ARM® System Control and Management Interface

Platform Design Document

Non-Confidential

Document number: DEN0056A

System Control and Management Interface Platform Design Document

Release information

The Change History table lists the changes that are made to this document.

Table 1 Change history

Date	Issue	Confidentiality	Change
May 2017	Issue A	Non-confidential	Version 1.0, first external release

Proprietary notice

Product Status

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

Web Address

http://www.arm.com

Contents

1 About this Document

This document describers an extensible operating system-independent software interface to perform various system control and management tasks, including power and performance management.

1.1 References

This document refers to the following documents.

Reference	Document Number	Title
[ACPI]		Advanced Configuration and Power Interface Specification.
[FDT]		Flattened Device Tree.
[PSCI]	DEN0022D	Power State Coordination Interface.
[PCSA]	DEN0050B	Power Control System Architecture Specification.
[ARMTF]		ARM Trusted Firmware.
[ARM]	DDI 0487	ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile.

1.2 Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
ACPI	Advanced Configuration and Power Interface
Agent	Entity that sends commands to the platform using SCMI. For example, the OSPM running on an AP or an on-chip management controller.
AP	Application processor, that is a processor that is running the operating system and applications in the system.
Command	A message that is sent from an agent to the platform.
Delayed response	A message that is sent from the platform to an agent to indicate completion of the work that is associated with an asynchronous command.
FDT	Flattened Device Tree
Message	An individual communication from an agent to the platform or from the platform to an agent.
Notification	A message that is sent from the platform to an agent to alert of a change in state.
OSPM	Operating System-directed Power Management. Typically, this acronym refers to the software components of an Operating System that interact with the power management interfaces of the platform.
Platform	Components in the system that implement SCMI protocols. An SCP is an example of a platform component that could implement the SCMI

	protocols.
PSCI	Power State Coordination Interface
SCMI	System Control and Management Interface, which is described in this specification.
SCP	System Control Processor, see [PCSA].

1.3 Feedback

ARM welcomes feedback on its documentation.

1.3.1 Feedback on this manual

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

- The title.
- The document and version number, ARM DEN 0056A.
- The page numbers to which your comments apply.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

2 Introduction

This document describes the *System Control and Management Interface* (SCMI), which is a set of operating system-independent software interfaces that are used in system management. SCMI is extensible and currently provides interfaces for:

- Discovery and self-description of the interfaces it supports.
- Power domain management, which is the ability to place a given device or domain into the various power-saving states that it supports.
- Performance management, which is the ability to control the performance of a domain that is composed of compute engines such as *application processors* (APs), GPUs, or other accelerators.
- Clock management, which is the ability to set and inquire rates on platform-managed clocks.
- Sensor management, which is the ability to read sensor data, and be notified of sensor value changes.

There is a strong trend in the industry to provide microcontrollers in systems to abstract various power, or other system management tasks, away from APs. These controllers usually have similar interfaces, both in terms of the functions that are provided by them, and in terms of how requests are communicated to them. The *Power Control System Architecture* (PSCA) describes how systems using this approach can be built. For detailed information about the PSCA, see [PSCA].

PSCA defines the concept of the *System Control Processor* (SCP), a processor that is used to abstract power and system management tasks from the APs. The SCP can take requests from APs and other system agents. It can coordinate these requests and place components in the platform into appropriate power and performance states. The SCMI interface is particularly relevant to these kinds of systems. The interface provides two levels of abstraction:

- **Protocols**
  Each group of related functions is referred to as a protocol. The SCMI interface structure is extensible, and therefore other protocols could be added in the future.
- **Transports**
  The protocols communicate through transports. A transport specification describes how protocol messages are communicated between agents using the interface and the platform components that implement the protocol messages.

The interface is intended to be described in firmware, using either the *Flattened Device Tree* (FDT) or *Advanced Configuration and Power Interface* (ACPI) specification. For more information, see [FDT] and [ACPI]. Because the protocols are intended to be generic, they result in generic kernel code to drive them. However, in the ACPI case, the interface can also be driven from ASL methods. This document is arranged into the following sections:

- Section 3 provides background into the interface structure.
- Section 4 describes protocols.
- Section 5 describes transports.

3 System Control and Management Interface structure

The SCMI is intended to allow agents such as an operating system to manage various functions that are provided by the hardware platform it is running on, including power and performance functions. As described in the introduction, SCMI provides two levels of abstraction, protocols and transports. Protocols define individual groups of system control and management messages. A protocol specification describes the messages that it supports. Transports describe the method by which protocol messages are communicated between agents and the platform. ARM strongly recommends that transports be operating system independent and capable of being virtualized.

Transports comply with the following rules:

- A transport might support multiple channels. Each agent has one or more dedicated channels. Channels cannot be shared between agents.
- Systems that use TrustZone technology can have both secure and non-secure channels. Data in a secure channel can only be read or written by secure memory accesses.

It is intended that protocols and transports are developed independently.

The protocols that are described in this document are intended to be used by power and performance management agents such as an operating system, also referred to as *Operating System-directed Power Management* (OSPM). Typical agents are:

- An OSPM that operates in Non-secure Exception levels.
- Secure-world software that is running on an AP.
- External entities in the system, such as a management controller in an enterprise system, or a modem in a mobile system.

The term *platform* is intended to describe the set of hardware components that interpret the messages and provide the necessary functionality. The term *agent* is used to describe the caller of the interface. Each agent that communicates with the platform must have its own set of dedicated channels. This requirement removes the need for create locking primitives across agents that are running entirely different software stacks. For example, a management controller and an operating system. In addition, dedicated channels provide a method for the platform to identify which agent is sending a message.

Figure 1 illustrates an example system that implements the SCMI interface. In this example, the platform includes an SCP that handles SCMI commands that are issued from APs. The latter communicates with the SCP through secure and non-secure channels. The figure also shows a device that uses SCMI protocols to manage its power and performance. As described in [PCSA], the SCP coordinates requests from all requesting agents and drives the hardware into appropriate power or performance states.

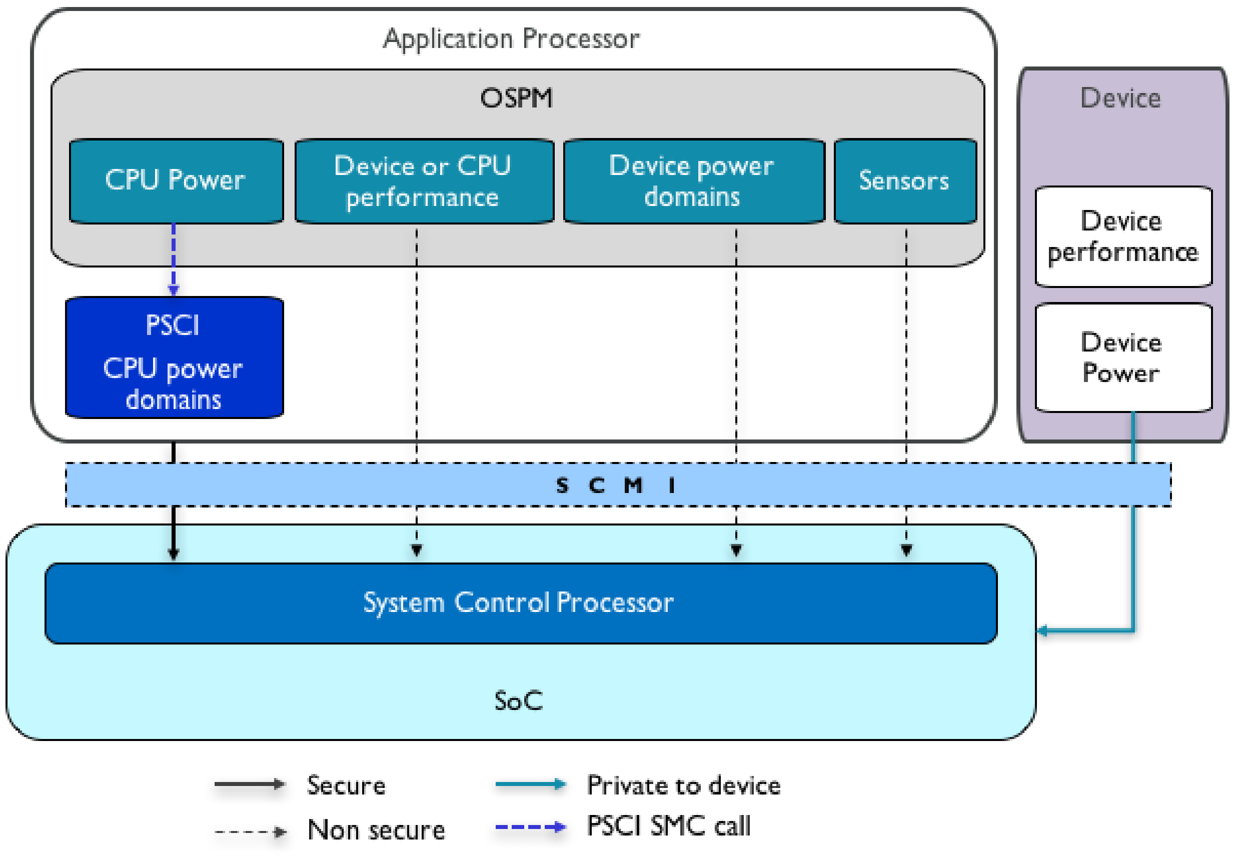

Figure 1 Example system that implements SCMI.

4 Protocols

4.1 Protocol structure

As described in *System Control and Management Interface structure*, a protocol is a group of messages. The following sections describe the message flow, the structure of messages, and discovery.

4.1.1 Agents, messages, and channels

The term agent is used to describe components that are clients of the SCMI interface. Agents have the following properties:

- Agents run a software stack with different privilege levels.
- Agent software stacks are independent from each other. This makes resource sharing, or the ability to write cross-agent locking primitives difficult. For example, one agent might be an operating system running on all APs, and another agent might be firmware running on a manageability controller.

Figure 2 describes how agents and the platform communicate. The diagram shows multiple agents communicating with the platform. Agents and the platform communicate over transport channels. Each agent has dedicated channels, which are used to send messages to, and receive messages from, the platform. Each channel is a bidirectional communication pipe between the agent and the platform. For a given channel either the agent or the platform is the master, or initiator, of communications. The master can place a message on a channel. At the other end, the slave processes the message, and, as a response, it might place return data on the channel. Depending on which entity is the master, a channel is one of two types:

- On **A**gent to **P**latform (A2P) channels, the agent is the master.
- On **P**latform to **A**gent (P2A) channels, the platform is the master.

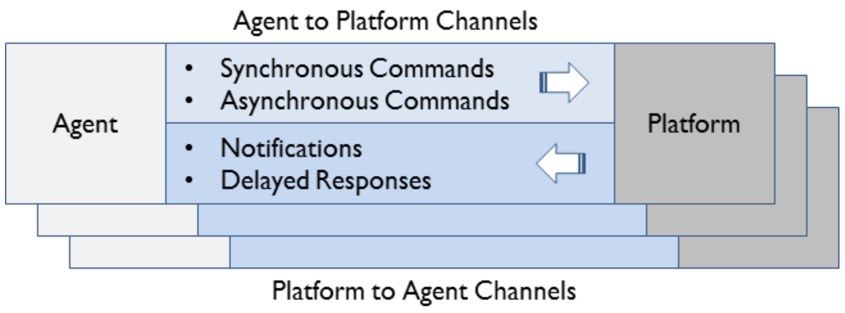

Figure 2 messages and channels

Each agent can have one or more A2P channels and one or more P2A channels. However, these channels have to be dedicated to that specific agent, and cannot be shared with other agents. This enables the platform to identify which agents are communicating with it.

The properties of channels are specific to the transport that is used to send messages. An A2P transport might support interrupt-driven communication to send messages, where the platform generates an interrupt when it processes the message. The interrupt alerts the agent that the channel can now be used to send a further message. The same holds true for a P2A transport, where the agent triggers an interrupt to the platform when it posts a command, which alerts the platform that it has a new command to process. On the other hand, a transport might only support polling-based communications. A transport can also support both methods, and allow the agent to choose.

Messages are used by agents to make requests to the platform. The messages can carry various parameters, including an identifier for the requested operation. In turn, the platform carries out the requested operation, and might generate data in response to the message. From this point of view, messages are analogous to remote procedure calls, which can carry various parameters, and can also provide return data. The platform can also send messages to an agent, typically to indicate completion of a long job, or to notify of an event.

Messages that are sent by agents on A2P channels are known as commands and fall into two categories:

- **Synchronous**
  Commands that free the channel when the requested work has been completed. The platform responds to these commands over the A2P channel that was used to send them. Therefore, the channel cannot be used to send another command until the previous synchronous command has completed.
- **Asynchronous**
  For these commands, the platform schedules the requested work to complete later in time. Therefore, these commands complete almost immediately to the calling agent, freeing the channel for new commands. The response to an asynchronous command indicates the success or failure in the ability to schedule the requested work. When the work has completed, the platform sends an additional delayed response message to the client over a P2A channel.

Messages that the platform can send to an agent over P2A channels also fall into two categories:

- **Delayed response**
  Messages sent to indicate completion of the work that is associated with an asynchronous command.
- **Notifications**
  These messages provide notifications of events taking place in the platform. Events might include changes in power state, performance state, or other platform status.

4.1.2 Message format

Messages are analogous to remote procedure calls, and therefore must be representative of the particular operation being requested, and any parameters or return values.

Each message carries a message header, which identifies the operation being requested. Each message belongs to a protocol. Therefore, the header of the message includes an 8-bit protocol identifier. This is known as the protocol_id. Within a protocol, each message is associated with a unique 8-bit identifier. This is known as the message_id.

A message can take several 32-bit arguments and can provide 32-bit return values. All parameters, message headers, and return arguments are expressed in little endian format. The endianness rule does not apply to strings. For all messages, any reserved field is set to all zeros.

Values for the protocol_id are described in Table 2.

