Arm Firmware Framework for Armv8-A

Document number	DEN0077A
Document quality	ALPHA0
Document version	1.1
Document confidentiality	Non-confidential

Arm Firmware Framework for Armv8-A

Release information

Date	Version	Changes
2021/Mar/15	v1.1 ALP0	• Added guidance for notifications • Added guidance for indirect messaging based upon notifications • Extended indirect messaging to the Secure world • Generalised guidance on scheduling • Clarfied guidance on states of an endpoint execution context • Added guidance on partition runtime models • Added guidance on interrupt management in the Secure world • Added guidance on power management • Added interfaces to discover the ID of the SPMC and SPMD • Added guidance to specify the security state of a memory region during retrieval • Added guidance to discover a SEPID
2020/Jul/24	REL	• Language fixes based upon feedback from editorial review • Removed reference to PSA from document title • Converted document to Arm spec format • Converted ffa_init_info C structure into a table • Clarified use of Sender ID field in FFA_FRAG_RX/TX • Fixed clash in FIDs of FFA_NORMAL_WORLD_RESUME and FFA_MEM_FRAG_RX • Clarified use of FFA_MSG_POLL with RX full interrupt • Clarified multi-endpoint memory management is an optional feature • Clarified how a receiver should request retransmission of a fragmented memory region description • Clarified 64-bit registers can be used in direct messaging

Date	Version	Changes
2020/Apr/24	EAC	• Replaced occurrences of SPCI with PSA FF-A • Added flag to identify other borrowers in a memory retrieve operation • Allowed time slicing of memory management operations at Non-secure physical SPCI instance • Replaced Cookie with Handle in fragmented and time-sliced memory management operations • Added separate ABIs for fragmented memory management operations • Allowed multiple retrievals by a Borrower of a memory region • Allowed retrieval by Hypervisor of a memory region on behalf of a VM • Replaced separate memory transaction descriptors with a single one • Removed Write-through attribute to cater for S2FWB • Specified coherency requirements for memory zeroing • Moved to 64-bit memory Handles • Clarifications to existing memory management guidance • Made guidance on power management IMPLEMENTATION DEFINED • Allowed discovery of minimum buffer size through FFA_FEATURES • Changed FFA_VERSION for negotiation of version number between caller and callee • Clarified usage and description of FFA_FEATURES • Added section on compliance requirements • Other errata fixes and language clarifications based on feedback from beta 1
2019/Dec/20	beta 1	• Added ability to pause and resume memory management transactions • Restricted indirect messaging to Normal world • Reworded guidance on Stream endpoint IDs (SEPIDs) • Added ABI to resume Normal world execution after a Secure interrupt • Reworded guidance on SPCI instances and Split SPM configuration • Added clearer guidance on optional and mandatory interfaces • Other errata fixes and language clarifications based on feedback from beta 0
2019/Nov/13	beta 0	• Replaced some occurrences of ARM with Arm • Non-confidential release of beta 0 spec
2019/Sep/17	beta 0	• Added guidance on partition manifest and setup • Significant rewrite of section on message passing • Added support for multi-component memory management • Added new interfaces for RX/TX management and deprecated old interfaces • Device reassignment has been removed from the scope of this release
2019/Apr/26	alpha 3 Draft 0	• Significant rewrite of section on message passing • Chapter on scheduling models has been removed • Significant rewrite of section on memory management • Chapter 5 has become Chapter 10. Its scope has been reduced temporarily due to preceding changes.
2018/Dec/21	alpha 2	• Changed content based on partner feedback since alpha 1 • There is a clear separation between message passing and scheduling • Introduced use of RX/TX buffers to enable message passing

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349 version 21.0

Contents

Arm Firmware Framework for Armv8-A

References

This section lists publications by Arm® and by third parties.

See Arm® Developer (http://developer.arm.com) for access to Arm® documentation.

[1] *Arm® System Memory Management Unit Architecture specification versions 3.0, 3.1 and 3.2*. See https://developer.arm.com/documentation/ihi0070/ca

[2] *Arm® System Memory Management Unit Architecture specification version 2.0*. See https://static.docs.arm.com/ihi0062/dc/IHI0062D_c_system_mmu_architecture_specification.pdf

[3] *Isolation using virtualization in the Secure world*. See https://developer.arm.com/products/architecture/security-architectures

[4] *SMC Calling Convention*. See https://developer.arm.com/documentation/den0028/latest

[5] *Arm® Architecture Reference Manual for the ARMv8-A architecture*. See https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf

[6] *Universally Unique IDentifier*. See https://tools.ietf.org/html/rfc4122

[7] *Reduced Virtual Interrupt Controller specification*. See https://developer.arm.com/architectures/system-architectures/software-standards/rvic

[8] *Arm® GIC architecture specification versions 3.0 and 4.0*. See https://static.docs.arm.com/ihi0069/e/Q1-IHI0069E_gic_architecture_specification_v3.1_19_01_21.pdf

[9] *VOLUME 4: Platform Initialization Specification, Management Mode Core Interface*. See http://www.uefi.org/sites/default/files/resources/PI_Spec_1_6.pdf

[10] *Management Mode Interface Specification*. See http://infocenter.arm.com/help/topic/com.arm.doc.den0060a/DEN0060A_ARM_MM_Interface_Specification.pdf

[11] *Secure Partition Memory Management*. See https://trustedfirmware-a.readthedocs.io/en/latest/components/secure-partition-manager-mm.html#secure-partition-memory-management

[12] *Power State Coordination Interface*. See https://static.docs.arm.com/den0022/d/Power_State_Coordination_Interface_PDD_v1_1_DEN0022D.pdf

Feedback

Arm welcomes feedback on its documentation.

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

- The title (Arm Firmware Framework for Armv8-A).
- The document ID and version (DEN0077A 1.1).
- The page numbers to which your comments apply.
- A concise explanation of your comments.

Arm® also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

The Armv8.4 architecture introduces the Virtualization extension in the Secure state. The Arm® SMMU v3.2 architecture [1] adds support for stage 2 translations for Secure streams to complement the Secure EL2 translation regime in an Armv8.4 PE. These architectural features enable *isolation* of mutually mistrusting software components in the Secure state from each other. *Isolation* is a mechanism for implementing the principle of least privilege:

A software component must be able to access only regions in the physical address space and system resources for example, interrupts in the GIC that are necessary for its correct operation.

Virtualization in the Secure state enables application of this principle in the following ways:

1. Firmware in EL3 can be isolated from software in S-EL1 for example, a Trusted OS.
2. Firmware components in EL3 can be isolated from each other by migrating vendor-specific components to a sandbox in S-EL1 or S-EL0.
3. Normal world software can be isolated from software in S-EL1 to mitigate against privilege escalation attacks.

This specification describes a software architecture that achieves the following goals.

1. Uses the Virtualization extension to isolate software images provided by an ecosystem of vendors from each other.
2. Describes interfaces that standardize communication between the various software images. This includes communication between images in the Secure world and Normal world.
3. Generalizes interaction between a software image and privileged firmware in the Secure state.

This software architecture is the *Firmware Framework*[1] for Arm® A-profile processors. The term *Framework* and abbreviation *FF-A* are used interchangeably with *Firmware Framework* in this specification.

[1]This document was called the *Secure Partition Client Interface (SPCI)* specification until its BETA1 release.

This Framework also goes beyond the preceding goals to ensure that the guidance can be used,

1. In the absence of the Virtualization extension in the Secure state. This provides a migration path for existing Secure world software images to a system that implements the Virtualization extension in the Secure state.
2. Between VMs managed by a Hypervisor in the Normal world. The Virtualization extension in the Secure state mirrors its counterpart in the Non-secure state (see also [2]). Therefore, a Hypervisor could use the Firmware Framework to enable communication and manage isolation between VMs it manages.

More rationale about the introduction of the Virtualization extension in Secure state and goals of the Firmware Framework is provided in the white-paper titled *Isolation using virtualization in the Secure world* [3].

1.1 Overview

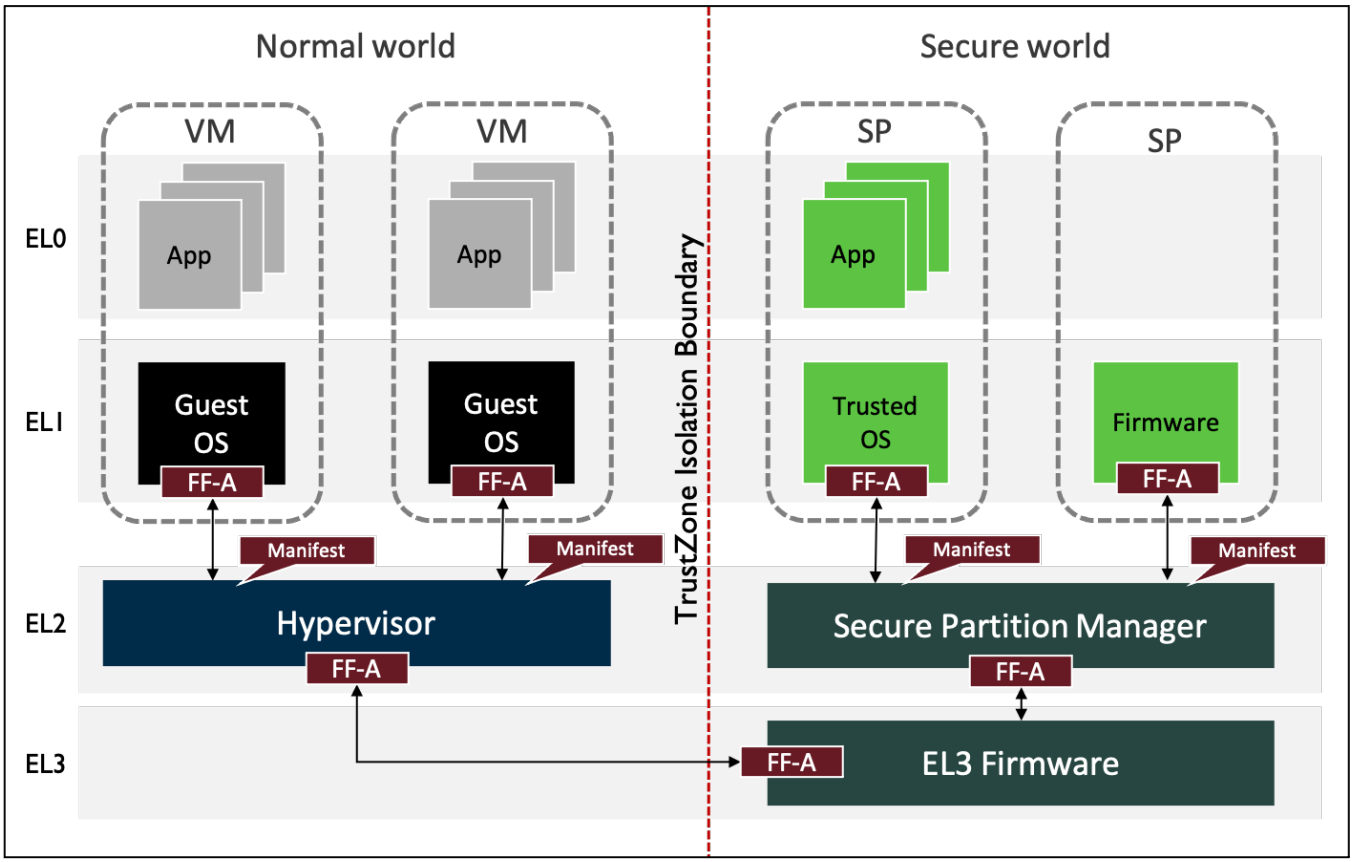

Figure 1.1: Firmware Framework with Secure EL2

The **building blocks** of the Firmware Framework described in this specification are as follows. Figure 1.1 illustrates an implementation of this Framework and its components.

1. **One or more partitions** that provide a sandboxed software execution environment. These could be VMs running in the Normal or Secure world. A Secure world VM is called a *Secure Partitions* (SP) to distinguish it from VMs in the Normal world.

   An SP typically encompasses the S-EL1 and S-EL0 Exception levels. The Firmware Framework supports SPs that run only in S-EL0 as well. A S-EL0 SP could be managed by software in S-EL1 or EL3. This is an IMPLEMENTATION DEFINED choice.

   The term *endpoint* is used interchangeably with the term *partition*.

   - In the Normal world, an endpoint could be a VM when the Virtualization extension is enabled or the OS Kernel when the Virtualization extension is disabled or unavailable. These endpoints are called *NS-Endpoints* in scenarios where it is not necessary to distinguish between them.
   - In the Secure world, an endpoint is an SP running in one of the following Exception levels:
     - Secure EL0.
     - Secure User mode.
     - Secure EL1.
     - Secure Supervisor mode.

These endpoints are called *S-Endpoints* in scenarios where it is not necessary to distinguish between them.

2. A **partition manifest** that describes the system resources a partition needs, identity of the partition to enable discovery of services that it implements and other attributes of the partition that govern its run-time behavior.
3. A **partition manager** (PM) that assigns system resources to partitions and manages isolation among them. In the Secure world, this component is called the *Secure Partition Manager* (SPM). In the Normal world it is a Hypervisor[2]. The SPM and Hypervisor are collectively referred to as the *Partition managers* in scenarios where they have the same responsibilities, and it is not necessary to distinguish between them. See 2.1 *Partition manager* for a description of this component.
4. **Application binary interfaces** that partitions can invoke at their Exception level boundaries for the following purposes.
    1. Discover the presence of a partition, its properties and services it implements.
    2. Message passing among partitions and partition managers.
    3. Memory management between partitions.

Table 1.1 summarizes software configurations supported by the Firmware Framework in each Security state with regard to availability of the Virtualization extension. Furthermore, each configuration in one Security state can co-exist with any configuration in the other Security state.

Table 1.1: Firmware Framework configurations

Config. No.	Security state	Virtualization extension enabled	Description
1.	Non-secure	No	OS in EL1 uses the Firmware Framework to communicate with one or more S-Endpoints.
2.	Non-secure	Yes	One or more VMs in EL1 use the Firmware Framework to: • Communicate with one or more S-Endpoints. • Communicate with each other. • Isolate themselves from each other.
3.	Secure	No	One or more S-Endpoints use the Firmware Framework as follows: • A single SP for example, a Trusted OS in S-EL1 uses the Framework to communicate with one or more NS-Endpoints. • One or more SPs in S-EL0 use the Firmware Framework to: – Communicate with one or more NS-Endpoints. – Communicate with each other. – Isolate themselves from each other.
4.	Secure	Yes	One or more SPs use the Firmware Framework to: • Communicate with one or more NS-Endpoints. • Communicate with each other. • Isolate themselves from each other.

[2] A hypervisor implementation could span EL1 and EL2. In this specification, this term refers to the layer of software that runs in EL2 and is responsible for providing isolation guarantees between VMs through use of the Arm® virtualization extension.

1.2 Document organization

The rest of this document is organized as follows.

1. Chapter 2 *Concepts* describes some fundamental concepts that are used to define the Firmware Framework architecture.
2. Chapter 3 *Setup* specifies the information contained in a partition manifest and how it is used to initialize a partition by a partition manager.
3. Chapter 4 *Message passing* describes the mechanisms that partitions can use for message passing.
4. Chapter 5 *Partition runtime models* describes the state transitions partitions are permitted make in the run-time models that their partition manager implements.
5. Chapter 6 *Interrupt management* specifies guidance on interrupt management in the Secure world.
6. Chapter 7 *Notifications* describes support for notifications. This is a mechanism that one partition can use to ring a *doorbell* of another partition.
7. Chapter 8 *Memory Management* describes the mechanisms that partitions can use for memory management.
8. Chapter 9 *Interface overview* provides an overview of the ABIs defined by the Firmware Framework.
9. ABIs used in the Firmware Framework for status reporting, setup and discovery of partitions, scheduling, messaging, memory management and notifications are specified in the following sections.
    - Chapter 10 *Status reporting interfaces*.
    - Chapter 11 *Setup and discovery interfaces*.
    - Chapter 12 *CPU cycle management interfaces*.
    - Chapter 13 *Messaging interfaces*.
    - Chapter 15 *Notification interfaces*.
    - Chapter 14 *Memory management interfaces*.
10. Chapter 16 *Appendix* provides guidance on the following additional topics.
    - 16.1 *S-EL0 & User mode partitions*.
    - 16.2 *Additional memory management features*.
    - 16.3 *Power Management*.
    - 16.4 *Legacy indirect messaging usage*.

Chapter 2
Concepts

2.1 Partition manager

The partition manager is responsible for initializing a partition as per the requirements stated in its manifest (see Chapter 3 *Setup*). A partition describes the regions in the system physical address space and resources it needs access to through its manifest. The partition manager uses the manifest to validate the resource requests and assign resources to the endpoint if the validation succeeds.

The following trust boundaries are defined by the Firmware Framework vis-a-vis the partition managers and partitions.

- The SPM is a part of the TCB for a system resource or physical address space range assigned to the Secure state.
- Both the Hypervisor and SPM are a part of the TCB for a system resource or physical address space range assigned to the Non-secure state.
- A VM must trust the Hypervisor and SPM to protect its resources from other endpoints.
- An SP must trust the SPM to protect its Secure resources from other SPs.
- An SP must not trust the state of any Non-secure resource it has access to. Therefore, it must not trust the Hypervisor or a NS-Endpoint that could also access the same resource.

A partition manager protects partition resources from other FF-A components by utilizing the following features implemented by the CPU and system architecture.

- The Arm® TrustZone Security extension is used to protect the Secure physical address space ranges and system resources assigned to FF-A components in the Secure state from components in the Non-secure state.
- Virtual memory-based memory protection provided by the Armv8-A VMSA is used to protect the physical address space ranges assigned to a Security state and FF-A component from other FF-A components. Its usage by the SPM is described in 2.2 *SPM architecture*. The Hypervisor uses this feature as follows.
  - If the *EL1&0 stage 2 translation regime, when EL2 is enabled* is implemented by a System Memory Management Unit (SMMU) in the Non-secure state, the Hypervisor uses it to restrict visibility of the Non-secure physical address space from a device upstream of the SMMU to only those regions that have been assigned to the VM that controls the device.
  - If the *Secure EL1&0 stage 2 translation regime, when EL2 is enabled* is implemented by a CPU in the Non-secure state, the Hypervisor uses it to restrict visibility of the Non-secure physical address space from a VM to only those regions that have been assigned to the VM.

The partition manager enables partitions to exchange messages (see Chapter 4 *Message passing*). It also enables a partition to manage access to memory regions that are assigned to it from other partitions (see Chapter 8 *Memory Management*).

In an implementation of this Framework, the SPM must use the concepts and interfaces described in this specification to fulfill the preceding responsibilities. A Hypervisor could use the Framework only for communication and memory management between the Normal world and Secure world. In this case, the Hypervisor must:

- Initialize VMs and isolate them from other VMs through IMPLEMENTATION DEFINED mechanisms.
- Implement a partition manager component that uses the Firmware Framework to enable communication and memory management between two endpoints. for example, this could be an FF-A driver implemented in the Hypervisor.

In this version of the Firmware Framework, it is assumed that a partition manager is initialized through an IMPLEMENTATION DEFINED mechanism.

2.2 SPM architecture

The responsibilities of the SPM are split between the two components as follows.

1. The *SPM Core* (SPMC) component which is responsible for:
   - Partition initialization and isolation at boot time.
   - Inter-partition isolation at run-time.
   - Inter-partition communication at run-time between:
     - S-Endpoints.
     - S-Endpoints and NS-Endpoints.
2. The *SPM Dispatcher* (SPMD) component which is responsible for:
   - *SPM Core* initialization at boot time.
   - Forwarding FF-A calls from Normal world to the *SPM Core*.
   - Forwarding FF-A calls from the *SPM Core* to the Normal world.

The term SPM is used when it is not necessary to distinguish between these two components. Some properties of the two components are as follows.

- Both components have access to the entire physical address space and are a part of the *Trusted computing base*.
- If the two components reside in separate Exception levels:
  - They must implement and report a mutually compatible version of the Firmware Framework. See 11.1.3 *SPM usage* for details.
  - They must use the ABIs defined in this specification for communication.
- The mechanism used by the SPMD to initialize the SPMC is IMPLEMENTATION DEFINED. The guidance provided in Chapter 3 *Setup* could be used by the implementation.
- The SPMD component must execute in either EL3 in AArch64 or the Monitor mode in AArch32 Execution states.

The Firmware Framework supports SPMC configurations listed in Table 2.1 & Table 2.2.

Table 2.1: SPMC configurations in AArch64 Execution state

SPM config. number	SPMD EL	SPMC EL	Virtualization extension enabled	Name of configuration
1.	EL3	S-EL1	No	S-EL1 SPMC
2.	EL3	S-EL2	Yes	S-EL2 SPMC
3.	EL3	EL3	No	EL3 SPMC

Table 2.2: SPMC configurations in AArch32 Execution state

SPM config. number	SPMD EL	SPMC EL	Virtualization extension enabled	Name of configuration
1.	EL3	Secure Supervisor	No	S-Supervisor SPMC
2.	Monitor	Secure Supervisor	No	S-Supervisor SPMC

Table 2.3 lists combinations of SPMC and SP configurations supported by this version of the Framework.

- The first row lists all possible SP configurations.
- The first column lists all possible SPMC configurations.
- An intersection of a row and a column indicates whether the SP configuration in the row is supported by the SPMC configuration in the column.

Table 2.3: Valid combinations of SPMC and SP configurations

SPMC config. name	S-EL0 SP	Secure User mode SP	S-EL1 SP	Secure Supervisor mode SP	Notes
S-EL1 SPMC	Yes	Yes	Yes	NA	See 2.2.2.1 *S-EL1 SPM core component*
S-Supervisor SPMC	NA	Yes	NA	Yes	See 2.2.2.2 *Secure Supervisor mode SPM core component*
EL3 SPMC	Yes	Yes	Yes[1]	No	See 2.2.2.3 *EL3 SPM core component*
S-EL2 SPMC	Yes	Yes	Yes	Yes	See 2.2.1 *SPM architecture with Secure EL2*

[1] Either a S-EL1 SP or S-EL0/User mode SPs must be supported but not both.

2.2.1 SPM architecture with Secure EL2

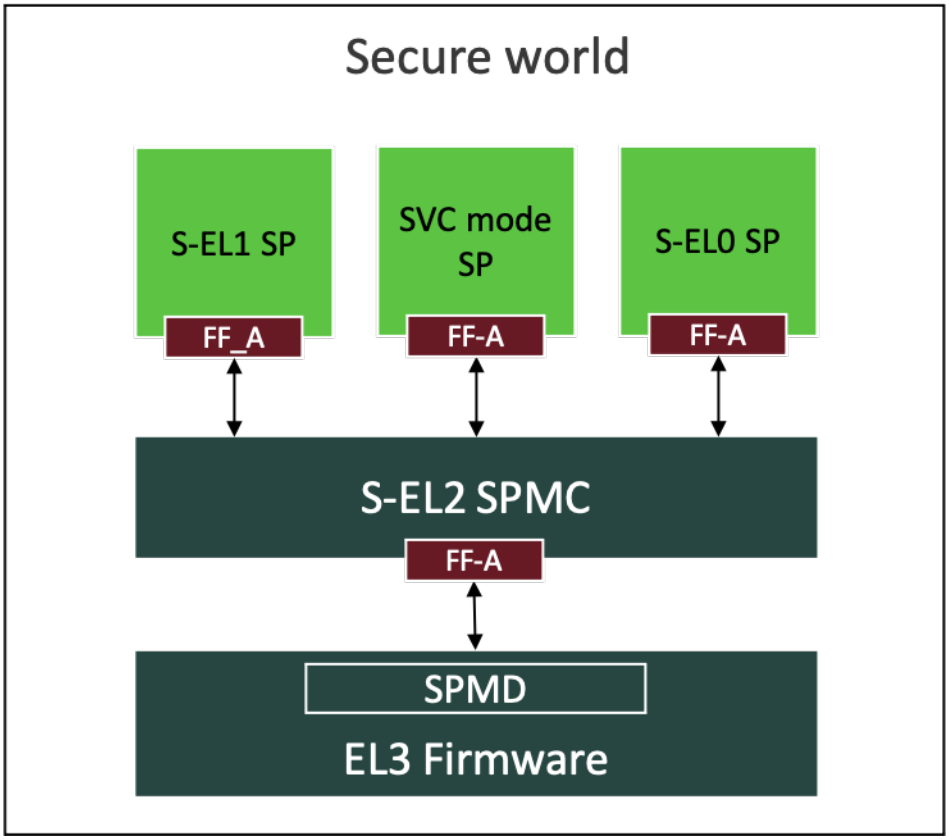

Figure 2.1: Example S-EL2 SPM Core and SP configuration

The S-EL2 SPMC (see SPM configuration 2 in Table 2.1) is fundamental to enforcing the principle of least privilege in the Secure state on Armv8.4 or later systems as described in Chapter 1 *Introduction*. It must support at least one of the SP configurations as follows.

1. The SPMC uses *Armv8.1 VHE* to manage one or more SPs that run in S-EL0 or Secure User mode. It fulfills all the responsibilities listed in 2.2 *SPM architecture*.

   The physical address space assigned to an SP is isolated from other FF-A components through the single stage of address translation implemented by the *Secure EL2&0 translation regime*.

2. The SPMC manages one or more SPs that run in S-EL1 or S-Supervisor mode. It fulfills all the responsibilities listed in 2.2 *SPM architecture*.

   The physical address space assigned to an SP is isolated from other FF-A components by the *Secure EL1&0 stage 2 translation regime, when EL2 is enabled*.

An example of these configurations is illustrated in Figure 2.1.

2.2.2 SPM architecture without Secure EL2

In the absence of Secure EL2, SPM could be used in the following scenarios.

- Reduce the size of the *Trusted computing base* on an Armv8.3 or earlier system by migrating EL3 & S-EL1 firmware components to one or more SPs that run in S-EL0 or Secure User mode.
  - The SPMC configurations described in 2.2.2.1 *S-EL1 SPM core component*, 2.2.2.2 *Secure Supervisor mode SPM core component* and 2.2.2.3 *EL3 SPM core component* could be used in this scenario.

  - See 16.1 *S-EL0 & User mode partitions* for an example use case of this configuration.

- Migrate a Secure world software stack that runs on Armv8.3 or earlier systems to the Firmware Framework in preparation for Armv8.4 or later systems.
  - The SPMC configurations described in 2.2.2.1 *S-EL1 SPM core component* and 2.2.2.2 *Secure Supervisor mode SPM core component* could be used in this scenario.

2.2.2.1 S-EL1 SPM core component

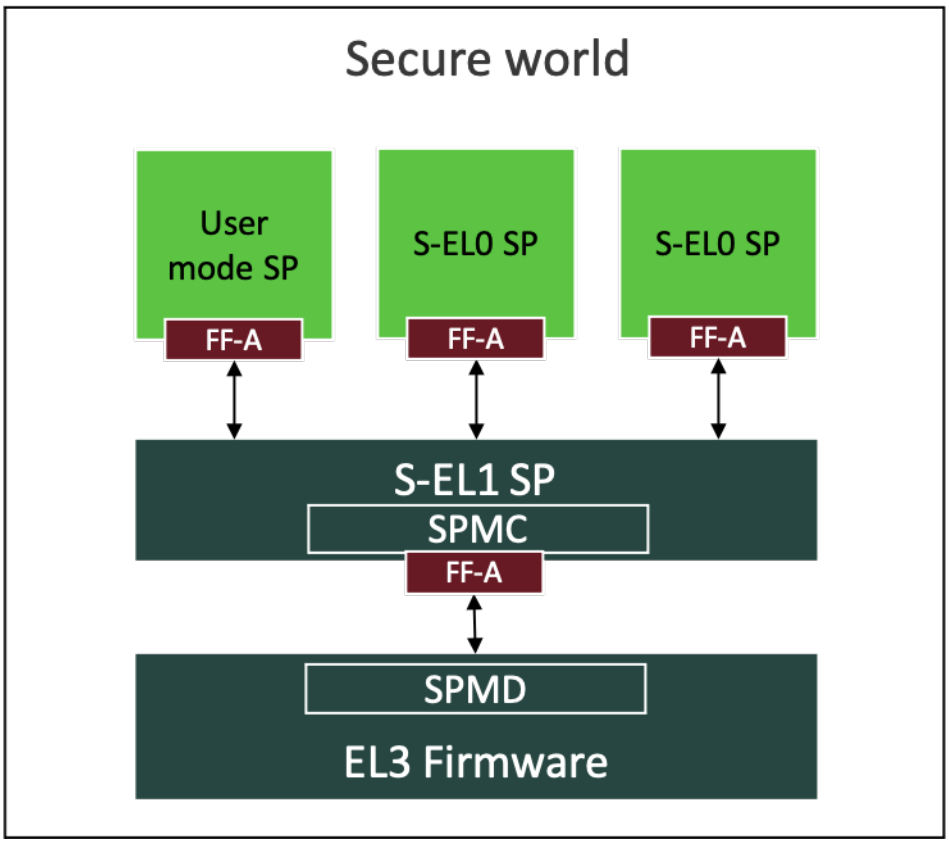

Figure 2.2: Example S-EL1 SPM Core and SP configuration

A S-EL1 SPMC must support at least one of the SP configurations as follows.

1. The SPMC manages one or more SPs that run in S-EL0 or Secure User mode. It fulfills all the responsibilities listed in 2.2 *SPM architecture*.

   The physical address space assigned to an SP is isolated from other FF-A components through the single stage of address translation implemented by the:

   - *Secure EL1&0 translation regime* for S-EL0 SPs.
   - *Secure PL1&0 translation regime* for Secure User mode SPs.

2. The SPMC manages a single SP that also runs in S-EL1. The SPMD, SPMC, and SP components have the same level of access to the physical address space and are a part of the *Trusted computing base*.

   In this configuration:

   - The Framework assumes that the SPMC is packaged in the SP software image.
   - The interface between the SPMC and the SP component is IMPLEMENTATION DEFINED for example, a set of C programming language APIs.

- Any FF-A calls targeted to the SP from the Normal world must be received by the SPMC and forwarded to the appropriate SP component through the IMPLEMENTATION DEFINED interface.
- The SPM and Normal world software cannot be isolated from the SP at boot time. See Chapter 3 *Setup* for more information on the implications of this constraint on partition setup and boot.
- The SPM and Normal world software cannot be isolated from the SP at run-time. See 8.5.1 *Component roles* for more information on the implications of this constraint on memory management transactions between the SP and the Normal world.
- The SP must be capable of receiving and sending messages just like the SPM. See 4.1.1 *Indirect messaging* & 4.4.1 *Discovery and setup* for more information on the implications of this constraint on communication between the SP and the Normal world.

Figure 2.2 illustrates a combination of these configurations.

2.2.2.2 Secure Supervisor mode SPM core component

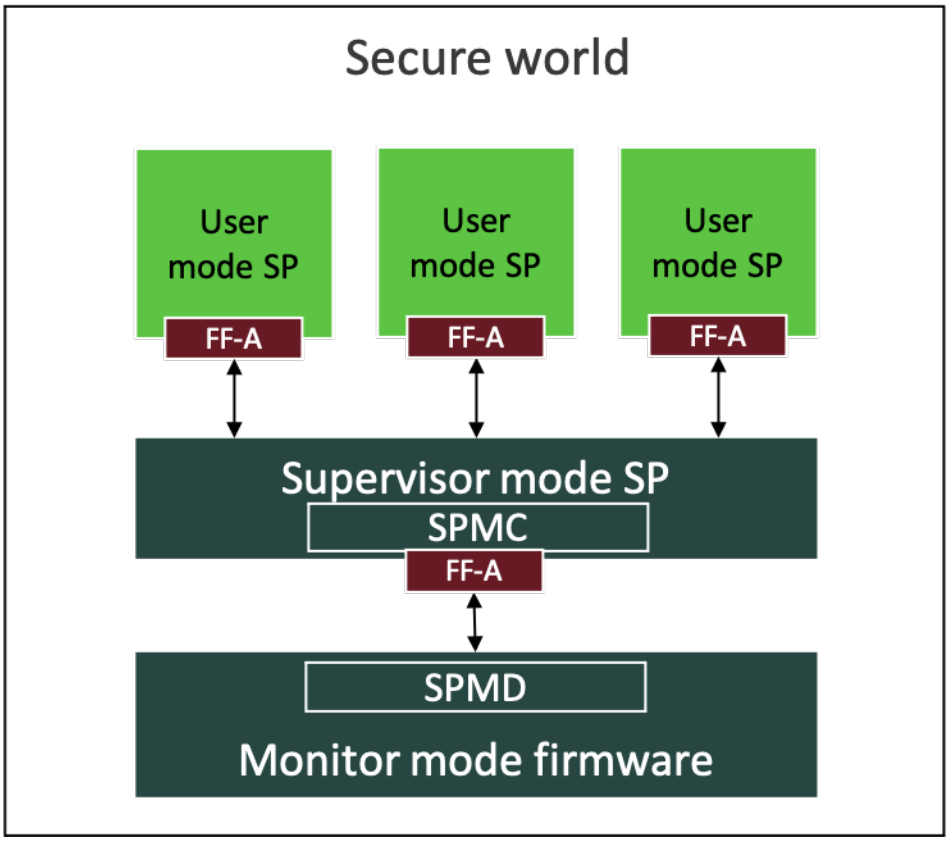

Figure 2.3: Example Supervisor mode SPM Core and SP configuration

The S-Supervisor SPMC must support the same SP configurations described in 2.2.2.1 *S-EL1 SPM core component* with the following caveats.

