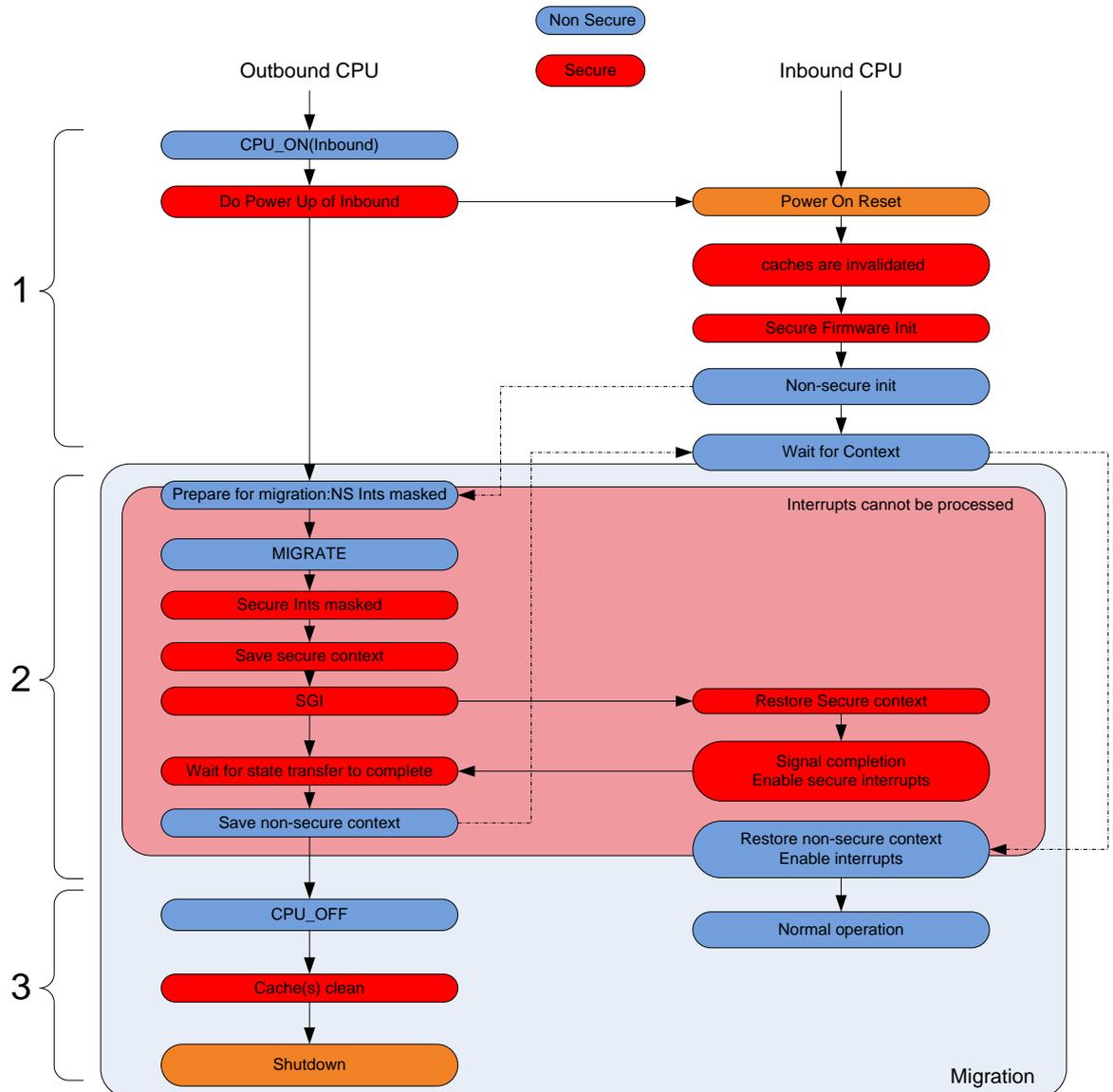
Figure 10 big.LITTLE migration

Before migration, the OSPM must request a powerup for the inbound core. The OSPM could be the Rich OS or a hypervisor, depending on which OS is managing the physical migration. This is shown in the area labelled 1 in Figure 10.

When the inbound core powers up, it performs basic initialization in the Secure world, and then moves to the Normal world. The OSPM must specify an entry point address for code that places the core into a holding pattern, waiting for valid context to restore.

Ideally, the OSPM should also provide a signal so that the outbound core knows when the inbound core is ready. Up until this point there is no need to disable interrupts, and the outbound core can continue to perform useful work.

Once the inbound core is available the migration process can take place. This is shown in the area labelled 2 in Figure 10. The OSPM initiates this by masking Non-secure interrupts and issuing a MIGRATE call.

The MIGRATE call must be synchronous, and transfers Secure context from the outbound core to the inbound core. The mechanism by which this is achieved is IMPLEMENTATION

DEFINED. The example in Figure 10 uses an SGI to alert the inbound core of the presence of valid secure context. This flow is shown in the area labelled 2 in the diagram:

1) The outbound core masks Secure interrupts, saves Secure context, and retargets its interrupts to the inbound core.