Table 2 Protocol identifiers

protocol_id	Description
`0x0 - 0xF`	Reserved.

`0x10`	Base protocol.
`0x11`	Power domain management protocol.
`0x12`	System power management protocol.
`0x13`	Performance domain management protocol.
`0x14`	Clock management protocol.
`0x15`	Sensor Management Protocol.
`0x16-0x7F`	Reserved for future use by this specification.
`0x80-0xFF`	Might be used for platform-specific extensions to this interface.

For all protocols and all transports, messages are sent to the platform using a 32-bit message header, which is described in Table 3.

Table 3 Message header format

Field	Description
Bits[31:28]	Reserved, must be zero.
Bits[27:18]	Token.
Bits[17:10]	protocol_id.
Bits[9:8]	Message type.
Bits[7:0]	message_id.

Each message type has additional requirements that are described below.

Commands

All commands, synchronous or asynchronous, have a message type of 0.

How the token field is used is entirely up to the caller. However, when a command returns, the platform must return the whole message header unmodified. The message header must always be the first parameter that is sent by an agent and returned by the platform.

In addition to the message header, commands return error status codes and can return more data. Any command that is sent with an unknown protocol_id or message_id must be responded to with a return value of NOT_SUPPORTED as the status code. Status codes are provided in section 4.1.4.

Delayed responses

Delayed responses have a message type of 2.

Delayed response messages are sent by the platform to the agent to indicate completion of work that was requested by an asynchronous command. The message header that is associated with a delayed response uses the format that is described in Table 3. The message_id of a delayed response matches that of its associated asynchronous command. The token in the message header matches the token of the associated asynchronous command. The payload that is associated with a delayed response includes a status error code, but might include additional data.

Notifications

Notifications have a message type of 3.

Notifications provide a mechanism for the platform to inform agents about events taking place in the platform. Optionally, the implementation can provide information about which agent caused an event. To this end, a notification payload carries an agent identifier, agent_id, as its first parameter. The agent_id is an integer identifier that can be used to codify the agent that generated an event. The agent_id uses the following rules:

- A value of 0 identifies the platform itself.
- Where implemented, agent_ids are sequential and start from one.
- Agent identifiers and their mapping to other components are platform-specific. Where necessary, this must be described to operating system through firmware table technologies such as FDT or ACPI.
- If agent identification is not supported, the implementation must set the agent_id to zero in notifications.

4.1.3 Protocol discovery

This specification encompasses various protocols. However, not every protocol has to be present in an implementation, because not every protocol is relevant for every market segment. Furthermore, the platform chooses which protocols it exposes to a given agent. The only protocol that must be implemented is the base protocol, which is described in section 4.2. The base protocol is used by an agent to discover which protocols are available to it.

All protocols, whether they are generic or vendor specific, must implement three messages with message_ids of `0x0`, `0x1`, and `0x2` as described in Table 4.

Table 4 Required messages

Message_id	Message	Description
`0x0`	**PROTOCOL_VERSION**	Returns the version of protocol.
`0x1`	**PROTOCOL_ATTRIBUTES**	Returns properties that are associated with the protocol implementation.
`0x2`	**PROTOCOL_MESSAGE_ATTRIBUTES**	Takes a message_id as a parameter and returns implementation details specific to that message.

Protocols might implement additional discovery messages.

Protocol versioning uses a 32-bit unsigned integer, where the upper 16 bits are the major revision, and the lower 16 bits are the minor revision.

The following rules apply to the version numbering:

- Higher numbers denote newer versions.

- Different major revision values indicate possibly incompatible messages. For two protocol versions, A and B, which differ in major revision, and where B is higher than A, the following might be true:
  - B can remove messages that were present in A.
  - B can add new messages that were not present A.
  - B can modify the behavior or parameters of messages that are also present in A.
- Minor revisions allow extensions, but must retain compatibility. For two protocol versions, A and B, that differ only in the minor revision, and where B is higher than A, the following must hold:
  - Every message in A must also be present in B, and work with compatible effect.
  - It is possible for revision B to have a higher message count than revision A.

4.1.4 SCMI status codes

Messages can return status codes to the sender. Negative 32-bit integers are used to return error status codes. Values 0 to -127 are reserved by this specification. Values below -127 can be used for vendor-specific errors.

Table 5 describes the error codes for SCMI messages.

Table 5 Status codes

Status code	Description
0	SUCCESS
-1	NOT_SUPPORTED
-2	INVALID_PARAMETERS
-3	DENIED
-4	NOT_FOUND
-5	OUT_OF_RANGE
-6	BUSY
-7	COMMS_ERROR
-8	GENERIC_ERROR
-9	HARDWARE_ERROR
-10	PROTOCOL_ERROR
-11 to -127	Reserved
< -127	Vendor specific

The specification of each SCMI message describes which error codes are appropriate to that message. However, unless otherwise specified, the following status codes apply to all command messages that are sent from an agent to the platform:

Code	Description
SUCCESS	Successful completion of the command.
INVALID_PARAMETERS	One or more parameters passed to the command are invalid or beyond legal limits.
NOT_SUPPORTED	The command is not implemented or not available to the calling agent.
COMMS_ERROR	The command could not be correctly transmitted to the platform.
GENERIC_ERROR	The command failed to be processed owing to an unspecified fault within the platform.
BUSY	The platform is out of resources and thus unable to process a command. While the platform might need to use this error when it is out of resources, ARM strongly recommends that the implementation ensures that sufficient resources are available to handle the more frequently issued commands in order to guarantee availability of service. In particular, the platform must guarantee service for the following commands: • System power protocol commands • AP/domain power management commands.
HARDWARE_ERROR	A hardware error occurred in a platform component during execution of a command.
PROTOCOL_ERROR	Returned when the receiver detects that the caller has violated the protocol specification.

4.2 Base protocol

This protocol describes the properties of the implementation and provide generic error management. The Base protocol provides commands to:

- Describe protocol version.
- Discover implementation attributes and vendor identification.
- Discover which protocols are implemented.
- Discover which agents are in the system.
- Register for notifications of platform errors.

This protocol is mandatory.

4.2.1 Commands

4.2.1.1 PROTOCOL_VERSION

This command returns the version of this protocol. For this version of the specification, the value that is returned must be `0x10000`, which corresponds to version 1.0.

message_id: `0x0`
protocol_id: `0x10`
This command is mandatory.

Return values

Name	Description
int32 status	See section 4.1.4 for status code definitions.
uint32 version	For this revision of the specification, this value must be `0x10000`.

4.2.1.2 PROTOCOL_ATTRIBUTES

This command returns the implementation details that are associated with this protocol.

message_id: `0x1`
protocol_id: `0x10`
This command is mandatory.

Return values

Name	Description	
int32 status	See section 4.1.4 for status code definitions.	
uint32 attributes	Bits[31:16]	Reserved, must be zero.
	Bits[15:8]	Number of agents in the system
	Bits[7:0]	Number of protocols that are implemented, excluding the base protocol.

If the platform does not support agent discovery, then it reports the number of agents in the system as zero, and all notifications carry a zero in the agent_id.

4.2.1.3 PROTOCOL_MESSAGE_ATTRIBUTES

On success, this command returns the implementation details associated with a specific message in this protocol.

The command returns the NOT_FOUND status code to indicate that the message identified by message_id is not provided by the platform implementation. Other status codes and their usage by this protocol are described in section 4.1.4.

message_id: `0x2`

protocol_id: `0x10`

This command is mandatory.

Parameters

Name	Description
uint32 message_id	message_id of the message.

Return values

Name	Description
int32 status	NOT_FOUND See section 4.1.4 for status code definitions.
uint32 attributes	Flags that are associated with a specific command in the protocol. For all commands in this protocol, this parameter has a value of 0.

4.2.1.4 BASE_DISCOVER_VENDOR

This command provides a vendor identifier ASCII string.

message_id: `0x3`

protocol_id: `0x10`

This command is mandatory.

Return values

Name	Description
int32 status	See section 4.1.4 for status code definitions.
uint8 vendor_identifier [16]	Null terminated ASCII string of up to 16 bytes with a vendor name.

4.2.1.5 BASE_DISCOVER_SUB_VENDOR

On success, this optional command provides a sub vendor identifier ASCII string.

message_id: `0x4`

protocol_id: `0x10`

This command is optional.

Return values

Name	Description
int32 status	See section 4.1.4 for status code definitions.
uint8 vendor_identifier [16]	Null terminated ASCII string of up to 16 bytes with a vendor name.

4.2.1.6 BASE_DISCOVER_IMPLEMENTATION_VERSION

This command provides a vendor-specific implementation 32-bit version. The format of the version number is vendor-specific, but version numbers must be strictly increasing so that a higher number indicates a more recent implementation.

message_id: `0x5`
protocol_id: `0x10`
This command is mandatory.

Return values

Name	Description
int32 status	See section 4.1.4 for status code definitions.
uint32 implementation_version	Format is vendor-specific.

4.2.1.7 BASE_DISCOVER_LIST_PROTOCOLS

This command allows the agent to discover which protocols it is allowed to access.

message_id: `0x6`
protocol_id: `0x10`
This command is mandatory.

Parameters

Name	Description
uint32 skip	Number of protocols to skip.

Return values

Name	Description
int32 status	See section 4.1.4 for status code definitions.
uint32 num_protocols	Number of protocols that are returned by this call.

uint32 protocols[1+(num_protocols-1)/4]	Array of protocol identifiers that are implemented, excluding the base protocol, with four protocol identifiers packed into each array elementThe PROTOCOL_ATTRIBUTES command can be used to determine the number of protocols implemented.

The following pseudocode illustrates how this command can be used.

```
int status = 0;
int skip = 0;
int total_protocols = 0;
int num_protocols = 0;

uint32 attributes = 0;
uint32* protocols = NULL;
invoke_PROTOCOL_ATTRIBUTES(&status,&attributes);

if (status)
      goto clean_up_and_return;

total_protocols = (attributes & NUM_PROTOCOLS_MASK) >>
                          NUM_PROTOCOLS_SHIFT;

if (!total_protocols)
      goto clean_up_and_return;

uint8* protocols;

do {
      invoke_BASE_DISCOVER_LIST_PROTOCOLS(skip,
            &status, &num_protocols, protocols);

      if (status)
            goto clean_up_and_return;


      for (int ix = 0; ix < num_protocols; ix++)
      {
            uint8 prot = protocols[ix];
            add_to_protocol_database(prot);
            skip++;
      }
} while (skip < total_protocols);
```

4.2.1.8 BASE_DISCOVER_AGENT

This optional command allows the caller to discover the name of an agent, described through an ASCII string of up to 16 bytes. Where agent identifiers, and this message, are supported, this command also allows the caller to obtain their agent identifier. A caller can discover if the command is implemented by issuing the PROTOCOL_MESSAGE_ATTRIBUTES command and passing its message_id. If the command is implemented, PROTOCOL_MESSAGE_ATTRIBUTES returns SUCCESS (0).

In addition to the standard status codes described in section 4.1.4, the command can return the NOT_FOUND error if the agent that is identified by agent_id does not exist. This would be the case if the agent identifier is larger than the number of agents that is reported through PROTOCOL_ATTRIBUTES.

On success, the call returns the agent identifier of the calling agent in the status field.

Agent identifiers, agent_id, describe agents in the system that can use the SCMI protocols. Not every agent can use all protocols, and some protocols can offer different views to different agents. An agent_id of 0 is reserved to identify the platform itself. If the command is not implemented, the caller does not interpret agent identifiers in notifications, and the platform sets agent_id to zero in notifications. Where supported, agent_id values are sequential, start from one, and are limited by the number of agents that is reported through PROTOCOL_ATTRIBUTES.

If called with an agent_id of 0, the string returned in the name parameter must start with the letters "platform".

message_id: `0x7` protocol_id: `0x10` This command is optional.	
Parameters	
Name	**Description**
uint32 agent_id	Identifier for the agent.
Return values	
Name	**Description**
int32 status	NOT_FOUND. See section 4.1.4 for status code definitions.
uint8 name[16]	Null terminated ASCII string of up to 16 bytes in length.

4.2.1.9 BASE_NOTIFY_ERRORS

An implementation can optionally provide notifications of errors in the platform to an agent that has registered through this command. A caller can discover if this command is implemented by issuing the PROTOCOL_MESSAGE_ATTRIBUTES command and passing the message_id of this command. If the command is implemented, PROTOCOL_MESSAGE_ATTRIBUTES returns SUCCESS (0).

Error notification is used to notify agents of commands that could not proceed due to unpredictable circumstances, such as internal hardware errors. Further information on the error notification and associated payload is provided in section 4.2.2.1, which describes the BASE_ERROR_EVENT notification.

message_id: `0x8` protocol_id: `0x10` This command is optional.	
Parameters	
Name	**Description**

uint32 notify_enable	Bits[31:1]	Reserved, must be zero.
	Bit[0]	Notify enable: If this value is zero, the platform does not send any BASE_ERROR_EVENT messages to the calling agent. If this value is one, the platform sends BASE_ERROR_EVENT messages to the calling agent when an error is detected. For more details on the BASE_ERROR_EVENT notification, see 4.2.2.1.

Return values

Name	**Description**
int32 status	INVALID_PARAMETERS See section 4.1.4 for status code definitions.

4.2.2 Notifications

4.2.2.1 BASE_ERROR_EVENT

These notifications are sent to any agent that has registered to receive them, provided the platform implements Base error notifications.

Errors that are reported by the platform are one of two types:

- **Fatal error**
  Indicates that the platform is no longer able to process commands. The error might be accompanied by the list of messages that were being processed when the failure took place.
- **Non-fatal error**
  Indicates that the platform was not able to process some commands, but it is still operational. The error notification is accompanied by the list of commands that could not be processed.

By definition, fatal error notifications cannot be guaranteed, and the platform must not rely on these notifications as a mechanism to enable recovery.