1. The SPMC manages one or more SPs that run only in Secure User mode.
2. The SPMC coexists with a single SP that also runs in Secure Supervisor mode.
3. The SPM is isolated from the Secure User mode SPs through the single stage of address translation implemented by the *Secure PL1&0 translation regime*.

This configuration is illustrated in Figure 2.3.

2.2.2.3 EL3 SPM core component

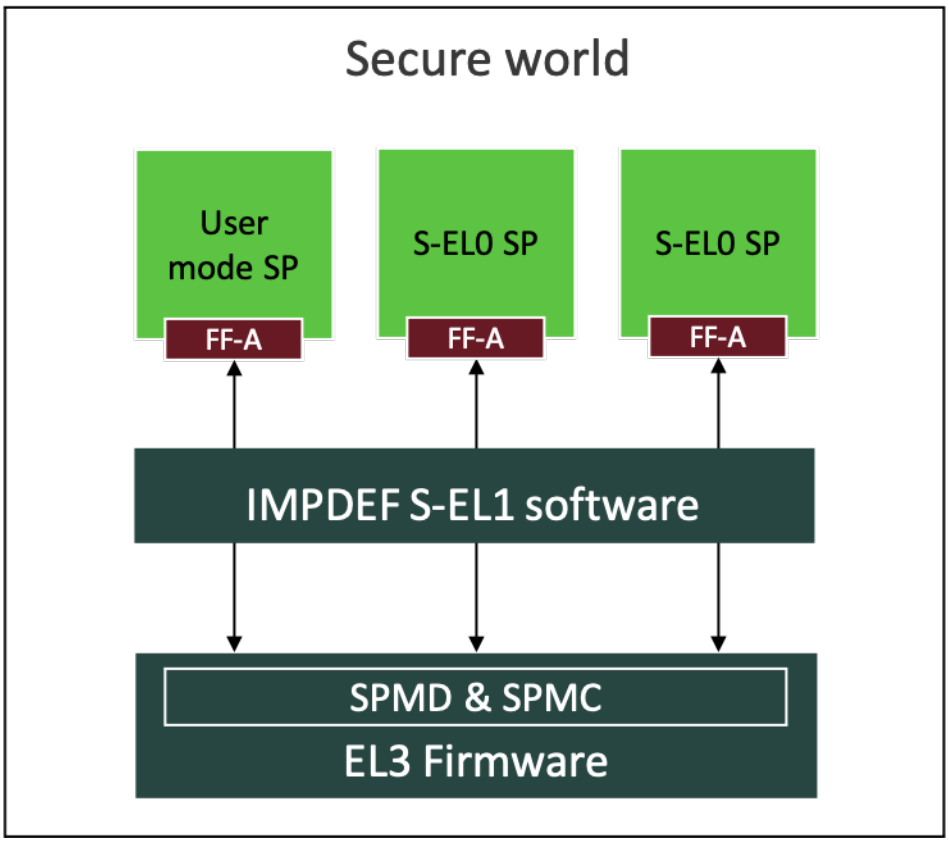

Figure 2.4: Example EL3 SPM Core and SP configuration

The EL3 SPMC co-exists with the SPMD and manages one of the following SP configurations. It fulfills all the responsibilities listed in *2.2 SPM architecture*.

1. One or more SPs that run in S-EL0 or Secure User mode. This configuration is illustrated in Figure 2.4.

   The physical address space assigned to an SP is isolated from other FF-A components through the single stage of address translation implemented by the:

   - *Secure EL1&0 translation regime* for S-EL0 SPs.
   - *Secure PL1&0 translation regime* for Secure User mode SPs.

2. A single SP that resides in S-EL1. The SPMD, SPMC, and SP components have the same level of access to the physical address space and are a part of the *Trusted computing base*. The roles of the SPMC and SPMD are combined such that they are collectively responsible for:

   - SP initialization at boot time.
   - Inter-partition communication between the SP and NS-Endpoints at runtime.

2.3 FF-A instances

An *FF-A instance* is a valid combination of two FF-A components at an Exception level boundary. These instances are used to describe the interfaces specified by the Firmware Framework. An interface is accessed at an FF-A instance through a conduit described in 2.4 *Conduits*. The responsibilities of the caller and callee in each interface depend on the FF-A instance at which it is invoked.

- An instance is *physical* if:
  - Each component can independently manage its translation regime.
  - The translation regimes of each component map virtual addresses to physical addresses.
- An instance is *virtual* if it is not physical.
- The instance between the SPMC and SPMD is called the *Secure physical FF-A instance*.
- The instance between the SPMC and an SP is called the *Secure virtual FF-A instance*.
- In the Normal world, the instance between:
  - The Hypervisor and a VM is called the *Non-secure virtual FF-A instance*.
  - The Hypervisor and SPMD is called the *Non-secure physical FF-A instance*.
  - The OS kernel and SPMD, in the absence of a Hypervisor is called the *Non-secure physical FF-A instance*.

Table 2.4 lists the valid Secure FF-A instances. Table 2.5 lists the valid Non-secure FF-A instances.

- Entries in the first row represent the higher Exception level at an Exception level boundary.
- Entries in the first column represent the lower Exception level at an Exception level boundary.
- Combinations of Exception levels that are not architecturally feasible are listed as *Not applicable (NA)*.
- Combinations of Exception levels that are not supported by the Firmware Framework are listed as *Invalid (INV)*.

Table 2.4: Secure FF-A instances

EL boundary	EL3	Monitor	S-EL2	S-EL1	Secure Supervisor
S-EL2	Secure physical	NA	NA	NA	NA
S-EL1	Secure physical	NA	Secure virtual	NA	NA
Secure Supervisor	Secure physical	Secure physical	Secure virtual	NA	NA
S-EL0	Secure virtual	NA	Secure virtual	Secure virtual	NA
User	Secure virtual	INV	Secure virtual	Secure virtual	Secure virtual

Table 2.5: Non-secure FF-A instances

EL boundary	EL3	Monitor	EL2	Hypervisor
EL2	Non-secure physical	NA	NA	NA
Hypervisor	Non-secure physical	Non-secure physical	NA	NA
EL1	Non-secure physical	NA	Non-secure virtual	Non-secure virtual
Supervisor	Non-secure physical	Non-secure physical	Non-secure virtual	Non-secure virtual

The definition of an *FF-A instance* when both FF-A components reside in the same Exception level is IMPLEMEN-

TATION DEFINED. This is applicable to the *Secure physical and virtual FF-A instances* described in 2.2.2.3 *EL3 SPM core component* and 2.2.2.1 *S-EL1 SPM core component* respectively. For example, the implementation could maintain a logical separation between the two components through the use of an API that has the same semantics as the FF-A ABIs at the same instance.

2.4 Conduits

The Framework defines interfaces to enable communication between various FF-A components (see Chapter 9 *Interface overview*). Each interface is accessible through one or more conduits as follows.

The SMC conduit as described in [4] should be used to invoke an interface by an FF-A component executing in EL1 or S-EL1. When an interface is invoked from EL1, the SMC execution must be trapped by the Hypervisor at EL2. Similarly, when an interface is invoked at S-EL1 and the SPM resides in S-EL2, the SMC execution must be trapped by the SPM. This implies that the SMC conduit provides the flexibility that is required to support implementations with and without a hypervisor in EL2 or SPM in S-EL2.

If an endpoint executing in EL1 or S-EL1 cannot use the SMC conduit, it must use the HVC conduit instead.

A S-EL0 SP must use the SVC (Supervisor Call) instruction as a conduit to call into S-EL1. The SMC32 and SMC64 calling conventions are mirrored as SVC32 and SVC64 calling conventions respectively.

The Firmware Framework enables message exchange between any two FF-A components that might be at the same or a different Exception level relative to each other. A request, its results, or an error status could be sent from:

- A lower EL to a higher EL
- A higher EL to a lower EL.

To fulfill this requirement, this version of the Framework uses the *ERET* instruction as a conduit for transmitting requests and responses from a higher EL to a lower EL.

The parameter register usage in an SMC, HVC, or SVC call is mirrored in an ERET call for example, *w0* contains a *function identifier* parameter in the ERET call. This ensures that messages can be passed at any FF-A instance irrespective of their direction of travel. An invocation through the SMC, HVC, or SVC conduits is completed through the ERET conduit. An invocation through the ERET conduit is completed through the SMC, HVC, or SVC conduits.

This usage of the *ERET* instruction as a conduit along with the SMC, HVC, and SVC conduits enables half-duplex communication between two FF-A components at an EL boundary at any FF-A instance.

The taxonomy of information transmitted through a conduit at an FF-A instance is as follows.

1. An interface invocation described in Chapter 9 *Interface overview*.
2. Results from the successful completion of the invoked interface.
3. Error code from an unsuccessful completion of the invoked interface.

Based on the preceding taxonomy, an interface invocation through one conduit at an FF-A instance can complete through another conduit in one of the following ways.

- A error code. The *FFA_ERROR* function is used to return the error code (see 10.2 *FFA_ERROR*).
- Results of the request. The *FFA_SUCCESS* function is used to return the results (see 10.3 *FFA_SUCCESS*).
- An invocation of another interface described in Chapter 9 *Interface overview*.

An invocation of a non-FF-A interface from a lower Exception level to a higher Exception level for example, through the SMC, HVC, or SVC conduits must not complete with an invocation of an FF-A function through the ERET conduit unless, the caller implements support to distinguish between the FF-A and non-FF-A register usage on completion. For example, *w0* would contain a status code in the latter case while it will contain a *function identifier* in the former case.

2.5 Execution state

The Armv8-A architecture defines two Execution states, AArch32 and AArch64 as described in [5]. The Execution states that are applicable to each FF-A component are as follows.

- The SPM in S-EL2 or EL3 must run in AArch64.
- The SPM in S-EL1 could run in AArch64 or AArch32.
- The Hypervisor in EL2 must run in AArch64.
- A S-EL0 SP could run in AArch64 or AArch32.
- An endpoint in S-EL1 or EL1 could run in either AArch64 or AArch32.

2.6 Memory types

Each memory region is assigned to either the Secure or Non-secure physical address space at system reset or during system boot. Normal world can only access memory regions in the Non-secure physical address space. Secure world can access memory regions in both address spaces. The Non-secure (NS) attribute bit in the translation table descriptor determines whether an access is to Secure or Non-secure memory. In this version of the Framework:

- Memory that is accessed with the NS bit set in the translation regime of any component is called *Normal memory*.
- Memory that is accessed with the NS bit cleared in the component translation regime is called *Secure memory*.

2.7 Memory granularity and alignment

The Firmware Framework specifies support to map a memory region in the translation regimes of the two FF-A components at an FF-A instance (see 4.2.2.3 *Buffer attributes* & Chapter 8 *Memory Management*). The translation regimes could use the same or a different translation granule size. To map the memory region correctly in both translation regimes, the following constraints must be met:

- If *X* is the larger translation granule size used by the two translation regimes, then the size of the memory region must be a multiple of *X*.
- The base address of the memory region must be aligned to *X*.

For example, at the Non-secure virtual FF-A instance, a VM and the Hypervisor could use translation granule sizes of 4K and 64K respectively. The size of any memory region that must be mapped in both their translation regimes must be a multiple of 64K and aligned to the 64K boundary.

An endpoint could specify its translation granule size in its partition manifest as described in 3.2.1 *Manifest for isolated partitions*. The Hypervisor and SPM could also use an IMPLEMENTATION DEFINED mechanism to determine the translation granule size of an endpoint.

An endpoint must discover the minimum size and alignment boundary (that is, the minimum value of *X*) to share a memory region with its partition manager through the *FFA_FEATURES* interface (see 11.2 *FFA_FEATURES*).

2.8 FF-A component identification and discovery

Partitions are identified by a 16-bit *ID* and a UUID (Unique Universal Identifier) (see [6]). This helps partitions discover the presence of other partitions and their properties.

A partition must use the *FFA_ID_GET* interface (also see 11.8 *FFA_ID_GET*) to discover its ID.

FF-A components can discover the identities and properties of other partitions through the *FFA_PARTITION_INFO_GET* interface. Once discovered, the IDs must be used in the messaging interfaces to identify the target of a message.

A unique ID must be assigned to each partition in the system. The mechanism used to assign an *ID* to a partition is IMPLEMENTATION DEFINED. The *ID* could be specified in the manifest of the partition or allocated at boot by the partition manager responsible for managing the partition. A Hypervisor could use the *FFA_PARTITION_INFO_GET* interface to determine the *IDs* assigned to SPs to avoid clashes with VM *IDs*.

The Framework supports identification of a partition manager by a unique 16-bit *ID* as well.

- The ID value *0* is reserved for the Hypervisor as described in [4].
- The ID values assigned to the SPMC and SPMD components are IMPLEMENTATION DEFINED.
  - In v1.1 of the Framework, the SPMC and SPMD must be assigned IDs. Other FF-A components can discover these IDs through the FFA_SPM_ID_GET interface (see 11.9 *FFA_SPM_ID_GET*).

2.9 Execution context

Each endpoint has one or more *execution contexts* depending on its implementation. An execution context comprises of general-purpose, system, and any memory mapped register state that must be maintained by a partition manager.

A partition manager is responsible for allocating, initializing, and running the execution context of an endpoint on a physical or virtual PE in the system. An execution context is identified by using a 16-bit ID. This ID is referred to as the *vCPU* or *execution context* ID. Each execution context must be allocated an ID that is unique among all execution contexts that belong to the endpoint.

An execution context of an endpoint represents a logical processor to the partition manager. The partition manager delegates message processing to an execution context of an endpoint. It is independent of threads implemented inside an endpoint to process the messages and logic to schedule these threads (see also 2.11 *Primary scheduler*). Figure 2.5 illustrates this relationship.

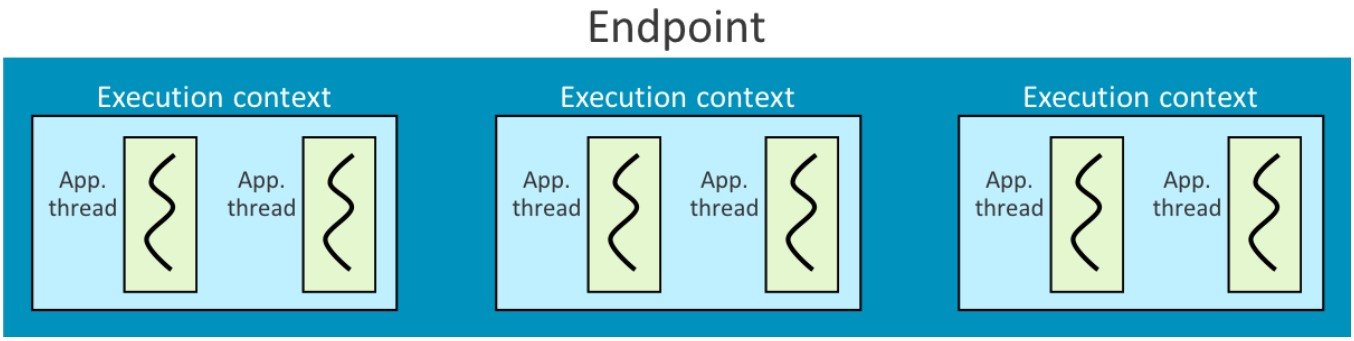

Figure 2.5: Example endpoint with execution contexts and threads

An endpoint must be one of the following types:

- Implements a single execution context and is not capable of Symmetric multi-processing. It runs only on a single PE in the system at any point of time. This type of endpoint is called a **UP** endpoint.
- Implements multiple execution contexts and is capable of Symmetric multi-processing. These contexts run concurrently on separate PEs in the system. These endpoints are called **MP** endpoints.

An execution context of an endpoint could be capable of *migrating*. Migration capability means that the partition manager could save the execution context of an endpoint on one PE. It could then restore the saved execution context on another PE and resume endpoint execution. The endpoint must not make any assumptions about the PE it runs on.

This version of the Framework requires the following:

- UP endpoints must be capable of migrating.
- Execution contexts of MP endpoints could be capable of migrating between PEs or could be fixed to a particular PE. The latter are called *pinned contexts*.
- The migration capability must be specified in the endpoint manifest (see 3.2.1 *Manifest for isolated partitions*).
- S-EL0 partitions must be UP.

The number of execution contexts an endpoint implements can differ from the number of PEs in the system. This must be specified in the manifest of the endpoint (see 2.10 *System resource management*). For example, a VM in the Normal world must use the manifest to inform the Hypervisor how many vCPUs it implements. The Hypervisor must maintain an execution context for each vCPU.

2.10 System resource management

Components in the Firmware Framework require access to the following system resources.

- Memory regions.
- Devices.
- CPU cycles.

The Framework associates the attributes of *ownership* and *access* with these resources. The Owner governs the following capabilities of non-Owners for each resource.

- The level of access a non-Owner has for using the resource. This could be exclusive, shared or no-access.
- The ability to grant access to the resource to other non-Owners. This is called access forwarding.

Also, the Owner could relinquish ownership to another component.

The Framework also specifies the transitions that result in a change of ownership and access attributes associated with a resource. A combination of these attributes and transitions determines how a resource is managed among components.

Rules associated with ownership and access of memory regions are described in Chapter 8 *Memory Management*.

Rules associated with ownership and access of CPU cycles are described in 2.11 *Primary scheduler*.

For a device that is upstream of an SMMU, its access to the physical address space is managed using the rules associated with management of memory regions (also see 8.2 *Direct memory access*).

For all devices, ownership and access attributes are associated with its *MMIO* region. A partition could request access and/or ownership of a device through its manifest (see Table 3.3). This is done through one of the following ways.

- A partition requests ownership and exclusive access to the MMIO region of a device during boot time (see Chapter 3 *Setup*). The corresponding partition manager assigns the MMIO region with these attributes to the partition.
- One or more partitions request access to the MMIO region of a device during boot time. The corresponding partition manager is the Owner of the MMIO region and grants access to all the partitions.

This version of the Framework does not permit:

- Ownership of a device MMIO region to be transferred to another partition during run-time.
- Access to a device MMIO region to be granted to another partition during run-time.
- Access to a device MMIO region to be revoked from a partition during run-time.

2.11 Primary scheduler

FF-A components require CPU cycles to do work. The Framework assumes a hierarchical model where a single FF-A component in the Normal world is the owner of CPU cycles across all PEs in the system. This component lends CPU cycles to other FF-A components.

This component is the Hypervisor if it is implemented in EL2. It could be one of the following.

1. The Host OS running in EL2 in the case of a Type 2 Hypervisor when the Virtualization host extension is used.
2. The Type 1 Hypervisor running in EL2.

This component is called the *primary endpoint* if it is implemented in an NS-Endpoint. It could be one of the following.

1. The OS kernel running in EL1 if the Virtualization extension is not used in the Normal world.
2. The Host OS running in EL1 in the case of a Type 2 Hypervisor when the Virtualization host extension is not implemented or used (see [5]).
3. A separate VM running in EL1 that has been delegated the responsibility of scheduling by the Hypervisor.

The scheduler implemented in the Hypervisor or primary endpoint is called the *primary scheduler*. This term is used in the context of CPU cycle allocation when it is not necessary to distinguish whether it is the Hypervisor or the primary endpoint that is owner of CPU cycles in the system.

An endpoint that does not implement the primary scheduler is called a *secondary endpoint*. A secondary endpoint could implement a *secondary scheduler* to manage allocated cycles among its threads. A secondary endpoint could be allocated CPU cycles,

1. By the primary scheduler. For example,
   - For every VM managed by a Hypervisor, it implements a thread for each vCPU of a VM. A vCPU receives CPU cycles when its thread is scheduled by the primary scheduler.
   - A Trusted OS has a counterpart driver in the primary endpoint. This driver is invoked by client applications to request Trusted OS services. The driver forwards requests to an execution context of the Trusted OS. It could do this as follows.
     – Manage a set of threads to run an execution context of the Trusted OS.
     – Run an execution context of the Trusted OS in the context of the client application thread that issued the request.

   In both examples, an execution context of a secondary endpoint is scheduled by the primary scheduler.
2. By another secondary endpoint. A variant of the above example could be where a Trusted OS has a counterpart driver in the VM scheduled by the Hypervisor instead of the primary endpoint. This driver is invoked by client applications installed in the VM to request Trusted OS services. The driver runs an execution context of the Trusted OS to handle the request. The client applications are scheduled by a secondary scheduler implemented in the VM.

   In this example, the primary scheduler in the Hypervisor schedules a secondary endpoint (VM). The secondary endpoint runs another secondary endpoint (Trusted OS SP).

The term *scheduler* is used in the context of CPU cycle allocation when it is not necessary to distinguish whether cycles are allocated by the primary or secondary scheduler.

Figure 2.6 illustrates an example of a primary endpoint. The primary scheduler manages threads that run execution contexts of VMs and SPs along with application threads. Application threads could in turn, run execution contexts of VMs and SPs as well.

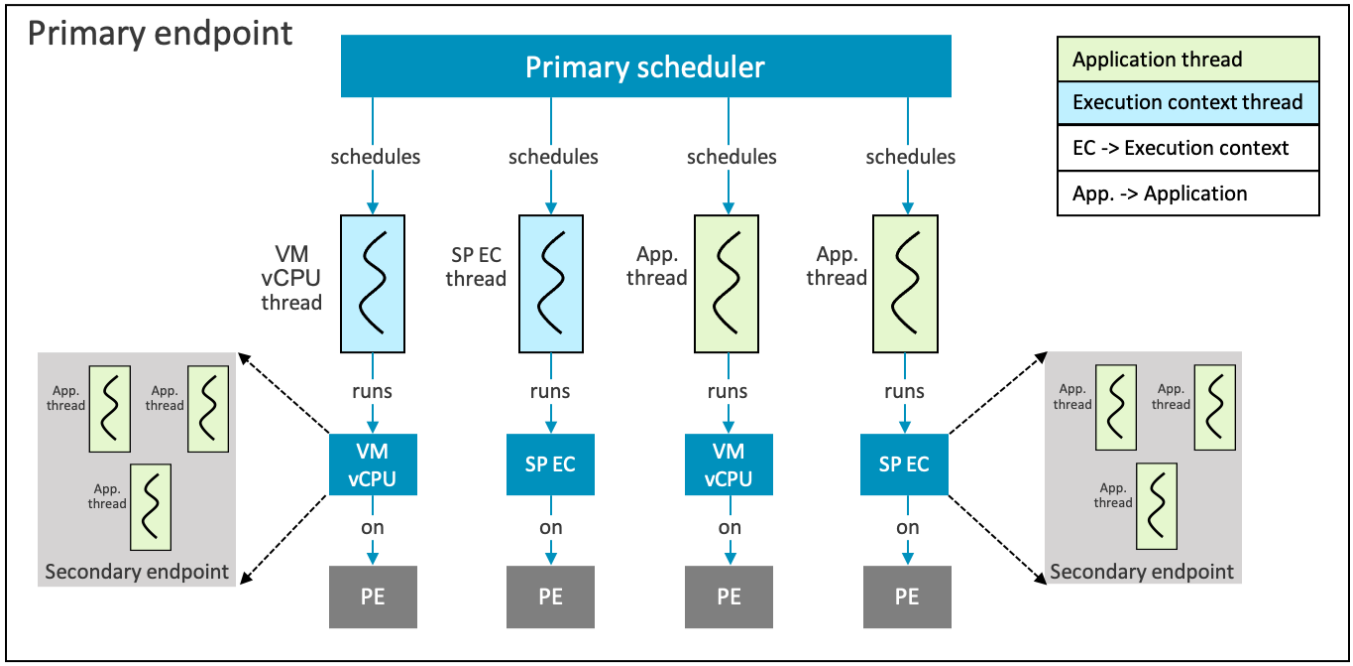

Figure 2.6: Example primary endpoint configuration

Figure 2.7 illustrates this example of a secondary endpoint. The secondary scheduler manages application threads, that could in turn, run execution contexts of SPs.

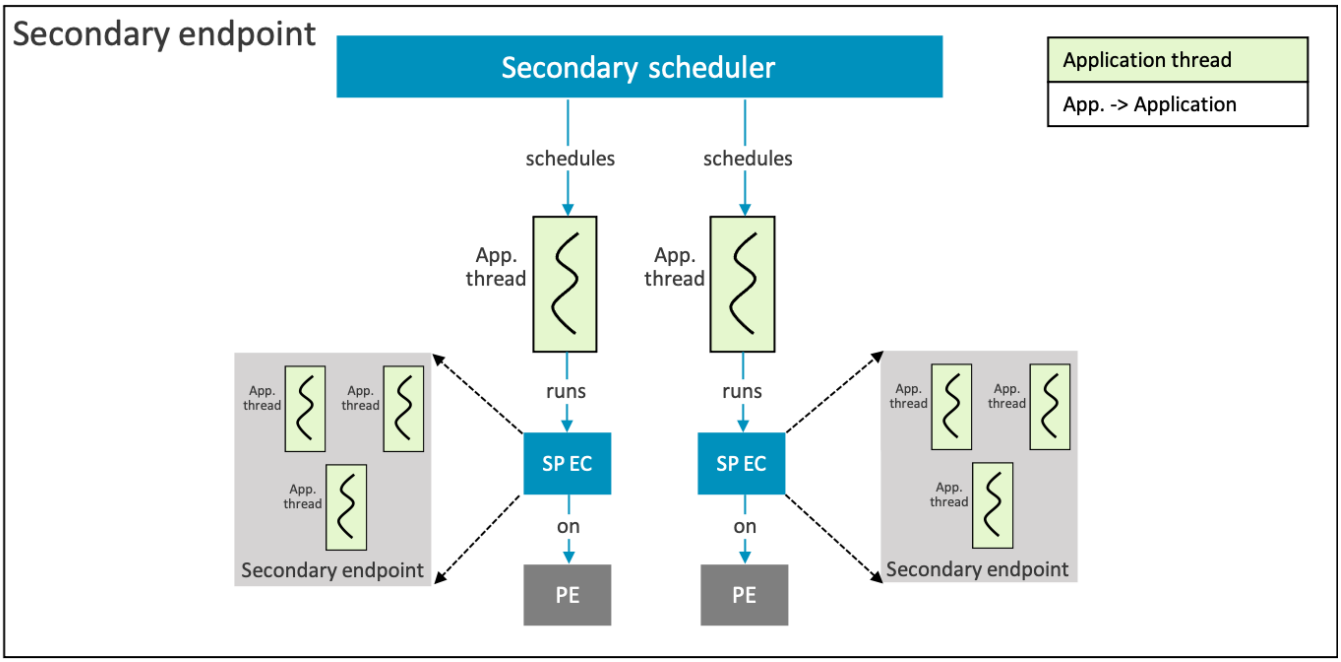

Figure 2.7: Example secondary endpoint configuration

Secondary endpoint services could be accessed during boot before the primary endpoint or Hypervisor is initialized. For example, a boot loader in the Normal world could access services provides by a SP.

The Framework assumes that the software components that perform boot subsume the role of the primary scheduler

before the Hypervisor or primary endpoint is initialized. Ownership of CPU cycles is relayed from one component to the next across the boot stages. Each component lends cycles to an endpoint if it accesses the services of the endpoint.

The Framework provides two ABIs to endpoints to allocate CPU cycles to other endpoints. These are,

1. FFA_MSG_SEND_DIRECT_REQ. See 13.2 *FFA_MSG_SEND_DIRECT_REQ*.
2. FFA_RUN. See 12.3 *FFA_RUN*.

2.12 Run-time states

Run-time refers to the stage during system boot when all the endpoints are initialized and application threads in an endpoint can access services implemented in other endpoints or partition managers through FF-A functions.

During run-time, the execution context of an endpoint can be in one of the following states from its perspective and that of the primary endpoint, SPM, and Hypervisor.

- *Waiting*. The execution context is waiting to be allocated CPU cycles to do work.
- *Running*. The execution context has been allocated CPU cycles and is doing work for example, running an application thread to process one or more messages.
- *Preempted*. The execution context was preempted by an interrupt while doing work.
- *Blocked*. The execution context is waiting for some work to complete on its behalf. It remains in this state until control is transferred back to it.

Transitions between these states are constrained by the following rules.

- An execution context in the *waiting* state only transitions to the *running* state.
- An execution context in the *running* state can transition to any other state.
- An execution context in the *blocked* state can only transition to the *running* state.
- An execution context in the *preempted* state only transitions to the *running* state.

An FF-A component could maintain additional IMPLEMENTATION DEFINED states. These are beyond the scope of this specification.

Guidance on transitions between these states is specified in 2.13 *Run-time state transitions*.

2.13 Run-time state transitions

FF-A ABIs are invoked with one or both of the SMC (as well as HVC and SVC) and ERET conduits (see 2.4 *Conduits*). Use of a conduit with or without an invocation of these ABIs triggers a *state transition*.

- An endpoint execution context uses the SMC, HVC and SVC conduits to trigger a state transition.
- A partition manager uses the ERET conduit to trigger a state transition for an execution context of an endpoint it manages.
- An interrupt that preempts an execution context in the *running* state also triggers a state transition.

State transitions based on states described in 2.12 *Run-time states* are of the following types.

1. Transitions that transfer control from one endpoint execution context to another and vice-versa. The interfaces whose invocation results in these transitions are listed below.

    1. FFA_MSG_SEND_DIRECT_REQ
    2. FFA_MSG_SEND_DIRECT_RESP
    3. FFA_RUN
    4. FFA_MSG_WAIT
    5. FFA_YIELD

    Each interface invocation is associated with two transitions.

    1. smc(Interface request)
    2. eret(Interface response)

    These transitions allow the endpoint execution to traverse between the *waiting*, *blocked* and *running* states.

2. Transitions that transfer control from an endpoint execution context to a Partition manager and back. The interfaces whose invocation results in these transitions are called *hypcalls*. These interfaces are listed below.

    - Partition setup and discovery interfaces in Chapter 11 *Setup and discovery interfaces*.
    - FFA_SECONDARY_EP_REGISTER interface in 16.3.2 *Secondary boot protocol*.
    - FFA_MSG_SEND2 messaging interface in Chapter 13 *Messaging interfaces*.
    - Memory management interfaces in
        - Chapter 14 *Memory management interfaces*.
        - 16.2 *Additional memory management features*.

    Each *hypcall* is associated with two transitions.

    1. smc(Hypcall request)
    2. eret(Hypcall response)

    A hypcall request transitions an endpoint execution context from the *running* to the *blocked* state.

    A hypcall response transitions an endpoint execution context from the *blocked* to the *running* state.

    A *hypcall* runs to completion between its two transitions from the perspective of the calling execution context.

3. Transitions that transfer control to an endpoint execution context in response to events such as a Secure interrupt or a power management message.

    A Secure interrupt could preempt another endpoint execution context. The latter enters the *preempted* state. Once the interrupt has been handled, the partition manager uses the eret() transition to put the endpoint execution context in the *running* state. Also see Chapter 6 *Interrupt management*.

    The Framework uses FFA_MSG_SEND_DIRECT_REQ and FFA_MSG_SEND_DIRECT_RESP interfaces to transmit power management messages between the SPMC and a SP execution context. These are described in 16.3.4 *Power Management messages*.

In both cases, the SP execution context enters the *running* state to handle the event.

Further guidance on state machines and runtime models is specified in Chapter 5 *Partition runtime models*.

Chapter 3
Setup

3.1 Overview

The Firmware Framework is responsible for partition and partition manager setup during a cold and warm boot (see 16.3 *Power Management*).

On the primary PE that performs the cold boot, execution contexts of partitions and partition managers are initialized to perform any system level setup.

In the Secure world,

- The SPMD initializes an execution context of the SPMC, if they reside in separate exception levels (see 2.2 *SPM architecture*).