2) The outbound core sends an SGI to the inbound core and waits.

3) The inbound core receives the SGI and restores the Secure context, unmasks Secure interrupts and then signals the outbound core.

4) The outbound core returns from the `MIGRATE` call.

If the inbound core does not respond to the SGI in a suitable time frame, or if the inbound core was not powered up, an `INTERNAL_FAILURE` error can be returned. When the inbound side returns from the interrupt caused by the SGI it goes back to its holding pattern, waiting for valid Non-secure context to restore.

A call to `MIGRATE` must be preceded by a call to `CPU_ON`. The calling OS can then use an IMPLEMENTATION DEFINED mechanism to determine that the core is powered up, and ready to migrate before issuing the call.

When migration of Secure context has completed, the OSPM saves Non-secure context, then releases the inbound core from its holding pattern. At this point the inbound core can restore the Non-secure context, re-enable interrupts, and resume normal operation. Interrupts are only disabled while Secure and Non-secure contexts are being transferred.

The area labelled 3 in Figure 10 describes the steps the outbound core carries out once the context has migrated to the inbound core. At this point the outbound core can prepare to shut down. The shutdown does not need to happen immediately. After migration, the outbound core can call `CPU_OFF`.

With supervisory software implementing a big.LITTLE migration solution, ARM recommends that the call is made by the migration stack, or by the Non-secure Trusted OS drivers, through notifications from the migration stack. On other multicore systems ARM recommends that the call is made by the Non-secure Trusted OS driver through notifications from the hot unplug stack. This component is supplied by the Trusted OS vendor.

Cluster migration can be implemented using this process by running it across all cores in the cluster simultaneously.

6.10   Compliance with the PSCI Specification

The functions that must be implemented to be fully compliant with a PSCI at a specified version are listed below:

Table 18 Mandatory and optional functions for a given version of PSCI

FUNCTION	PSCI 0.2	PSCI 1.0
PSCI_VERSION	Mandatory	Mandatory
CPU_SUSPEND	Mandatory	Mandatory
CPU_OFF	Mandatory	Mandatory
CPU_ON	Mandatory	Mandatory
AFFINITY_INFO	Mandatory	Mandatory

MIGRATE	Optional	Optional
MIGRATE_INFO_TYPE	Optional(*)	Optional(*)
MIGRATE_INFO_CPU	Optional(**)	Optional(**)
SYSTEM_OFF	Mandatory	Mandatory
SYSTEM_RESET	Mandatory	Mandatory
PSCI_FEATURES	Not Applicable	Mandatory
CPU_FREEZE	Not Applicable	Optional
CPU_DEFAULT_SUSPEND	Not Applicable	Optional
NODE_HW_STATE	Not Applicable	Optional
SYSTEM_SUSPEND	Not Applicable	Optional
PSCI_SET_SUSPEND_MODE	Not Applicable	Optional
PSCI_STAT_RESIDENCY	Not Applicable	Optional(***)
PSCI_STAT_COUNT	Not Applicable	Optional(***)

(*)Note that for issue B.b of this document `MIGRATE_INFO_TYPE` was compulsory, and `NOT_SUPPORTED` was not a valid return value. This was fixed in issue C of this document, which acts a release of PSCI1.0, and errata fix release for PSCI 0.2. Therefore this change applies to all versions of PSCI from 0.2 onwards.

(**) Mandatory if MIGRATE is implemented.

(***) If present then both PSCI_STAT_RESIDENCY and PSCI_STAT_COUNT must be implemented.

7 History

7.1 Changes between PSCIv1.0 and PSCIv0.2

PSCI1.0 version of the specification has the following functional changes from version PSCI0.2:

- Added `PSCI_FEATURES`.

- Added `CPU_FREEZE`.

- Added `CPU_DEFAULT_SUSPEND`.

- Added `NODE_HW_STATE`.

- Added `SYSTEM_SUSPEND`.

- Added OS Initiated and platform coordinated mode descriptions. This affects returns values for `CPU_SUSPEND` which now adds `DENIED` as a possible return value.

- Added `PSCI_SET_SUSPEND_MODE`.

- Added `PSCI_STAT_RESIDENCY`.

- Added `PSCI_STAT_COUNT`.

- Made `MIGRATE_INFO_TYPE` optional by letting it return `NOT_SUPPORTED`. This indicates that `MIGRATE` is not required. This change applies retroactively to version 0.2 of PSCI as an errata change.

- Relaxed requirement that caches on all CPUs must be cleaned in the `SYSTEM_OFF` function (SWARCH-13).

- Clarified early returns in the `CPU_SUSPEND` case, both the return value and return path (SWARCH-14, SWARCH-5).

- Clarified mandatory functions (SWARCH-4).

- Fixed `MIGRATE target_cpu` parameter type (SWARCH-16).