Error notifications must not be used as mechanism to report that a command cannot be executed as requested due to constraints that arise in normal operation.

On initial boot of an agent, these notifications must be disabled by default to that agent.

message_id: `0x0`
protocol_id: `0x10`
This command is optional.

Parameters

Name	**Description**

uint32 agent_id	Identifier of the agent that caused this notification.	
uint32 error_status	Bit[31]	Fatal. Set if error is fatal and platform cannot continue. Cleared if error is non-fatal but commands have failed.
	Bits[30:10]	Reserved, must be zero.
	Bits[9:0]	Command count, number of commands in the command list. A value of zero is possible if the error cannot be attributed.
{uint32 message_header unit32 status}[N]	Each entry in the command list is a tuple, where the first entry is the message header of the command, and second is an error status code that is associated with the command. The size of the list is specified by the command count sub-field.	

4.3 Power domain management protocol

This protocol is intended for management of power states of power domains.

The power domain management protocol provides commands to:

- Describe the protocol version.
- Discover implementation attributes.
- Set the power state of a domain.
- Get the current power state of a domain.
- Optionally get notifications when power domains change state.
- Optionally return statistics on residency and usage count of a given power state.

4.3.1 Power domain management protocol background

In this document, a power domain is defined as a group of components that are powered together. For example, a set of components that share a power source, and can only be turned ON or OFF as a group, form a power domain. Power domains have the following properties:

They can include one or more devices.

- Power domains must at least support ON and OFF, but can support additional power states.
- In the ON state, the domain is operational and devices within it can run.
- In the OFF state, the domain has no power supplied to it and devices within it cannot run.

Domains can have dependencies on other domains. For example, a parent domain can include a child domain. In such a case, if the child domain is ON, the parent domain is also necessarily ON.

The protocol does not cover discovery of power states that are supported by a domain, or description of the properties of the states, for example associated latencies, context loss, or domain dependencies. This information is expected to be provided to the caller by way of firmware tables in FDT or ACPI.

Protocol commands take integer identifiers to identify the power domain that they apply to. The identifiers are sequential and start from 0.

The protocol can be used to manage the power state of application processors, or to manage the power state of other devices in the system.

Operating systems that are running on application processors must not directly use SCMI to manage the power state of these processors. Instead, power states for domains that include APs must be managed using PSCI calls from the operating system. When the OSPM calls a PSCI function, the PSCI implementation, which is described in [PSCI, ARMTF], can communicate with the platform using this protocol over secure channels. This protocol allows SCMI to provide an implementation for PSCI functions designed to manage the power of application processors, such as CPU_DEFAULT_SUSPEND, CPU_SUSPEND, CPU_FREEZE, CPU_ON and CPU_OFF. These functions map to various use cases including idle, secondary core boot, and hot plug. The list does not include system power state transitions such as system shutdown or reset, which are covered by the system power management protocol instead, as described in section 4.4.

Agents that are not running on application processors can register to receive notifications of power state changes to these power domains.

Non-secure channels can be used to manage power domains for devices that do not include application processors. Any agent can register for power state change notifications for these domains.

An implementation can include devices that are intended for use only by secure entities in the system such as a trusted OS. Power domains for such devices must be managed through secure channels.

Agents other than the OSPM can manage power domains. In a multi-agent system, domains that are presented to a given agent might be exclusive to it, or they might be shared with others. If the domain is shared, the platform must track power state change requests from each agent. The power domain must be in the shallowest state that is requested among the agents. These conditions follow a platform-coordinated model analogous to that described in [ACPI] and [PSCI]. Platform policy dictates which agents can see which domains.

For all messages in this protocol, the interpretation of the power state parameter is specific to the combination of agent and the power domain that the agent is managing. A power domain with Application Processors that is managed by a PSCI agent must support representation of the power state parameter based on definitions in [PSCI]. On the other hand, for power domains pertaining to devices, the power state parameter must minimally represent two pre-defined states, ON and OFF. Power state encoding for device power domains is described in Table 6.

Table 6: Power State Parameter Layout for Device Power Domains

Bit field	Description
31	Reserved. Must be zero.
30	StateType If set to 0, indicates that context is preserved. If set to 1, indicates that context is lost.
29:28	Reserved. Must be zero.
27:0	StateID

	A value of zero when StateType is set to 0 represents the ON state. A value of zero when StateType is set to 1 represents the OFF state. All other values are IMPLEMENTATION_DEFINED.

4.3.2 Commands

4.3.2.1 PROTOCOL_VERSION

On success, this command returns the Protocol version. For this version of the specification, the return value must be `0x10000`, which corresponds to version 1.0.

message_id: `0x0`
protocol_id: `0x11`
This command is mandatory.

Return values

Name	**Description**
int32 status	See section 4.1.4 for status code definitions.
uint32 version	For this revision of the specification, this value must be `0x10000`.

4.3.2.2 PROTOCOL_ATTRIBUTES

This command returns the implementation details associated with this protocol.

message_id: `0x1`
protocol_id: `0x11`
This command is mandatory.

Return values

Name	**Description**	
int32 status	See section 4.1.4 for status code definitions.	
uint32 attributes	Bits[31:16]	Reserved, must be zero.
	Bits[15:0]	Number of power domains.

uint32 statistics_address_low	The lower 32 bits of the physical address where the statistics shared memory region is located. The address must be in the memory map of the calling agent. This field is invalid and must be ignored if the statistics_len field is set to 0.
uint32 statistics_address_high	The upper 32 bits of the physical address where the statistics shared memory region is located. The address must be in the memory map of the calling agent. This field is invalid and must be ignored if the statistics_len field is set to 0.
uint32 statistics_len	The length in bytes of the statistics shared memory region. A value of 0 in this field indicates that the platform doesn't support the statistics shared memory region.

The statistics shared memory region is described in section 4.3.4.

4.3.2.3 PROTOCOL_MESSAGE_ATTRIBUTES

On success, this command returns the implementation details associated with a specific message in this protocol. In addition to the standard status codes that are described in section 4.1.4, the command can return NOT_FOUND if the message that is identified by message_id is not implemented.

This command can be used to inquire if power state change notifications are supported, by passing POWER_STATE_NOTIFY message identifier to the call. If the platform returns SUCCESS then it supports power state change notifications. Otherwise, if the platform returns NOT_FOUND, then it is an indication that notifications are not implemented, or that notifications are not available to the calling agent. The notifications commands are described in sections 4.3.2.7 and 4.3.3.1.

message_id: `0x2`

protocol_id: `0x11`

This command is mandatory.

Parameters

Name	**Description**
uint32 message_id	message_id of the message.

Return values

Name	**Description**
int32 status	NOT_FOUND See section 4.1.4 for status code definitions.
uint32 attributes	Flags that are associated with a specific command in the protocol. In the current version of the specification, this value is always 0.

4.3.2.4 POWER_DOMAIN_ATTRIBUTES

This command returns the attribute flags associated with a specific power domain. The command returns the NOT_FOUND status code if the domain identified by domain_id does not exist.

message_id: `0x3`
protocol_id: `0x11`
This command is mandatory.

Parameters

Name	Description
uint32 domain_id	Identifier for the domain. Domain identifiers are limited to 16 bits, and the upper 16 bits of this field are ignored by the platform.

Return values

Name	Description	
int32 status	NOT_FOUND See section 4.1.4 for status code definitions.	
uint32 attributes	Bit[31]	Power state change notifications support. Set to 1 if power state change notifications are supported on this domain. Set to 0 if power state change notifications are not supported on this domain.
	Bit[30]	Power state asynchronous support. Set to 1 if power state can be set asynchronously. Set to 0 if power state cannot be set asynchronously.
	Bit[29]	Power state synchronous support. Set to 1 if power state can be set synchronously. Set to 0 if power state cannot be set synchronously.
	Bits[28:0]	Reserved, must be zero.
uint8 name[16]	Null terminated ASCII string of up to 16 bytes in length describing the power domain name.	

For some agents, the platform might only allow registration and receipt of notifications for power domains, and disallow setting of power states of those domains.

4.3.2.5 POWER_STATE_SET

This command allows an agent to set the power state of a power domain. Power domains can be managed synchronously or asynchronously:

- **Synchronous Mode**
  A call with valid parameters completes and frees the channel when the domain has transitioned to the desired power state.
- **Asynchronous Mode**
  The call completes immediately and the caller can register for notifications if it wishes to observe the power state transition. These notifications are described in section 4.3.3.1.

When this command is used for power domains that include application processors, the Async flag is ignored. This call must return to the calling AP before that AP is powered down. Following this call, the AP executes some instructions before invoking a *Wait for Interrupt* (WFI) instruction [ARM]. The platform controller that implements SCMI begins the transition to the required power state when it observes the WFI. The method used by the platform controller to observe the WFI is IMPLEMENTATION DEFINED. For these power domains, this protocol can be used to implement PSCI CPU_SUSPEND, CPU_ON, CPU_FREEZE, CPU_DEFAULT_SUSPEND and CPU_OFF functions.

In addition to the standard usage of status codes described in section 4.1.4, the command involves special significance for the following status codes, as described:

- SUCCESS for a power domain that can only be set synchronously, this status is returned after the power domain has transitioned to the desired state. For a power domain that is managed asynchronously, this status is returned if the command parameters are valid and the power state change has been scheduled.
- NOT_FOUND if the power domain identified by domain_id does not exist.
- INVALID_PARAMETERS if the requested power state does not represent a valid state for the power domain that is identified by domain_id.
- NOT_SUPPORTED if the request is not supported.

A power domain can contain other power domains. If the caller wants to change the state of a power domain and one of its parents, the power domain parameter must identify the child. The required power state for the child domain, and its parents, must be encoded in the power state parameter. How this is encoded in the power_state parameter is IMPLEMENTATION DEFINED.

message_id: `0x4`
protocol_id: `0x11`
This command is mandatory.

Parameters

Name	**Description**

uint32 flags	Bits[31:1]	Reserved, must be zero.
	Bit[0]	Async flag. Set to 1 if power transition must be done asynchronously. Set to 0 if power state transition must be done synchronously. The async flag is ignored for application processor domains.
uint32 domain_id	Identifier for the power domain.	
uint32 power_state	Platform-specific parameter identifying the power state of the domain. For device power domains, this parameter is encoded as described in Table 6.	

Return values

Name	**Description**
int32 status	One of the following: • SUCCESS • NOT_FOUND • INVALID_PARAMETERS • NOT_SUPPORTED See section 4.1.4 for status code definitions.

4.3.2.6 POWER_STATE_GET

This command allows the calling agent to request the current power state of a power domain.

—— Note ——

It is possible for the power_state value returned by this command to be stale by the time the command completes, as another state change request could have been initiated and completed in the interim.

In addition to the standard status codes described in section 4.1.4, the command can return the error NOT_FOUND if the power domain identified by domain_id does not exist.

message_id: `0x5`
protocol_id: `0x11`
This command is mandatory.

Parameters

Name	**Description**

uint32 domain_id	Identifier for the power domain.

Return values

Name	Description
int32 status	NOT_FOUND See section 4.1.4 for status code definitions.
uint32 power_state	Platform-specific parameter identifying the power state of this domain.

4.3.2.7 POWER_STATE_NOTIFY

This command allows the caller to request notifications from the platform for state changes in a specific power domain. These notifications are sent using the POWER_STATE_CHANGED notification, which is described in section 4.3.3.1. In addition to the standard status codes described in section 4.1.4, the command can return the error NOT_FOUND if the power domain identified by domain_id does not exist.

Notification support is optional, and PROTOCOL_MESSAGE_ATTRIBUTES must be used to discover whether this command is implemented.

These notifications must be disabled by default during initial boot of the platform.

message_id: `0x6`
protocol_id: `0x11`
This command is optional.

Parameters

Name	Description	
uint32 domain_id	Identifier for the power domain.	
uint32 notify_enable	Bits[31:1]	Reserved must be zero.
	Bit[0]	Notify enable. This bit can have one of the following values: 0, which indicates that the platform does not send any POWER_STATE_CHANGED messages to the calling agent. 1, which indicates that the platform does send POWER_STATE_CHANGED messages to the calling agent when a domain changes power state. See section 4.3.3.1 for more details about the POWER_STATE_CHANGED notification.

Return values

Name	Description

int32 status	NOT_FOUND INVALID_PARAMETERS See section 4.1.4 for status code definitions.

4.3.3 Notifications

4.3.3.1 POWER_STATE_CHANGED

If an agent has registered to receive power state change notifications for the power domain that is identified by domain_id, the platform sends these notifications to that agent when a power state transition commences, including transitions to an ON state.

The platform is not required to guarantee sending a notification to an agent for every state transition. In particular, if a number of power states transitions take place in quick succession, the platform is allowed to issue a notification for the last transition only.

Note that notified power states might not match those requested by the agent that is notified. The power state that is finally selected by the platform might differ from that requested by an agent, due to coordination with other requests on the same domain.

message_id: `0x0`
protocol_id: `0x11`
This command is optional.

Parameters

Name	**Description**
uint32 agent_id	Identifier of the agent that caused the power transition.
uint32 domain_id	Identifier of the power domain whose power state was changed.
uint32 power_state	The value of the power state that the power domain transitioned to. These notifications take place when the transition has completed.

4.3.4 Power state statistics shared memory region

Optionally, the platform can provide a statistics shared memory region that is associated with the power state protocol. Whether support is present is indicated by the PROTOCOL_ATTRIBUTES command, which is described in section 4.3.2.2. The PROTOCOL_ATTRIBUTES command also provides the address and the size of the shared memory region. The region provides usage counts and residency information for each power state that is used by each power state domain. The memory must be accessible from the Non-secure world, and OSPM must map it as non-cached normal memory or device memory. For a given power domain, and for each power state in a domain, statistics in the shared memory region track the number of times the state has been used and the amount of time the domain has been in the state. The statistics must be updated regardless of the agent in the system that placed a domain into a given power state. After a system reset, suspend, or shutdown, all the statistics must be initialized to zero. Time measurements are in microseconds.