  It could use the following guidance to perform the initialization.

  - The SPMC manifest to determine information such as the entry point address, execution state and Framework version of the SPMC (see 3.2.2 *Manifest for non-isolated partitions and SPMC*).
  - Guidance on programming general-purpose and system registers prior to invoking the SPMC entry point (see 3.3 *Register state*).
  - Protocol for passing any information to the SPMC (see 3.4 *Protocol for passing data*).
  - Protocol for indicating completion of initialization (see 3.5 *Protocol for completing execution context initialization*).

- The SPMC initializes each SP. An SP could co-reside with the SPMC (see 2.2.2.1 *S-EL1 SPM core component* and 2.2.2.2 *Secure Supervisor mode SPM core component*). The SPMC uses an IMPLEMENTATION DEFINED mechanism to initialize the SP. Information required to do this could be encoded in the SPMC manifest (see 3.2.2 *Manifest for non-isolated partitions and SPMC*).

An SP could reside in a separate exception level from the SPMC as follows.

- It has the same level of access to the physical address space as the SPMC. It is called a *non-isolated SP*. Also see 2.2.2.3 *EL3 SPM core component*.
- Its physical address space is isolated from the SPMC and other FF-A components. It is called an *isolated SP*. Also see,
  - 2.2.1 *SPM architecture with Secure EL2*.
  - 2.2.2.1 *S-EL1 SPM core component*.
  - 2.2.2.2 *Secure Supervisor mode SPM core component*.

The SPMC must use the SP manifest described in 3.2.1 *Manifest for isolated partitions* to initialize its execution context as follows.

1. Validate the contents of the manifest.
2. Configure the partition as per the properties described in the manifest.
3. Assign the requested physical address space ranges and system resources to the partition.
4. Isolate an *isolated SP* by ensuring it only has visibility of resources that it has requested.
5. Program the general-purpose and system register prior to invoking the SP entry point as described in 3.3 *Register state*.
6. Use protocol described in 3.4 *Protocol for passing data* for passing any information to the SP.
7. Use the runtime model described in 5.5 *Runtime model for SP initialization* to initialize the SP execution context.

In the Normal world,

- The Hypervisor or the OS kernel is initialized through an IMPLEMENTATION DEFINED mechanism after the Secure world hands control to the Normal world during cold boot.
- The Hypervisor initializes each VM through an IMPLEMENTATION DEFINED mechanism. It could use the guidance for initializing an *isolated SP* for doing this.

3.2 Manifests

3.2.1 Manifest for isolated partitions

The following information must be specified in the manifest of a partition.

- Partition properties as described in Table 3.1.
- Memory regions as described in Table 3.2.
- Devices as described in Table 3.3.
- Partition boot protocol as described in Table 3.10.

The following aspects of the partition manifest are IMPLEMENTATION DEFINED.

- Format of the manifest.
- Time of creation of manifest. This could be at:
  - Build time.
  - Boot time.
  - Combination of both.
- Mechanism used by the Hypervisor and SPM to obtain the information in the manifest and interpret its contents.

Table 3.1: Partition properties

Information fields	Mandatory	Description
FF-A version	Yes	• Version of FF-A expected by the partition at the FF-A instance it will execute.
UUID	Yes	• UUID of service implemented by this partition. • UUID can be shared by multiple instances of partitions that offer the same service. • For example, – If there are multiple instances of a Trusted OS, then the UUID can be shared by all instances. – The TEE driver in the HLOS can use the UUID with the *FFA_PARTITION_INFO_GET* interface to determine the: ∗ Number of Trusted OSs. ∗ The partition ID of each instance of the Trusted OS.
Partition ID	No	• Pre-allocated partition ID.
Auxiliary IDs	No	• List of pre-allocated 16-bit IDs that could be used in memory management transactions to allow a partition manager to handle the transaction in an IMPLEMENTATION DEFINED manner.
Name	No	• Name of the partition for example, for debugging purposes.

Information fields	Mandatory	Description
Number of execution contexts	Yes	• Number of vCPUs that a VM or SP wants to instantiate. • In the absence of virtualization, this is the number of execution contexts that a partition implements. • If value of this field = 1 and number of PEs > 1 then the partition is treated as UP & migrate capable. • If the value of this field > 1 then the partition is treated as an MP capable partition irrespective of the number of PEs.
Run-time EL	Yes	• EL1 or Secure EL1. • Secure EL0.
Execution state	Yes	• AArch64. • AArch32.
Load address	No	• Absence of this field indicates that the partition is position independent and can be loaded at any address chosen at boot time.
Entry point offset	No	• Absence of this field indicates that the entry point is at offset 0x0 from the base of the partition binary image. • If present, this field specifies the offset of the entry point from the base of the partition binary image.
Translation Granule	No	• 4KB (default value if not specified). • 16KB. • 64KB.
Boot order	No	• A unique number among all partitions that specifies if this partition must be booted before others. • For example, a partition could provide a service that other partitions need to initialize themselves. The manifest of this partition can use this field to ensure it is booted before others.
RX/TX information	No	• Reference to memory region entries in this manifest that describes the RX/TX buffers expected by the partition. • The memory region entries must specify the base addresses of both buffers. • The size and attributes fields must fulfill the requirements specified in 4.2.2.3 *Buffer attributes*.
Messaging method	Yes	• This field specifies which messaging methods are supported by the partition. This could be one or both of direct and indirect messaging. These methods are described in Chapter 4 *Message passing*. The following information must be provided in the manifest: • Indirect messaging is supported. This always includes support for both sending and receiving indirect messages. • Direct messaging is supported. 4.4.1 *Discovery and setup* specifies the information that must be provided. • Managed exits are supported. 6.4.1 *Managed exit* specifies the information that must be provided.

Information fields	Mandatory	Description
Notification support	No	• This field specifies if the partition supports receipt of notifications as described in Chapter 7 *Notifications*. • Absence of this field indicates that the partition cannot receive notifications.
Primary Scheduler implemented	No	• Presence of this field indicates that the partition implements the primary scheduler. • Run-time EL must be EL1 if this field is specified.
Run-time model	No	• If the run-time EL is S-EL0 or User mode then this field specifies the run-time model that the SPM must enforce for this SP. – *Run to completion*. SP execution must not be preempted. An execution context of this SP must only transition between the *waiting* and *running* states described in 2.12 *Run-time states*. – *Preemptible*. SP execution can be preempted. An execution context of this SP can transition between all states described in 2.12 *Run-time states*. This is the default run-time model for a S-EL0/User mode SP if this field is not specified in the partition manifest. • This field is deprecated in v1.1 of the Framework in favor of the Message processing and Secure interrupt handling scheduling model fields. If specified, the SPMC must convert the specified run-time model into the appropriate scheduling model.
Message processing scheduling model	No	• This field specifies the actions that the SPMC must take in response to interrupts while processing a message as described in 6.5.1 *Overview*. • This is a mandatory field if the partition does not specify the *Secure Interrupt handling scheduling model*.
Secure interrupt handling scheduling model	No	• This field specifies the actions that the SPMC must take in response to interrupts while handling a Secure interrupt as described in 6.5.1 *Overview*. • This is a mandatory field if the partition does not specify the *Message processing scheduling model*.
Tuples of (Name, SEPID , SMMU ID, Stream IDs)	No	• If present, then each tuple specifies the association between its members that the partition manager must create. The members are as follows. – *Stream endpoint* ID that this endpoint is a *proxy* for. The dependent device must not be assigned to this endpoint (see 8.2.1 *Stream endpoint*). – *SMMU ID* identifies the SMMU instance on a system with multiple SMMUs. – One or more *Stream IDs* associate the device that generates them with the *SEPID* in the SMMU identified by *SMMU ID*. – An optional *Name* for the SEPID for debugging purposes.
Power management messages	No	• This field specifies the power management messages the SP is interested in receiving. See 16.3.4 *Power Management messages*.

Information fields	Mandatory	Description
Cold boot reason register	No	• Presence of this field indicates that the partition expects that the entry point offset field must be reused for a secondary cold boot (see 16.3 *Power Management* and 16.3.2 *Secondary boot protocol*). • The reset reason is encoded in a general-purpose register as follows. – Value of 0 in the register indicates a primary cold boot. – Value of 1 in the register indicates a secondary cold boot. • The register is specified in this field. Register must be between *w0/x0-w7/x7*. The width of the register is derived from its Execution state specified in the partition manifest.

Table 3.2: Memory regions

Information fields	Mandatory	Description
Base address	No	• Absence of this field indicates that a memory region of specified size and attributes must be mapped into the partition translation regime. The PM must describe the memory region to the partition through an IMPLEMENTATION DEFINED mechanism. • If present, this field could specify a PA, VA (for S-EL0 partitions) or IPA (for S-EL1 and EL1 partitions). This information must be specified using an IMPLEMENTATION DEFINED mechanism. – If a PA is specified, then the memory region must be identity mapped with the same IPA or VA as the PA. – If a VA or IPA is specified, then the memory could be identity or non-identity mapped. • If present, the address must be aligned to the Translation granule size.
Page count	Yes	• Size of memory region expressed as a count of 4K pages. • For example, if the memory region size is 16K, value of this field is 4.
Attributes	Yes	• Memory access permissions. – Instruction access permission. – Data access permission. • Memory region attributes. – Memory type. – Shareability attributes. – Cacheability attributes. • Memory Security state. – Non-secure for a NS-Endpoint. – Non-secure or Secure for an S-Endpoint.
Name	No	• Name of the memory region for example, for debugging purposes.

Table 3.3: Device regions

Information fields	Mandatory	Description
Physical base address	Yes	• PA of base of a device MMIO region. • If the MMIO region is not physically contiguous, then an entry for each physically contiguous constituent region must be specified. • Each entry must specify the PA and size of the constituent region. The size must be expressed as a count of 4K pages.
Page count	Yes	• Total size of MMIO region expressed as a count of 4K pages. • For example, if the MMIO region size is 16K, value of this field is 4.
Attributes	Yes	• Memory attributes must be Device-nGnRnE. • Instruction access permission must be not executable. • Data access permissions must be one of the following: – Read/write. – Read-only. • Security attributes must be: – Non-secure for a NS-Endpoint. – Non-secure or Secure for an S-Endpoint.
Interrupts	No	• List of physical interrupt IDs. • Attributes of each interrupt ID. – Interrupt type. ∗ SPI. ∗ PPI. ∗ SGI. – Interrupt configuration. ∗ Edge triggered. ∗ Level triggered. – Interrupt Security state. ∗ Secure. ∗ Non-secure. – Interrupt priority value. – Target execution context/vCPU for each SPI. ∗ This field is optional even if other interrupt properties are specified since interrupt affinity could be managed through an IMPLEMENTATION DEFINED interface between the endpoint and its partition manager.
SMMU ID	No	• If present, then on a system with multiple SMMUs, this field must help the partition manager determine which SMMU instance is this device upstream of. • Absence of this field implies that the device is not upstream of an SMMU.
Stream IDs	No	• List of Stream IDs assigned to this device. • Absence of Stream ID list indicates that the device is not upstream of an SMMU.

Information fields	Mandatory	Description
Exclusive access and ownership	No	• If present, this field implies that this endpoint must be granted exclusive access and ownership of the MMIO region of the device. • Absence of this field implies that access to the MMIO region of the device could be shared among multiple endpoints.
Name	No	• Name of the device region for example, for debugging purposes.

3.2.2 Manifest for non-isolated partitions and SPMC

The following aspects of the partition manifest are IMPLEMENTATION DEFINED.

- Format of the manifest.
- Time of creation of the manifest. This could be at:
  - Build time.
  - Boot time.
  - Combination of both.
- Mechanism used by the SPMD to obtain information in the manifest and interpret its contents.

Table 3.4: Properties of a non-isolated SP or SPMC

Information fields	Mandatory	Description
FF-A version	Yes	• Version of Firmware Framework implemented by the SPMC component. See 11.1 *FFA_VERSION* for more information about the usage of this field.
UUID	No	• UUID to identify the single SP.
SP and/or SPMC ID	No	• Pre-allocated ID of the SP and/or the SPMC.
Name	No	• Name of the partition for example, for debugging purposes.
Execution state	Yes	• AArch64. • AArch32.
Load address	No	• Absence of this field indicates that the SPMC image is position independent and can be loaded at any address chosen at boot time.
Entry point offset	No	• Absence of this field indicates that the entry point is at offset 0x0 from the base of the SPMC binary image. • If present, this field specifies the offset of the entrypoint from the base of the SP binary image.
FF-A boot protocol usage	No	• See Table 3.10.

Information fields	Mandatory	Description
Power management messages	No	• This field specifies the power management messages the SP is interested in receiving. See 16.3.4 *Power Management messages*.
Cold boot reason register	No	• Presence of this field indicates that the partition expects that the entry point offset field must be reused for a secondary cold boot (see 16.3 *Power Management* and 16.3.2 *Secondary boot protocol*). • The reset reason is encoded in a general-purpose register as follows. – Value of 0 in the register indicates a primary cold boot. – Value of 1 in the register indicates a secondary cold boot. • The register is specified in this field. Register must be between *w0/x0-w7/x7*. The width of the register is derived from its Execution state specified in the partition manifest.
Notification support	No	• This field specifies if the non-isolated partition supports receipt of notifications as described in Chapter 7 *Notifications*. • Absence of this field indicates that the partition cannot receive notifications.

3.2.3 Independent peripheral device manifest

This manifest must be used by *independent peripheral devices* to describe their properties to a partition manager. See 8.2 *Direct memory access* for more details.

Table 3.5: Device properties

Information fields	Mandatory	Description
FF-A version	Yes	• Version of the Firmware Framework expected by the device.
Name	No	• Name of the partition for example, for debugging purposes.
Translation Granule	Yes	• 4KB. • 16KB. • 64KB.
SEPID	Yes	• Pre-allocated Stream endpoint ID.

Table 3.6: Memory regions accessible by the device

Information fields	Mandatory	Description
Base address	Yes	• This field could specify a PA or IPA. This distinction must be specified using an IMPLEMENTATION DEFINED mechanism. – If a PA is specified, then the memory region must be identity mapped with the same IPA as the PA. – If an IPA is specified, then the memory could be identity or non-identity mapped. • The address must be aligned to the Translation granule size.
Page count	Yes	• Size of memory region expressed as a count of 4K pages. • For example, if the memory region size is 16K, value of this field is 4.
Properties	Yes	• Memory region properties (see 8.11 *Memory region properties*). • Security attributes. – Non-secure for a Non-secure device. – Non-secure or Secure for a Secure device.
Name	No	• Name of the memory region for example, for debugging purposes.

Table 3.7: Device regions

Information fields	Mandatory	Description
Physical base address	Yes	• PA of base of a device MMIO region. • If the MMIO region is not physically contiguous, then an entry for each physically contiguous constituent region must be specified. • Each entry must specify the PA and size of the constituent region. The size must be expressed as a count of 4K pages.
Properties	Yes	• Memory type must be Device-nGnRnE. • Instruction access permission must be not executable. • Data access permissions must be one of the following: – Read/write. – Read-only. • Security attributes must be: – Non-secure for a Non-secure device. – Non-secure or Secure for a Secure device.
Page count	Yes	• Total size of MMIO region expressed as a count of 4K pages. • For example, if the MMIO region size is 16K, value of this field is 4.
SMMU ID	Yes	• On a system with multiple SMMUs, this field must help a partition manager determine which SMMU instance is this device upstream of.

Information fields	Mandatory	Description
Stream IDs	Yes	• List of Stream IDs assigned to this device.
Name	No	• Name of the device region for example, for debugging purposes.

3.3 Register state

The partition manager must program system and general-purpose registers that influence partition execution as follows.

- The MMU must be disabled for a partition that does not run in S-EL0 in either Execution state. The MMU must be enabled for S-EL0 partition that runs in either Execution state.
- The partition manager must ensure that all memory regions allocated to a partition are clean to the Point of Coherency. Also, there must be no stale cached copies of executable memory held in any instruction caches visible to a PE on which the execution contexts of the partition may execute.

  This could be achieved by executing cache maintenance instructions, after initializing the memory regions for a partition.
- The state of other System registers is IMPLEMENTATION DEFINED. If the partition manager must program a System register to fulfill a specific partition requirement then this must be encoded in its manifest through an IMPLEMENTATION DEFINED mechanism.
  - For example, an S-EL0 partition could want the instruction alignment check to be disabled by setting SCTLR_EL1.A, bit[1] = b'0.
- The state of general-purpose registers is IMPLEMENTATION DEFINED. Also see Table 3.10.

3.4 Protocol for passing data

The partition manager could also pass an array of *name-value-size* pairs to a partition execution context when it is entered. This information is encoded in an initialization descriptor specified in Table 3.9.

- The partition must specify the information it expects to be populated in an initialization descriptor in its manifest through an IMPLEMENTATION DEFINED mechanism.
- In this version of the Firmware Framework, it is assumed that information in an initialization descriptor is passed only to that execution context of a partition which is initialized by the partition manager on the primary PE.
- The partition manager must fulfill the following requirements for the memory region where an initialization descriptor is populated.
  - Size of memory region must be a multiple of the translation granule size used by the partition.
  - Address of memory region must be aligned to the translation granule size used by the partition.
  - The memory region must be mapped in the translation regime of the partition that is managed by the Hypervisor (see 2.1 *Partition manager*) or SPM (see 2.2 *SPM architecture*).
  - The memory region must be mapped with the same memory attributes as the RX/TX buffers as described in 4.2.2.3 *Buffer attributes* in the partition translation regime managed by the Hypervisor or SPM.
  - Boot information must be populated at offset 0 in the memory region
  - The address of boot information must be passed in the general-purpose register specified in the partition manifest (see Table 3.10).

Table 3.8: Name value size tuple descriptor

Field	Byte length	Byte offset	Description
Name	16	0	• Name of an object passed to the partition.
Value	8	16	• Value of the object identified by the *Name* field.
Size	8	24	• Size of the object identified by the *Name* field.

Table 3.9: Initialization descriptor

Field	Byte length	Byte offset	Description
Signature	4	0	• ASCII string *"FF-A"* to identify this descriptor.
NVS count	4	4	• Count of *Name value size* tuple descriptors.
NVS array	–	8	• Array of *Name value size* tuple descriptors. See Table 3.8.

Table 3.10: Boot protocol information

Information fields	Mandatory	Description
FF-A boot protocol usage	No	• Presence of this field indicates that the partition expects the address of an initialization descriptor to be passed in a general-purpose register (see Table 3.9). • The register in which the address of an initialization descriptor will be passed must be specified. Register must be between *w0/x0-w3/x3*. The width of the register is derived from its Execution state specified in the partition manifest.

3.5 Protocol for completing execution context initialization

A partition must use the *FFA_MSG_WAIT* (also see 12.1 *FFA_MSG_WAIT*) interface or an IMPLEMENTATION DEFINED mechanism to indicate completion of initialization of its execution context to the partition manager.

A partition must use the *FFA_ERROR* (also see 10.2 *FFA_ERROR*) interface or an IMPLEMENTATION DEFINED mechanism to report an error during initialization of its execution context to the partition manager.

The runtime model that the SPMC uses for initializing an execution context of a SP is described in 5.5 *Runtime model for SP initialization*.

Chapter 4
Message passing

4.1 Overview

The Firmware Framework defines a set of ABIs that enable FF-A components to exchange messages with each other. A message exchange comprises of the following phases.

1. Transmission of the message payload from the Sender to the Receiver. The mechanisms specified by the Framework to do this are described in 4.2 *Message transmission*.
2. Allocation of CPU cycles to the Receiver to process the message on a PE in the system. Cycles could be allocated by,
    1. The scheduler implemented in the Sender. This method is described in 4.1.2 *Direct messaging*.
    2. A scheduler implemented in another FF-A component in the Normal world. This method is described in 4.1.1 *Indirect messaging*.
3. Message processing using the allocated cycles. The role of the Framework during message processing is described in Chapter 5 *Partition runtime models*.

Any FF-A component can send or receive messages. In any message exchange between two endpoints, one or both partition managers validate and forward the message from the sender to the receiver. A partition manager is called the *Relayer* when it performs this role. In all messaging scenarios, in the absence of a Hypervisor, the SPM subsumes its responsibilities. In a configuration with the S-EL1 or S-Supervisor SPMC, the role of the SPM as a Relayer is subsumed by the SP.

4.1.1 Indirect messaging

In this method, the message Sender requests a scheduler in another NS-endpoint or Hypervisor to allocate CPU cycles to the Receiver for processing the message. The Sender could make progress concurrently with the Receiver either on the same or a different PE. The Sender either polls or is notified when a response from the Receiver is available.

The term **Indirect messaging** is used to describe this method for CPU cycle allocation along with interfaces to transmit the message payload. A detailed description of the usage of this method is provided in 4.3 *Indirect messaging usage*. Figure 4.1 illustrates this method. It assumes that both the Sender and Receiver run on the same PE.

The *FFA_MSG_SEND2* ABI is used by a Sender to send a message to a Receiver using indirect messaging. Also see 13.1 *FFA_MSG_SEND2*.

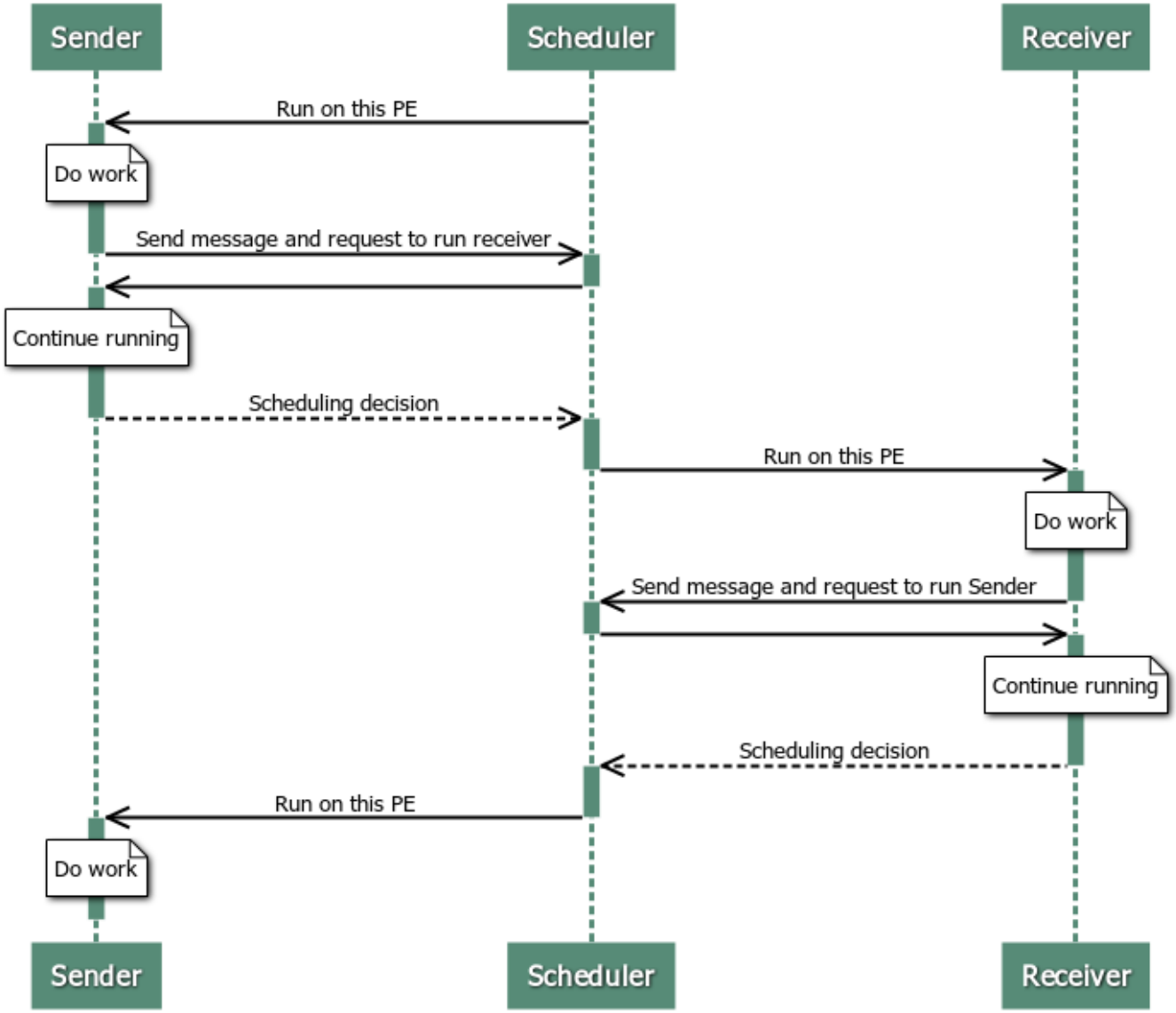

Figure 4.1: Indirect messaging

4.1.2 Direct messaging

In this method, the message Sender lends CPU cycles to the Receiver so that it can make progress. The Receiver is *scheduled* by the Sender. The Sender does not make progress until either a response is returned, or execution is returned back to it. Both the Sender and Receiver run on the same PE.

The term **Direct messaging** is used to describe this method for CPU cycle allocation along with interfaces to transmit the message payload. A detailed description of this method is provided in 4.4 *Direct messaging usage*. Figure 4.2 illustrates this method.

This method is used for messaging between an endpoint and the Hypervisor or SPM. It is also used for messaging between endpoints in the following scenarios.

- The scheduler is not available. For example,
  - It is not initialized.

  - It is not possible to communicate with the scheduler.
- The latency associated with scheduler communication cannot be tolerated by the use case.
- The Receiver must be run on the same PE as the Sender.

The Framework allows this method to be used for passing messages in only the following scenarios.

- Between FF-A endpoints under the constraints described in 4.4 *Direct messaging usage*.
- From an endpoint to the Hypervisor or SPM.
- From the Hypervisor or SPM to an endpoint.
- Between the Hypervisor and SPM.

The following ABIs are used to implement direct messaging between endpoints.

- *FFA_MSG_SEND_DIRECT_REQ*. This interface is used by a Sender to send a request message payload to a Receiver, lend CPU cycles to the Receiver and wait for a response to arrive. Also see 13.2 *FFA_MSG_SEND_DIRECT_REQ*.
- *FFA_MSG_SEND_DIRECT_RESP*. This interface is used by a Sender to send a response message payload to a Receiver, return CPU cycles to the Receiver and wait for a new message to arrive. Also see 13.3 *FFA_MSG_SEND_DIRECT_RESP*.
- *FFA_INTERRUPT*. This interface is used by the Relayer to inform the Sender that direct message processing in the Receiver was preempted.
- *FFA_RUN*. This interface is used by the Sender to resume a preempted Receiver.

All other ABIs specified by the Framework use direct messaging for communication between endpoints and Hypervisor or SPM and, between the Hypervisor and SPM.

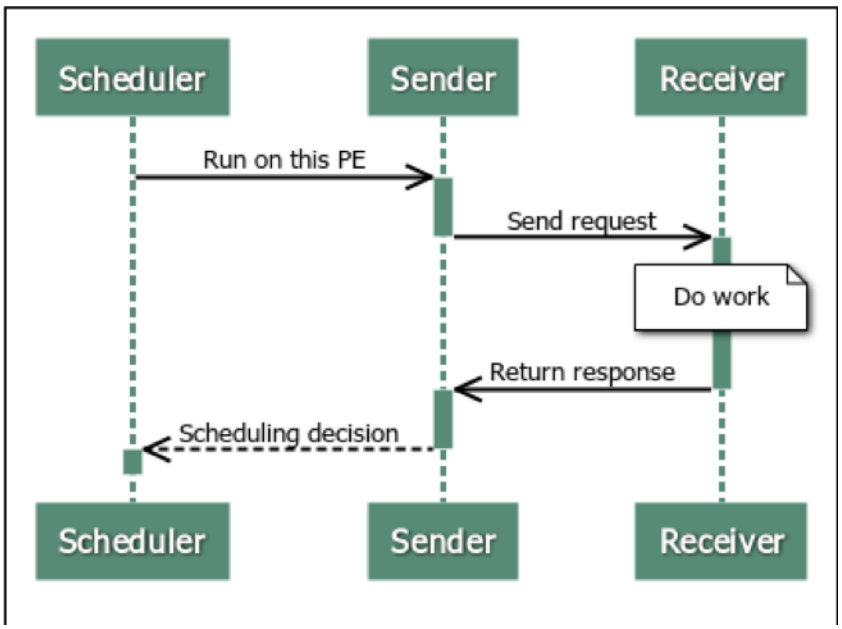

Figure 4.2: Direct messaging

4.2 Message transmission

4.2.1 Overview

Message payloads are exchanged between two FF-A components through general purpose registers and/or a single pair of shared memory regions to transmit and receive messages called *RX/TX buffers* (see also 4.2.2 *RX/TX buffers*).

- Direct messaging can use both these mechanisms along with the ABIs described in 4.1.2 *Direct messaging*.
- Indirect messaging must use only the RX/TX message buffers along with the ABIs described in 4.1.1 *Indirect messaging*.

The Framework defines a message as all information encoded in,

1. The input parameter registers *w0/x0-w7/x7* in an FF-A ABI definition.
2. The RX/TX buffers if they are used in an FF-A ABI definition.

Each message has a header and a payload. The header describes properties of a message such as,

- Type of message.
  - E.g., the *function ID* parameter in w0/x0.
- Source and target of the message.
  - E.g., the *source and target endpoint* parameters in w1 in FFA_MSG_SEND_DIRECT_REQ.
- Size of the message.
  - E.g., the *total length* parameter in w1 in FFA_MEM_SHARE.

The version of the message header and payloads is the same as the version of the Firmware Framework as returned by **FFA_VERSION** (see 11.1 *FFA_VERSION*).

The header is encoded in the parameter registers, RX/TX buffers or both. This depends upon the ABI definition. The Framework uses the message header to decide how it must handle the message. For example, in response to an FF-A ABI invocation, a partition manager decides if it must interpret the message payload.

There are two types of messages.

1. Messages with payloads that are defined by the communication Framework for example, memory management messages. They have the same definition in any implementation of a particular version of the Firmware Framework. Messages with these payloads are called **Framework messages**.

   Framework message payloads can be interpreted by the Relayer, Sender and Receiver. They are used when:

   - Relayer participation is required to validate or modify message contents before delivery to the Receiver.
   - The Hypervisor or SPM is the destination of the message payload. It processes the message and provides a response.

   In this version of the Firmware Framework, Framework messages are exchanged only in the following scenarios.

   - Between an endpoint and Hypervisor or SPM.
   - Between the Hypervisor or SPM and an endpoint.
   - Between the Hypervisor and SPM.
   - Between the SPM and Hypervisor.

2. Messages with payloads that are defined by the services implemented inside a partition. The format of these messages is specific to the service or partition implementation. Messages with these payloads are called **Partition messages**.

   Partition message payloads are only interpreted by the Sender and Receiver endpoints. A Relayer only uses the header information to route them correctly. Hence, by definition these messages are only exchanged between partitions.

The properties of Framework and Partition messages influence direct and indirect messaging as follows.

- Direct messaging can be used to transmit both Framework and partition messages. Framework messages can be transmitted in both RX/TX buffers and registers. Partition messages can only be transmitted in registers.
- Indirect messaging can be used to only transmit Partition messages in the RX/TX buffers.

Table 4.1 lists valid combinations of the following attributes of a message exchange.

1. Messaging method.
2. Message type.
3. Message payload location.

It also lists examples of ABIs used to transmit messages for a valid combination of these attributes.

Table 4.1: Combinations of messaging and message transmission mechanisms

Messaging method	Message type	Message payload location	Message transmission interface
Direct	Partition	Register	• FFA_MSG_SEND_DIRECT_REQ. • FFA_MSG_SEND_DIRECT_RESP.
Direct	Partition	RX/TX	• Invalid usage.
Direct	Framework	Register	• Any interface to send or receive information from the Hypervisor or SPM for example, – FFA_VERSION. – FFA_RX_RELEASE. – FFA_YIELD. – FFA_RXTX_MAP. – FFA_RXTX_UNMAP. – FFA_RUN.
Direct	Framework	RX/TX	• Any interface to send or receive information from the Hypervisor or SPM for example, – FFA_MEM_DONATE. – FFA_MEM_SHARE. – FFA_MEM_LEND. – FFA_MEM_RELINQUISH. – FFA_MEM_RETRIEVE_REQ. – FFA_MEM_RETRIEVE_RESP. – FFA_MEM_RECLAIM.
Indirect	Partition	Register	• Invalid usage.
Indirect	Partition	RX/TX	• FFA_MSG_SEND2.
Indirect	Framework	Register	• Invalid usage.
Indirect	Framework	RX/TX	• Invalid usage.