- Removed `NOT_PRESENT` as a possible return code for `AFFINTY_INFO`. This was an erratum (SWARCH-6).

- States now pertain to power domain levels rather than affinity levels. This ensure that affinity levels (as defined by MPIDR) are independent from the power domain levels actually implemented

- Relaxed `AFFINITY_INFO` so that it does not need to work at affinity levels higher than zero. `NODE_HW_STATE` provides alternative functionality.

- Introduced the `INVALID_ADDRESS` return code on functions that take address parameters (`CPU_SUSPEND`, `CPU_ON`, `CPU_DEFAULT_SUSPEND`, `SYTEM_SUSPEND`).

- Closer alignment between PSCI and the SMC Calling Conventions [4], by relaxing the need to return `INVALID_PARAMETERS` when there is a mismatch between the execution state of the caller (AArch32/64) and the SMC function identifier (SMC32/SMC64) From PSCI1.0, every AArch32 call to an SMC64 call will return `NOT_SUPPORTED`, and an AArch64 OS is free to call an SMC32 call (SWARCH-1). This affects: `CPU_SUSPEND`, `CPU_ON`, `AFFINITY_INFO`, `MIGRATE`, and `MIGRATE_INFO_CPU`.

7.2   Changes in PSCIv0.2 from first proposal

This version of the specification has the following functional changes from the first draft proposal version of PSCI:

- Provided values for Function ID parameter of all functions.

- Added context ID parameters to `CPU_ON` and `CPU_SUSPEND`.

- Removed `DENIED` return value of `CPU_SUSPEND`.

- Removed `power_state` parameter for `CPU_OFF`.

- Removed `INVALID_PARAMETERS` for `CPU_OFF`.

- Specified the format for the `target_cpu` parameter of a `CPU_ON` call to be based on MPIDR.

- Removed `DENIED` return value of `CPU_ON`.

- Added `ALREADY_ON` return value of `CPU_ON`.

- Added `ON_PENDING` return value of `CPU_ON`.

- Added `INTERNAL_FAILURE` return value of `CPU_ON` and `MIGRATE`.

- Added `NOT_PRESENT` return value to `MIGRATE`.

- Renamed -3 error code from "Core Not Available" to `DENIED` for `MIGRATE`.

- Added the following functions:

    - `PSCI_VERSION`.

    - `AFFINITY_INFO`.

    - `MIGRATE_INFO_TYPE`.

    - `MIGRATE_INFO_UP_CPU`.

    - `SYSTEM_OFF`.

    - `SYSTEM_RESET`.

- Defined a version number for PSCI, at the time of writing, 0.2.

- Made all functions apart from `MIGRATE` and `MIGRATE_INFO_UP_CPU` compulsory, and consequently removed `NOT_SUPPORTED` return code from `CPU_ON`, `CPU_SUSPEND`, and `CPU_OFF`.

    - Note that the errata fix in release C makes `MIGRATE_INFO_UP_CPU` optional by allowing `NOT_SUPPORTED` as a return code.

8 Glossary

The following table describes some of the terms used in this document.

Table 19 Glossary terms

Term	Description
ACPI	The Advanced Configuration and Power Interface specification. This defines a standard for device configuration and power management by an OS.
FDT	Flattened Device Tree. This is a hardware description standard. Firmware tables are constructed that describe the hardware. These tables are passed to the OS at boot time. An OS can interrogate the data they contain when it needs to discover the hardware properties of a device.
HVC	The Hypervisor Call instruction or the associated exception. This requests a hypervisor function, causing the core to enter EL2.
MP (Multi-Processor)	A software stack, typically an OS that supports simultaneous operation across multiple cores.
Normal World	The execution environment when the core is in the Non-secure state.
OSPM	Operating System-directed Power Management. Typically, this acronym refers to the components of an OS that provide power management for the platform. In this document OSPM refers to software components that are involved in the selection and application of power states for individual cores or clusters, or overall system states.
Rich OS	Application OS such as Linux or Windows.
Secure Platform Firmware (SPF)	Owned by the silicon vendor and OEM. This firmware layer is the first thing that runs at boot on an application processor. It provides a number of services, including platform initialization, the installation of the Trusted OS, and routing of Secure Monitor Calls. Some calls are destined for the SPF and some are destined for the Trusted OS. SPF can run in EL3 and secure EL1 on ARMv8 systems using AArch64 in EL3. For ARMv7 systems, or ARMv8 systems using AArch32 at EL3, SPF executes in EL3. The SPF must include the implementation that acts on power management requests made via the Power State Coordination Interface.
Secure World	The execution environment when the core is in the Secure state. When the core is executing in EL3 it is in Secure state.
SMC	The Secure Monitor Call instruction or the associated exception. This requests a Secure Monitor function, causing the core to enter EL3.
Trusted OS	This is the OS running in the Secure world. It provides a Trusted Execution Environment.
UP (Uniprocessor)	A software stack, typically an OS that does not support simultaneous operation across multiple cores.