The design of the statistics shared memory region allows the platform implementation to choose which power domains are included. However, if a domain is included, all its power states must be represented, including time that is spent in an ON state.

The format of the frame is described in Table 7.

Table 7 Power state statistics shared memory region

Field	Byte Length	Byte Offset	Description
Signature	`0x4`	`0x0`	`0x504F5752` ('POWR').
Revision	`0x2`	`0x4`	For this revision, this field must be zero.
Attributes	`0x2`	`0x6`	For this revision, this field must be zero.
Number of domains	`0x2`	`0x8`	Number of domains for which statistics are collected.
Reserved	`0x6`	`0xA`	Must be zero.
Power domain offset array	`0x4` × (Total number of power domains)	`0x10`	For each power domain, this array provides a 4-byte offset, from the start of the shared memory area, to the memory location of the power domain entry in the data section. The entry is described in Table 8. A value of zero for the offset of a given power domain indicates that statistics are not collected for that domain.
Power domain data section	--	--	This area must start at an offset of `0x10` + `0x4` × (Number of power domains), or higher.

The power domain data section contains an entry for each power domain for which statistics are collected. The format for each entry is described in Table 8.

Table 8 Power domain entry

Field	Byte Length	Byte Offset	Description
Number of power states	`0x2`	`0x0`	Number of power state entries in the power state array.
Current power state	`0x2`	`0x2`	Index into power state array for current power state.
Reserved	`0x4`	`0x4`	Must be zero.

Time of last change	`0x8`	`0x8`	Timestamp in microseconds since boot of the last power state transition, including to a running state.
Power state array	`N × 0x18`	`0x10`	Where N is the number of power states. Described in Table 9.

The format for each entry in the power state array is described in Table 9.

Table 9 Power state entry

Field	Byte Length	Byte Offset	Description
power_state	`0x4`	`0x0`	Identifier for the power state.
Reserved	`0x4`	`0x4`	Must be zero.
Usage count	`0x8`	`0x8`	Number of times this domain has entered the power state. This value must be updated when the domain transitions into the power state.
Residency	`0x8`	`0x10`	Amount of time in microseconds domain has been resident in the power state. This value must be updated when the domain transitions out of the power state.

For 64-bit statistics, races can arise between the platform updating a statistic and the reader accessing it. For example, the platform can use a 32-bit controller to update a statistic and thus require two accesses. On the other hand, the reader can be a 32 or 64-bit processor. Races might arise between the write accesses by the platform and the read accesses by the processor, leading to a stale value being reported. To prevent this problem, the reader must read the statistic twice, and compare the values that were obtained. If the two reads match, the statistic was read successfully, otherwise further reads must be done until the last two reads match.

4.4 System power management protocol

This protocol is intended for system shutdown, suspend and reset.

The system power protocol provides commands to:

- Describe the protocol version.
- Discover implementation attributes.
- Shut down the system.
- Suspend the system.

- Reset the system.
- Request a graceful shutdown or reset.
- Allow an agent to forcibly power down or reset the system.

4.4.1 System power management protocol background

The OSPM must be able to power down or reset the whole system it is running on. ACPI provides S states (S1-S5) for this purpose. In turn, PSCI provides SYSTEM_RESET, SYSTEM_RESET2, SYSTEM_SUSPEND and SYSTEM_OFF. On some systems, other agents might be required to initiate a system power down or reset. This protocol is designed to allow more than one agent to request these types of system power transitions. It is envisaged that, in the common case, there are up to three agents:

- On application processors, a PSCI implementation. The PSCI implementation fulfills OSPM calls to SYSTEM_OFF, SYSTEM_SUSPEND, SYSTEM_RESET and SYSTEM_RESET2 functions. In order to do so, the PSCI implementation uses the SCMI protocol to request system power down or reset transitions.
- Particularly in enterprise systems, there might be a management agent that can request a shutdown or a reset, either gracefully, through cooperation with the OSPM, or forcibly.
- The OSPM, which might receive notifications for a graceful shutdown request.

An agent can request the system to forcibly shut down or reset. The platform responds by performing the action that is requested. An agent can also request a graceful shutdown or reset. In this case, the platform will send a notification to the OSPM that will, in turn, initiate the requested action. To this end, the protocol allows an agent to request notifications of system power state transition requests generated by other agents. Table 10 describes the expected behavior for the various operations that are provided by this interface, depending on the calling agent.

Table 10 System power management operations, and expected responses depending on type of agent

Operation	Type of agent	Response
Request a power state transition	OSPM	Deny request as NOT_SUPPORTED, as the agent is not secure.
	PSCI implementation on application processor	Shutdown or reset as requested.
	Management agent	Shutdown or reset as requested.
Request a graceful power state transition	OSPM or PSCI implementation	Deny the request as NOT_SUPPORTED.
	Management agent	Allow the request and notify the OSPM agent.
Request for notification of power state transition requests	OSPM	Allow, as this agent will initiates a shutdown or reset in response to the notification.

	Management agent	Allow, to enable the management to confirm that the OSPM has requested shutdown or reset (through PSCI).
	PSCI	Deny. NOT_SUPPORTED, because it is not required to handle notifications.

Notifications of system power state transitions are not propagated to the agent that requests the transition.

The protocol supports four kinds of system transitions:

- System powerup or shutdown.
- System suspend, as defined in PSCI for SYSTEM_SUSPEND, this is effectively a suspend to RAM scenario analogous to S3 in ACPI.
- Architectural system resets, which are resets that are defined by this specification. These resets include system cold reset and system warm reset.
- Vendor defined transitions.

A system cold reset is equivalent to power cycling the system. All components in the system are powered down or held in reset. Components that are involved in the system boot are powered up or released from reset. In this context, the term cold boot refers to the expected boot flow after the first application of power to the system.

A system warm reset is one that preserves all memory that is visible to application processors. Similar to cold reset, all components in the system, except those involved in the provision of system memory to application processors, are powered down or held in reset. This definition of system memory does not extend to caches or to memory mapped I/O. As in the cold reset case, only those components that are involved in a system boot are powered up or released from reset.

The view of the system that is affected by a system power state transition depends on the target segment and type of device being implemented. In some implementations, the system that is being powered down includes all the agents that can use this interface, as well as the platform controller that implements it. In this case we say that this protocol has a full system view. However, for some platform implementations, the platform controller that implements this SCMI protocol might be in a dedicated always-on domain, such that it is not included in the system power transitions. In this case, we say that this protocol has an OSPM system view, and the system power state transitions only affect those parts of the system that the OSPM controls. In this latter kind of system, if an agent requests a system shutdown, the platform controller remains powered, so that it can service further commands, for example, a command to power up the system. Table 11 describes the expected behavior for the various operations that are provided by this interface, depending on the calling agent and the view of the system implemented.

Table 11 System power management operations, and expected responses depending on view

System view	Operation (all are forcible)	Calling Agent	Expected behavior

OSPM	Shutdown or system suspend	PSCI on behalf of the OSPM	Calling agent view of the system is shut down or suspended. System power state notifications to other agents are sent at the point at which it is possible to request a system power up.
		Management agent	Message returns at the point at which it is possible to request a system power up.
	Reset	PSCI on behalf of the OSPM	Calling agent view of the system is reset. System power state notifications to other agents are sent when it is possible to request forcible system shutdown or reset.
		Management agent.	OSPM view is reset. The message returns when it is possible to request forcible system shutdown or reset.
	Power-up	PSCI on behalf of the OSPM	Not supported.
		Management agent.	OSPM view is powered up. The message returns at the point at which forcible system power state requests are possible.
	Get system power state	PSCI on behalf of the OSPM	Not supported.
		Management agent	Message returns system power state of OSPM view.
Full	Shutdown or suspend	PSCI on behalf of the OSPM	Whole system is shut down or suspended. System power state notifications to other agents are sent at the point at which PSCI makes its request.
		Management agent	Whole system is shut down or suspended. Notifications in this case are not required.
	Reset	PSCI on behalf of the OSPM	System is Reset. System power state notifications to other agents are sent at the point at which PSCI makes its request.
		Management agent	System is Reset. Notifications in this

		case are not required.
Power up or get system state	PSCI on behalf of the OSPM	Not supported.
	Management agent	Not supported.

In both full and OSPM view implementations, the behavior towards a PSCI or an OSPM agent remains unchanged. The change in behavior is only visible to an external agent, such as a management agent. Commands to power up or get system state are only present in systems that implement the OSPM view.

4.4.2 Commands

4.4.2.1 PROTOCOL_VERSION

On success, this command returns the version of this protocol. For this version of the specification, the value returned must be `0x10000`, which corresponds to version 1.0.

message_id: `0x0`
protocol_id: `0x12`
This command is mandatory.

Return values

Name	**Description**
int32 status	See section 4.1.4 for status code definitions.
unt32 version	For this revision of the specification, this must be `0x10000`.

4.4.2.2 PROTOCOL_ATTRIBUTES

This command returns the implementation details associated with this protocol.

message_id: `0x1`
protocol_id: `0x12`
This command is mandatory.

Return values

Name	**Description**	
int32 status	See section 4.1.4 for status code definitions.	
attributes	Bits[31:0]	Reserved, must be zero.

4.4.2.3 PROTOCOL_MESSAGE_ATTRIBUTES

On success, this command returns the implementation details associated with a specific message in this protocol. In addition to the standard status codes described in section 4.1.4, the command can return the following errors:

- NOT_FOUND if the message identified by the message_id is not provided by the implementation.
- If notifications are not supported for the calling agent, this command returns NOT_SUPPORTED for the SYSTEM_POWER_STATE_NOTIFY command, which is described in section 4.4.2.4, and the SYSTEM_POWER_STATE_NOTIFIER notification, which is described in section 4.4.3.1.

message_id: `0x2`

protocol_id: `0x12`

This command is mandatory.

Parameters

Name	**Description**
uint32 message_id	message_id of the message

Return values

Name	**Description**	
int32 status	NOT_FOUND. See section 4.1.4 for status code definitions.	
uint32 attributes	Flags associated with a specific command in the protocol. If message_id is for SYSTEM_POWER_STATE_SET the attributes have the following format:	
	Bit[31]	System warm reset support Set to 1 if system warm reset is supported. Set to 0 if system warm reset is not supported.
	Bit[30]	System suspend support. Set to 1 if system suspend is supported Set to 0 if system suspend is not supported
	Bits[29:0]	Reserved, must be zero. For all values of message_id, this value is zero.

4.4.2.4 SYSTEM_POWER_STATE_SET

This command is used to power down or reset the system.

In addition to the standard status codes described in section 4.1.4, the command can return the following errors:

- INVALID_PARAMETERS if the requested power state is not valid.

- NOT_SUPPORTED if the requested state is not supported for the calling agent.
- DENIED for system suspend requests when there are application processors, other than the caller, in a running or idle state.

System power-up must only be available to agents other than a PSCI implementation on systems that implement OSPM view, as discussed in section 4.4.1.

message_id: `0x3`
protocol_id: `0x12`
This command is mandatory.

Parameters

Name	Description
uint32 flags	This parameter has the following format: Bits[31:1] Reserved, must be zero. Bit[0] Graceful request. This flag is ignored for power up requests. Set to 1 if the request is a graceful request. Set to 0 if the request is a forceful request.
uint32 system_state	Can be one of: `0x0` System shutdown. `0x1` System cold reset. `0x2` System warm reset. `0x3` System power-up. `0x4` System suspend. `0x5` – `0x7FFFFFFF` Reserved, must not be used. `0x80000000` – `0xFFFFFFFF` Might be used for vendor-defined implementations of reset, power-up, or shutdown. These can include additional parameters. The prototype for vendor-defined call is beyond the scope of this specification.

Return values

Name	Description
int32 status	Specific status codes outlined above in 4.4.2.4 for particular cases as described. See section 4.1.4 for other status code definitions.

4.4.2.5 SYSTEM_POWER_STATE_GET

This command must only be available to agents other than a PSCI implementation on systems that implement OSPM view, as discussed in section 4.4.1. The command is to get the power state of the system.

message_id: `0x4`
protocol_id: `0x12`
This command is mandatory in an OSPM view implementation.

Return values

Name	**Description**
int32 status	See section 4.1.4 for status code definitions.
uint32 system_state	Can be one of: `0x0` System shutdown. `0x3` System power-up. `0x4` System suspend. `0x5` – `0x7FFFFFFF` Reserved, must not be used. `0x80000000` – `0xFFFFFFFF` Might be used for vendor defined states.

4.4.2.6 SYSTEM_POWER_STATE_NOTIFY

This command is used to request notification of system power state requests. This command might be used:

- By the OSPM to receive notifications of graceful system power state requests.
- By a management agent to be notified that the OSPM requested a forceful transition.

In addition to the standard status codes that are described in section 4.1.4, the command can return the error NOT_SUPPORTED if notifications are not supported or available to the calling agent.

On initial boot of an agent, these notifications must be disabled by default to that agent.

message_id: `0x5`
protocol_id: `0x12`
This command is mandatory in an OSPM view implementation.

Parameters

Name	**Description**

uint32 notify_enable	Bits[31:1]	Reserved, must be zero.
	Bit[0]	Notify enable: If this value is zero, the platform does not send any SYSTEM_POWER_STATE_NOTIFIER messages to the calling agent. If this value is set to one, the platform does send SYSTEM_POWER_STATE_NOTIFIER messages commands to the calling agent. See section 4.4.3.1 for details about SYSTEM_POWER_STATE_NOTIFIER notifications.

Return values

Name	Description
int32 status	NOT_SUPPORTED INVALID_PARAMETERS See section 4.1.4 for status code definitions.