4.2.2 RX/TX buffers

A RX/TX buffer pair is shared between two FF-A components at an FF-A instance.

- The FF-A component at the lower EL is the *Consumer* of the RX buffer and *Producer* of the TX buffer.
- The FF-A component at the higher EL is the *Producer* of the RX buffer and the *Consumer* of the TX buffer.

The endianness of all message payloads populated in the RX/TX buffers is *little-endian*.

In the Normal world,

- Each VM has a Non-secure buffer pair. It is shared with the Hypervisor and SPMC.
- The OS kernel has a Non-secure buffer pair. It is shared with the SPMC.
- The Hypervisor has a Non-secure buffer pair. It is shared with the SPMC.

In the Secure world,

- Each SP has a Secure buffer pair. It is shared with the SPMC.
- The SPM is split into the SPMD and SPMC components as described in 2.2 *SPM architecture*. In configurations where the SPMC resides in a separate Exception level from the SPMD (see Table 2.1 & Table 2.2), it is IMPLEMENTATION DEFINED whether the two SPM components share an RX/TX buffer pair.

These message buffer configurations are illustrated in Figure 4.3.

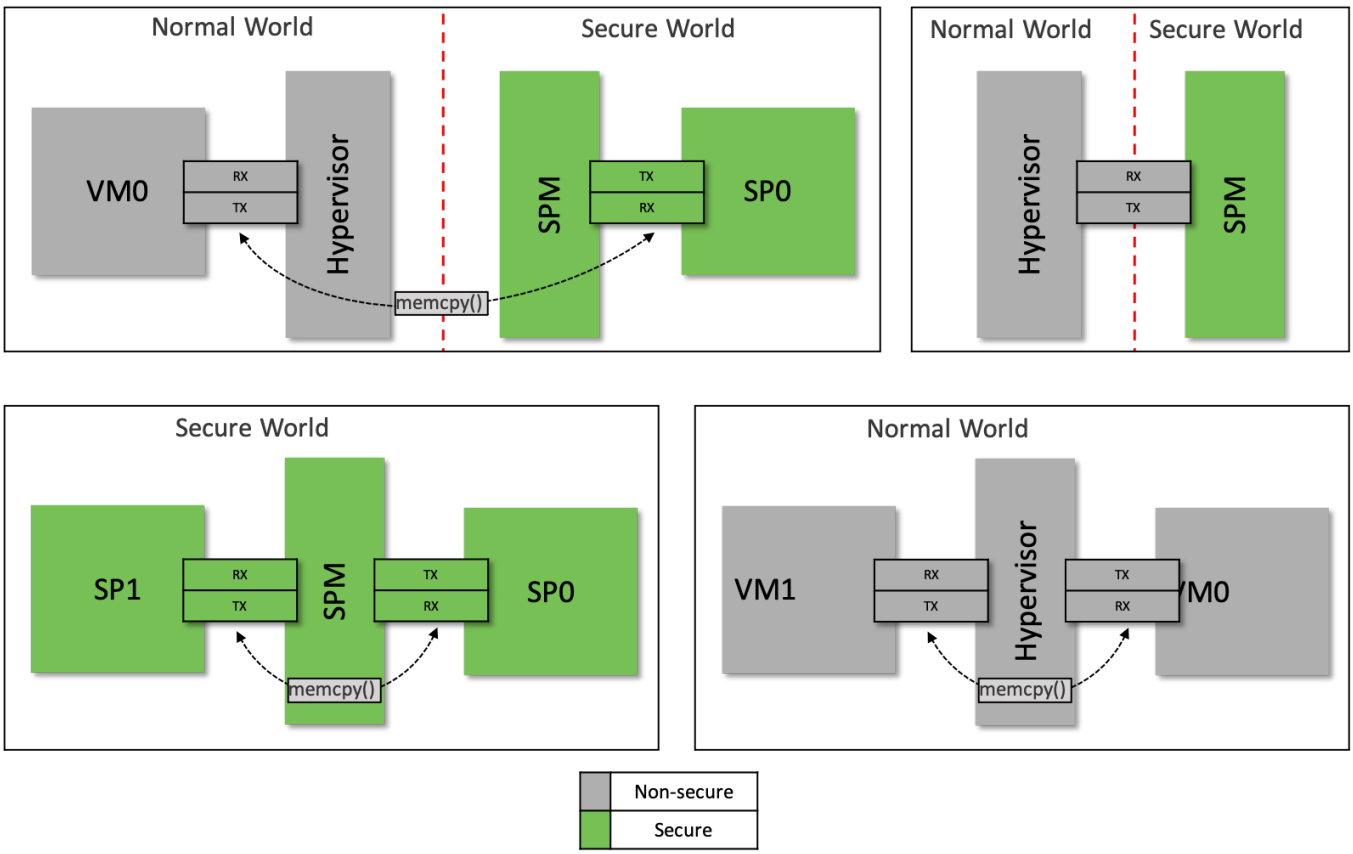

Figure 4.3: Configurations of RX/TX buffer pair between FF-A components

Mechanisms for message transmission through RX/TX buffers are described in 4.2.2.1 *Buffer-based message transmission*.

Mechanisms for discovery and setup of a RX/TX buffer pair are described in 4.2.2.2 *Buffer setup*.

Requirements for correctly mapping a RX/TX buffer pair in the translation regimes of both FF-A components at any FF-A instance are described in 4.2.2.3 *Buffer attributes*.

4.2.2.1 Buffer-based message transmission

4.2.2.1.1 Transmission of partition messages

The following common rules govern transmission of partition messages.

1. Partition messages are populated at the base of a TX or RX buffer as per the encoding described in Table 4.2.

   This encoding avoids the use of the FFA_MSG_POLL interface by the Consumer of a RX buffer to determine the identity of the message Sender and message length.

2. The FFA_MSG_SEND2 ABI is used transmit a partition message from the TX buffer of the Sender endpoint to the RX buffer of the Receiver endpoint.
3. A message is transmitted between VMs by copying it from the TX buffer of the Sender VM to the RX buffer of the Receiver VM. The message copy is done by the Hypervisor.
4. A message is transmitted between SPs by copying it from the TX buffer of the Sender SP to the RX buffer of the Receiver SP. The message copy is done by the SPMC.
5. A message is transmitted from a VM to a SP by copying it from the TX buffer of the Sender VM to the RX buffer of the Receiver SP. The invocation of FFA_MSG_SEND2 is forwarded by the Hypervisor to the SPMC. The message copy is done by the SPMC.
6. A message is transmitted from a SP to a VM by copying it from the TX buffer of the Sender SP to the RX buffer of the Receiver VM. The message copy is done by the SPMC.

Table 4.2: Encoding of a partition message

Field	Byte length	Byte offset	Description
Flags	4	–	• Bits[31:0]: Reserved for future use. MBZ and ignored.
Reserved (MBZ)	4	4	
Message Offset	4	8	• Offset from the beginning of the buffer to the start of message payload.
Sender/Receiver IDs	4	12	• Sender and Receiver endpoint IDs. – Bits[31:16]: Sender endpoint ID. – Bits[15:0]: Receiver endpoint ID.
Message size	4	16	• Length of message in bytes in the RX buffer.

4.2.2.1.2 Transmission of framework messages

The following common rules govern transmission of framework messages.

1. A message is transmitted from a VM to the Hypervisor in the TX buffer of the Sender VM.
2. A message is transmitted from the Hypervisor to a VM in the RX buffer of the Receiver VM.
3. A message is transmitted from a VM to the SPMC in two steps.
   1. It is transmitted from the VM to the Hypervisor in the TX buffer of the Sender VM.
   2. It is transmitted from the Hypervisor to the SPMC in the TX buffer of the Hypervisor. The Hypervisor copies it from the Sender VM's TX buffer to its TX buffer.

4. A message is transmitted from the SPMC to a VM in two steps.
    1. It is transmitted from the SPMC to the Hypervisor in the RX buffer of the Hypervisor.
    2. It is transmitted from the Hypervisor to the VM in the RX buffer of the Receiver VM. The Hypervisor copies it from its RX buffer to the Receiver VM's RX buffer.
5. A message is transmitted from a SP to the SPMC in the TX buffer of the Sender SP.
6. A message is transmitted from the SPMC to a SP in the RX buffer of the Receiver SP.
7. A message is transmitted from the Hypervisor to the SPMC in the TX buffer of the Hypervisor.
8. A message is transmitted from the SPMC to the Hypervisor in the RX buffer of the Hypervisor.
9. Transmission of framework messages from a SP to the Hypervisor or a NS-endpoint is not supported in this version of the Framework.

Framework messages are transmitted as described above in invocations of the following ABIs.

1. FFA_MEM_DONATE
2. FFA_MEM_LEND
3. FFA_MEM_SHARE
4. FFA_MEM_RETRIEVE_REQ
5. FFA_MEM_RETRIEVE_RESP
6. FFA_MEM_RELINQUISH
7. FFA_RXTX_MAP
8. FFA_PARTITION_INFO_GET

4.2.2.2 Buffer setup

This version of the Framework enables setup of RX/TX buffer pairs between FF-A components as per the following rules.

1. Allocation of a buffer pair for an endpoint can be done by the endpoint or its partition manager.

    In the former case, the endpoint allocates the buffer pair and uses FFA_RXTX_MAP ABI (see 11.5 *FFA_RXTX_MAP*) to map it in the partition manager's translation regime.

    In the latter case,

    1. The partition manager manages a stage of address translation in the translation regime of the endpoint as described in 2.2 *SPM architecture*.
    2. The endpoint requests buffer allocation in its manifest by specifying their base addresses (as IPAs or VAs) and size as described in 3.2.1 *Manifest for isolated partitions*.
    3. The partition manager maps the buffer pair in the stage of translation regime it manages on behalf of the endpoint and its own translation regime.

    If the endpoint is a VM, in both cases, the Hypervisor uses the FFA_RXTX_MAP ABI to map the buffer pair in the SPMC's translation regime as well.
2. The Hypervisor allocates the buffer pair it shares with the SPM. It uses the FFA_RXTX_MAP ABI to map this buffer pair in the SPMC's translation regime.
3. Buffer pairs shared between the SPMC and a SP are not visible to an FF-A component in the Normal world.
4. An endpoint uses the FFA_RXTX_UNMAP ABI (see 11.6 *FFA_RXTX_UNMAP*) to unmap the buffer pair from the partition manager's translation regime.

    If the endpoint is a VM, the Hypervisor uses the FFA_RXTX_UNMAP ABI to unmap the buffer pair from the SPMC's translation regime as well.
5. The Hypervisor uses the FFA_RXTX_UNMAP ABI to unmap the buffer pair it shares with the SPMC from the SPMC's translation regime.

Figure 4.4 illustrates an example RX/TX buffer setup where the:

- SPM allocates the buffer pair on behalf of the SP.
- Hypervisor registers its buffer pair with the SPM.
- VM allocates and registers its buffer pair with the Hypervisor and SPM.
- VM unregisters its buffer pair with the Hypervisor and SPM.

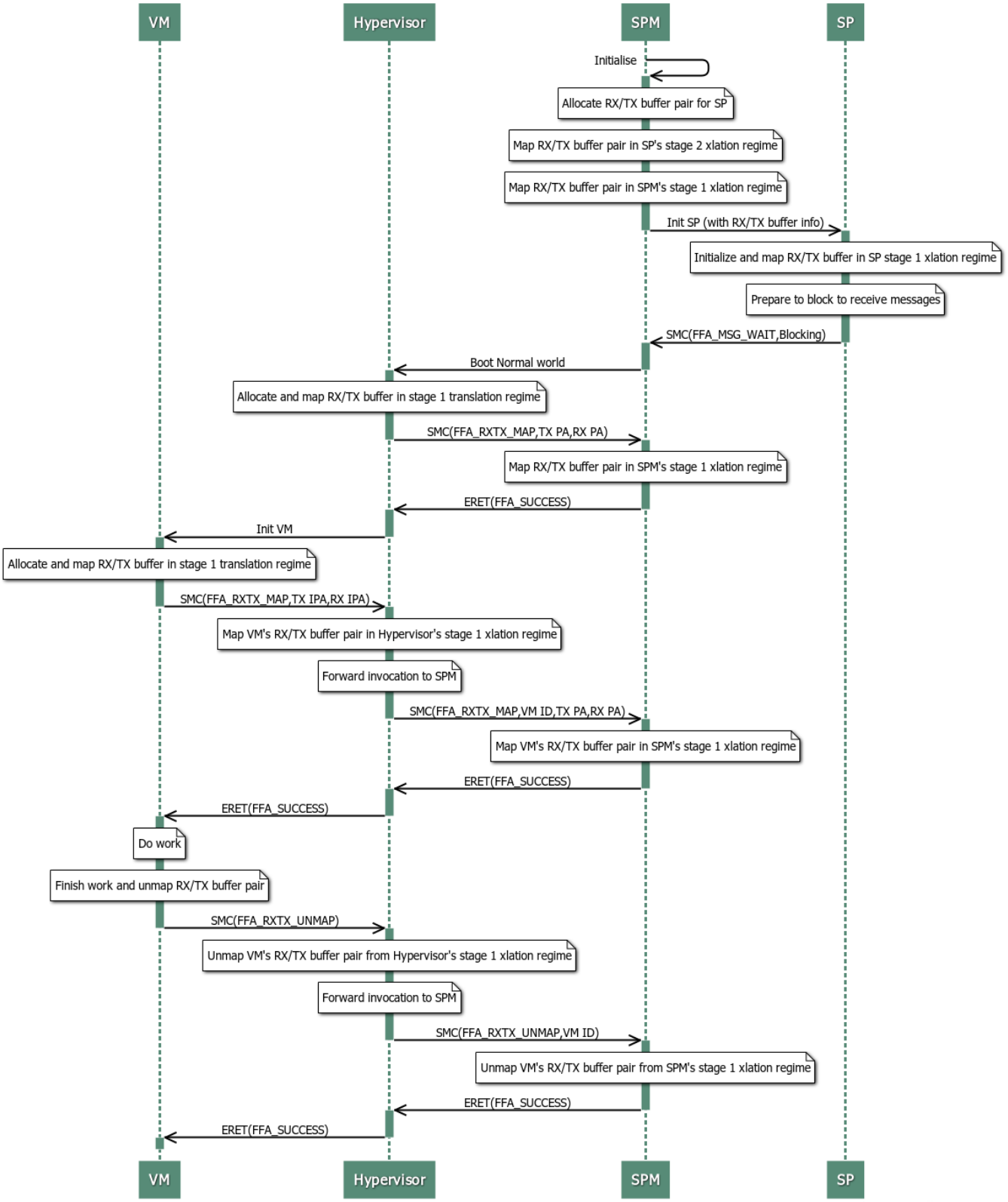

Figure 4.4: RX/TX Buffer setup

4.2.2.3 Buffer attributes

Endpoints and partition managers must ensure that buffer pairs are setup with attributes that follow the rules listed below.

1. The size of the RX and TX buffers in a pair are the same and a multiple of the larger translation granule size used by the FF-A components at an FF-A instance.
2. The alignment of the RX and TX buffers in a pair is equal to the larger translation granule size used by the FF-A components at an FF-A instance (see also 2.7 *Memory granularity and alignment*).
3. An endpoint discovers the minimum size and alignment boundary for the RX/TX buffers by passing the function ID of the *FFA_RXTX_MAP* ABI as input in the *FFA_FEATURES* interface (see 11.2 *FFA_FEATURES*).
4. All buffer pairs are mapped with the following memory region attributes in all stages of a translation regime in the system.
   - Normal memory.
   - Write-Back Cacheable.
   - Non-transient Read-Allocate.
   - Non-transient Write-Allocate.
   - Inner Shareable.
   - Memory used for buffer pairs shared between an SP and SPMC must be mapped as Secure memory.
   - Memory used for buffer pairs shared between a Normal world FF-A component and the SPMC must be mapped as Non-secure memory.
   - Table 4.3 describes the minimum permission requirements of RX/TX buffer.

Table 4.3: RX/TX buffer minimum permission requirements

Buffer Type	Producer	Consumer	Description
RX	RW, XN	RO, XN	• Producer must have write access to populate message payload. • Consumer must have at least read access to read message payload.
TX	RW, XN	RO, XN	• Producer must have Write-access to populate message payload. • Consumer must have at least read access to copy the message payload to the target RX buffer. • Consumer must also have Write- access to modify message payload if required.

4.2.2.3.1 Coherency requirements

A buffer pair could be accessed with different memory region attributes from the translation regime of the Producer and Consumer, if address translation is disabled in one of them.

To avoid memory coherency issues in this scenario, the FF-A component that has address translation disabled must perform cache maintenance on the buffer in scenarios listed in Table 4.4. The cache maintenance must ensure that the buffer contents at any intermediate cache levels are not out of sync with the buffer contents at the *Point of coherence* (see [5]).

- As a Producer, this must be done before the Consumer reads the buffer (see 4.2.2.4 *Buffer synchronization*).
- As a Consumer, this must be done before reading the buffer populated by the Producer.

Table 4.4: RX/TX buffer cache maintenance requirements

Config No.	Address translation in Producer	Address translation in Consumer	Cache maintenance required
1.	Disabled	Disabled	No
2.	Disabled	Enabled	Yes
3.	Enabled	Disabled	Yes
4.	Enabled	Enabled	No

4.2.2.4 Buffer synchronization

The RX and TX buffers are written to by a Producer and read by a Consumer as described in Table 4.5. Concurrent accesses to these buffers from both entities on either side of an FF-A instance must be synchronized to preserve the integrity of their contents.

Table 4.5: Producers and Consumers of RX/TX buffers

Buffer Type	Producers	Consumers
VM RX	Hypervisor, SPMC	VM
VM TX	VM	Hypervisor, SPMC
OS Kernel RX	SPMC	OS Kernel
OS Kernel TX	OS Kernel	SPMC
SP RX	SPMC	SP
SP TX	SP	SPMC
Hypervisor RX	SPMC	Hypervisor
Hypervisor TX	Hypervisor	SPMC

4.2.2.4.1 Buffer states and ownership

The Framework defines buffer states and ownership rules that must be followed by the Producer and Consumer of each buffer.

- Each buffer is either in *empty* or *full* (has a message in it) states at any given time. This state must be tracked internally by the Producer and Consumer using an IMPLEMENTATION DEFINED mechanism.
- A buffer is in the *empty* state immediately after being mapped in both the Producer and Consumer's translation regimes.
- The Producer of a buffer owns it when it is empty.
- The Consumer of a buffer owns it when it is full.
- The Producer writes to the buffer when it is empty.
- The Consumer reads from the buffer when it is full.

4.2.2.4.2 Transfer of buffer ownership

After a Producer has written to a buffer, it must transfer its ownership to the Consumer for reading the message. Equally, the Consumer must transfer ownership back to the Producer after it has read the message. This is done as per the rules stated below.

1. Ownership transfer for the TX buffer takes place as follows.
    1. For a partition message,
        1. An invocation of the FFA_MSG_SEND2 ABI transfers the ownership from the Producer to the Consumer.
        2. Completion of an FFA_MSG_SEND2 ABI invocation transfers the ownership from the Consumer to the Producer.
    2. For a framework message,
        1. An invocation of an FF-A ABI that uses the TX buffer of the caller transfers the ownership from the Producer to the Consumer. In this version of the Framework, the following memory management ABIs use the TX buffer.
            - FFA_MEM_DONATE.
            - FFA_MEM_LEND.
            - FFA_MEM_SHARE.
            - FFA_MEM_RETRIEVE_REQ.
            - FFA_MEM_RELINQUISH.
            - FFA_MEM_FRAG_TX.
        2. Completion of an FF-A ABI that uses the TX buffer of the caller transfers the ownership from the Consumer to the Producer.
2. Ownership transfer for the RX buffer takes place as follows.
    1. For a partition message,
        1. Completion of an FFA_NOTIFICATION_GET ABI invocation by the Consumer, that signals the *RX buffer full notification*, transfers the ownership from the Producer to the Consumer. Also see 7.8.1 *RX buffer full notification*.
    2. For a framework message,
        1. Completion of the FFA_PARTITION_INFO_GET ABI transfers the ownership of the caller's RX buffer from the Producer to the Consumer.
        2. An invocation of the FFA_MEM_RETRIEVE_RESP ABI uses the RX buffer of the callee and transfers the ownership from the Producer to the Consumer.
    3. For both types of messages, an invocation of the following FF-A ABIs transfers the ownership from the Consumer to the Producer.
        1. FFA_MSG_WAIT.

2. FFA_RX_RELEASE.

4.2.2.4.3 Management of buffer ownership between Hypervisor and SPMC

Both the Hypervisor and SPMC are producers of a VM's RX buffer. They could both contend for the buffer in certain scenarios. For example, the Hypervisor transmits a message from VM0 to VM1 and the SPMC transmits a message from SP0 to VM1 simultaneously.

The Framework defines the FFA_RX_ACQUIRE ABI to solve this contention as described below. Also see 11.3 *FFA_RX_ACQUIRE*.

1. A VM's RX buffer is owned by the SPMC after it is mapped into its translation regime (see 4.2.2.2 *Buffer setup*).
2. The Hypervisor uses FFA_RX_ACQUIRE ABI to acquire ownership of a VM's RX buffer from the SPMC, prior to writing to the buffer.
3. The VM transfers ownership of its RX buffer to the Hypervisor as described in 4.2.2.4.2 *Transfer of buffer ownership*.
4. The Hypervisor uses FFA_RX_RELEASE ABI to relinquish ownership of the VM's RX buffer to the SPMC.

The Hypervisor does not need to acquire and release ownership of a VM's RX buffer if the SPMC does not implement the FFA_RX_ACQUIRE ABI. For example, in a scenario where no SP supports indirect messaging.

Implementation Note

A buffer could be shared among multiple Producers, Consumers, and multiple instances of the same Producer and Consumer (also see Table 4.5). Both the Producers and the Consumers must use an IMPLEMENTATION DEFINED synchronization mechanism to protect the buffer from concurrent accesses that are internal to them. A Producer or Consumer could implement additional states internally to prevent concurrent accesses. Such states are outside the scope of this version of the Firmware Framework.

For example, multiple instances of the SPM will run concurrently on different PEs. As the Producer for an RX buffer or as a Consumer for a TX buffer, the SPM could use a spinlock to protect each buffer from accesses made concurrently by its own instances.

4.3 Indirect messaging usage

4.3.1 Discovery and setup

An endpoint that can receive messages through indirect messaging must specify this property in its manifest (see 3.2.1 *Manifest for isolated partitions*). The scheduler that runs this endpoint can discover its presence and the number of execution contexts it implements through the following mechanisms.

1. The FFA_PARTITION_INFO_GET interface. See 11.7 *FFA_PARTITION_INFO_GET*.
2. An IMPLEMENTATION DEFINED mechanism for example, Device tree.

In version 1.0 of the Framework, only VMs are allowed to send and receive messages through indirect messaging. Also see 16.4 *Legacy indirect messaging usage*.

4.3.2 Message delivery

The Framework defines the **FFA_MSG_SEND2** interface to transmit a message from the TX buffer of a Sender to the RX buffer of a Receiver and inform the scheduler that the Receiver must be run. 13.1 *FFA_MSG_SEND2* describes the **FFA_MSG_SEND2** interface.

4.3.3 Scheduling the Receiver

The Relayer informs the primary scheduler that the Receiver has a message in its RX buffer and must be scheduled. The primary scheduler either runs the Receiver itself or informs the secondary scheduler responsible for running the Receiver.

In this version of the Framework, the Relayer and schedulers use a Framework notification for performing these actions. See Chapter 7 *Notifications* & 7.8.1 *RX buffer full notification* for details.

Once the Receiver starts processing the message after a scheduling decision, the runtime model presented to it by its partition manager is described in Chapter 5 *Partition runtime models*.

4.4 Direct messaging usage

In a direct message exchange, transmission of the message from the Sender to the Receiver takes place in tandem with allocation of CPU cycles to the Receiver to process the message.

The Framework assumes that direct messaging is used by a Sender as an equivalent of invoking a procedure or function in the Receiver. The Receiver executes the function and returns the results through another direct message.

- For Framework messages, execution of the function in the Hypervisor or SPM runs to completion from the perspective of the Sender.
- For Partition messages, execution of the function in an endpoint could run to completion or be preempted by interrupts one or more times. In the latter case, the communication framework is responsible for resuming function execution.

Direct messaging is used for:

1. Exchanging Framework messages with the Hypervisor and SPM in the configurations listed in Table 4.6. These messages can be exchanged in both RX/TX buffers and registers.

   In v1.1 of the Framework, the FFA_MSG_SEND_DIRECT_REQ and FFA_MSG_SEND_DIRECT_RESP ABIs can be used to,

   - Exchange framework messages between the SPMD and SPMC.
   - Exchange framework messages between the SPMC and an SP.

   The SPMD and SPMC IDs are used to specify the Sender and Receiver in these ABIs (see 11.9 *FFA_SPM_ID_GET*).

   The Framework messages that can be exchanged through this mechanism are described in 16.3.4 *Power Management messages*.

2. Exchanging Partition messages between endpoints in the configurations listed in Table 4.7. These messages can be exchanged only in registers.

Table 4.6: Valid configurations for exchanging Framework messages through direct messaging

Config no.	Sender	Receiver	Relayer
1.	VM	Hypervisor	•
2.	NS-Endpoint	SPM	Hypervisor (if present)
3.	SP	SPM	•
4.	SP	Hypervisor	SPM
5.	Hypervisor	VM	•
6.	Hypervisor	SPM	•
7.	Hypervisor	SP	SPM

Config no.	Sender	Receiver	Relayer
8.	SPM	NS-Endpoint	Hypervisor (if present)
9.	SPM	Hypervisor	•
10.	SPM	SP	•

Table 4.7: Valid configurations for exchanging Partition messages through direct messaging

Config no.	Sender	Receiver	Relayer
1.	VM	VM	Hypervisor
2.	NS-Endpoint	SP	Hypervisor (if present) and SPM
3.	SP	SP	SPM
4.	SP	NS-Endpoint	SPM and Hypervisor (if present)

4.4.1 Discovery and setup

An endpoint could be capable of receiving direct messages, sending direct messages or both. A Sender of direct requests must be able to receive direct responses. A Receiver of direct requests must be able to send direct responses.

The ability to send or receive direct messages must be specified in the manifest of the endpoint (see Table 3.1 in 3.2.1 Manifest for isolated partitions).

In a direct message exchange, an execution context of the Receiver must be available on the same PE as the Sender to receive and process the message. To fulfil this requirement, the Receiver must make one of the following implementation choices.

- The Receiver is implemented as a **UP** endpoint. This enables the SPM or Hypervisor to migrate the endpoint execution context to the PE on which a direct messaging request is made.
- The Receiver is implemented as a **MP** endpoint. In this case, the number of execution contexts that the endpoint implements must be equal to the number of PEs in the system. Each execution context must be pinned to a PE at system boot. This enables the SPM or Hypervisor to guarantee availability of an endpoint execution context for direct messages on the same PE as the Sender.

This implementation choice must be specified in the manifest of the endpoint (see Table 3.1 in 3.2.1 Manifest for isolated partitions).

A partition manager can discover the properties of an endpoint it manages through the endpoint manifest. It can discover the properties of endpoints it does not manage through the **FFA_PARTITION_INFO_GET** interface

(see 11.7 *FFA_PARTITION_INFO_GET*). An endpoint could use the same interface to determine properties of other endpoints as well.

In version 1.0 of the Framework, an attempt to send an indirect message to an endpoint that only supports receipt of direct requests must be rejected,

- By the Hypervisor if the Sender is a VM.
- By the SPM if the Sender is an SP, Hypervisor, or NS-Endpoint.

In this version of the Firmware Framework, a partition manager can only send and receive Framework messages through direct messaging. To support this model, it must be implemented as per the constraints listed as follows for an **MP** endpoint.

- It must have as many execution contexts as PEs in the system.
- Each execution context runs only on the PE where it was initialized during boot. Hence, it can be considered to be *pinned* to that PE.

In the SPM configuration where the SPMC coexists with an SP at S-EL1 or Secure Supervisor mode (see Table 2.3), the SP must be implemented as per the constraints that apply to the SPM implementation.

4.4.2 Message delivery and Receiver execution

The Framework defines the FFA_MSG_SEND_DIRECT_REQ and FFA_MSG_SEND_DIRECT_RESP interfaces (also see 13.2 *FFA_MSG_SEND_DIRECT_REQ* & 13.3 *FFA_MSG_SEND_DIRECT_RESP*) to transmit direct messages between a Sender and Receiver.

- The Sender sends a request message to the Receiver using the FFA_MSG_SEND_DIRECT_REQ interface.
- The Receiver sends a response message to the Sender using the FFA_MSG_SEND_DIRECT_RESP interface.

13.2.1 *Component responsibilities for FFA_MSG_SEND_DIRECT_REQ* describes how a message is transmitted using the *FFA_MSG_SEND_DIRECT_REQ* interface and the responsibilities of the participating components in all the configurations listed in Table 4.7.

13.3.1 *Component responsibilities for FFA_MSG_SEND_DIRECT_RESP* describes how a message is transmitted using the *FFA_MSG_SEND_DIRECT_RESP* interface and the responsibilities of the participating components in all the configurations listed in Table 4.7.

Figure 4.5 illustrates an example flow in which a VM sends a direct message to an SP through the *FFA_MSG_SEND_DIRECT_REQ* interface. The SP processes the messages and returns the results using the *FFA_MSG_SEND_DIRECT_RESP* interface.

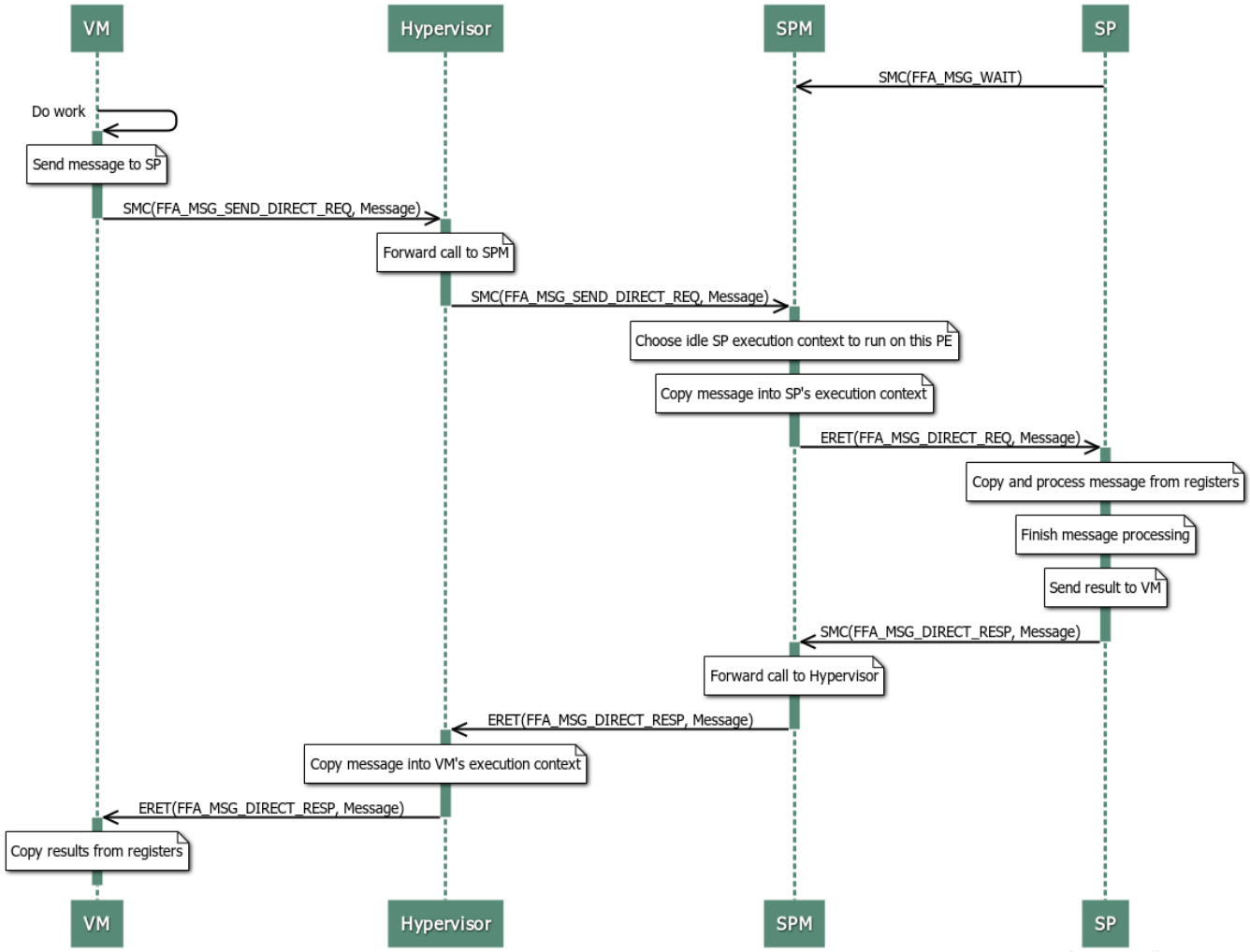

Figure 4.5: Example direct message exchange between a VM and SP

Chapter 5
Partition runtime models

5.1 Overview

The runtime model of an endpoint describes the transitions, its execution contexts are permitted to make between states post CPU cycle allocation.