4.4.3 Notifications

4.4.3.1 SYSTEM_POWER_STATE_NOTIFIER

If an agent has registered for system power state notifications with SYSTEM_POWER_STATE_NOTIFY, the platform sends this notification to the agent. Typically, the agent is either:

- The OSPM that initiates a system power state transition in response to this notification.The OSPM needs this notification to become aware that a remote entity such as the management agent is requesting a graceful power state transition.
- A management agent that initiated a graceful power state transition and is waiting for the OSPM to perform a power state transition in response. The management agent needs this notification to confirm that the platform controller has successfully received the power state transition request from the PSCI.

message_id: `0x0`
protocol_id: `0x12`
This command is optional.

Parameters

Name	Description
uint32 agent_id	Identifier for the agent that caused the system power state transition.

uint32 flags	This parameter has the following format: Bits[31:1] Reserved, must be zero. Bit[0] Graceful request. Set to 1 if the notification indicates that a system power state transition has been gracefully requested. Set to 0 if the notification indicates that a system power state has been forcibly requested.
uint32 system_state	System power state that the system has transitioned to, or which has been requested. Can be one of: `0x0` System shutdown. `0x1` System cold reset. `0x2` System warm reset. `0x3` System power-up. `0x4` System suspend. `0x5` – `0x7FFFFFFF` Reserved, must not be used. `0x80000000` – `0xFFFFFFFF` Might be used for vendor-defined implementations of reset, power-up, or shutdown. These can include additional parameters. The prototype for vendor-defined call is beyond the scope of this specification.

4.5 Performance domain management protocol

This protocol is intended for performance management of groups of devices or APs that run in the same performance domain. Performance domains must not be confused with power domains. A performance domain is defined by a set of devices that always have to run at the same performance level. For a given performance domain, there is a single point of control that affects all the devices in the domain, making it impossible to set the performance level of an individual device in the domain independently from other devices in that domain. For example, a set of CPUs that share a voltage domain, and have a common frequency control, is said to be in the same performance domain. The commands in this protocol provide functionality to:

- Describe the protocol version.
- Describe attribute flags of the protocol.
- Set the performance level of a domain.
- Read the current performance level of a domain.

- Return the list of performance levels supported by a performance domain, and the properties of each performance level.
- Optionally return statistics on residency and usage count of a performance level in performance domains.

4.5.1 Performance domain management protocol background

The command set operates in an abstract integer performance scale. The implementation can choose what this scale represents. For example, in some systems, the values in the scale might represent actual frequencies, while in others they might represent a percentage of the maximum performance of the domain. In all cases, the scale must be linear, meaning that a value of 2X delivers twice the performance as compared to a value of X.

Although this protocol uses an abstract scale to represent performance levels, the underlying implementation only provides a discrete set of performance levels. Each of these levels has an associated power cost. The protocol provides a command to discover these levels and their associated power cost. The power can be expressed in mW or in an abstract scale. Vendors are not obliged to reveal power costs if it is undesirable, but a linear scale is required.

Protocol commands take integer identifiers to describe which performance domain a given command applies to. The identifiers are sequential and start from 0.

In a multi-agent system, a given agent exclusively owns the performance of a set of domains. Agents, other than the platform agents, are not allowed to change the performance of domains they do not own. However, an agent can be allowed to set limits on the performance of a domain it does not own. Agents are also allowed to read performance data, or register for notifications issued on performance changes.

A performance domain can be characterized by three distinct levels that are advertised by the platform. These distinct levels are described in Table 12.

Table 12 Performance Domain Levels with Special Significance

Performance Level	Description
Highest Performance	This is the theoretical maximum performance level of the domain.
Sustained Performance	This is the maximum performance level that the platform can sustain under normal conditions. In exceptional circumstances, such as thermal runaway, the platform may not be be able to guarantee this level.
Lowest Performance	This is the lowest performance level supported by the domain.

4.5.2 Commands

4.5.2.1 PROTOCOL_VERSION

On success, this command returns the version of this protocol. For this version of the specification, the value returned must be `0x10000`, which corresponds to version 1.0.

message_id: `0x0` protocol_id: `0x13` This command is mandatory.

Return values

Name	Description
int32 status	See section 4.1.4 for status code definitions.
uint32 version	For this revision of the specification, this must be `0x10000`.

4.5.2.2 PROTOCOL_ATTRIBUTES

This command returns the attribute flags associated with this protocol.

message_id: `0x1` protocol_id: `0x13` This command is mandatory.

Return values

Name	Description	
int32 status	See section 4.1.4 for status code definitions.	
uint32 attributes	Bits[31:17]	Reserved, must be zero.
	Bit[16]	Power values expressed in mW: Set to 1 if the value described for a power consumption of performance level is expressed in mW. Set to 0 if the value described for a power consumption of performance level is expressed in a proprietary scale.
	Bits[15:0]	Number of performance domains.
uint32 statistics_address_low	The lower 32 bits of the physical address where the statistics shared memory region is located. The address must be in the memory map of the calling agent. If the statistics_len field is 0, then this field is invalid and must be ignored.	
uint32 statistics_address_high	The upper 32 bit of the physical address where the shared memory region is located. The address must be in the memory map of the calling agent. If the statistics_len field is 0, then this field is invalid and must be ignored.	

uint32 statistics_len	The length in bytes of the shared memory region. A value of 0 in this field indicates that the platform doesn't support the statistics shared memory region.

The statistics shared memory region is described in section 4.5.4.

4.5.2.3 PROTOCOL_MESSAGE_ATTRIBUTES

On success, this command returns the implementation details associated with a specific message in this protocol. In addition to the standard status codes described in section 4.1.4, the command can return the error NOT_FOUND if the message identified by the message_id is not provided by the implementation.

This command can be used to enquire if performance level or limit change notifications are supported by the platform. This is achieved by passing message identifiers for the PERFORMANCE_NOTIFY_LEVEL or PERFORMANCE_NOTIFY_LIMITS messages to the call. The platform then returns a status code of NOT_FOUND to indicate that notifications are not implemented, or that they are not available to the calling agent. The notification commands are described in sections 4.5.2.11 and 4.5.2.10.

message_id: `0x2`
protocol_id: `0x13`
This command is mandatory.

Parameters

Name	**Description**
uint32 message_id	message_id of the message.

Return values

Name	**Description**
int32 status	NOT_FOUND. See section 4.1.4 for status code definitions.
uint32 attributes	Flags associated with a specific command in the protocol. For all commands in this protocol, this parameter has a value of 0.

4.5.2.4 PERFORMANCE_DOMAIN_ATTRIBUTES

This command returns attributes that are specific to a given domain. In addition to the standard status codes described in section 4.1.4, the command can return the error NOT_FOUND if the performance domain identified by domain_id does not exist.

message_id: `0x3`

protocol_id: `0x13`

This command is mandatory.

Parameters

Name	**Description**
uint32 domain_id	Identifier for the performance domain.

Return values

Name	**Description**
int32 status	NOT_FOUND See section 4.1.4 for status code definitions.

uint32 attributes	Bit[31]	Can set limits. Set to 1 if calling agent is allowed to set the performance limits on the domain. Set to 0 if a calling agent is not allowed to set limits on the performance limits on the domain.
	Bit[30]	Can set performance level. Set to 1 if calling agent is allowed to set the performance of a domain. Set to 0 if a calling agent is not allowed to set the performance of a domain. Only one agent can set the performance of a given domain.
	Bit[29]	Performance limits change notifications support. Set to 1 if performance limits change notifications are supported for this domain. Set to 0 if performance limits change notifications are not supported for this domain.
	Bit[28]	Performance level change notifications support. Set to 1 if performance level change notifications are supported for this domain. Set to 0 if performance level change notifications are not supported for this domain.
	Bits[27:0]	Reserved and set to zero.
uint32 rate_limit	Bits[31:20]	Reserved and set to zero.
	Bit[19:0]	Rate Limit in microseconds. A value of zero indicates that this field is not supported by the platform.
uint32 sustained_freq	Frequency base corresponding to the sustained performance level, to be used for informational purposes only. Expressed in units of kHz.	
uint32 sustained_perf_level	The performance level value that corresponds to the sustained performance delivered by the platform.	
uint8 name[16]	Null terminated ASCII string of up to 16 bytes in length describing a domain name.	

4.5.2.5 PERFORMANCE_DESCRIBE_LEVELS

This command allows the agent to ascertain the discrete performance levels that are supported by the platform, and their respective power costs. On success, the command returns an array that consists of several performance level entries, each of which describes an expected performance and power cost. The power cost can be expressed in milliwatts or in an abstract scale. How the numbers in that scale convert to the actual wattage is IMPLEMENTATION DEFINED, but the conversion must be linear, meaning that a power of 2X is twice the power of X. The size of the array, which is also returned, depends on the number of return values that a given transport can support. Therefore, it might not be possible to return information for all performance levels with just one call. To solve this problem, the interface allows multiple calls.

In addition to the standard status codes described in section 4.1.4, the command can return the error NOT_FOUND if the performance domain identified by domain_id does not exist.

message_id: `0x4`
protocol_id: `0x13`
This command is mandatory.

Parameters

Name	**Description**
uint32 domain_id	Identifier for the performance domain.
uint32 level_index	Index to the first level to be described in the return level array.

Return values

Name	**Description**	
int32 status	NOT_FOUND See section 4.1.4 for status code definitions.	
uint32 num_levels	Bits[31:16]	Number of remaining performance levels.
	Bits[15:12]	Reserved, must be zero.
	Bits[11:0]	Number of performance levels that are returned by this call.

{uint32, uint32, uint32} perf_levels[N]	Array of performance levels to be described. Each array entry is composed of three 32-bit words with the following format: uint32 entry[0] Performance level value. uint32 entry[1] Power cost. uint32 entry[2] Attributes Bits[31:16] Reserved, must be zero. Bit[15:0] Worst-case transition latency in microseconds to move from any supported performance to the level indicated by this entry in the array.

The following pseudocode describes how the command can be used to discover information about every supported performance level for the performance domain:

```
uint16 level_index = 0;
int32 status = 0;
struct number_of_perf_levels {
            uint perf_levels_array_len:12;
            uint reserved: 4;
            uint remaining:16;
} num_levels = {0,0,0};

struct perf_level_data {
            uint32 power;
            uint32 perf_value;
            uint16 reserved;
            uint16 transition_latency;
};

struct perf_level_data perf_levels[];

do {
            invoke_PERFORMANCE_DESCRIBE_LEVELS(
                        domain_id, level_index,
                        &status, &num_levels, perf_levels);

            if (status)
                                    goto clean_up_and_return;
    num_levels.perf_levels_array_len
            add_levels_to_database(domain_id, level_index,  // process
                        num_levels.perf_levels_array_len,
                        perf_levels);

            level_index += num_levels.perf_levels_array_len;
}          while(num_levels.remaining);
```

4.5.2.6 PERFORMANCE_LIMITS_SET

This command allows the caller to set limits on the performance level of a domain. In addition to the standard status codes described in section 4.1.4, the command can return the following statuses:

- SUCCESS if the command successfully set the limits of operation. If setting a limit requires modifying the current performance level of the domain, the command can return before this change has been completed. However, the change in performance level must still take place.
- NOT_FOUND if the performance domain identified by domain_id does not exist.
- OUT_OF_RANGE if the limits set lie outside the highest and lowest performance levels that are described by PERFORMANCE_DESCRIBED_LEVELS.
- DENIED if the calling agent is not permitted to change the performance limits for the domain, as described by PERFORMANCE_DOMAIN_ATTRIBUTES.

message_id: `0x5`
protocol_id: `0x13`
This command is mandatory.

Parameters

Name	**Description**
uint32 domain_id	Identifier for the performance domain.
uint32 range_max	Maximum allowed performance level.
uint32 range_min	Minimum allowed performance level.

Return values

Name	**Description**
int32 status	NOT_FOUND OUT_OF_RANGE DENIED See section 4.1.4 for status code definitions.

4.5.2.7 PERFORMANCE_LIMITS_GET

This command allows the agent to ascertain the range of allowed performance levels. The returned value reflects the currently set limits for the performance domain. These limits might have been set implicitly by the platform, or by a call to PERFORMANCE_LIMIT_SET. Performance requests outside the range result in OUT_OF_RANGE errors. In addition to the standard status codes described in section 4.1.4, the command can return the error NOT_FOUND if the performance domain identified by domain_id does not exist.

On success, the range return value provides the minimum and maximum allowed performance level.

message_id: `0x6`
protocol_id: `0x13`
This command is mandatory.

Parameters

Name	**Description**
uint32 domain_id	Identifier for the performance domain.

Return values

Name	**Description**
int32 status	NOT_FOUND See section 4.1.4 for status code definitions.
uint32 range_max	Maximum allowed performance level.
uint32 range_min	Minimum allowed performance level.

4.5.2.8 PERFORMANCE_LEVEL_SET

This command allows the agent to set the performance level of a domain. This command can return before the domain has transitioned to the required performance level. The platform simply has to acknowledge that it has received the command. The agent can register for performance level notifications to ascertain whether a performance transition has taken place. For further details, see section 4.5.3.2.

In addition to the standard status codes describes in section 4.1.4, the command can return the following errors:

- NOT_FOUND if the performance domain identified by domain_id does not exist.
- OUT_RANGE if the requested performance level is outside the currently allowed range.
- DENIED if the calling agent is not permitted to change the performance level for a domain, as described by PERFORMANCE_DOMAIN_ATTRIBUTES.

message_id: `0x7`
protocol_id: `0x13`
This command is mandatory.

Parameters

Name	**Description**
uint32 domain_id	Identifier for the performance domain.
uint32 performance_level	Requested performance level.

Return values

Name	Description
int32 status	SUCCESS NOT_FOUND OUT_OF_RANGE DENIED See section 4.1.4 for status code definitions.

4.5.2.9 PERFORMANCE_LEVEL_GET

On success, this command returns the current performance level of a domain. Note the performance level value that is returned by this command might be stale by the time the command completes, as a subsequent performance change might have been initiated in the meantime.

In addition to the standard status codes described in section 4.1.4, the command can return the following errors:

NOT_FOUND if the performance domain identified by domain_id does not exist.

message_id: `0x8`
protocol_id: `0x13`
This command is mandatory.