- The states are described in 2.12 *Run-time states*.
- The state transitions are described in 2.13 *Run-time state transitions*.

The Framework specifies the following mechanisms to allocate CPU cycles to an endpoint execution context.

1. The FFA_RUN interface is used to allocate CPU cycles to an execution context of an endpoint for message processing. The runtime model for this execution context is described in 5.2 *Runtime model for FFA_RUN*.
2. The FFA_MSG_SEND_DIRECT_REQ interface is used to allocate CPU cycles to an execution context of an endpoint for message processing. The runtime model for this execution context is described in 5.3 *Runtime model for FFA_MSG_SEND_DIRECT_REQ*.
3. A Secure interrupt targeted to a SP preempts the Normal world. The SPMC runs an SP execution context in the *waiting* state for handling the Secure interrupt. The runtime model for this execution context is described in 5.4 *Runtime model for Secure interrupt handling*. Also see 6.2 *Secure interrupt signaling mechanisms*.
4. The SPMC runs an SP execution context to initialize the SP during boot. The runtime model for this execution context is described in 5.5 *Runtime model for SP initialization*.

The following common rules and guidelines govern the use of runtime models in the Framework.

1. An endpoint execution context in the *running* state could use FFA_RUN and FFA_MSG_SEND_DIRECT_REQ ABIs to call into another endpoint execution context. This sequence could be repeated for any number of times. All endpoint execution contexts in the sequence become a part of a *call chain*.

The Framework specifies rules in the applicable runtime models to prevent *loops* forming in a call chain i.e. an endpoint execution context allocates cycles to another endpoint execution context which is already a part of the call chain.

2. The partition manager of an endpoint applies a runtime model,

    1. From when the endpoint execution context transitions from the *waiting* to the *running* state.
    2. To when the endpoint execution context next transitions from the *running* to the *waiting* state.

    The endpoint execution context could enter the *blocked* and *preempted* states multiple times before entering the *waiting* state to return control back to its partition manager.

3. An endpoint execution context can invoke hypcalls in all runtime models.
4. The partition manager returns *DENIED* as the error code, if an invalid transition is attempted by an endpoint execution context.
5. The partition manager returns *DENIED* as the error code, if a valid transition is attempted by an endpoint execution context that will result in a *loop* in the call chain.

5.2 Runtime model for FFA_RUN

Figure 5.1 illustrates the state machine specified by the runtime model presented to an endpoint execution context that is allocated CPU cycles through the FFA_RUN interface. Rules that govern this runtime model are listed below.

1. It can use the smc(FFA_MSG_SEND_DIRECT_REQ) transition to send a message and allocate CPU cycles to any endpoint execution context apart from those in a call chain that leads to the currently running endpoint execution context. The execution context enters the *blocked* state.
2. It can use the smc(FFA_RUN) transition to allocate CPU cycles to any endpoint execution context apart from those in a call chain that leads to the currently running endpoint execution context. The execution context enters the *blocked* state.
3. It cannot use the smc(FFA_MSG_SEND_DIRECT_RESP) transition to send a message, relinquish control back to any endpoint and enter the *waiting* state.
4. It uses the smc(FFA_MSG_WAIT) transition to relinquish control back to the endpoint execution context that allocated CPU cycles to it and enter the *waiting* state. For example, to signal completion of message processing.
5. It uses the smc(FFA_YIELD) transition to relinquish control back to the endpoint execution context that allocated CPU cycles to it and enter the *blocked* state. For example, to wait until an internal lock is not available.

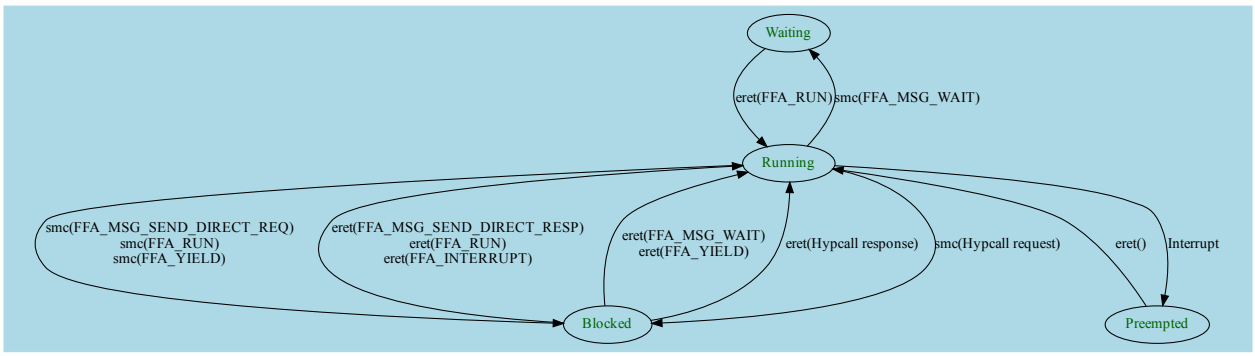

Figure 5.1: State machine for runtime model with FFA_RUN

5.3 Runtime model for FFA_MSG_SEND_DIRECT_REQ

Figure 5.2 illustrates the state machine specified by the runtime model presented to an endpoint execution context that is allocated CPU cycles through the FFA_MSG_SEND_DIRECT_REQ interface. Rules that govern this runtime model are listed below.

1. It can use the smc(FFA_MSG_SEND_DIRECT_REQ) transition to send a message and allocate CPU cycles to any endpoint execution context apart from those in a call chain that leads to the currently running endpoint execution context. The execution context enters the *blocked* state.
2. It can use the smc(FFA_RUN) transition to allocate CPU cycles to any endpoint execution context apart from those in a call chain that leads to the currently running endpoint execution context. The execution context enters the *blocked* state.
3. It uses the smc(FFA_MSG_SEND_DIRECT_RESP) transition to return a response and relinquish control to the endpoint execution context that allocated CPU cycles to it and enter the *waiting* state. For example, to signal completion of message processing.

   It cannot use the smc(FFA_MSG_SEND_DIRECT_RESP) transition to send a message, relinquish control back to any other endpoint execution context and enter the *waiting* state.
4. It cannot use the smc(FFA_MSG_WAIT) transition to relinquish control back to any endpoint and enter the *waiting* state.
5. It cannot use the smc(FFA_YIELD) transition to relinquish control back to the endpoint and enter the *blocked* state.

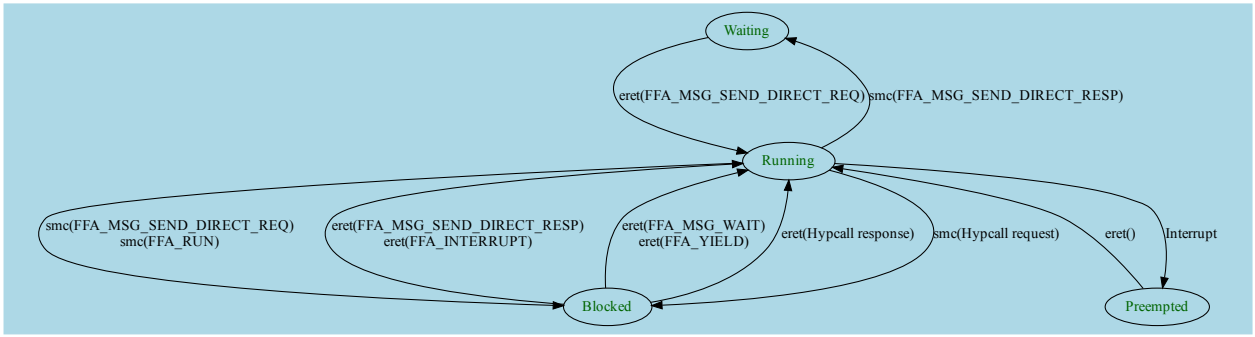

Figure 5.2: State machine for runtime model with FFA_MSG_SEND_DIRECT_REQ

5.4 Runtime model for Secure interrupt handling

Figure 5.3 illustrates the state machine specified by the runtime model presented to an endpoint execution context that is allocated CPU cycles by its partition manager to handle a Secure interrupt. Rules that govern this runtime model are listed below.

1. This runtime model is only applicable to interrupts that are signaled to an SP execution in the *waiting* state. This is described in 6.2 *Secure interrupt signaling mechanisms*.
2. It uses the smc(FFA_MSG_WAIT) transition to relinquish control back to its partition manager and enter the *waiting* state after handling the interrupt.
3. It can use the smc(FFA_MSG_SEND_DIRECT_REQ) transition to send a message and allocate CPU cycles to any SP. The execution context enters the *blocked* state.
4. It cannot use the smc(FFA_YIELD) transition to relinquish control back to any endpoint and enter the *blocked* state as it was scheduled by its partition manager.
5. It cannot use the smc(FFA_MSG_SEND_DIRECT_RESP) transition to send a message, relinquish control to any endpoint and enter the *waiting* state as it was scheduled by its partition manager.
6. It can use the smc(FFA_RUN) transition to resume a request that was made earlier through the smc(FFA_MSG_SEND_DIRECT_REQ) transition. The target of the smc(FFA_RUN) transition is in a *preempted* state. The calling execution context enters the *blocked* state.

If a Secure interrupt is handled by an SP execution context in the *running*, *blocked* or *preempted* states, the existing runtime model of the execution context is preserved. For example,

- The SPMC could signal a Secure interrupt to a S-EL1 SP in the *running* state under the runtime model for FFA_MSG_SEND_DIRECT_REQ. The runtime model of the SP does not change during interrupt handling.
- The SPMC could signal a Secure interrupt to a S-EL1 SP in the *running* state under the runtime model for Secure interrupt handling. This implies that another Secure interrupt is signaled to the SP while it is already handling another Secure interrupts. The runtime model of the SP does not change during interrupt handling.

Also see Chapter 6 *Interrupt management*.

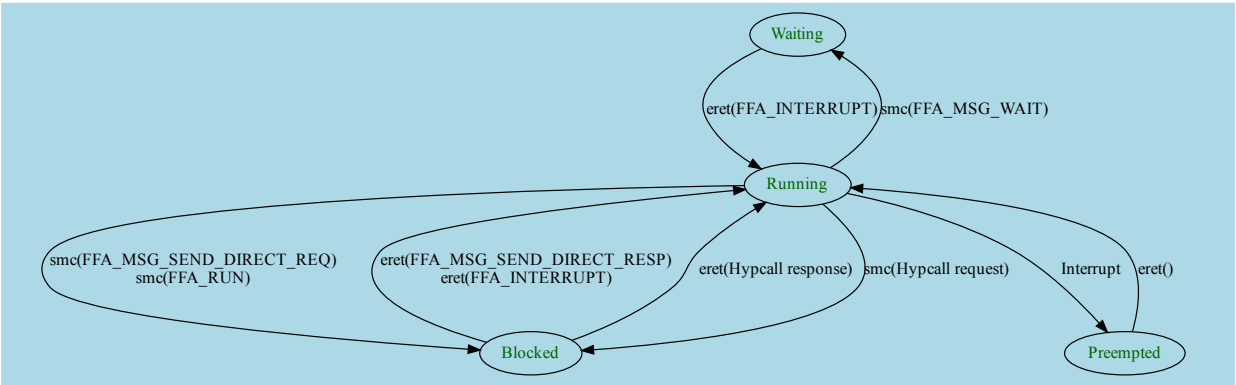

Figure 5.3: State machine for runtime model with Secure interrupt handling

5.5 Runtime model for SP initialization

Figure 5.4 illustrates the state machine specified by the runtime model presented to a SP execution context that is allocated CPU cycles by the SPMC to initialize its state. Rules that govern this runtime model are listed below.

1. It can use the smc(FFA_MSG_SEND_DIRECT_REQ) transition to send a message and allocate CPU cycles to any SP execution context that has already been initialized. The execution context enters the *blocked* state.
2. It uses the smc(FFA_MSG_WAIT) transition to signal successful initialization to the SPMC and enter the *waiting* state.
3. It uses the smc(FFA_ERROR) transition to signal failed initialization to the SPMC and enter the *waiting* state.
4. It cannot use the smc(FFA_YIELD) transition to relinquish control back to any endpoint and enter the *blocked* state as it was scheduled by the SPMC.
5. It cannot use the smc(FFA_MSG_SEND_DIRECT_RESP) transition to send a message, relinquish control to any endpoint and enter the *waiting* state as it was scheduled by the SPMC.
6. It cannot use the smc(FFA_RUN) transition to allocate CPU cycles to an execution context of another endpoint and enter the *blocked* state.

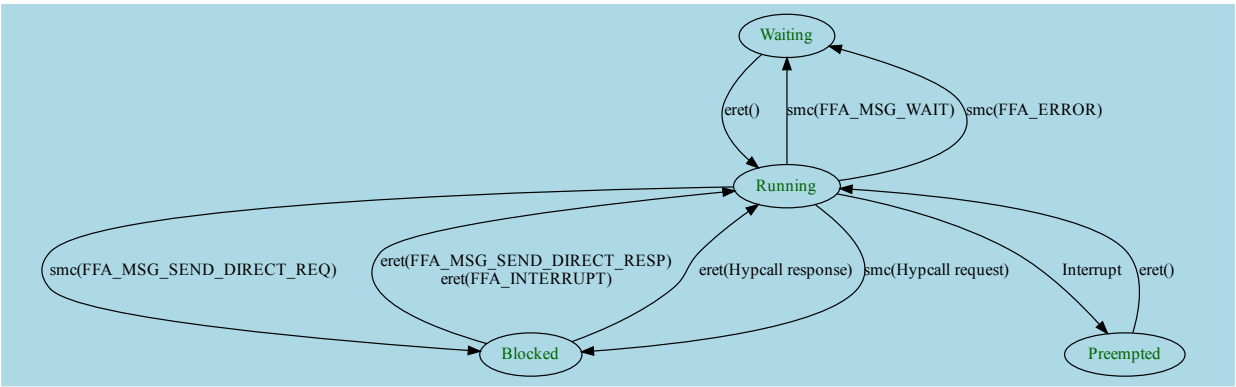

Figure 5.4: State machine for runtime model for initializing an SP execution context

Chapter 6
Interrupt management

6.1 Overview

A physical interrupt can trigger on a PE where an endpoint execution context is in the *running* state. It could be targeted to this execution context or another FF-A component in the system. Alternatively, a physical interrupt targeted to an endpoint execution context could trigger when the context is in the *waiting*, *blocked* or *preempted* states.

This chapter provides guidance regarding management of interrupts in such scenarios under the following assumptions.

1. The guidance is applicable to FF-A components in the Secure world. Management of interrupts in the Normal world is IMPLEMENTATION DEFINED. Furthermore, this guidance is applicable to configurations where,
    1. The SPMC has exclusive access to the physical GIC. This guidance could be extended to a configuration where the SPMC shares access to the physical GIC with a trusted S-EL1 SP in an IMPLEMENTATION DEFINED manner. This is beyond the scope of this specification.
    2. A S-EL1 SP only has access to the virtual GIC. The SPMC signals interrupts through the virtual IRQ and FIQ lines and the ERET conduit. The model of the GIC presented by the SPMC to the SP is IMPLEMENTATION DEFINED. For example, it could implement the Reduced virtual interrupt controller specification [7] to present this model to an SP.
    3. A S-EL0 SP has no access to the virtual GIC. The SPMC signals interrupts through the ERET conduit.
2. The GIC implements support for Secure EL2 introduced in version 3.1 of the Arm GIC architecture. This assumption is applicable to S-EL1 SPs managed by a SPMC in S-EL2.
3. The GIC implements version 2.0 or later of the Arm GIC architecture. This assumption is applicable to S-EL0 SPs managed by a SPMC in EL3 or S-EL1.

4. Secure interrupts are configured as G1S or G0 interrupts if the GIC architecture version is 3.0 or later.
5. Non-secure interrupts are configured as G1NS interrupts if the GIC architecture version is 3.0 or later.
6. Secure interrupts are configured as G0 interrupts if the GIC architecture version is 2.0.
7. Non-secure interrupts are configured as G1 interrupts if the GIC architecture version is 2.0
8. SCTLR_EL1.UMA=0 during execution in a S-EL0 SP. It is not allowed to mask physical or virtual FIQs in the PSTATE register.
9. SPs managed by a SPMC in S-EL2 are treated as S-EL0 SPs if S-EL0 is specified as the run-time EL in the partition manifest. Also see 3.2.1 *Manifest for isolated partitions*.
10. Secure interrupts are routed to EL3 when execution is in the Non-secure state by programming SCR_EL3.FIQ=1.
11. All interrupts are routed to the SPMC when execution is in the Secure state. For example, with a SPMC in S-EL2, this could be done by programming,
    - SCR_EL3.FIQ=0 and SCR_EL3.IRQ=0.
    - HCR_EL2.IMO=1 and HCR_EL2.FMO=1.

The guidance in this chapter covers the following scenarios related to interrupt management.

1. When a physical interrupt triggers, its ability to preempt a running SP execution context and be handled depends upon the *scheduling model* used to run it by the SPMC. These scheduling models are described in 6.5 *SP scheduling models*.
2. The interrupt could preempt a SP execution context and be targeted to another FF-A component. Guidance on managing the state of the SP execution context in this scenario and informing its scheduler about the preemption is specified in 6.4 *Preemption during message processing*.
3. A Secure interrupt targeted to a SP execution context could preempt execution in the Normal world or another SP execution context. The mechanisms used by the SPMC and SP to signal start and completion of interrupt handling are specified in,
    - 6.2 *Secure interrupt signaling mechanisms*
    - 6.3 *Secure interrupt completion mechanisms*

    For S-EL1 SPs, these mechanisms are applicable to virtual Secure interrupts that are signaled in response to physical Secure interrupts.

The runtime model that a SP must use while handling a Secure interrupt is specified in 5.4 *Runtime model for Secure interrupt handling*.

6.2 Secure interrupt signaling mechanisms

The mechanisms used by the SPMC to signal a pending Secure interrupt to a SP execution context are,

1. The FFA_INTERRUPT interface with the ERET conduit. This mechanism is used for signaling to both S-EL1 and S-EL0 SPs.
2. The vIRQ signal. This mechanism is only used for signaling to S-EL1 SPs.

The choice of mechanism depends upon the type of SP and the run-time state of the SP execution context.

- Table 6.1 describes how the SPMC signals to a S-EL0 SP that it has a pending interrupt.
- Table 6.2 describes how the SPMC signals to a S-EL1 SP execution context that it has a pending interrupt.

Table 6.1: Secure interrupt signaling to a S-EL0 SP

No.	SP type	SP state	Conduit	Interface and parameters	Description
1.	S-EL0	Waiting	ERET	FFA_INTERRUPT, Interrupt ID	• SPMC signals to a S-EL0 SP execution context that an interrupt is pending and its ID. • SPMC resumes execution of the SP execution context through the ERET instruction.
2.	S-EL0	Blocked	NA	NA	• The SPMC does not signal an interrupt to a S-EL0 SP in the *blocked* state.
3.	S-EL0	Preempted	NA	NA	• The SPMC cannot signal an interrupt to a S-EL0 SP in the *preempted* state.
4.	S-EL0	Running	NA	NA	• The SPMC cannot signal an interrupt to a S-EL0 SP in the *running* state.

Table 6.2: Secure interrupt signaling to a S-EL1 SP

No.	SP type	SP state	Conduit	Interface and parameters	Description
1.	S-EL1	Waiting	ERET, vIRQ	FFA_INTERRUPT, Interrupt ID	• SPMC signals to a S-EL1 SP that an interrupt is pending and its ID. • SPMC also pends the vIRQ signal to allow the S-EL1 SP to handle the interrupt in the handler context. • SPMC resumes execution of the SP through the ERET instruction.

No.	SP type	SP state	Conduit	Interface and parameters	Description
2.	S-EL1	Blocked	ERET, vIRQ	FFA_INTERRUPT	• SPMC signals to a S-EL1 SP that an interrupt is pending. The ID of the interrupt is not specified since the SP could be running in an application context and not have any use of the ID. • SPMC also pends the vIRQ signal to allow the S-EL1 SP to handle the interrupt in a separate handler context. • SPMC resumes execution of the SP through the ERET instruction.
3.	S-EL1	Preempted	vIRQ	NA	• SPMC pends the vIRQ signal to allow the S-EL1 SP to handle the interrupt in a separate handler context. • The SP handles the interrupt when its execution is subsequently resumed.
4.	S-EL1	Running	ERET, vIRQ	NA	• SPMC pends the vIRQ signal to allow the S-EL1 SP to handle the interrupt in a separate handler context. • SPMC resumes execution of the SP through the ERET instruction.

When execution in Normal world is preempted by a Secure interrupt, the SPMD uses the FFA_INTERRUPT ABI with the ERET conduit to signal the interrupt to a SPMC in S-EL2 or S-EL1.

6.3 Secure interrupt completion mechanisms

A SP signals completion of Secure interrupt handling to the SPMC through the following mechanisms.

1. If the SP was in the *waiting* state when the interrupt was signaled to it, completion is signaled through an invocation of the FFA_MSG_WAIT interface. Also see 5.4 *Runtime model for Secure interrupt handling*.
2. If the SP was in the *blocked* state when the interrupt was signaled to it, completion is signaled through an invocation of the FFA_RUN interface.
3. A S-EL1 SP drops the priority of the virtual Secure interrupt after handling it. The virtual interrupt was

signaled by the SPMC in S-EL2. For example,

1. The SP could drop the priority of the virtual Secure interrupt by writing to the *ICV_EOIR0_EL1 or ICV_EOIR1_EL1* registers in the virtual CPU interface specified in [8]. The SPMC traps this access by setting *ICH_HCR_EL2.TALL1 == 1*.
2. The SP could drop the priority of the virtual Secure interrupt by invoking a para-virtualized SMC interface implemented by the SPMC.

If a SP can use multiple mechanisms to signal completion of Secure interrupt handling, the SPMC treats the first invocation of a mechanism as the signal.

The SPMC in S-EL2 or S-EL1 uses the FFA_NORMAL_WORLD_RESUME ABI to indicate completion of Secure interrupt handling to the SPMD if execution in Normal world was preempted by the Secure interrupt. Also see 12.4 *FFA_NORMAL_WORLD_RESUME*.

6.4 Preemption during message processing

An endpoint execution context in the running state could be interrupted by a physical interrupt targeted to another FF-A component. In this scenario,

- The execution context enters the *preempted* state on being interrupted.
- The state of the execution context is saved by the SPMC upon interruption.
- The state of the execution context is restored by the SPMC upon resumption.
- If it is a Non-secure interrupt, the SPMC takes one of the following actions.
  - It informs the scheduler of the execution context about the preemption through the FFA_INTERRUPT interface (also see 10.4 *FFA_INTERRUPT*).

    The scheduler uses the FFA_RUN interface to resume the SP execution context subsequently.
  - It informs the SP execution context about the pending interrupt. This enables the SP to perform any bookkeeping before relinquishing control to the Normal world. See 6.4.1 *Managed exit* for details.
- If it is a Secure interrupt, the SPMC signals it to the target SP as per the scheduling policy of the *running* and target SPs. Scheduling policies described in 6.5 *SP scheduling models*. Signaling is done as described in 6.2 *Secure interrupt signaling mechanisms*.
- The execution context is resumed on the same PE where it was preempted or on a different PE if it is not pinned to the original PE.

Figure 6.1 illustrates an example flow where *Client 0* in a NS-Endpoint sends a direct message to the single execution context EC0 on CPU0 of an UP-Migrate capable SP. Message processing in SP EC0 is preempted by a Non-secure interrupt. It is later resumed on CPU1 by the NS-Endpoint.

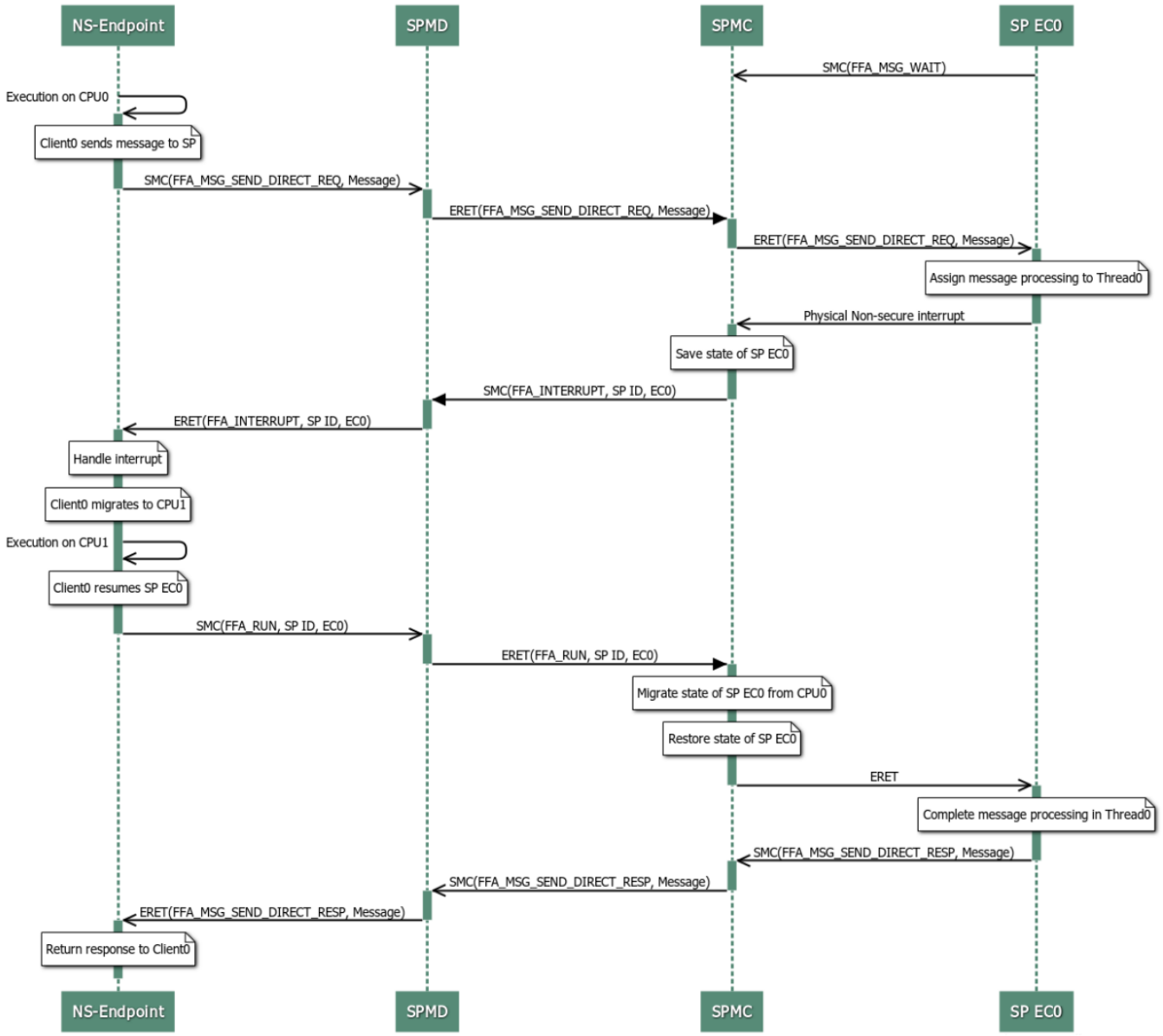

Figure 6.1: Example endpoint preemption flow

6.4.1 Managed exit

6.4.1.1 Overview

A managed exit is a mechanism in which a running SP execution context is notified about a pending physical Non-secure interrupt. This allows the SP to manage its internal state before relinquishing control to the Normal world where the interrupt is handled.

A managed exit stands in contrast to preemption of an SP execution context in the running state. In this case, the SP does not get an opportunity to manage its internal state before control is handed to the Normal world.

A managed exit could be used for the following reasons.

1. It enables other application threads running in the SP execution context to make progress while one or more application threads have been preempted.

2. It ensures that the CPU cycles allocated to an SP execution context are used to process the request that the scheduler has issued instead of a request from another endpoint.
3. It ensures that critical events can be conveyed to the endpoint in time.

   For example, the OS could issue a power state transition event through a PSCI function on a PE. The SPMC could need to inform SP execution contexts pinned to that PE about this event. This cannot be done if a SP execution context is in a *preempted* state. Also see 16.3.4 *Power Management messages*.
4. It enables application threads in an MP SP with pinned execution contexts to be migrated to a different PE. They could be then resumed under the execution context pinned on that PE. This is in contrast to the SP execution context, and therefore all its application threads, remaining in a preempted state, on the PE where they were preempted until later resumed.

   Figure 6.2 illustrates a managed exit flow using this reason as an example where *Client 0* in a NS-Endpoint sends a direct message to MP capable SP. The SP has access to the virtual GIC and two execution contexts *EC0* and *EC1* which are pinned to *CPU0* and *CPU1* respectively. SP EC0 stops message processing and performs a managed exit in response to a Non-secure physical interrupt. Message processing is later resumed on CPU1 by the NS-Endpoint.

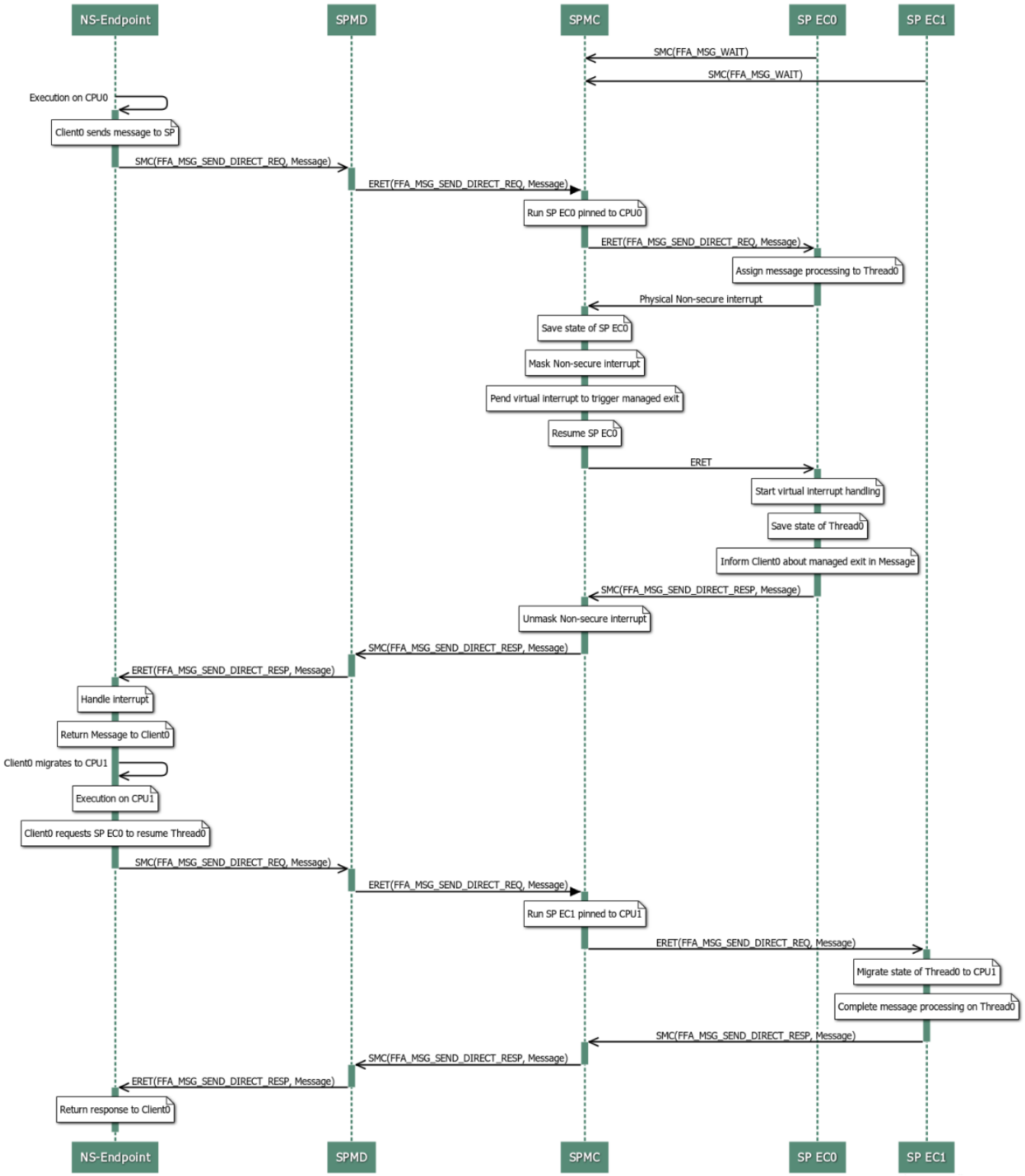

Figure 6.2: Example managed exit flow

6.4.1.2 Rules and guidelines

Use of a managed exit by the SPMC and a SP is subject to the following rules and guidelines.