Parameters

Name	Description
uint32 domain_id	Identifier for the performance domain.

Return values

Name	Description
int32 status	NOT_FOUND See section 4.1.4 for status code definitions.
uint32 performance_level	Current performance level of the domain.

4.5.2.10 PERFORMANCE_NOTIFY_LIMITS

This command allows the agent to request notifications from the platform for changes in the allowed maximum and minimum performance levels. These notifications are sent using the PERFORMANCE_LIMITS_CHANGED command which is described in section 4.5.3.1. In addition to the standard status codes described in section 4.1.4, the command can return the following errors:

- NOT_FOUND if the performance domain identified by domain_id does not exist.
- DENIED if notifications are not supported for the indicated performance domain.

If no domains support limit notifications, the command can be omitted. PROTOCOL_MESSAGE_ATTRIBUTES, which is described in section 4.5.2.4, can be used to determine whether this command is implemented.

On initial boot of an agent, by default, these notifications must be disabled from being sent to that agent.

message_id: `0x9`
protocol_id: `0x13`
This command is optional.

Parameters

Name	**Description**	
uint32 domain_id	Identifier for the performance domain	
uint32 notify_enable	Bits[31:1]	Reserved, must be zero.
	Bit[0]	Notify enable: If this value is zero, the platform does not send any PERFORMANCE_LIMITS_CHANGED messages to the agent. If this value is set to one, the platform does send PERFORMANCE_LIMITS_CHANGED messages to the agent. See section 4.5.3.1 for more details about PERFORMANCE_LIMITS_CHANGED notifications.

Return values

Name	**Description**
int32 status	DENIED NOT_FOUND INVALID_PARAMETERS See section 4.1.4 for status code definitions.

4.5.2.11 PERFORMANCE_NOTIFY_LEVEL

This command allows the agent to request notifications from the platform when the performance level for a domain changes in value. These notifications are sent using the PERFORMANCE_LEVEL_CHANGED command which is described in section 4.5.3.2. In addition to the standard status codes described in section 4.1.4, the command can return the following errors:

- NOT_FOUND if the performance domain identified by domain_id does not exist.
- NOT_SUPPORTED if notifications are not supported for the indicated performance domain.

If no domains support level change notifications the command can be omitted. PROTOCOL_MESSAGE_ATTRIBUTES, which is described in section 4.5.2.4, can be used to determine whether this command is implemented.

On initial boot of an agent, by default, these notifications must be disabled from being sent to that agent.

message_id: `0xA`
protocol_id: `0x13`
This command is optional.

Parameters

Name	**Description**	
uint32 domain_id	Identifier for the performance domain	
uint32 notify_enable	Bits[31:1]	Reserved, must be zero.
	Bit[0]	Notify enable: If this value is zero, the platform does not send any PERFORMANCE_LEVEL_CHANGED messages to the agent. If this value is set to one, the platform does send PERFORMANCE_LEVEL_CHANGED messages to the agent. See section 4.5.3.2 for more details about the PERFORMANCE_LEVEL_CHANGED notifications.

Return values

Name	**Description**
int32 status	NOT_SUPPORTED NOT_FOUND INVALID_PARAMETERS See section 4.1.4 for status code definitions.

4.5.3 Notifications

4.5.3.1 PEFORMANCE_LIMITS_CHANGED

If an agent has registered for limit change notifications for the domain that is identified by domain_id, the platform sends this notification to the agent when the performance limits for that domain change.

The platform is not required to guarantee sending a notification to an agent for every limits change. In particular, if several changes take place in quick succession, the platform is allowed to only issue a notification for the last change.

message_id: `0x0`

protocol_id: `0x13`

This command is optional.

Parameters

Name	**Description**
uint32 agent_id	Identifier for the agent that caused the performance limit change.
uint32 domain_id	Identifier for the performance domain whose limit was changed.
uint32 range_max	Maximum allowed performance level.
uint32 range_min	Minimum allowed performance level.

4.5.3.2 PERFORMANCE_LEVEL_CHANGED

If an agent has registered to receive performance level change notifications for the domain that is identified by domain_id, the platform sends this notification to the agent when the performance level of that domain changes.

The platform is not required to guarantee sending a notification to an agent for every level change transition. In particular, if several performance changes happen in quick succession, the platform is allowed to only issue a notification for the last transition.

The level values that are passed in notifications might not match the values that are requested by the agent that is notified. The level that is finally selected by the platform might differ from the level that is requested by an agent, due to thermal or power constraints.

message_id: `0x1`

protocol_id: `0x13`

This command is optional.

Parameters

Name	**Description**
uint32 agent_id	Identifier for the agent that caused the performance level change.
uint32 domain_id	Identifier for the performance domain whose level was changed.
uint32 performance_level	Current performance level of the domain.

4.5.4 Performance domain statistics shared memory region

Optionally, the platform can provide a statistics memory region that is associated with the performance domain management protocol. Whether support is present is indicated by the

PROTOCOL_ATTRIBUTES command, which is described in section 4.5.2.2. This command also provides the address and size of the shared memory region. For a given performance domain, and for each performance level in that domain, statistics in the shared memory region track the number of times that the level has been used and the amount of time that the domain has been in that performance level. The statistics must be updated regardless of the agent in the system that placed a domain into a given performance level. After a system reset, suspend, or shutdown, all the statistics must be initialized to zero when the system first starts up. Time measurements are in microseconds.

For APs, the shared memory must be accessible from the Non-secure world, and must be mapped as non-cached normal memory or device memory. The format of the shared memory structure is described in Table 13.

Table 13 Performance level statistics memory region

Field	Byte Length	Byte Offset	Description
Signature	`0x4`	`0x0`	`0x50455246` ('PERF').
Revision	`0x2`	`0x4`	For this revision, this value must be zero.
Attributes	`0x2`	`0x6`	For this revision, this value must be zero.
Number of domains	`0x2`	`0x8`	Number of domains for which statistics are collected.
Reserved	`0x6`	`0xA`	Must be zero.
Performance domain offset array	`0x4 ×` (Number of domains)	`0x10`	For each performance domain, this array provides a 4-byte offset, from the start of the shared memory area, to the memory location of the performance domain entry in the data section. The entry format is described in Table 14. A value of zero for the offset of a given performance domain indicates that statistics are not collected for that domain.
Performance domain data section	--	--	This area must start at an offset of `0x10` + `0x4 ×` (Number of performance domains), or higher.

The performance domain data section contains entries for each power domain. The format for each entry is described in Table 14.

Table 14 Performance domain entry

Field	Byte Length	Byte Offset	Description
Number of performance levels	`0x2`	`0x0`	Number of performance level entries in the performance levels array.

Current performance level	`0x2`	`0x2`	Index into performance level array for current performance level.
Reserved	`0x4`	`0x4`	Must be zero.
Time of last change	`0x8`	`0x8`	Timestamp in microseconds since boot of the last performance level transition.
Performance level array	`N × 0x18`	`0x10`	Performance level array, where N is the number of performance levels. Described in Table 15.

The format for each entry in the performance level array is described in Table 15.

Table 15 Performance level array entry

Field	**Byte Length**	**Byte Offset**	**Description**
performance_level	`0x4`	`0x0`	Performance level.
reserved	`0x4`	`0x4`	Reserved, must be set to zero.
Usage count	`0x8`	`0x8`	Number of times this domain has used this performance level. This value must be updated when the domain transitions into the performance level.
Residency	`0x8`	`0x10`	This value represents the amount of time domain has been running at the performance level, and is given in microseconds. This value must be updated every time the domain transitions to different performance level.

Accessing multi-word statistics can cause races between platform write accesses and the read accesses by agents in the system. This problem and its solution are described in section 4.3.4.

4.6 Clock management protocol

This protocol is intended for management of clocks. It is used to enable or disable clocks, and to set rates. The protocol provides commands to:

- Describe the protocol version.
- Discover implementation attributes.
- Describe a clock.
- Enable or disable a clock.
- Set the rate of the clock synchronously or asynchronously.

4.6.1 Clock management protocol background

This protocol can be used for managing clock rates. It is not to be confused with the performance management protocol, which is used to manage the speed of compute engines such as application processors or GPUs. Examples of usage for the clock protocol might be setting rates for LCD clocks or I^2C buses.

The protocol does not cover discovery of clock dependencies, which must be described through firmware tables instead.

Protocol commands take integer identifiers to describe which clock a given command applies to. The identifiers are sequential and start from 0.

A given clock must be owned and exposed to a single agent.

4.6.2 Commands

4.6.2.1 PROTOCOL_VERSION

On success, this command returns the version of this protocol. For this version of the specification, the return value must be `0x10000`, which corresponds to version 1.0.

message_id: `0x0`
protocol_id: `0x14`
This command is mandatory.

Return values

Name	**Description**
int32 status	See section 4.1.4 for status code definitions.
uint32 version	For this revision of the specification, this value must be `0x10000`.

4.6.2.2 PROTOCOL_ATTRIBUTES

This command returns the implementation details associated with this protocol.

message_id: `0x1`
protocol_id: `0x14`
This command is mandatory.

Return values

Name	**Description**	
int32 status	See section 4.1.4 for status code definitions.	
uint32 attributes	Bits[31:24]	Reserved, must be zero.
	Bits[23:16]	Maximum number of pending asynchronous clock rate changes supported by the platform.
	Bits[15:0]	Number of clocks.

4.6.2.3 PROTOCOL_MESSAGE_ATTRIBUTES

On success, this command returns the implementation details associated with a specific message in this protocol. In addition to the standard status codes described in section 4.1.4, the command can return the error NOT_FOUND if the message identified by the message_id is not provided by the implementation.

message_id: `0x2`

protocol_id: `0x14`

This command is mandatory.

Parameters

Name	**Description**
uint32 message_id	message_id of the message.

Return values

Name	**Description**
int32 status	NOT_FOUND See section 4.1.4 for status code definitions.
uint32 attributes	Flags that are associated with a specific command in the protocol. For all commands in this protocol, this parameter has a value of 0.

4.6.2.4 CLOCK _ATTRIBUTES

This command returns the attributes that are associated with a specific clock. In addition to the standard status codes described in section 4.1.4, the command can return the error NOT_FOUND if the clock identified by clock_id does not exist, or is not owned by the calling agent.

message_id: `0x3`

protocol_id: `0x14`

This command is mandatory.

Parameters

Name	**Description**
uint32 clock_id	Identifier for the clock device.

Return values

Name	**Description**

int32 status	NOT_FOUND See section 4.1.4 for status code definitions.	
uint32 attributes	Bits[31:1]	Reserved, must be zero.
	Bits[0]	Enabled/disabled If set to 1, the clock device is enabled. If set to 0, the clock device is disabled.
uint8 clock_name[16]	A NULL terminated ASCII string with the clock name, of up to 16 bytes.	

4.6.2.5 CLOCK_DESCRIBE_RATES

This command allows the agent to ascertain the valid rates to which the clock can be set to. On success, the command returns an array, which contains a number of rate entries. Clocks can support many rates. In this case, individually describing each rate is too onerous, and the array contains only the highest and the lowest rate, as well as the number of supported rates. In either case, the size of the array is returned, which depends on the number of return values a given transport can support. Therefore, it might not be possible to return the whole array with just one call. To solve this problem, the interface allows multiple calls.

In addition to the standard status codes described in section 4.1.4, the command can return the error NOT_FOUND if the clock identified by clock_id does not exist.

message_id: `0x4`
protocol_id: `0x14`
This command is mandatory.

Parameters

Name	**Description**
uint32 clock_id	Identifier for the clock device.
uint32 rate_index	Index to the first rate value to be described in the return rate array.

Return values

Name	**Description**
int32 status	NOT_FOUND See section 4.1.4 for status code definitions.

uint32 num_rates_flags	Bits[31:16] Number of remaining rates. Bits[15:13] Reserved, must be zero. Bit[12] Return format: If this bit is set to 1, the Rate Array is a triplet that constitutes a segment of the form: ▪ rates[0] is the lowest physical rate that the clock can synthesize in the segment. ▪ rates[1] is the highest physical rate that the clock can synthesize in the segment. ▪ rates[2] is the step size between two successive physical rates that the clock can synthesize within the segment. If this bit is set to 0, each element of the Rate Array represents a discrete physical rate that the clock can synthesize. Bits[11:0]: Number of rates that are returned by this call.
{uint32, uint32} rates [N]	Rate Array: If Bit 12 of the num_rates_flags field is set to 0, each array entry is composed of two 32-bit words and has the following format: Lower word: Lower 32 bits of the physical rate in Hertz. Upper word: Upper 32 bits of the physical rate in Hertz. If Bit 12 of the num_rates_flags field is set to 1, then each entry is a member of a segment {lowest rate, highest rate, step size} as described above.

For an example of using this kind of API, see 4.5.2.5.

4.6.2.6 CLOCK_RATE_SET

This command allows the caller to set the clock rate of a clock synchronously or asynchronously.

In addition to the standard status codes described in section 4.1.4, the command can return the following errors:

- NOT_FOUND if the clock identified by clock_id does not exist.
- INVALID_PARAMETERS if the requested rate is not supported by the clock.

- BUSY if there are too many asynchronous clock rate changes pending. PROTOCOL_ATTRIBUTES provides the maximum number of pending asynchronous clock rate changes supported by the platform.

The command returns when the clock rate has been changed.

message_id: `0x5`

protocol_id: `0x14`

This command is mandatory.

Parameters

Name	**Description**	
uint32 flags	Bits[31:4]	Reserved, must be zero.
	Bit[3:2]	Round up/down: If Bit 3 is set to 1, the platform rounds up/down autonomously to choose a physical rate closest to the requested rate, and Bit 2 is ignored. If Bit 3 is set to 0, then the platform rounds up if Bit 2 is set to 1, and rounds down if Bit 2 is set to 0.
	Bit[1]	Ignore delayed response: If async flag, bit 0, is set to 1 and this bit is set to 1, the platform does not send a CLOCK_RATE_SET delayed response. If async flag, bit 0, is set to 1 and this bit is set to 0, the platform does send a CLOCK_RATE_SET delayed response. If async flag, bit 0, is set to 0, then this bit field is ignored by the platform.
	Bit[0]	Async flag: Set to 1 if clock rate is to be set asynchronously. In this case the call is completed with CLOCK_RATE_SET_COMPLETE message if bit 1 is set to 0. For more details, see section 4.6.3.1. A SUCCESS return code in this case indicates that the platform has successfully queued this command. Set 0 to if the clock rate is to be set synchronously. In this case, the call with return the clock rate setting has been completed.
uint32 clock_id	Identifier for the clock device.	
uint32 rate[2]	Requested clock rate as a 64-bit entity.	

Return values	
Name	**Description**
int32 status	NOT_FOUND INVALID_PARAMETERS BUSY See section 4.1.4 for status code definitions.

4.6.2.7 CLOCK_RATE_GET

This command allows the calling agent to request the current clock rate.

Note

If the clock rate is set asynchronously, the rate value that is returned by this command might be stale by the time the command completes.

In addition to the standard status codes described in section 4.1.4, the command can return the error NOT_FOUND if the clock identified by clock_id does not exist.

message_id: `0x6` protocol_id: `0x14` This command is mandatory.	
Parameters	
Name	**Description**
uint32 clock_id	Identifier for the clock device.
Return values	
Name	**Description**
int32 status	NOT_FOUND See section 4.1.4 for status code definitions.
uint32 rate[2]	Clock rate as a 64-bit entity.

4.6.2.8 CLOCK_CONFIG_SET

This command allows the calling agent to configure a clock device.

In addition to the standard status codes described in section 4.1.4, the command can return the error NOT_FOUND if the clock identified by clock_id does not exist.

message_id: `0x7`
protocol_id: `0x14`
This command is mandatory.

Parameters

Name	**Description**	
uint32 clock_id	Identifier for the clock device.	
uint32 attributes	Bits[31:1]	Reserved, must be zero.
	Bit[0]	Enable/Disable:
		If set to 1, the clock device is enabled.
		If set to 0, the clock device is disabled.

Return values

Name	**Description**
int32 status	NOT_FOUND See section 4.1.4 for status code definitions.

4.6.3 Delayed responses

4.6.3.1 CLOCK_RATE_SET_COMPLETE

If the agent has changed the clock rate asynchronously through CLOCK_RATE_SET, the platform sends this delayed response to the agent when the clock rate changes.

message_id: `0x5`
protocol_id: `0x14`
This command is optional.

Parameters

Name	**Description**
int32 status	SUCCESS if clock rate was set successfully. Vendor-specific error. See section 4.1.4 for status code definitions.
uint32 clock_id	Identifier for the clock device.
uint32 rate[2]	Value of the rate that the clock transitioned to.

4.7 Sensor management protocol

This protocol provides functions to manage platform sensors, and provides the following commands:

- Describe the protocol version.
- Describe the attribute flags of the protocol.
- Discover sensors that are implemented and managed by the platform.
- Read a sensor synchronously or asynchronously as allowed by the platform.
- Obtain and program sensor attributes, if applicable.
- Receive notifications on specific changes to sensor data, for example when a sensor value crosses a threshold.
- Specify a region of shared memory for conveying sensor values, if supported by the platform.

4.7.1 Sensor management protocol background

The protocol supports accessing sensors through one of the following mechanisms:

- Synchronous Access – This method is recommended for sensors whose data is immediately available or is internally cached by the platform, and can be returned immediately to the requesting agent. Examples include platform event counters, or sensor data samples that are stored in internal memory within the platform.
- Asynchronous Access – This method is recommended for sensors whose data is not cached by the platform or for sensors that are slow to read. An example of this could be an on-die thermal sensor.
- Event Notification – The agent can register for receiving notifications on specific sensor values, conditions, or states of interest.
- Shared Memory – In this scheme, the platform periodically updates the sensor value in an area of memory that is shared between agents and the platform.

Agents can discover the access mechanisms that are supported by a particular sensor by examining the attributes that are advertised for the sensor. The platform can support multiple access mechanisms.

4.7.2 Commands from Agents to Platform

4.7.2.1 PROTOCOL_VERSION

On success, this command returns the version of this protocol. For this version of the specification, the return value must be `0x10000`, which corresponds to version 1.0.

message_id: `0x0`
protocol_id: `0x15`

This command is mandatory.

Return values

Name	**Description**
int32 status	See section 4.1.4 for status code definitions.

uint32 version	For this revision of the specification, this value must be `0x10000`.

4.7.2.2 PROTOCOL_ATTRIBUTES

This command returns the implementation details associated with this protocol.

message_id: `0x1`
protocol_id: `0x15`

This command is mandatory.

Return values

Name	**Description**	
int32 status	See section 4.1.4 for status code definitions.	
uint32 attributes	Bits[31:24]	Reserved, must be zero.
	Bits[23:16]	Maximum number of outstanding asynchronous commands that is supported by the platform.
	Bits[15:0]	Number of sensors that is present and managed by the platform.
uint32 sensor_reg_address_low	This value indicates the lower 32 bits of the physical address where the sensor shared memory region is located. The address must be in the memory map of the calling agent. If the sensor_reg_len field is 0, then this field is invalid and must be ignored by the agent.	
uint32 sensor_reg_address_high	This value indicates the upper 32 bits of the physical address where the shared memory region is located. The address must be in the memory map of the calling agent. If the sensor_reg_len field is 0, then this field is invalid and must be ignored by the agent.	
uint32 sensor_reg_len	This value indicates the length in bytes of the shared memory region. A value of 0 in this field indicates that the platform does not implement the sensor shared memory.	

The sensor shared memory region is described in section 4.7.5.

4.7.2.3 PROTOCOL_MESSAGE_ATTRIBUTES

On success, this command returns the implementation details associated with a specific message in this protocol.

If the message is not supported or implemented by the platform, then this command returns a NOT_FOUND status code. This allows calling agents to comprehend which commands are supported on a particular platform, and configure themselves accordingly.

message_id: `0x2`
protocol_id: `0x15`

This command is mandatory.

Parameters

Name	Description
uint32 message_id	message_id of the message.

Return values

Name	Description
int32 status	NOT_FOUND, if the command associated with message_id is not implemented or supported by the platform. Other status codes according to section 4.1.4 might be returned for general error or status reporting.
uint32 attributes	Attributes that are associated with the message that is specified by message_id. Currently, this field returns the value of `0`.

4.7.2.4 SENSOR_ DESCRIPTION_GET

This command can be used for sensor discovery on the platform. On success, it returns an array of Sensor Descriptors as described in 4.7.2.4.1.

message_id: `0x3`
protocol_id: `0x15`

This command is mandatory.

Parameters

Name	Description
uint32 desc_index	Index of the first sensor descriptor to be read in the sensor descriptor array.

Return values

Name	Description
int32 status	See section 4.1.4 for status code definitions.

uint32 num_sensor_flags	Bits[31:16]	Number of remaining sensor descriptors.
	Bits[15:12]	Reserved, must be zero.
	Bits[11:0]	Number of sensor descriptors that are returned by this current call.
SENSOR_DESC desc[N]	An array of sensor descriptors, of format described in 4.7.2.4.1.	

4.7.2.4.1 Sensor Descriptor

The SENSOR_DESC structure describes the sensor properties, such as the unique identifier for the sensor, its name, reading types and other characteristics.

uint32 sensor_id	Identifier for the sensor.	
uint32 sensor_attributes_low	Bits[31]	Asynchronous sensor read support. If this flag is set to 1, then this sensor can be read asynchronously through the SENSOR_READING_GET command, and its value is returned in the SENSOR_READING_COMPLETE delayed response. If this flag is set to 0, the sensor must be only be read using a synchronous call to SENSOR_READING_GET command.
	Bits[30:8]	Reserved for future use.
	Bits[7:0]	Number of trip points supported.

uint32	sensor_attributes_high	Bits[31:22]	sensor_update_interval: Bits[31:27] sec – Seconds Bits[26:22] mult – two's complement format representing the power-of-10 multiplier that is applied to the Seconds field. The time duration between successive updates to the sensor value. The representation is in the [sec] x $10^{[mult]}$ format, in units of seconds. This field is set to 0 if the sensor doesn't require a minimum update interval.
		Bits[21:16]	Reserved
		Bits[15:11]	The power-of-10 multiplier in two's-complement format that is applied to the sensor unit specified by the SensorType field.
		Bits[10:8]	Reserved
		Bits[7:0]	SensorType: The type of sensor and the measurement system it implements, as described in Table 16.
uint8	sensor_name[16]		A NULL terminated ASCII string with the sensor name, of up to 16 bytes.

Table 16 Sensor Type Enumerations[1]:

Enum	Sensor Unit Description	Enum	Sensor Unit Description	Enum	Sensor Unit Description
0	None	**30**	Cubic Feet	**60**	Bits
1	Unspecified	**31**	Meters	**61**	Bytes
2	Degrees C	**32**	Cubic Centimeters	**62**	Words (data)
3	Degrees F	**33**	Cubic Meters	**63**	Doublewords
4	Degrees K	**34**	Liters	**64**	Quadwords
5	Volts	**35**	Fluid Ounces	**65**	Percentage
6	Amps	**36**	Radians	**66**	Pascals
7	Watts	**37**	Steradians	**67**	Counts
8	Joules	**38**	Revolutions	**68**	Grams
9	Coulombs	**39**	Cycles	**69**	Newton-meters

10	VA	**40**	Gravities	**70**	Hits
11	Nits	**41**	Ounces	**71**	Misses
12	Lumens	**42**	Pounds	**72**	Retries
13	Lux	**43**	Foot-Pounds	**73**	Overruns/Overflows
14	Candelas	**44**	Ounce-Inches	**74**	Underruns
15	kPa	**45**	Gauss	**75**	Collisions
16	PSI	**46**	Gilberts	**76**	Packets
17	Newtons	**47**	Henries	**77**	Messages
18	CFM	**48**	Farads	**78**	Characters
19	RPM	**49**	Ohms	**79**	Errors
20	Hertz	**50**	Siemens	**80**	Corrected Errors
21	Seconds	**51**	Moles	**81**	Uncorrectable Errors
22	Minutes	**52**	Becquerels	**82**	Square Mils
23	Hours	**53**	PPM (parts/million)	**83**	Square Inches
24	Days	**54**	Decibels	**84**	Square Feet
25	Weeks	**55**	DbA	**85**	Square Centimeters
26	Mils	**56**	DbC	**86**	Square Meters
27	Inches	**57**	Grays	**-**	All others – reserved
28	Feet	**58**	Sieverts		
29	Cubic Inches	**59**	Color Temperature Degrees K	**255**	OEM Unit

1: This table is based on the Distributed Management Task Force (DMTF) specification number DSP 0249 (Platform Level Data Model specification).

4.7.2.5 SENSOR_TRIP_POINT_NOTIFY

This command is used by the agent to globally control generation of notifications on cross-over events for the trip-points that have been configured using the SENSOR_TRIP_POINT_CONFIG command.

message_id: `0x4`
protocol_id: `0x15`

This command is optional.

Parameters

Name	Description	
uint32 sensor_id	Identifier for the sensor.	
uint32 sensor_event_control	Bits[31:1]	Reserved.
	Bit[0]	Globally controls generation of notifications on crossing of configured trip-points pertaining to the specified sensor. If this bit is set to 1, notifications are sent whenever the sensor value crosses any of the trip-points that have been configured using the SENSOR_TRIP_POINT_CONFIG command. If this bit is set to 0, no notifications are sent for any of the trip-points.

Return values

Name	Description
int32 status	INVALID_PARAMETERS See section 4.1.4 for status code definitions.

4.7.2.6 SENSOR_TRIP_POINT_CONFIG

This command is used for selecting and configuring a trip-point of interest. Following the successful completion of this command, the platform generates the SENSOR_TRIP_POINT_EVENT event whenever the sensor value crosses the programmed trip point value, provided notifications have been enabled for trip-points globally using the SENSOR_TRIP_POINT_NOTIFY command.

An agent can use this command for various use-cases. For example:

- The agent can invoke this command twice to program the upper and lower values of a hysteresis band, respectively.
- For a counter-type sensor that is required to fire a notification on reaching a certain count, the agent can issue this command to program the count value.

message_id: `0x5`
protocol_id: `0x15`

This command is mandatory if at least one of the implemented sensors in the platform supports trip points.

Parameters

Name	Description
uint32 sensor_id	Identifier for the sensor.

uint32 trip_point_ev_ctrl	Bits[31:12]	Reserved.
	Bits[11:4]	trip_point_id: Identifier for the selected trip point. This value should be equal to or less than the total number of trip points that are supported by this sensor as advertised in its descriptor.
	Bits[3:2]	Reserved for future use.
	Bits[1:0]	Event control for the trip-point: If set to 0, disables event generation for this trip-point (this is the default state) If set to 1, enables event generation when this trip-point value is reached or crossed in a positive direction If set to 2, enables event generation when this trip-point value is reached or crossed in a negative direction If set to 3, enables event generation when this trip-point value is reached or crossed in either direction.
uint32 trip_point_val_low	Lower 32 bits of the sensor value corresponding to this trip-point. The default value is 0.	
uint32 trip_point_val_high	Higher 32 bits of the sensor value corresponding to this trip-point. The default value is 0.	

Return values

Name	**Description**
int32 status	See section 4.1.4 for status code definitions.

4.7.2.7 SENSOR_READING_GET

This command requests the platform to provide the current value of the sensor that is represented by sensor_id. For synchronous mode of access, the platform provides the sensor reading in the response to this command itself. For asynchronous accesses, the platform returns the sensor value in the SENSOR_READING_COMPLETE delayed response.