1. A SP requests a managed exit in its partition manifest (see Table 3.1 in 3.2.1 *Manifest for isolated partitions*) if it runs in a privileged Exception level. This is one of the following.
   - Secure EL1.
   - Secure Supervisor mode.
2. The SPMC ensures that the state of the Non-secure interrupt that triggers a managed exit does not change in the GIC through any software action until the managed exit has completed.
3. The SPMC ensures that a managed exit is performed for all SPs that have,
   1. Requested this mechanism through their partition manifests and
   2. Entered the *preempted*, *blocked* or *running* states after the most recent switch of execution from the Normal world to the Secure world on the current PE.
4. The SPMC can impose an IMPLEMENTATION DEFINED timeout within which a SP must complete the managed exit.

   The SPMC takes an IMPLEMENTATION DEFINED action if the timeout expires before the managed exit is completed.
5. The SPMC masks Non-secure interrupts while a managed exit is in progress.
6. The SPMC can signal a Secure interrupt to a SP that is performing a managed exit. The SP handles these scenarios through an IMPLEMENTATION DEFINED mechanism.
7. An SP execution context uses the *FFA_MSG_SEND_DIRECT_RESP* interface to complete a managed exit if it was allocated cycles through the *FFA_MSG_SEND_DIRECT_REQ* interface (see 5.3 *Runtime model for FFA_MSG_SEND_DIRECT_REQ*).
8. An SP execution context uses the *FFA_MSG_WAIT* interface to complete a managed exit if it was allocated cycles through the *FFA_RUN* interface (see 5.2 *Runtime model for FFA_RUN*).
9. A SP that has been asked to perform a managed exit could relinquish control without acknowledging the managed exit signal. The SPMC treats this as a valid response to the managed exit request and destroys any internal state to track the progress of the managed exit.

6.4.1.3 Signaling mechanism

A managed exit is signaled by the SPMC to a SP execution context as described below.

1. A S-EL2 SPMC uses the vFIQ or vIRQ signals to signal a managed exit to a SP. The vFIQ signal is used if the SP does not explicitly indicate in its partition manifest that the vIRQ signal must be used. An example flow using this signaling mechanism is illustrated in Figure 6.3.

   The mechanism used by a non-S-EL2 SPMC and a SP for signaling a managed exit is IMPLEMENTATION DEFINED.
2. If the vIRQ signal is used by a SP, the SPMC reserves an interrupt ID to allow the SP to distinguish between a managed exit request and other interrupts.

   This ID can be discovered through the FFA_FEATURES interface (see 11.2 *FFA_FEATURES*) and Table 11.10.

   The managed exit interrupt is signaled as a G1S interrupt to the SP. The interrupt is an SGI or a PPI.

   An example flow using this signaling mechanism is illustrated in Figure 6.4.

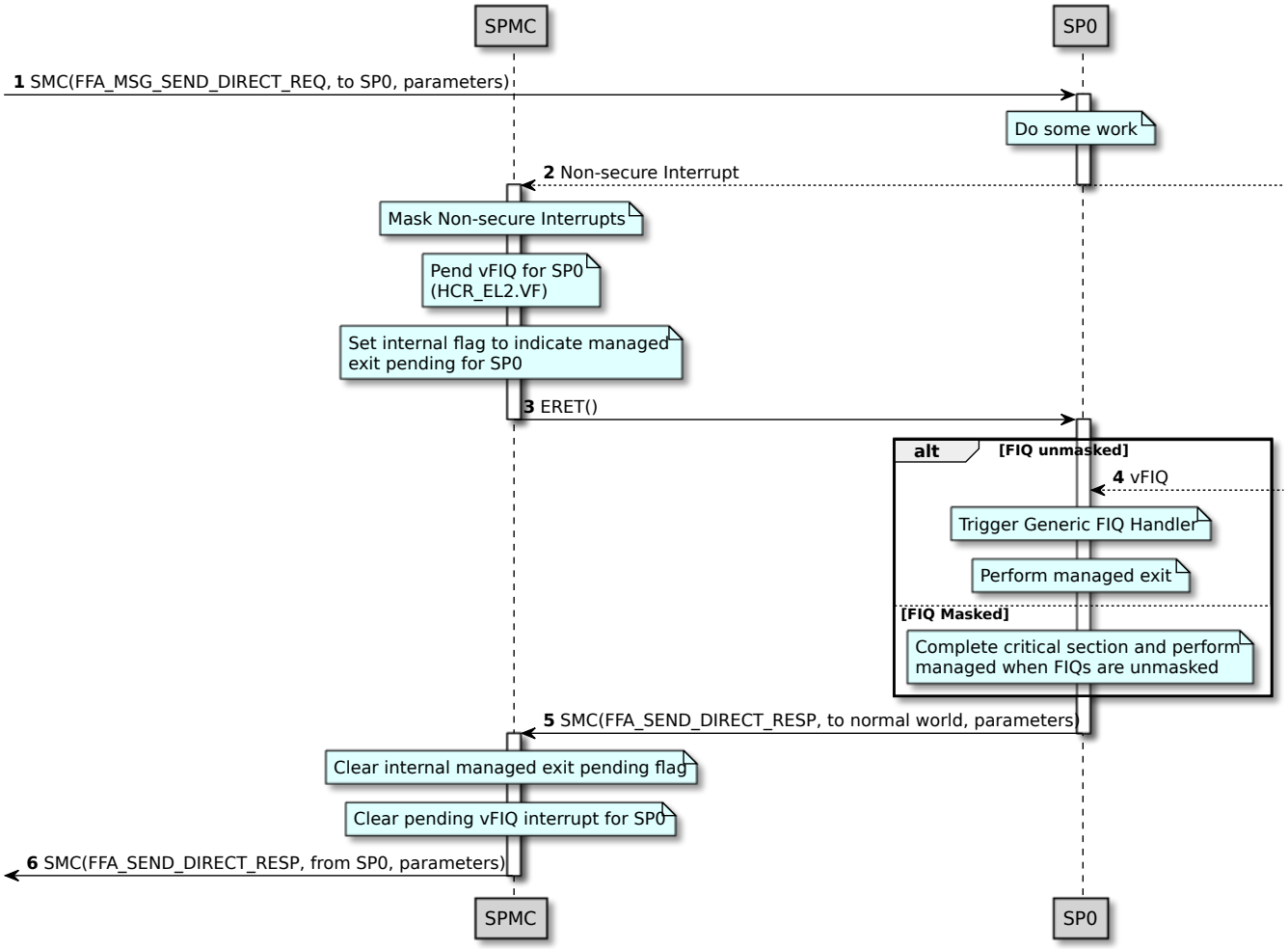

Figure 6.3: Managed exit signaling through a vFIQ

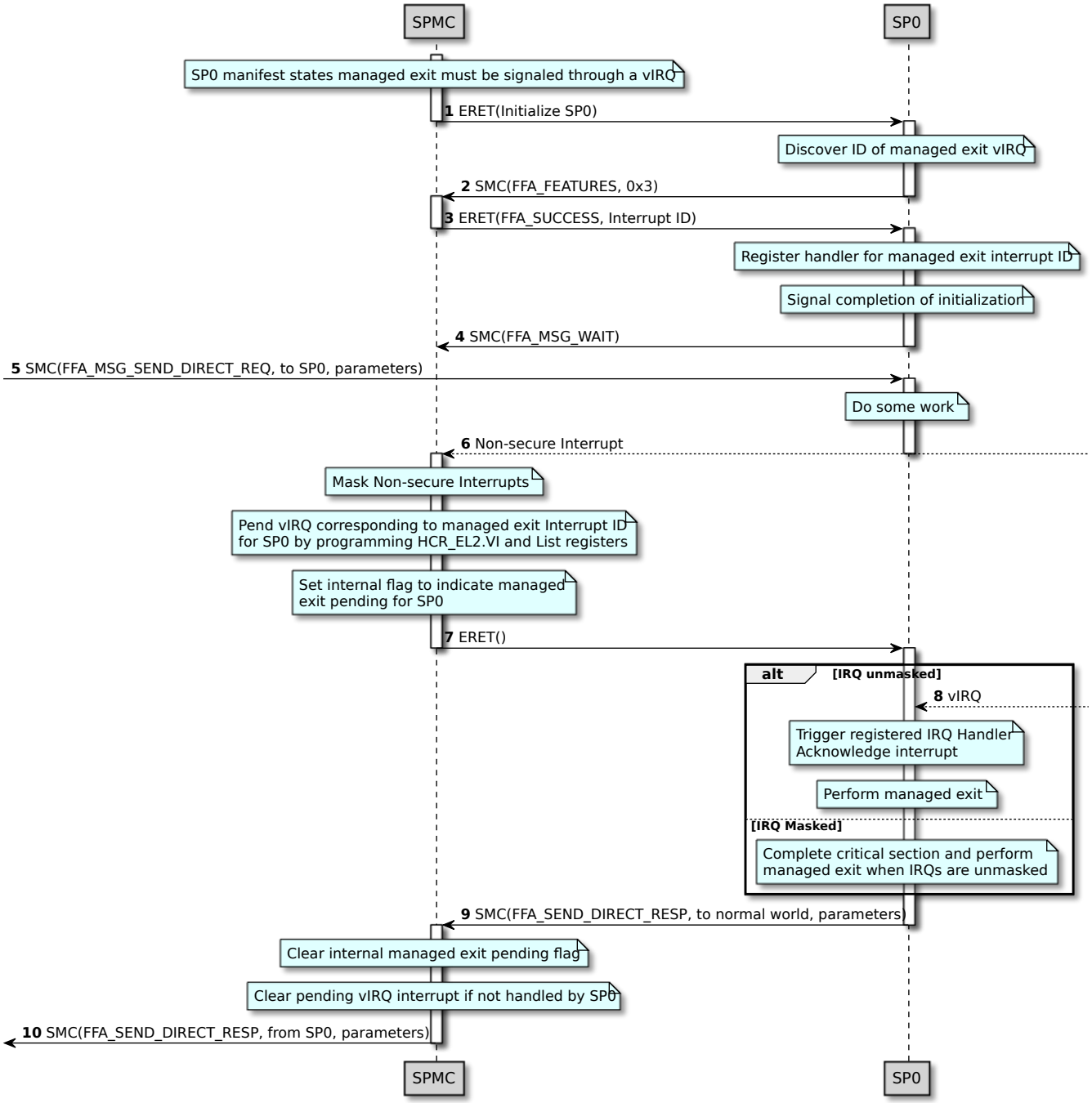

Figure 6.4: Managed exit signaling through a vIRQ

6.4.1.4 Example flows

Multiple SPs could be in a call chain where each SP is blocked on the next SP. Between any two adjacent SPs in the chain, a managed exit could be requested by one of them, none of them or both of them.

Figure 6.5, Figure 6.6, Figure 6.7 and Figure 6.8 illustrate how the SPMC returns control to the Normal world in response to a Non-secure interrupt in each of these scenarios. The first two SPs in the call chain are considered. The same sequence would apply to any other pair of adjacent SPs in a call chain with more than two SPs. The Normal world would be replaced by the SP preceding the pair.

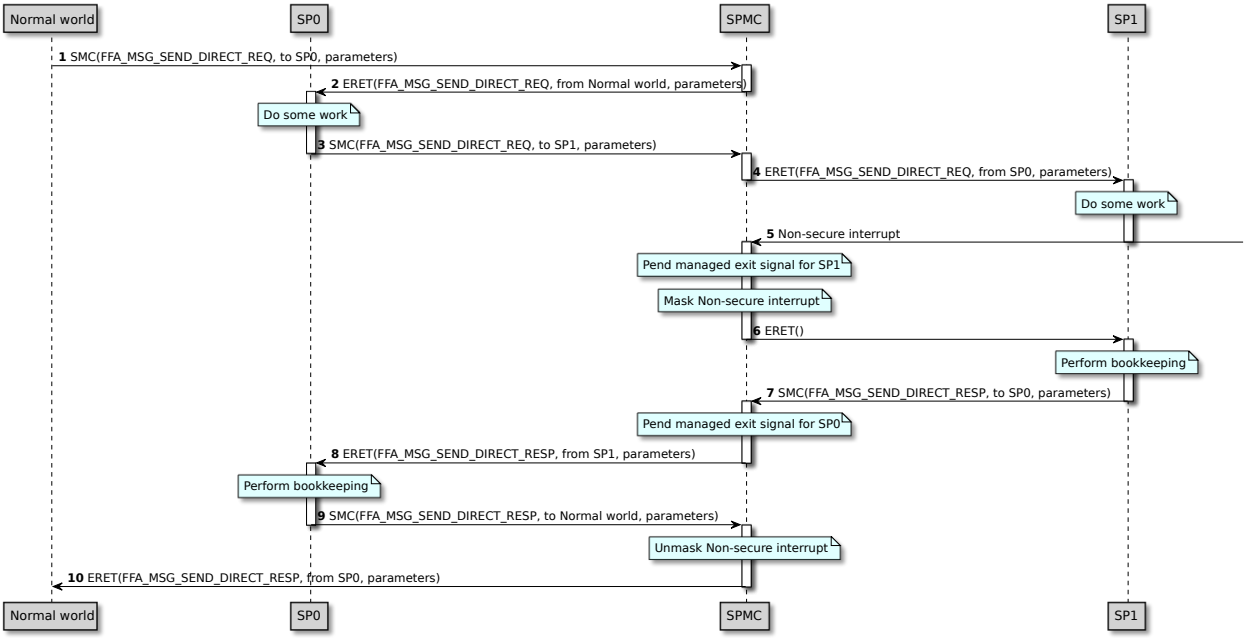

Figure 6.5: Managed exit is supported by SP0 and SP1

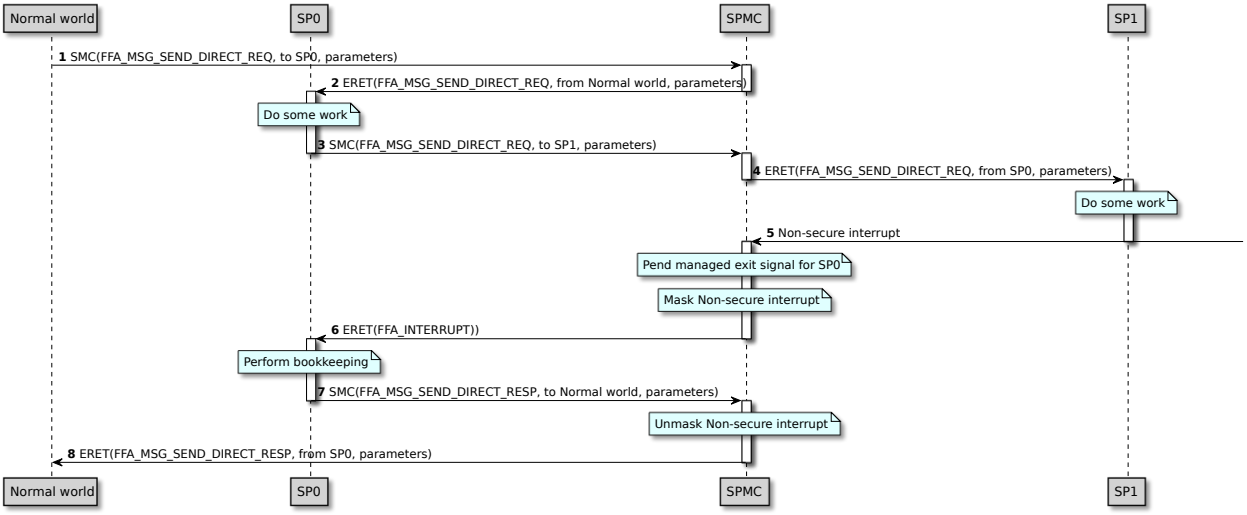

Figure 6.6: Managed exit is supported by SP0 but not by SP1

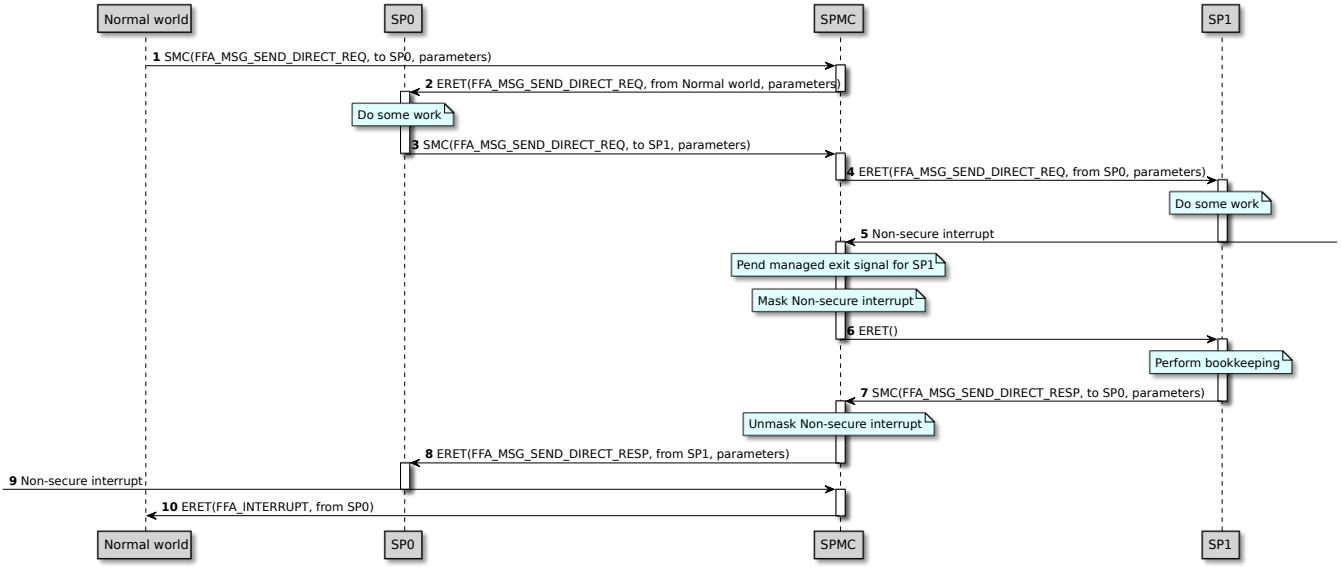

Figure 6.7: Managed exit is supported by SP1 but not SP0

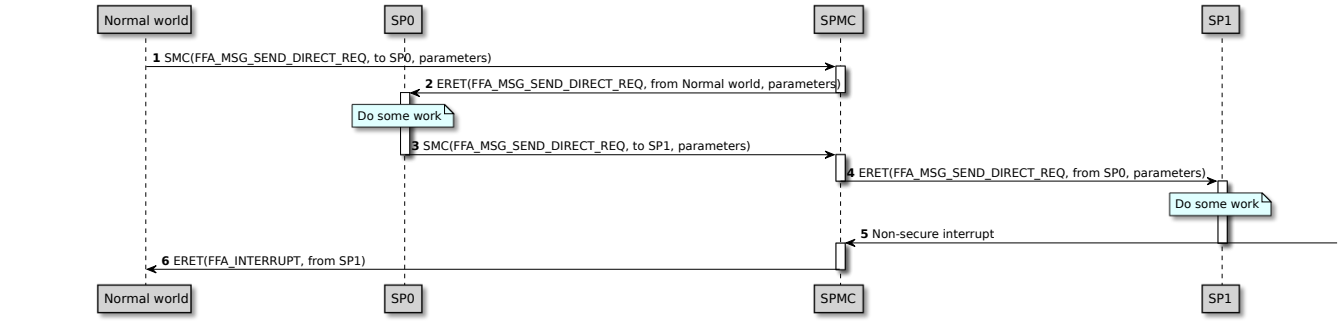

Figure 6.8: Managed exit is not supported by SP1 and SP0

6.5 SP scheduling models

6.5.1 Overview

The scheduling model of an SP specifies the action that the SPMC takes in the following scenarios.

1. An execution context of the SP is in the *running* state. It could be running to,
    1. Handle a Secure interrupt or process a message in response to FFA_MSG_SEND_DIRECT_REQ invocation by another SP in a *call chain* that was initiated by the SP that started handling a Secure interrupt.
    2. Process a message in response to FFA_RUN or FFA_MSG_SEND_DIRECT_REQ invocation by the Normal world or another SP in a *call chain* that was initiated by the SP that was run by the Normal world.

    Also see Chapter 5 *Partition runtime models*.
2. An interrupt of the following types triggers.
    1. A physical Non-secure interrupt. The term *NS-Int* is used to refer to this interrupt.
    2. A physical Secure interrupt targeted to another SP. The term *Other S-Int* is used to refer to this interrupt.

3. A physical Secure interrupt targeted to this SP. The term *Self S-Int* is used to refer to this interrupt.

The actions specified by a SP apply to these interrupts as described below.

1. For a Secure physical interrupt, the Framework assumes that the SPMC signals the corresponding virtual interrupt to the target SP. This is applicable to both S-EL0 and S-EL1 SPs as described in 6.2 *Secure interrupt signaling mechanisms*. An action specified by a SP pertains to the virtual interrupt.
2. For an *NS-Int*, the action applies to the physical Non-secure interrupt. The physical interrupt is handled by an FF-A component in the Normal world.

The actions that can be taken by the SPMC are listed below.

1. The interrupt is handled. The running SP execution context enters the *preempted* state. The SPMC runs the FF-A component that handles the interrupt.
    1. For an *NS-Int*, the SPMC hands control to the Normal world.
    2. For a Secure interrupt, the virtual interrupt is handled by the target SP.

    The term *Interrupt handled* is used to refer to these actions collectively.
2. The *NS-Int* is handled after the S-EL1 SP performs a managed exit (see 6.4.1 *Managed exit*). The running SP execution context enters the *waiting* state. The SPMC hands control to the Normal world. The term *ME* is used to refer to a *managed exit*. The term *Interrupt handled with ME* is used to refer to this action.
3. The interrupt is deferred. It is handled when the running SP execution context,
    1. Enters the *waiting* state.
    2. Finishes handling a Secure interrupt and signals completion as described in 6.3 *Secure interrupt completion mechanisms*.

    The term *Interrupt deferred* is used to refer to this action.

The actions can be listed in an order of decreasing permissiveness as follows.

1. *Interrupt handled*
2. *Interrupt handled with ME*
3. *Interrupt deferred*

6.5.2 Rules and guidelines

The following rules and guidelines govern the specification of a scheduling model.

1. A SP specifies actions in response to interrupts while processing a message. This set of actions is called the *Scheduling model for message processing*.
2. A SP specifies actions in response to interrupts while handling a Secure interrupt. This set of actions is called the *Scheduling model for interrupt handling*.
3. The scheduling model used when the SP is processing a message can be different from the scheduling model used when the SP is handling a Secure interrupt.

    If the two models are different, the SPMC switches between them as described below.
    1. A switch to the *Scheduling model for interrupt handling* is made when a *Self S-Int* is signaled to the SP. Also see 6.2 *Secure interrupt signaling mechanisms*.
    2. A switch to the *Scheduling model for message processing* is made when the SP signals completion of *Self S-Int* handling. Also see 6.3 *Secure interrupt completion mechanisms*.
4. The action *Interrupt handled with ME* cannot be specified by a S-EL0 SP in either scheduling model.
5. The action *Interrupt handled with ME* cannot be specified by a S-EL1 SP in the *Scheduling model for interrupt handling*.

6. The action specified for a type of interrupt when the SP is processing a message cannot be less permissive than the action specified for the same type of interrupt when the SP is handling a Secure interrupt.
7. The action specified in response to an *Other S-Int* in the *Scheduling model for message processing* is *Interrupt handled*.

   This implies that a SP cannot defer another SP's interrupts while processing a message.
8. The action specified by a SP in the *Scheduling model for interrupt handling* for *Self S-Ints* and *Other S-Ints* could be subject to rules enforced by the SPMC as described below.
   1. A Secure interrupt is always handled if it has a higher physical priority than the interrupt being handled by the SP. The action specified by the SP has no effect.
   2. A Secure interrupt is always deferred if it has a lower physical priority than the interrupt being handled by the SP. The action specified by the SP has no effect.
   3. A Secure interrupt is handled if the following conditions are true.
      1. The SP specifies the *Interrupt handled* action for this type of interrupt.
      2. The interrupt has a physical priority equal to the interrupt being handled by the SP.
   4. A Secure interrupt is deferred if the following conditions are true.
      1. The SP specifies the *Interrupt deferred* action for this type of interrupt.
      2. The interrupt has a physical priority equal to the interrupt being handled by the SP.
9. A SP execution context enters the *preempted* state in response to an *Other S-Int* if it specifies the *Interrupt handled* action. The SPMC resumes the preempted SP execution context after the *Other S-Int* is handled.
10. A SP could be preempted by an *Other S-Int* if it specifies the *Interrupt handled* action. The SPMC ensures that the scheduling model of the preempted SP is preserved while the *Other S-Int* is handled.

    For example,

    - SP0 specifies the *Interrupt deferred* action for *NS-Ints* while running.
    - SP1 specifies the *Interrupt handled* action for *NS-Ints* while running.
    - SP0 is preempted by an *Other S-Int* targeted to SP1.
    - An *NS-Int* triggers while SP1 is running.
    - The SPMC switches to the Normal world to allow the *NS-Int* to be handled.

    This sequence violates the scheduling model specified by SP0. The SPMC could mitigate against this scenario by running SP1 at a physical priority level that defers *NS-Ints*.
11. A SP could be scheduled by another SP that was handling a *Self S-Int* or processing a message. The SPMC ensures that the scheduling model of the scheduler SP is preserved while the scheduled SP runs.

    For example,

    - SP0 specifies the *Interrupt deferred* action for *NS-Ints* while running.
    - SP1 specifies the *Interrupt handled* action for *NS-Ints* while running.
    - SP0 schedules SP1 by invoking FFA_MSG_SEND_DIRECT_REQ.
    - An *NS-Int* triggers while SP1 is running.
    - The SPMC switches to the Normal world to allow the *NS-Int* to be handled.

    This sequence violates the scheduling model specified by SP0. The SPMC could mitigate against this scenario by running SP1 at a physical priority level that defers *NS-Ints*.

6.5.2.1 Valid actions for S-EL0 SP scheduling models

Based upon the rules and guidelines listed in 6.5.2 *Rules and guidelines*,

1. Table 6.3 specifies the valid combinations of actions in the scheduling model for message processing for a S-EL0 SP.

2. Table 6.4 specifies the valid combinations of actions in the scheduling model for interrupt handling for a S-EL0 SP.

Table 6.3: Valid actions in scheduling model for message processing for a S-EL0 SP

Config. No.	NS-Int	Self S-Int	Other S-Int
1.	Handled	Deferred	Handled
2.	Deferred	Deferred	Handled

Table 6.4: Valid actions in scheduling model for interrupt handling for a S-EL0 SP

Config. No.	NS-Int	Self S-Int	Other S-Int
1.	Handled	Deferred	Handled
2.	Deferred	Deferred	Handled
3.	Deferred	Deferred	Deferred

6.5.2.2 Valid actions for S-EL1 SP scheduling models

The Framework specifies the following additional rules for S-EL SP w.r.t scheduling models.

1. A S-EL1 SP cannot specify the *Interrupt deferred* action for *Self S-Ints* and *NS-Ints* while processing a message.
2. The action specified by a S-EL1 SP in response to an *NS-Int* in the *Scheduling model for message processing*, must be less or equally permissive as the action taken in response to a Secure interrupt in the same scheduling model.

   For example, while processing a message, a SP cannot choose the *Interrupt handled* action for an *NS-Int* and the *Interrupt deferred* action for *Self S-Ints* and *Other S-Ints*.
3. The action specified by a S-EL1 SP in response to an *NS-Int* in the *Scheduling model for Interrupt handling*, must be less or equally permissive as the action taken in response to a Secure interrupt in the same scheduling model.

   For example, while handling a Secure interrupt, a SP cannot choose the *Interrupt handled* action for an *NS-Int* and the *Interrupt deferred* action for *Self S-Ints* and *Other S-Ints*.

Based upon the rules and guidelines listed in this section and 6.5.2 *Rules and guidelines*,

1. Table 6.5 specifies the valid combinations of actions in the scheduling model for message processing for a S-EL1 SP.
2. Table 6.6 specifies the valid combinations of actions in the scheduling model for interrupt handling for a S-EL1 SP.

Table 6.5: Valid actions in scheduling model for message processing for a S-EL1 SP

Config. No.	NS-Int	Self S-Int	Other S-Int
1.	Handled	Handled	Handled
2.	Handled with ME	Handled	Handled

Table 6.6: Valid actions in scheduling model for interrupt handling for a S-EL1 SP

Config. No.	NS-Int	Self S-Int	Other S-Int
1.	Handled	Handled	Handled
2.	Deferred	Handled	Deferred
3.	Deferred	Handled	Handled
4.	Deferred	Deferred	Deferred
5.	Deferred	Deferred	Handled

6.5.3 Reference of possible actions

Table 6.7 lists all possible actions that can be taken in response to an *NS-Int* while a SP execution context is running. It also specifies which combination of actions is valid along with rationale.

Table 6.7: List of actions in response to a *NS-Int*

No.	Action during message processing	Action during Secure interrupt handling	Valid	Description
1.	Interrupt handled	Interrupt handled with ME	No	• Managed exit mechanism is not available during Secure interrupt handling.
2.	Interrupt handled	Interrupt handled	Yes	• *NS-Ints* trigger a switch to the Normal world.

No.	Action during message processing	Action during Secure interrupt handling	Valid	Description
3.	Interrupt handled	Interrupt deferred	Yes	• *NS-Ints* trigger a switch to the Normal world during message processing. • *NS-Ints* remain pending during Secure interrupt handling.
4.	Interrupt handled with ME	Interrupt handled with ME	No	• Managed exit mechanism is not available during Secure interrupt handling.
5.	Interrupt handled with ME	Interrupt handled	No	• Action during Secure interrupt handling cannot be more permissive than action during message processing.
6.	Interrupt handled with ME	Interrupt deferred	Yes	• *NS-Ints* trigger a switch to the Normal world during message processing after the SP execution context performs a managed exit. • *NS-Ints* remain pending during Secure interrupt handling.
7.	Interrupt deferred	Interrupt handled with ME	No	• Action during Secure interrupt handling cannot be more permissive than action during message processing.
8.	Interrupt deferred	Interrupt handled	No	• Action during Secure interrupt handling cannot be more permissive than action during message processing.
9.	Interrupt deferred	Interrupt deferred	Yes	• Only applicable to S-EL0 SPs. A S-EL1 SP cannot defer an *NS-Int* during message processing.

Table 6.8 lists all possible actions that can be taken in response to an *Other S-Int* while a SP execution context is running. It also specifies which combination of actions is valid along with rationale.

Table 6.8: List of actions in response to an *Other S-Int*

No.	Action during message processing	Action during Secure interrupt handling	Valid	Description
1.	Interrupt handled	Interrupt handled	Yes	• As described in rule 8 in 6.5.2 *Rules and guidelines*.
2.	Interrupt handled	Interrupt deferred	Yes	• As described in rule 8 in 6.5.2 *Rules and guidelines*.
3.	Interrupt deferred	Interrupt handled	No	• Action during Secure interrupt handling cannot be more permissive than action during message processing.
4.	Interrupt deferred	Interrupt deferred	No	• As described in rule 7.

Table 6.9 lists all possible actions that can be taken in response to a *Self S-Int* while a SP execution context is running. It also specifies which combination of actions is valid along with rationale.

Table 6.9: List of actions in response to a *Self S-Int*

No.	Action during message processing	Action during Secure interrupt handling	Valid	Description
1.	Interrupt handled	Interrupt handled	Yes	• Only applicable to S-EL1 SPs. Interrupt is signaled as described in Table 6.2.
2.	Interrupt handled	Interrupt deferred	Yes	• Only applicable to S-EL1 SPs. Interrupt is signaled as described in Table 6.2 during message processing. • *Self S-Int* remains pending during Secure interrupt handling.
3.	Interrupt deferred	Interrupt handled	No	• Action during Secure interrupt handling cannot be more permissive than action during message processing.

No.	Action during message processing	Action during Secure interrupt handling	Valid	Description
4.	Interrupt deferred	Interrupt deferred	Yes	• Only applicable to S-EL0 SPs. A S-EL1 SP cannot defer *Self S-Ints* during message processing as specified in rule 1 in 6.5.2.2 *Valid actions for S-EL1 SP scheduling models*.

6.5.4 Discovery and setup

A SP specifies its *Scheduling model for message processing* and *Scheduling model for interrupt handling* in its partition manifest (see 3.2.1 *Manifest for isolated partitions*). A SP must specify at least one model. The model is specified as a list of valid actions in response to an interrupt. The valid actions are listed in the following tables.

1. Table 6.9
2. Table 6.8
3. Table 6.7

Table 6.10 describes example scheduling models specified by a S-EL1 SP that is not co-resident with any other SP and is managed by a S-EL2 SPMC.

- The actions corresponding to *Other S-Ints* are not applicable.
- During message processing, the SP performs a managed exit in response to an *NS-Int*. It handles *Self S-Ints*.
- During Secure interrupt handling, the SP defers an *NS-Int*. It defers *Self S-Ints* as well.

Table 6.10: Example scheduling models for a single S-EL1 SP

Interrupt type	Action taken during message processing	Action taken during Secure interrupt handling
NS-Int	Interrupt handled with ME	Interrupt deferred
Other S-Int	Not applicable	Not applicable
Self S-Int	Interrupt handled	Interrupt deferred

Table 6.11 describes example scheduling models specified by a S-EL1 SP that is co-resident with at least one other SP. All SPs are managed by a S-EL2 SPMC.

- During message processing, the SP performs a managed exit in response to an *NS-Int*. It handles *Self S-Ints*. The SPMC ensures *Other S-Ints* are handled too.
- During Secure interrupt handling, the SP defers an *NS-Int*. It defers *Self S-Ints*. The SPMC ensures *Other S-Ints* of same priority are deferred and those of higher priority are handled.

Table 6.11: Example scheduling models for a S-EL1 SP resident with at least another SP

Interrupt type	Action taken during message processing	Action taken during Secure interrupt handling
NS-Int	Interrupt handled with ME	Interrupt deferred
Other S-Int	Interrupt handled	Interrupt deferred

Interrupt type	Action taken during message processing	Action taken during Secure interrupt handling
Self S-Int	Interrupt handled	Interrupt deferred

6.5.4.1 Support for legacy run-time models

Version 1.0 of the Framework allows a S-EL0 SP to specify its run-time model in its partition manifest. It can specify the *Run to completion* or the *Preemptible* models. These models are deprecated in the current version of the Framework. To maintain backwards compatibility, the SPMC must convert these run-time models to a scheduling model as described below.

- The *Run to completion* model is recommended for S-EL0 SPs that only handle Secure interrupts. Hence, a definition of the *Scheduling model for message processing* is not required. The *Scheduling model for interrupt handling* is described in Table 6.12.

Table 6.12: Scheduling model for legacy run-to-completion run-time model

Interrupt type	Action taken during Secure interrupt handling
NS-Int	Interrupt deferred
Other S-Int	Interrupts of same priority are deferred. Only interrupts of higher priority are handled
Self S-Int	Interrupt deferred

- The *Preemptible* model is recommended for S-EL0 SPs that only process messages. Hence, a definition of the *Scheduling model for interrupt handling* is not required. The *Scheduling model for message processing* is described in Table 6.13.

Table 6.13: Scheduling model for legacy preemptible run-time model

Interrupt type	Action taken during message processing
NS-Int	Interrupt handled
Other S-Int	Interrupt handled
Self S-Int	Not applicable

Chapter 7
Notifications

7.1 Overview

The notification mechanism enables a requester endpoint (henceforth called the *Sender*) to notify a service provider endpoint (henceforth called the *Receiver*) about an event with non-blocking semantics.

A notification is akin to the doorbell between two endpoints in a communication protocol that is based upon the doorbell/mailbox mechanism. The term *doorbell* is used in lieu of *notification* in contexts where it makes it easier to understand a concept under discussion.

The Framework is responsible for the delivery of the notification from the Sender to the Receiver without blocking the Sender.

The Receiver endpoint relies on another software component for allocation of CPU cycles to handle a notification. This component is the primary or a secondary scheduler (see 2.11 *Primary scheduler*). It is called the *Receiver's scheduler* in the context of notifications in the rest of this specification.

The Framework is responsible for informing the Receiver's scheduler that the Receiver must be run since it has a pending notification.

Figure 7.1 illustrates the notification mechanism and its participants.

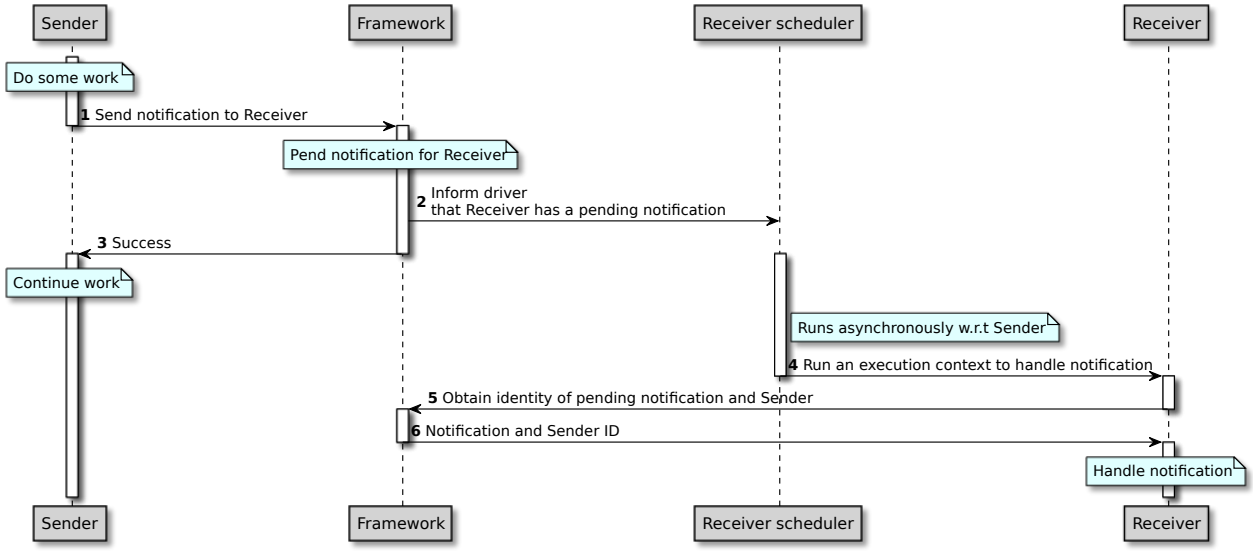

Figure 7.1: Example notification flow

Support for notifications in the Framework is governed by the following common rules. Rules specific to a particular aspect of notification support are specified the following sub-sections.

1. Each endpoint is provided with 64 notifications that can be signaled to it by only SPs in the system. These are called *SP notifications*.
2. Each endpoint is provided with 64 notifications that can be signaled to it by only VMs in the system. These are called *VM notifications*.
3. The partition manager of each endpoint provides it with 64 notifications that can be signaled by the partition managers in the system.
    1. 32 notifications are reserved for signaling by the SPMC
    2. 32 notifications are reserved for signaling by the Hypervisor

    These notifications are called *Framework Notifications*. See 7.8 *Framework Notifications*.
4. The identity of a notification is its bit position in a bitmap managed by the partition manager on behalf of a Receiver.
5. In the framework notifications bitmap, the lower 32 bits are reserved for signaling by the SPMC.
6. In the framework notifications bitmap, the top 32 bits are reserved for signaling by the Hypervisor.
7. The Partition manager reserves memory for each notification bitmap at the time of endpoint creation. Also see 7.3 *Notification bitmap setup*.
8. The Framework provides an interface to the Sender to specify the notification to signal to the Receiver. Also see 15.5 *FFA_NOTIFICATION_SET*.

    A Sender signals a notification by requesting its Partition manager to set the corresponding bit in the notifications bitmap of the Receiver.
    1. If the Sender is a VM, the bit is set in the VM notifications bitmap of the Receiver.
    2. If the Sender is a SP, the bit is set in the SP notifications bitmap of the Receiver.
9. The VM notifications and Hypervisor framework notifications bitmaps for a VM are written to by the Hypervisor.
10. The VM notifications and Hypervisor framework notifications bitmap for a SP are written to by the SPMC.

11. The SP notifications and SPMC framework notification bitmaps for both VMs and SPs are written to by the SPMC.
12. The Framework provides an interface to the Receiver to determine the identity of the notification and its Sender. Also see 15.6 *FFA_NOTIFICATION_GET*.
13. The Framework provides no guarantees when a notification will be handled by the Receiver.
14. The Framework does not provide a mechanism for a Sender to determine if the Receiver has handled the notification. If required, the Sender and Receiver must enable this through an IMPLEMENTATION DEFINED mechanism.

Guidance on discovering support for notifications is provided in 7.7 *Feature discovery*.

7.1.1 Use cases

The Framework provides guidance for support of notifications to address the requirements of the following types of use cases.

1. The blocking semantics associated with message exchange using direct messaging (see 4.1.2 *Direct messaging*) are not desirable in a scenario where the Sender endpoint must make progress in tandem with the Receiver endpoint processing its request. For example,
   - A secondary endpoint is scheduled by the primary scheduler and requests services implemented in a Trusted OS SP. It is not desirable to allocate cycles to the SP from the quota allocated to the secondary endpoint by the primary scheduler.
   - The Trusted OS could request a service provided by another SP. It might too not want to allocate cycles to the SP from the quota allocated to it by its scheduler.
2. An asynchronous signaling mechanism is required by the Secure world to notify the Normal world. For example,
   1. A Secure interrupt preempts the Normal world
   2. The Secure interrupt is handled in a SP
   3. The SP needs to signal the Normal world about an event signaled by the Secure interrupt e.g., completion of an operation previously requested by the Normal world.

   The SP cannot send a direct message to the Normal world and block until the response is received. This is because the Normal world is in a preempted state. Hence, a non-blocking mechanism is required that enables the SP to notify the Normal world.

   In the same example above, it is possible that the SP only performs *top-half* interrupt handling and requires CPU cycles to perform *bottom-half* interrupt handling. These cycles are allocated by the SP's scheduler in the Normal world. The SP cannot send a direct message. It needs another mechanism to signal to its Scheduler that it must be run.

7.2 Notification bitmap permissions

The following rules govern the permissions an FF-A component has on a notification bitmap of an endpoint.

1. Each endpoint has read-write permissions on each of its bitmaps.
2. Permissions of the Hypervisor and SPMC on the notification bitmap of each type of endpoint are described in Table 7.1.
3. Permissions of VMs and SPs on the notification bitmap of each type of endpoint are described in Table 7.2.

Table 7.1: Hypervisor and SPMC permissions on an endpoint notification bitmap

Endpoint type	Notifications bitmap	SPMC	Hypervisor
SP	SP	RW	NA
SP	VM	RW (Directed by Hypervisor)	RW
SP	SPMC framework	RW	NA
SP	HYP framework	RW (Directed by Hypervisor)	RW
VM	SP	RW	RO
VM	VM	NA	RW
VM	SPMC framework	RW	RO
VM	HYP framework	NA	RW

Table 7.2: VM and SP permissions on an endpoint notification bitmap

Endpoint type	Notifications bitmap	Implemented in	Other SP permissions	Other VM permissions
SP	SP	SPMC	Write-only	NA
SP	VM	SPMC	NA	Write-only
SP	SPMC framework	SPMC	NA	NA
SP	HYP framework	SPMC	NA	NA
VM	SP	SPMC	Write-only	NA
VM	VM	Hypervisor	NA	Write-only
VM	SPMC framework	SPMC	NA	NA
VM	HYP framework	Hypervisor	NA	NA

7.3 Notification bitmap setup

An endpoint's notification bitmaps are setup before it configures its notifications and before other endpoints and partition managers can start signaling these notifications. Also see 7.4 *Notification configuration* and 7.5 *Notification signaling*.

The following rules govern the setup of a notification bitmap of an endpoint.

1. For a VM, the Hypervisor reserves memory for its VM and Hypervisor framework notification bitmaps before initializing it.
2. For a VM, the SPMC reserves memory for its SP and SPMC framework notification bitmaps before the Hypervisor initializes it.
3. The Hypervisor uses the FFA_NOTIFICATION_BITMAP_CREATE interface to request the SPMC to allocate the SP and SPMC framework notification bitmaps for the VM prior to its initialization (see 15.1 *FFA_NOTIFICATION_BITMAP_CREATE*).
4. The Hypervisor does not initialize a VM if memory cannot be reserved for all its notification bitmaps.
5. For a SP, the SPMC reserves memory for its VM, SP and framework notification bitmaps before initializing it.
6. The SPMC does not initialize a SP if memory cannot be reserved for its notification bitmaps.
7. The Hypervisor uses the FFA_NOTIFICATION_BITMAP_DESTROY interface to inform the SPMC when it destroys a VM (see 15.2 *FFA_NOTIFICATION_BITMAP_DESTROY*). The SPMC frees memory for the VM's SP and SPMC framework notification bitmaps.

Within an endpoint, there could be one or more consumers of its VM and SP notifications. The mechanism used by the endpoint to manage access to its notifications amongst their consumers is IMPLEMENTATION DEFINED.

Figure 7.2 illustrates how the Hypervisor and SPMC create notification bitmaps on behalf of a VM and SP respectively.

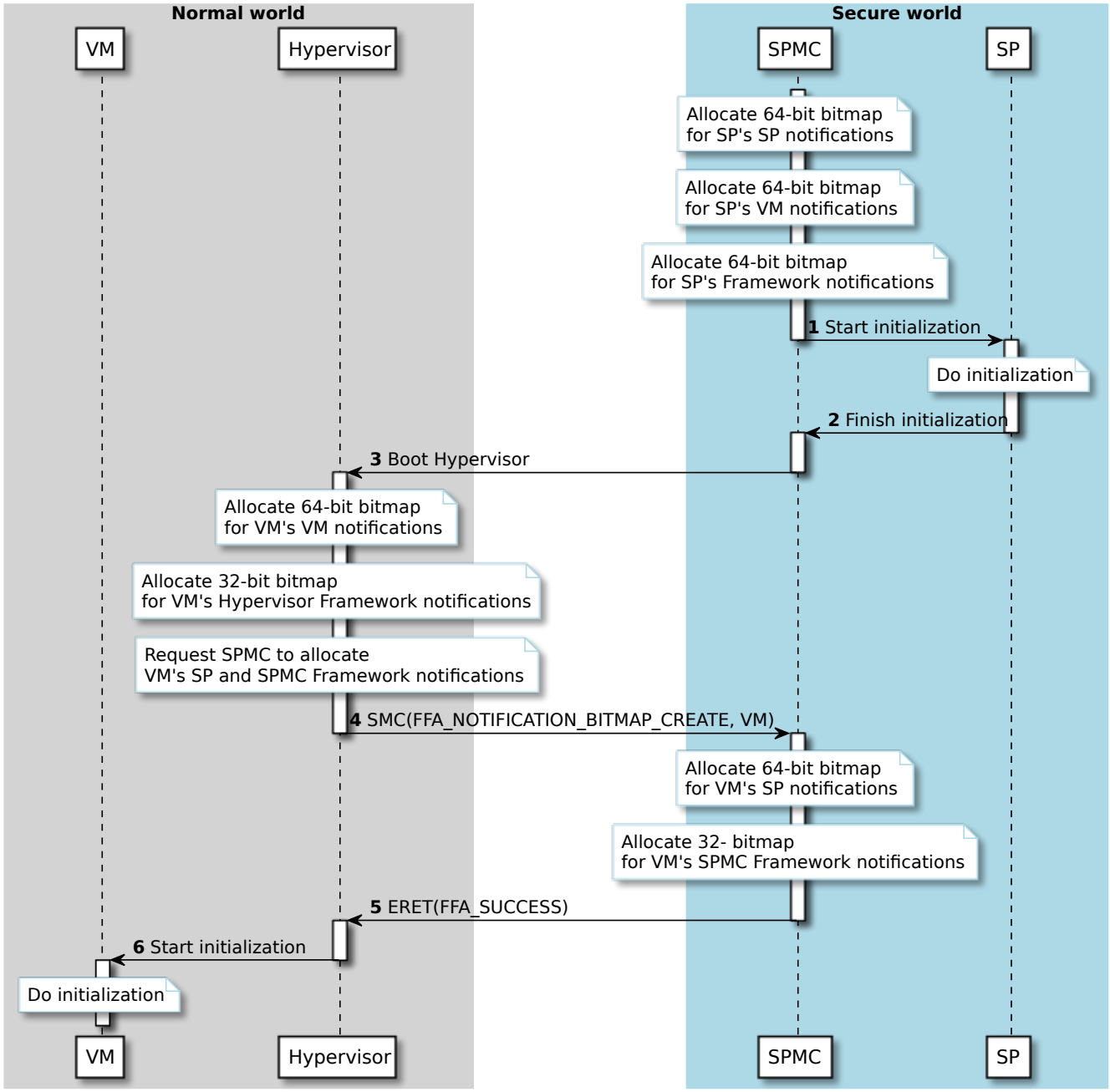

Figure 7.2: Notification bitmap creation for a VM and SP

7.4 Notification configuration

A Receiver and its scheduler configure a notification as described below before it can be signaled by other endpoints and partition managers. Also see 7.5 *Notification signaling*.

1. The Receiver and its scheduler configure support for handling interrupts used by the Framework for notification signaling. See 7.4.1 *Notification interrupt setup*.
2. The Receiver binds a non-framework notification to an endpoint that is allowed to signal it. See 7.4.2 *Notification binding*.

7.4.1 Notification interrupt setup

The following rules govern the configuration of interrupts used by the Framework for signaling notifications.

1. The Framework uses the *Schedule Receiver interrupt* to inform the Receiver's scheduler that the Receiver must be run to handle a pending notification.
2. The Framework uses the *Notification pending interrupt* to inform the Receiver that it has a pending notification. This is a virtual interrupt and is used by the following type of Receivers.
    1. A VM running under a Hypervisor.
    2. A S-EL1 SP running under a S-EL2 SPMC.
3. A Receiver's scheduler obtains the description of the *Schedule Receiver interrupt* by invoking the FFA_FEATURES interface (see 11.2 *FFA_FEATURES*).

    Feature ID *0x2* is allocated to obtain a description of the *Schedule Receiver interrupt*.

    The description of the *Schedule Receiver interrupt* is encoded as specified in Table 11.10.
4. A Receiver obtains the description of the *Notification pending interrupt* by invoking the FFA_FEATURES interface (see 11.2 *FFA_FEATURES*).

    Feature ID *0x1* is allocated to obtain a description of the *Notification pending interrupt*.

    The description of the *Notification pending interrupt* is encoded as specified in Table 11.10.

Figure 7.3 illustrates an example setup of the Receiver scheduler interrupt in the primary endpoint for a Receiver endpoint.

- The Receiver endpoint has a counterpart driver in the primary endpoint. The primary endpoint implements an FF-A driver that allows access to Framework functionality to other drivers including the Receiver endpoint driver. The Receiver endpoint driver runs an execution context of the Trusted OS in response to requests from a client application or a pending notification.
- The FF-A driver discovers the Receiver scheduler interrupt.
- The Receiver endpoint driver registers a callback function with the FF-A driver.
- The FF-A driver calls this function if there is a pending notification for the Receiver endpoint and it must be scheduled by its driver.

From the Framework's perspective, the primary scheduler is the Receiver's scheduler in this example. Within the primary endpoint, the Receiver endpoint driver is the Receiver's scheduler.

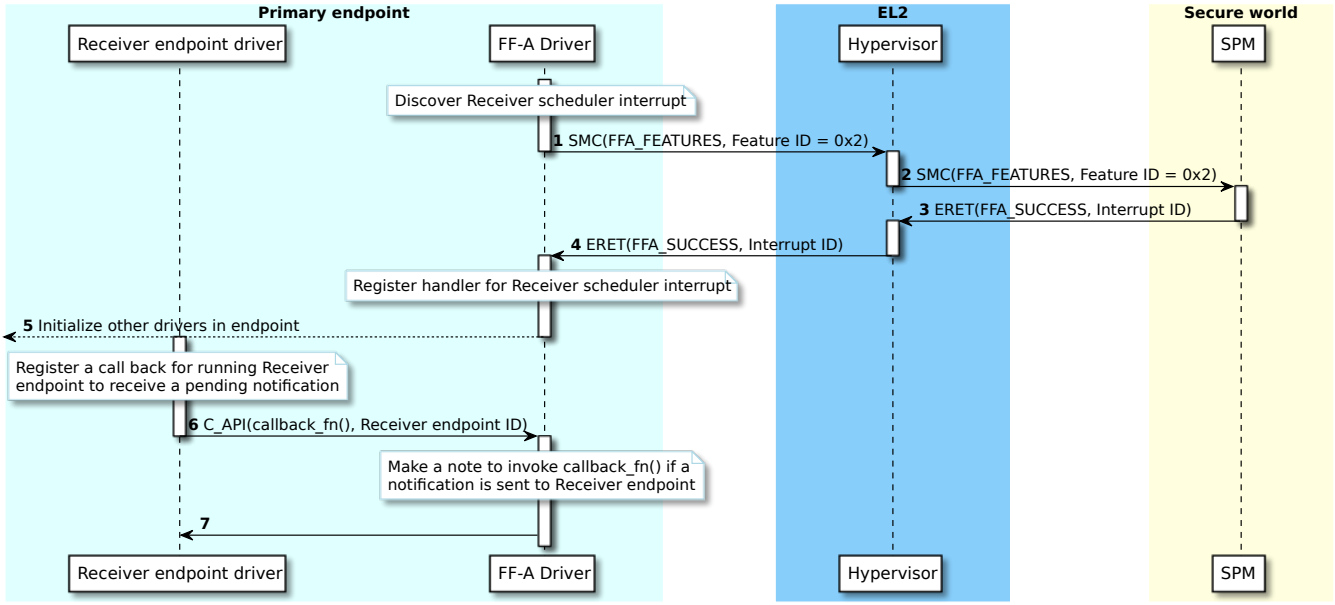

Figure 7.3: Receiver scheduler interrupt setup in primary endpoint

Figure 7.4 illustrates an example setup of the notification pending interrupt in a Receiver endpoint.

- The Receiver endpoint implements a service driver that can receive notifications. It also implements an FF-A driver that allows access to Framework function to the service driver.
- The FF-A driver discovers the notification pending interrupt.
- The Receiver service driver requests the FF-A driver to allocate a set of notification IDs. The notifications are used by clients to access this service.
- The Receiver service driver registers a callback function with the FF-A driver.
- The FF-A driver calls this function if there is a pending notification allocated to the Receiver service driver.

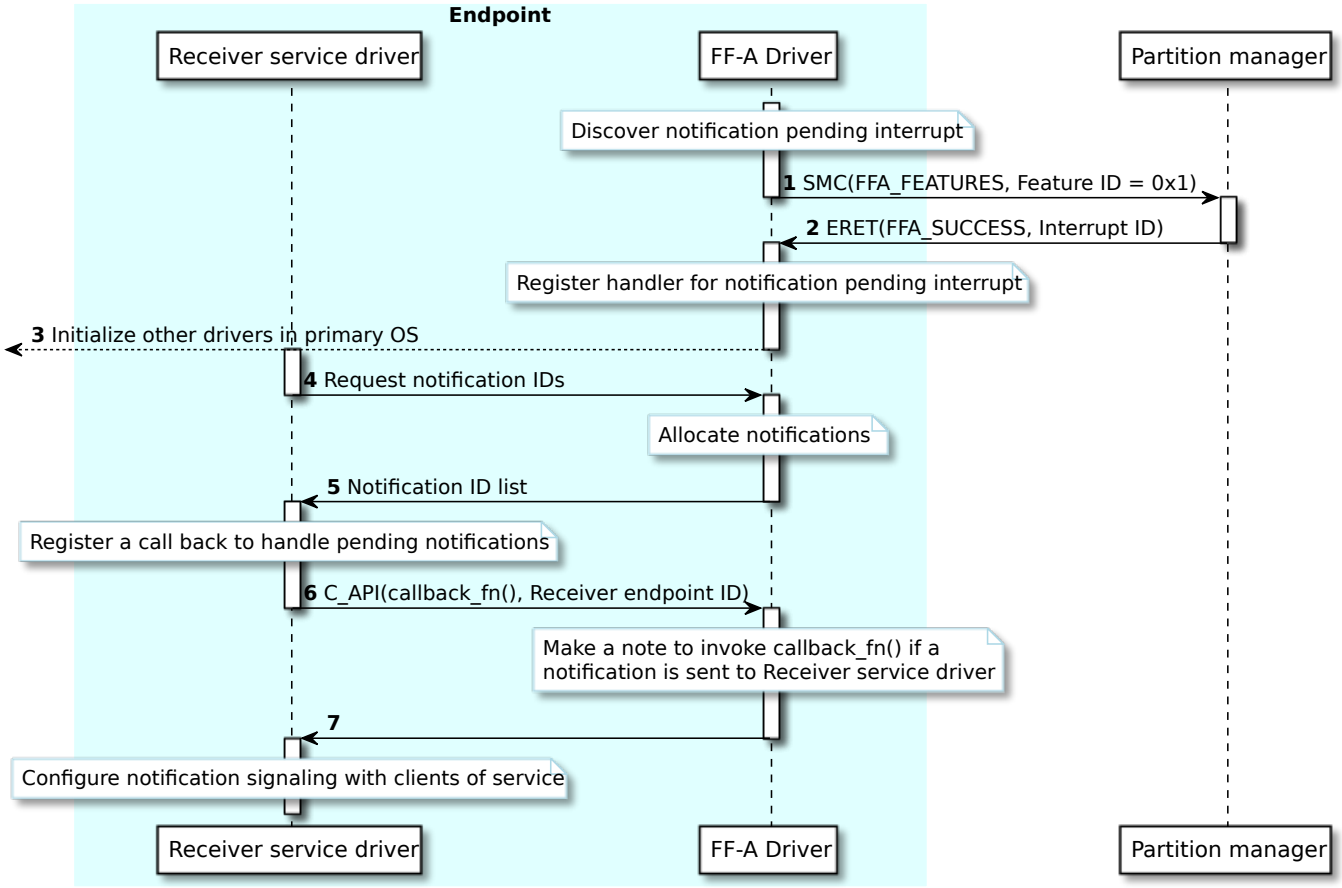

Figure 7.4: Notification pending interrupt setup in a Receiver endpoint

7.4.1.1 Interrupt properties

The following rules govern the properties of the *Notification pending interrupt* and *Schedule Receiver interrupt*.

1. The type of interrupts should be inferred from the interrupt ID specified in Table 11.11. For example, in the Arm GIC architecture, the interrupt ID indicates whether it is a PPI, SGI or SPI.
    1. If the interrupt is a PPI, the same interrupt ID is used for this interrupt on all PEs in the system.
    2. If the interrupt is an SGI, it is not signaled such that multiple PEs receive the interrupt independently and concurrently. The interrupt is signaled so that only a single PE receives it.

       The Arm GIC architecture allows signaling of an SGI through the targeted list model. In this model, upon a write to the *ICC_SGIxR_EL1* or *ICC_ASGI1R_EL1* register, multiple PEs could receive the interrupt independently. The above rule disallows this signaling model. Instead, an SGI can be signaled only to the current PE like a PPI.
2. Both interrupts are edge-triggered.
3. The Security state of the *Notification pending interrupt* is the same as the Security state of the endpoint it is targeted to.
4. The Security state of the *Schedule Receiver interrupt* is Non-secure.

   The delivery of the physical *Schedule Receiver interrupt* from the Secure state to the Non-secure state depends upon the state of the interrupt controller as configured by the Hypervisor. This is beyond the control of the Secure world. It is possible that the interrupt gets lost.

- For example, the *Schedule Receiver interrupt* could be a PPI and signaled on a PE when the Hypervisor is about to turn the PE off through a PSCI CPU_OFF call. The interrupt would not be handled by the Hypervisor in this scenario.

The Framework makes the following recommendation w.r.t use of an SGI as the *Schedule Receiver interrupt*.

- The Arm GIC specification defines 16 SGIs. It recommends that they are equally divided between the Non-secure and Secure states. General-purpose operating systems in the Non-secure state typically do not have SGIs to spare. The usage of SGIs in the Secure state is limited. It is more likely that software in the Secure world does not use all the SGIs allocated to it. Arm recommends that the Secure world software *donates* an unused SGI to the Normal world for use as the *Schedule Receiver interrupt*. This implies that Secure world software must configure the SGI in the GIC as a Non-secure interrupt before presenting it to the Normal world through the FFA_FEATURES ABI as described in 7.4.1 *Notification interrupt setup*.

7.4.2 Notification binding

A Receiver must bind a non-framework notification to a Sender before the latter can signal the notification to the former. Effectively, the Receiver assigns one or more *doorbells* to a specific Sender. Only the Sender can ring these *doorbells*.

The following rules govern the binding of notifications.

1. A Receiver uses the FFA_NOTIFICATION_BIND interface to bind one or more notifications to the Sender. (see 15.3 *FFA_NOTIFICATION_BIND*).
2. A notification is not bound to any Sender endpoint at the time of the Receiver initialization.
3. A notification is signaled and pended only if it is bound to a Sender endpoint.
4. The notification bitmap in which a notification is bound to a Sender endpoint is determined by the security state of the Sender endpoint.
    1. If the Sender is a VM, the VM notifications bitmap is used.
    2. If the Sender is a SP, the SP notifications bitmap is used.
5. A Receiver endpoint un-binds a notification from a Sender endpoint to stop the notification from being signaled. It uses the FFA_NOTIFICATION_UNBIND interface to do this (see 15.4 *FFA_NOTIFICATION_UNBIND*).
6. A notification is unbound only if it is not in a pending state.
7. A notification is one of the following types.
    - It is signaled to and handled by a specific execution context or vCPU of the Receiver endpoint. These notifications are called *Per-vCPU notifications*. The vCPU is specified by the Sender.
    - It is signaled to the Receiver endpoint and is handled by an execution context or vCPU that is chosen by the Receiver's scheduler or partition manager through an IMPLEMENTATION DEFINED mechanism. These notifications are called *Global notifications*.

    Also see 7.5 *Notification signaling*.
8. The type of notification is specified by the Receiver endpoint when the notification is bound to the Sender endpoint.
9. The type of *Notification pending interrupt* used by a Receiver is a PPI or SGI, if the Receiver implements one or more *per-vCPU* notifications.

    This constraint enables delivery of the interrupt to the vCPU specified as the target of the notification.
10. An unbound notification is neither global nor per-vCPU i.e., it does not have a type associated with it.