When the platform notices failure or fault conditions in the sensor or its associated logic or circuitry, it returns the HARDWARE_ERROR status. Other errors pertain to the interface itself, and are enumerated in 4.1.4.

Agents should assess the sensor attributes to determine the optimal mode of access for the sensor. A slow sensor like a temperature sensor can be more optimally read asynchronously, while a shared memory-based sensor can be read synchronously.

message_id: `0x6`
protocol_id: `0x15`

This command is mandatory.

Parameters

Name	**Description**	
uint32 sensor_id	The identifier for the sensor to be read	
uint32 flags	Bits[31:1]	Reserved
	Bit[0]	Async flag: Set to 1 if the sensor is to be read asynchronously. Set to 0 to if the sensor is to be read synchronously.

Return values

Name	**Description**
int32 status	See section 4.1.4 for status code definitions. If this is an asynchronous call, then the returned status code pertains to this command itself, and any error that occurs during the actual sensor read operation is reported subsequently with the SENSOR_READING_COMPLETE delayed response.
uint32 sensor_value_low	Lower 32 bits of the sensor value. This value is invalid if an error status is returned.
uint32 sensor_value_high	Higher 32 bits of the sensor value. This value is invalid if an error status is returned.

4.7.3 Delayed Responses from Platform to Agent

4.7.3.1 SENSOR_READING_COMPLETE

This response is the delayed response to an asynchronous SENSOR_READING_GET command issued by an agent. When the platform determines that there are certain failure conditions in the sensor itself, such as a fault in the sensor hardware or related circuitry or logic, it returns HARDWARE_ERROR to report that condition to the caller. Other errors apply to the interface itself, and are enumerated in 4.1.4.

message_id: `0x6`
protocol_id: `0x15`

This response is mandatory and is generated if the caller used the asynchronous method to read the sensor.

Return Values

Name	**Description**
int32 status	An appropriate status code, as described in section 4.1.4.
uint32 sensor_id	Identifier for the sensor.
uint32 sensor_value_low	Value that is read from the sensor. Lower 32 bits of the sensor value. This value is invalid if an error status is returned.
uint32 sensor_value_high	Value that is read from the sensor. Lower 32 bits of the sensor value. This value is invalid if an error status is returned.

4.7.4 Notifications

4.7.4.1 SENSOR_TRIP_POINT_EVENT

This notification is issued by the platform when a sensor crosses a specific trip point that the agent had requested event notification for, by using the SENSOR_TRIP_POINT_CONFIG command.

The platform might read sensors periodically using polling, or program sensors to generate interrupts on trip points, depending on implementation. If the sensor value changes such that it crosses several trip-points between successive reads by the platform, then the platform might minimally send only one notification to the agent to represent the multiple cross-over condition.

Message_id: `0x0`
protocol_id: `0x15`

This notification is optional.

Return Values

Name	**Description**
uint32 agent_id	Refers to the agent that caused this event. For the current version of the specification, this field is set to 0 to indicate that the platform is the generator of all sensor events.
Uint32 sensor_id	Identifier for the sensor that has tripped

uint32	trip_point_desc	Bits[31:17]	Reserved.
		Bit[16]	Direction. If set to 1, indicates that the trip point was reached or crossed in the positive direction. If set to 0, indicates that the trip point was reached or crossed in the negative direction.
		Bits[15:8]	Reserved for future use.
		Bits[7:0]	trip_point_id The identifier for the trip point that was crossed or reached.

4.7.5 Sensor Values Shared Memory

Optionally, the platform might provide sensor values through the shared memory region that is associated with the sensor management protocol. Whether support is present is indicated by the PROTOCOL_ATTRIBUTES command, which is described in section 4.7.2.2. This command also provides the address and the size of the shared memory region. The memory must be accessible from the Non-secure world, and OSPM must map it as non-cached normal memory or device memory.

The format of the frame is described in Table 17.

Table 17 Sensor shared memory region

Field	Byte Length	Byte Offset	Description
Signature	`0x4`	`0x0`	`0x53454E53` ('SENS').
Revision	`0x2`	`0x4`	For this revision, this value must be zero.
Attributes	`0x2`	`0x6`	For this revision, this value must be zero.
Number of sensors	`0x2`	`0x8`	Number of sensors.
Reserved	`0x6`	`0xA`	Must be zero.
Sensor domain offset array	`0x4 ×` Number of sensors	`0x10`	For each sensor, this array provides a 4-byte offset, from the start of the shared memory area, to the memory location where the sensor value is stored. A value of 0 indicates that the sensor value is not reported through shared memory. The array is indexed by sensor_id.
Sensor values data	--	`0x10 + 0x4` × (Number	Each sensor value is stored on a 64-bit aligned boundary, with a number that might

section	of sensors)	be up to 64 bits.

Accessing multi-word values might cause races between platform write accesses and the read accesses by agents in the system. This problem and its solution are described in section 4.3.4.

5 Transports

Transports describe how messages are exchanged between agents and the platform.

5.1 Mailbox transport

This form of transport relies on the use of shared memory between the platform and the agents.

The transport optionally supports interrupt based communication, where, on completion of the processing of a message, the caller receives an interrupt. Polling for completion is also supported.

The transport can be used to provide an agent to platform, or a platform to agent channel. Each channel in the transport includes:

- **Mailbox memory area**
  This is an area of memory that is shared between the caller and the callee. At any point in time, the shared memory is owned by the caller or the callee. The ownership is reflected by a **channel status** word in the mailbox memory area. The channel is said to be free when the memory area is owned by caller, and busy when it passed to callee. When a channel is free, the caller can write a message and associated payload to this shared memory area. After this, the caller updates the status, and relinquishes ownership of the shared memory by marking the channel as busy. The callee can use the shared memory to pass return values that are associated with the processing of the message. When the callee has completed processing the message, it updates the status to indicate that the channel is now free. The layout of the memory area is described in section 5.1.2.
- **Doorbell**
  A mechanism that the caller can use to alert the callee of the presence of a message. Typically, this mechanism is implemented as a register in caller, which, when written, raises an interrupt on the callee.
- **Completion interrupt**
  This transport supports polling or interrupt driven modes of communication. In interrupt mode, when the callee completes processing a message, it raises an interrupt on the caller. Hardware support for completion interrupts is optional.

5.1.1 Message communications flow

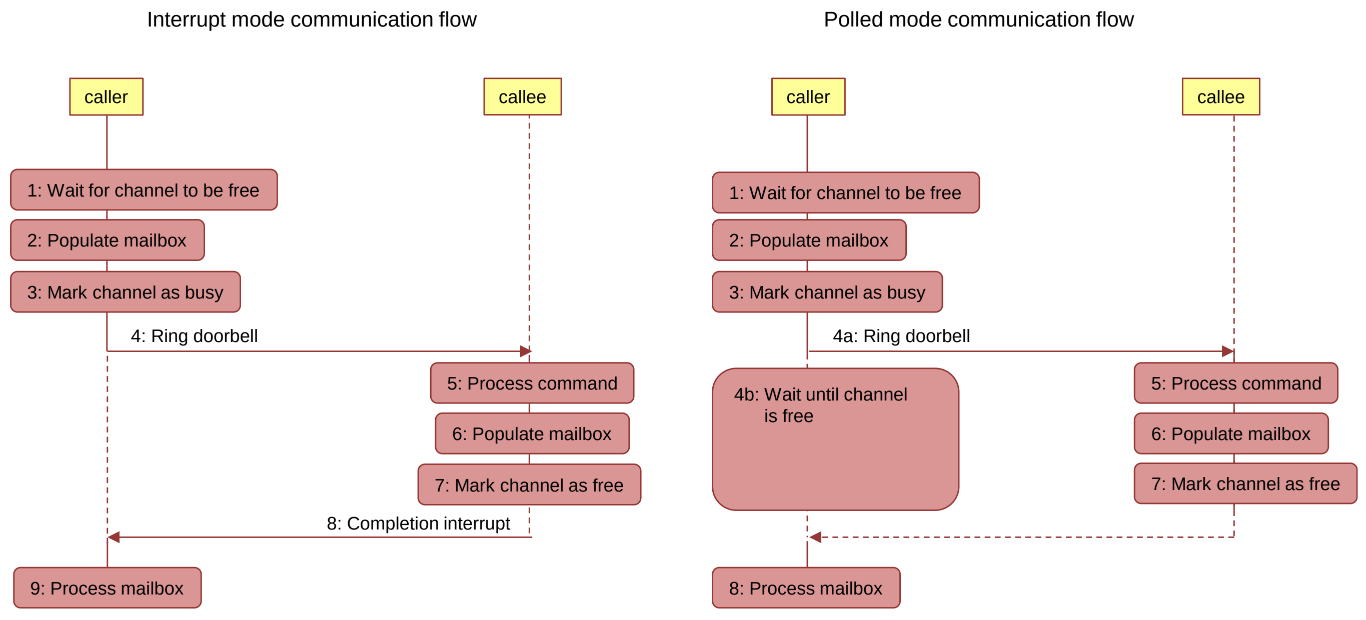

Figure 3 Communications flow

A flow chart for sending a message from the caller to the callee using interrupt mode is shown on the left of Figure 3.

The steps are as follows:

1. The caller must ensure that the channel is free.
2. The caller populates the mailbox with the message and its payload.
3. The caller marks the channel as busy by updating the status.
4. The caller rings the doorbell. This signals the callee that a pending message is in the mailbox.
5. The callee processes the command in mailbox.
6. Optionally, the callee updates the mailbox area with any return data that are associated with the message processing.
7. The callee marks the channel as free by updating the status.
8. The callee issues a completion interrupt to the caller.
9. Optionally the caller processes the contents of the mailbox area.

A flow chart for sending a message using polling mode is shown on the right of Figure 3. The main difference is that the caller has to poll for command completion by checking the status of the channel, as there is no completion interrupt.

The caller must ensure the appropriate ordering of memory operations so that all updates to the mailbox must be visible to the callee before ringing the doorbell. Equally, the callee must ensure that all mailbox changes are visible to the caller before updating the status.

If the caller contains multiple processing elements that can share a transport channel, then appropriate locking must be put in place to ensure that only one processing element can use the channel at any one time. The channel must be locked until the message processing completes and the results are processed by the caller.

5.1.2 Mailbox memory

For a given channel, the layout of the memory that is shared between the agent and platform is described in Table 18.

Table 18 Layout of the mailbox

Field	Byte Length	Byte Offset	Description
Reserved	`0x4`	`0x0`	Reserved, must be zero.
Channel status	`0x4`	`0x4`	The field has the following format: Bits[31:1] Reserved, must be zero. Bit[1] Channel error This bit is set to one if the previous message was not transmitted due to a communications error. The caller must clear it when it has ownership of the channel. Bit[0] Channel free: This bit is set to one if the channel is free. This bit is cleared to zero if the channel is busy.
Reserved	`0x8`	`0x8`	IMPLEMENTATION DEFINED field.
Mailbox flags	`0x4`	`0x10`	Mailbox flags are described in Table 19.
Length	`0x4`	`0x14`	Length in bytes of the Message header and Payload areas (4+N).
Message header	`0x4`	`0x18`	Message header field as described in section 4.1, Table 3.
Message Payload	`N`	`0x1C`	Array of 32-bit values that are used to hold any parameters or return values. The arguments are sent out in the same order they are declared in a protocol command. Return values are sent back in the same order as they are declared in a protocol command. If a message is not known to the callee, the payload must contain NOT_SUPPORTED as the first return value. Status codes are provided in section 4.1.4.

When interrupt driven communication is supported, the mailbox transport allows the caller to choose between interrupt and polling driven communications. This can be done on any transfer, and is useful when the caller wants to operate in a fire and forget fashion, without having to handle interrupts. To make the choice, the mailbox flags are used. The format of the flags is described in Table 19.

Table 19 Mailbox flags

Field	Description
Bits[31:1]	Reserved, must be zero.
Bit[0]	Interrupt communication enable: Set to 1 if the command should complete via an interrupt. Set to 0 if the command should not result in an interrupt assertion.

5.1.2.1 Mailbox transport firmware representation guidelines

An operating system on an agent needs a description of the mailbox transport and its properties before using it. ARM recommends using firmware technologies such as FDT and ACPI for this purpose. This section details the properties that are required to be defined for each channel.

Doorbell register

For agent to platform channels, a doorbell register is required to alert the platform that a message is present in the mailbox area. Writing to it requires a read-modify-write sequence. Firmware tables can be used to describe the properties of the register to an OSPM running on the AP. The properties that must be described are shown in Table 20.

Table 20 Properties of the doorbell register

Field	Description
Register address	Physical address of the register that is written to, to issue a command to the platform.
Preserve Mask	Mask of bits that must be preserved when modifying the doorbell register to issue a command.
Modify Mask	Mask of bits that must be set when modifying the doorbell register to issue a command.

Channels can share a register address for the doorbell, but in this case must have unique preserve and modify masks.

For platform to agent channels, a message interrupt must be described. This interrupt is raised by the platform on notification or delayed response messages. Not describing this interrupt implies that that platform messages have to be polled by agents.

Mailbox shared memory address and size

The physical address of the mailbox, and its size, must be described to the OSPM.

Completion interrupt

For agent-to-platform channels, where interrupt mode is supported, the properties of the completion interrupt must be described by agent firmware. The properties of the completion interrupt to be described are covered in Table 21.

Table 21 Properties of the completion interrupt

Field	Description
Interrupt identifier	Identifier for the interrupt asserted by the platform on command completion.
Interrupt properties	Whether interrupt is level or edge triggered.
Register address	If the interrupt is level sensitive, the physical address of the interrupt clearing register that must be written to, to clear the interrupt.
Preserve Mask	If the interrupt is level sensitive, mask of bits that must be preserved when accessing the register to clear the interrupt.
Modify Mask	If the interrupt is level sensitive, mask of bits that must be set when accessing the register to clear the interrupt.

If the interrupt is level-sensitive, it can be shared by more than one channel. In this case, the preserve- and modify-masks must be unique for each channel.