Figure 7.5 illustrates an example flow of how a VM can bind a global notification to a SP

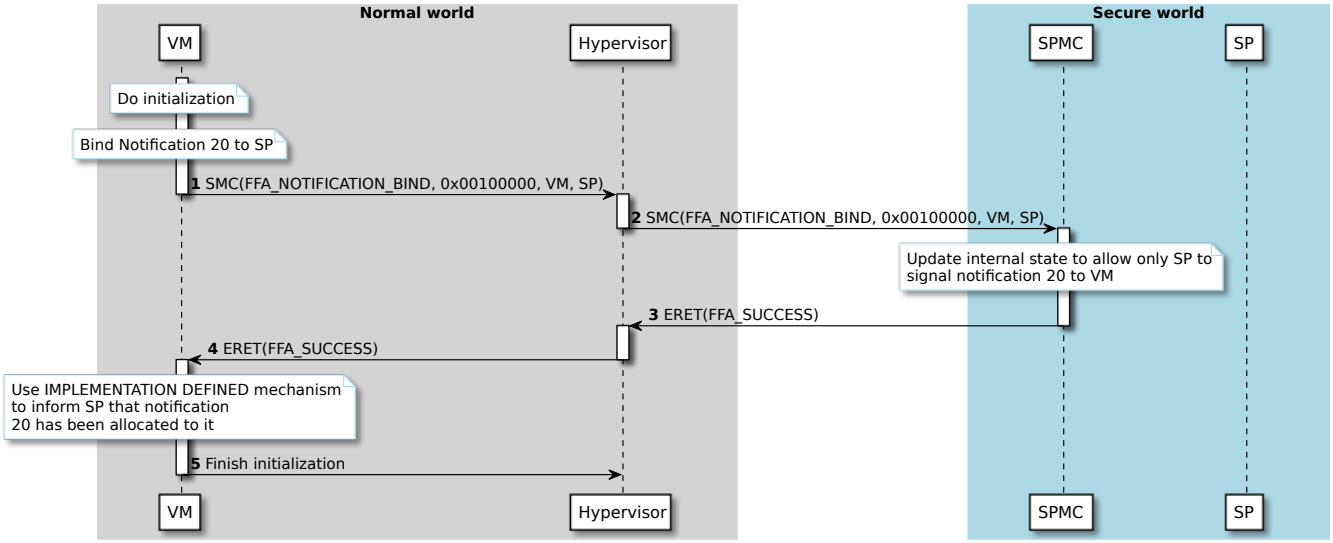

Figure 7.5: Binding a global notification from VM to SP

An IMPLEMENTATION DEFINED mechanism is used by a Receiver and a Sender to negotiate the notification ID that the Sender will use to signal to the Receiver. Figure 7.6 illustrates an example flow of how,

- A SP binds a global notification to a VM.
- The VM discovers the identity of the notification.

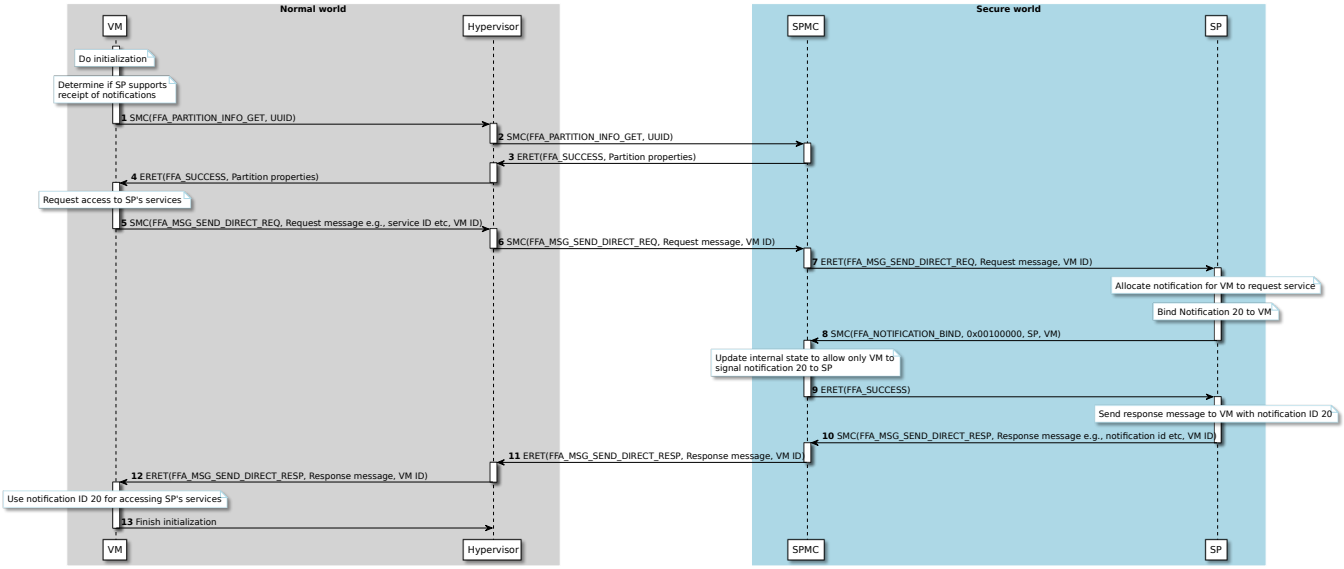

Figure 7.6: Notification binding between a VM and SP

7.5 Notification signaling

Notification signaling is performed in the three phases.

1. The Sender requests the Receiver's partition manager to ring a doorbell that was bound to the Sender by the Receiver.
2. The Sender's partition manager informs the Receiver's scheduler that one of the Receiver's doorbells has been rung.
3. The Receiver is run by its scheduler. It obtains the identity of the doorbell that was rung from its partition manager.

The following rules govern the signaling of notifications.

1. A Sender uses the FFA_NOTIFICATION_SET interface to signal a notification to the Receiver (see 15.5 *FFA_NOTIFICATION_SET*).
2. The notification bitmap in which a notification is signaled to the Receiver is determined by the security state of the Sender endpoint.
    1. If the Sender is a VM, the VM notifications bitmap is used.
    2. If the Sender is a SP, the SP notifications bitmap is used.
3. For a global notification pended by a Sender, subsequent invocations of the FFA_NOTIFICATION_SET interface by the same Sender for the same notification have no effect until the notification is cleared.
4. For a per-vCPU notification pended by a Sender, subsequent invocations of the FFA_NOTIFICATION_SET interface by the same Sender for the same notification and Receiver vCPU have no effect until the notification is cleared for that Receiver vCPU.
5. A Receiver determines that it has a pending notification through one or more of the following mechanisms.
    1. The partition manager signals the virtual *Notification pending interrupt* to the Receiver.

       The interrupt is signaled when the target execution context of the Receiver next enters the *running* state.
        1. For a *per-vCPU* notification, the target execution context is specified by the Sender in the invocation of the FFA_NOTIFICATION_SET interface.
        2. For a *global* notification, the target execution context depends the type of interrupt.
            1. The interrupt is a PPI or SGI. It is signaled to an execution context determined by the partition manager of the Receiver through an IMPLEMENTATION DEFINED mechanism.
            2. The interrupt is a SPI. It is signaled to the execution context it is targeted to by the Receiver.
    2. The Receiver's scheduler uses the FFA_MSG_SEND_DIRECT_REQ interface to run and inform the Receiver through a partition message that it has a pending notification.
    3. The Receiver uses the FFA_NOTIFICATION_GET interface to poll if it has pending notifications.
6. A Receiver endpoint uses the FFA_NOTIFICATION_GET interface to retrieve its pending notifications (see 15.6 *FFA_NOTIFICATION_GET*).
7. A pending notification is cleared by a partition manager when it is retrieved by the Receiver endpoint as described below.
    1. The Hypervisor clears a pending notification in the VM and Hypervisor notifications bitmap of a VM.
    2. The SPMC clears a pending notification in the SP and SPMC notifications bitmap of a VM.
    3. The SPMC clears a pending notification in all notifications bitmap of a SP.

8. The *Schedule Receiver interrupt* (see 7.4.1 *Notification interrupt setup*) is used by the Partition manager of the Sender to inform the Receiver's scheduler that the Receiver has one or more notifications pending.

   This interrupt is used only if the Partition manager and the Receiver's scheduler reside in separate exception levels.

9. The Receiver's scheduler uses the FFA_NOTIFICATION_INFO_GET interface to retrieve the list of endpoints that have pending notifications and must be run (see 15.7 *FFA_NOTIFICATION_INFO_GET*).

10. A notification could be signaled by a Sender in the Secure world to a VM. The Hypervisor needs to determine which VM and vCPU (in case a per-vCPU notification is signaled) has a pending notification in this scenario. It obtains this information through an invocation of the FFA_NOTIFICATION_INFO_GET ABI at the Non-secure physical FF-A instance.

7.5.1 Example signaling flows

This section describes some example notification signaling flows between the Normal and Secure worlds. The following scenarios are considered.

1. SP0 sends a notification to SP1.
2. SP0 sends a notification to VM0.
3. SP0 sends a notification to its scheduler.

For the sake of simplicity, the following assumptions have been made.

1. Schedulers of all Receivers are implemented in the primary endpoint.
2. The primary endpoint is responsible for handling physical G1NS interrupts. The Hypervisor does not signal the virtual *Notification pending interrupt* to the primary endpoint.
3. There could be multiple PEs in the system. However, the scenarios encountered in notification signaling due to the presence of multi-processing are ignored.
4. SP0 is an MP-capable partition. Each execution context of SP0 is pinned to a physical PE on the system. Also see 4.4.1 *Discovery and setup*.
5. The endpoints bind the following notifications as described in 7.4.2 *Notification binding*.
   1. SP1 binds global notification 5 to SP0.
   2. VM0 binds global notification 0 to SP0.
   3. SP0's scheduler in the primary endpoint binds per-vCPU notification 1 to SP0.
6. Each endpoint uses an IMPLEMENTATION DEFINED mechanism to inform another endpoint about a notification it can signal.

   For example, a SP's scheduler could inform the SP about a notification that it can signal by sending it a direct message through the *FFA_MSG_SEND_DIRECT_REQ* ABI.
7. The *Schedule Receiver interrupt* is a physical PPI or a SGI that is signaled on the same PE on which the notification is signaled.
8. The discovery and setup associated with the *Schedule Receiver interrupt* and *Notification pending interrupt* is performed by the endpoints and their schedulers as described in 7.4.1 *Notification interrupt setup*.

7.5.1.1 SP0 signals a notification to SP1, VM0 and its scheduler

Figure 7.7 illustrates an example flow where SP0 sends notifications to SP1, VM0 and its scheduler in the primary endpoint while handling a Secure interrupt that preempted the Normal world. It is assumed that the execution context of SP0 and VM0 on the PE where the interrupt triggers is in a *waiting* state.

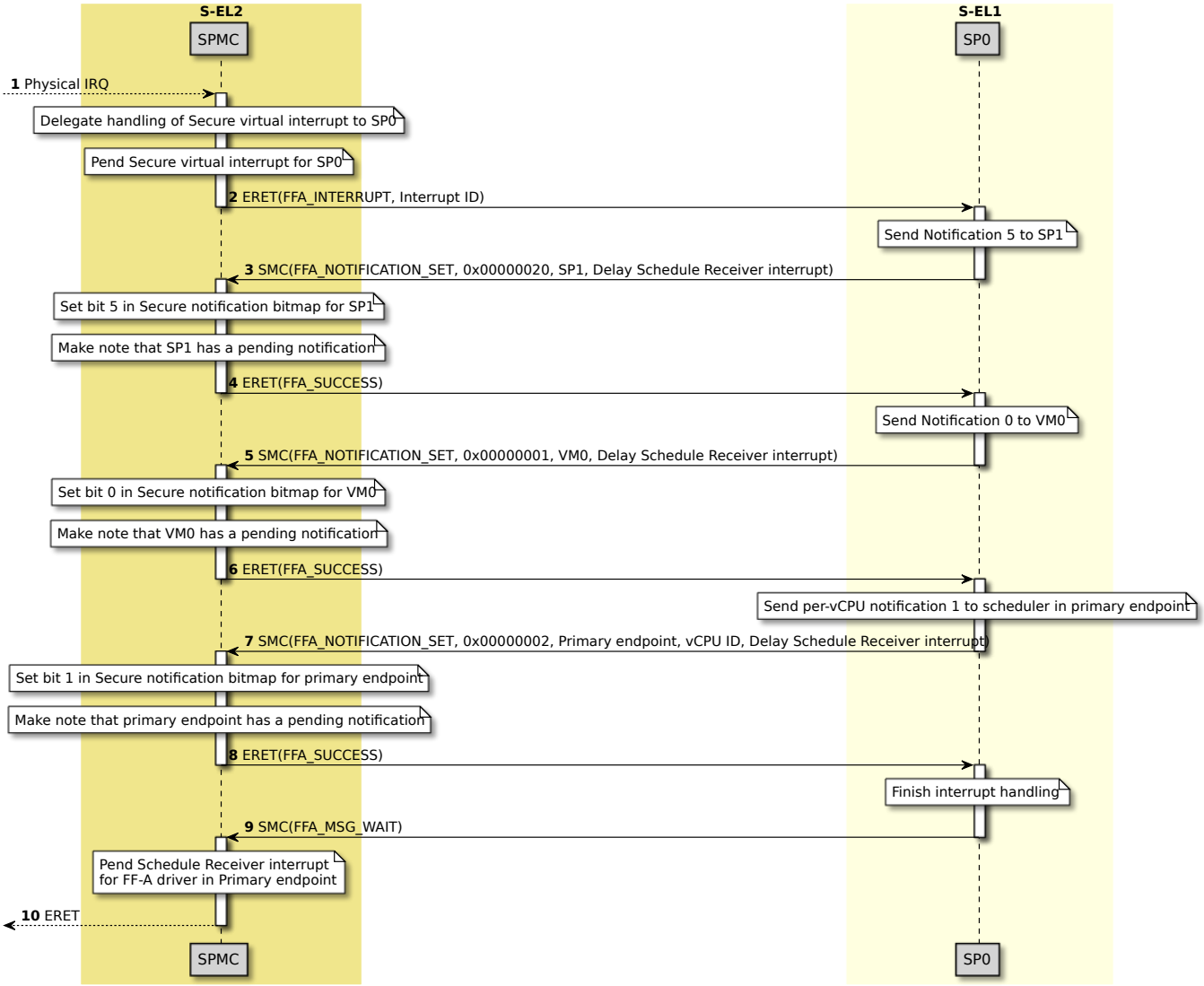

Figure 7.7: Signaling from SP0 to SP1, VM0 and its scheduler

7.5.1.2 Primary endpoint handles Schedule Receiver interrupt

Figure 7.8 illustrates an example flow where the FF-A driver in the primary endpoint handles the *Schedule Receiver interrupt*.

Figure 7.9 illustrates an example flow where the SP1 driver in the primary endpoint schedules an SP1 execution context in response to the *Schedule Receiver interrupt*.

Figure 7.10 illustrates an example flow where the VM0 driver in the primary endpoint schedules a VM0 execution context in response to the *Schedule Receiver interrupt*.

Figure 7.11 illustrates an example flow where the SP0 driver in the primary endpoint handles the notification pended by SP0.

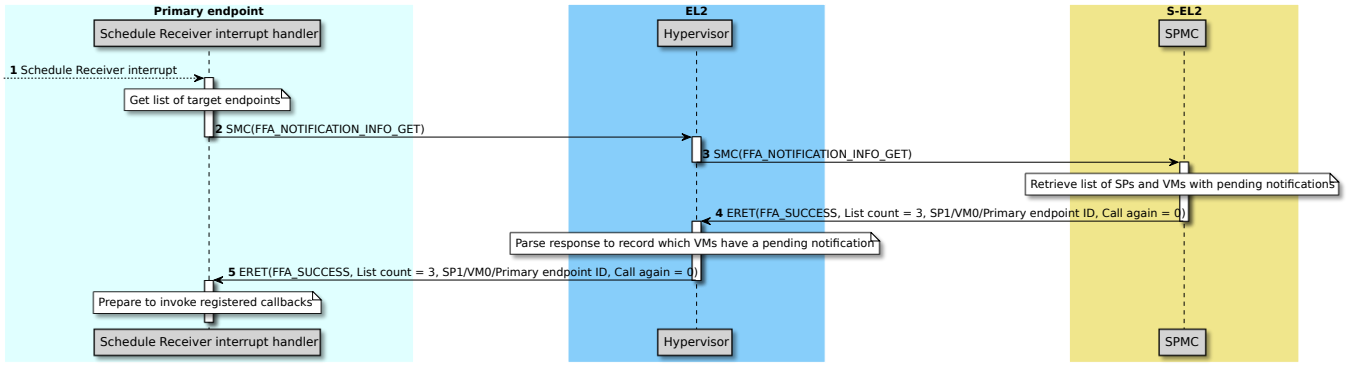

Figure 7.8: Schedule Receiver interrupt handling in primary endpoint

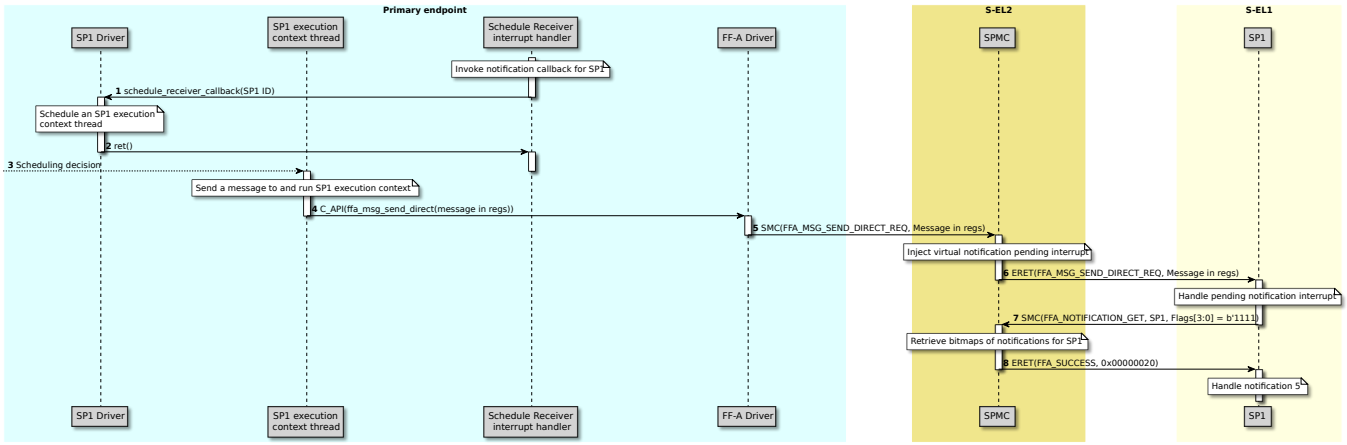

Figure 7.9: SP1 driver in primary endpoint schedules SP1

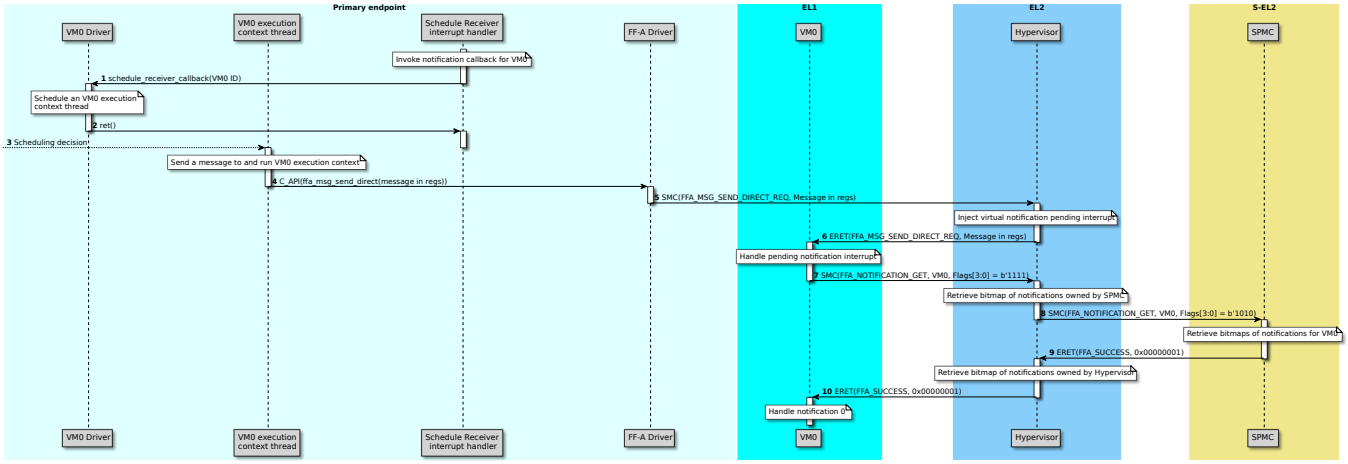

Figure 7.10: VM0 driver in primary endpoint schedules VM0

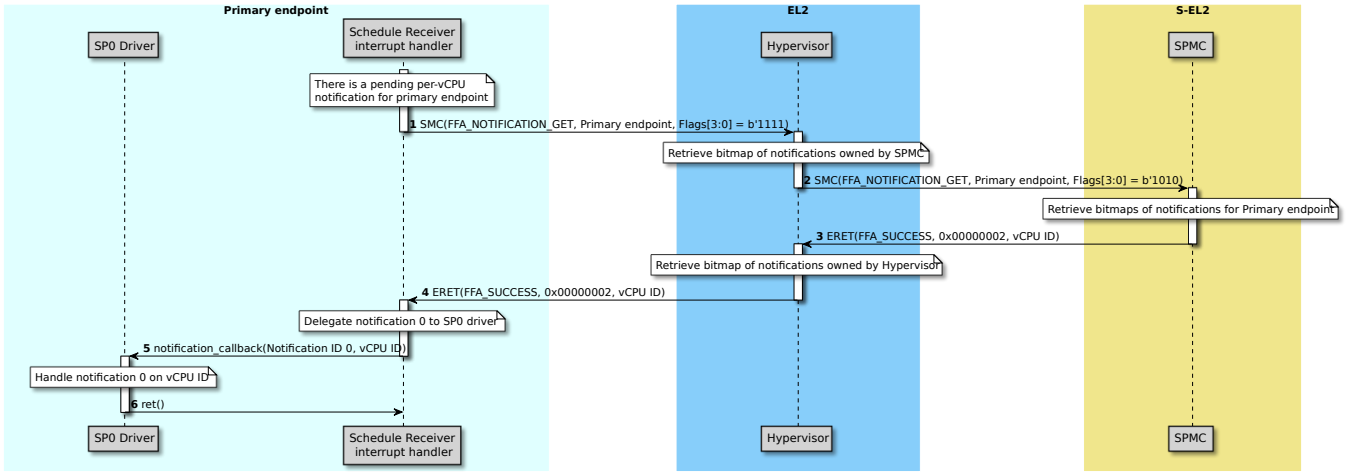

Figure 7.11: SP0 driver in primary endpoint receives a notification

7.6 Notification state machine

Figure 7.12 describes the state diagram of a notification.

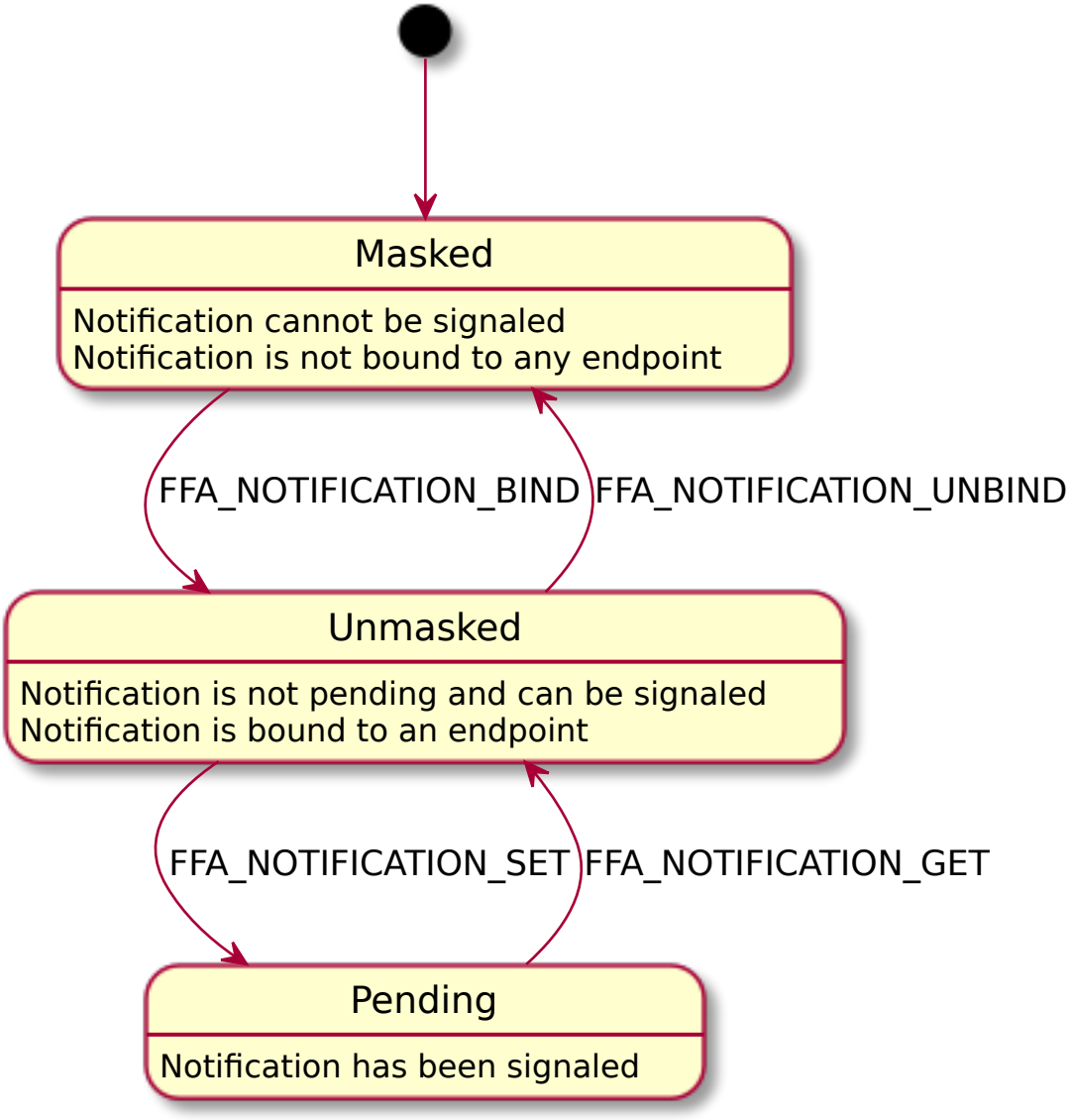

Figure 7.12: Notification state transition diagram

7.7 Feature discovery

The following rules govern discovery of support for notifications and its compliance.

1. Support for receipt of notifications is optional. If an endpoint implements this support, it specifies this in its manifest (see Chapter 3 *Setup*).

   In the rest of this document, unless explicitly stated otherwise, it is assumed that VMs and SPs support receipt of notifications.

2. A partition manager can choose to not implement support for notifications. It does not initialize an endpoint if this support is requested through the endpoint manifest.

3. An FF-A component in the Normal world uses the FFA_PARTITION_INFO_GET interface to determine if another endpoint supports receipt of notifications (see 11.7 *FFA_PARTITION_INFO_GET*).

4. An invocation of the FFA_FEATURES interface with Feature IDs *0x1* and *0x2* or any notification ABI, completes with an invocation of the *FFA_ERROR* interface with the *NOT_SUPPORTED* error code, if the callee does not support notifications.

5. An invocation of the FFA_FEATURES interface with Feature ID *0x1* or any Notification ABI, apart from FFA_NOTIFICATION_INFO_GET and FFA_NOTIFICATION_SET, by an endpoint, completes with an invocation of the *FFA_ERROR* interface with the *NOT_SUPPORTED* error code, if the endpoint does not support receipt of notifications.

6. An invocation of any notification ABI by an endpoint that does not support receipt of notifications completes with an invocation of the *FFA_ERROR* interface with the *NOT_SUPPORTED* error code.

7.8 Framework Notifications

Framework notifications are doorbells that are rung by the partition managers to signal common events to an endpoint. These doorbells cannot be rung by an endpoint directly. A partition manager can signal a Framework notification in response to an FF-A ABI invocation by an endpoint.

In this version of the Framework, the following doorbells are supported.

1. RX buffer full notification. See 7.8.1 *RX buffer full notification*.

7.8.1 RX buffer full notification

This notification is signaled by a partition manager during transmission of a partition message through indirect messaging to,

1. Inform the message Receiver's scheduler that the Receiver must be run.
2. Inform an endpoint that it has a pending message in its RX buffer.

Also see 4.3 *Indirect messaging usage*.

The following rules govern usage of this notification.

1. This notification is signaled by setting *Bit[0]* in the framework notifications bitmap of an endpoint.
    1. This notification is reserved in both the SPMC and Hypervisor framework notifications bitmaps of every endpoint.
    2. This notification is signaled to only those endpoints that can receive messages through indirect messaging.
2. In response to an FFA_MSG_SEND2 invocation by a Sender endpoint, the Framework performs the following actions after the message is copied from the TX buffer of the Sender to the RX buffer of the Receiver.
    1. The notification is pended in the framework notification bitmap of the Receiver.
        1. If the Sender is a SP, the notification is pended in the SPMC framework notifications bitmap of the Receiver.
        2. If the Sender is a VM, the notification is pended in the Hypervisor framework notifications bitmap of the Receiver.
        3. If the Receiver is a SP, the notification is pended by the SPMC irrespective of whether the Sender is a VM or a SP.
        4. If the Receiver is a VM, the notification is pended by the SPMC is the Sender is a SP. It is pended by the Hypervisor is the Sender is a VM.
    2. The partition manager of the endpoint that contains Receiver's scheduler pends the *Schedule Receiver* interrupt for this endpoint.

       The Receiver receives the notification as described in 7.5 *Notification signaling* and copies out the message from its RX buffer.

Chapter 8
Memory Management

8.1 Overview

The Firmware Framework describes mechanisms and interfaces that enable FF-A components to manage access and ownership of memory regions in the physical address space to fulfill use cases such as:

- DRM protected video path.
- Communication with a VM with pre-configured machine learning frameworks,
- Biometric authentication and Secure payments.

FF-A components can use a combination of Framework and Partition messages to manage memory regions in the following ways.

1. The Owner of a memory region can transfer its ownership to another FF-A component.
2. The Owner of a memory region can relinquish access to it and grant access to one or more FF-A components.
3. The Owner of a memory region can share access to it with one or more FF-A components.
4. The Owner of a memory region can reclaim access to it by requesting FF-A components to relinquish access to the memory region.

8.2 Direct memory access

The Framework enables FF-A components to manage access to the physical address space from a device that is upstream of an SMMU using the memory management transactions described in 8.5 *Memory management transactions*.

As per the Arm® SMMU architecture, each transaction generated by a device is associated with a *Stream ID*. This Stream ID could be one of many that the device is configured to use. A Stream ID is used to determine the stage 1 and stage 2 address translations that must be used for the transaction. It is also possible that one or both stages of translation could be bypassed for a Stream ID in the SMMU.

If enabled, the stage 2 translations corresponding to a Stream ID control access to the physical address space that the device has. A set of stage 2 translation tables could map to one or more Stream IDs. The Framework manages stage 2 translations in the SMMU as described in 8.3 *Address translation regimes*.

The Hypervisor programs the SMMU to create and manage the association between a Non-secure Stream ID and the stage 2 translations its transactions must use.

The SPM programs the SMMU to create and manage the association between a Secure Stream ID and the stage 2 translations its transactions must use.

The Framework does not manage the stage 1 translations and their association with Stream IDs in the SMMU on behalf of the device. This should be done by an endpoint through an IMPLEMENTATION DEFINED mechanism.

8.2.1 Stream endpoint

A set of SMMU stage 2 translations maintained by a partition manager is called a *Stream endpoint*. Each Stream endpoint is assigned a 16-bit ID called the *Stream endpoint ID* or *SEPID*.

Stream endpoints associated with a Secure Stream ID are called *Secure SEPIDs*

Stream endpoints associated with a Non-secure Stream ID are called *Non-secure SEPIDs*

Endpoints that run on a PE are referred to as *PE endpoints* to differentiate them from Stream endpoints. The term *endpoint* is used when it is not required to distinguish between these types of endpoints.

There is a 1:N (N >= 1) mapping between a SEPID and Stream IDs assigned to different devices that is, the stage 2 translations corresponding to the SEPID could be *shared* by one or more Stream IDs.

SEPIDs are used in memory management transactions to:

- Grant and revoke access to a physical memory region to a device.
- Transfer ownership of a physical memory region from or to a device.

SEPID values must be distinct from those assigned to PE endpoints. A SEPID is discoverable through the *FFA_PARTITION_INFO_GET* interface (also see 11.7 *FFA_PARTITION_INFO_GET*).

This version of the Framework considers two types of devices.

1. Devices that can act as initiators and recipients of memory management transactions. These devices are called *independent* peripheral devices. Each device must specify the following information in its partition manifest (see 3.2.3 *Independent peripheral device manifest*).

   - A SEPID assigned to the device at boot time.
   - The SMMU ID that the device is upstream of.
   - Each Stream ID the device can generate.
   - Regions in the physical address space that must be mapped in the translation tables corresponding to the SEPID at boot time.

   This information enables the partition manager to create an association between a device and a SEPID at boot time.

A partition manager or PE endpoint and an *independent* device must use an IMPLEMENTATION DEFINED mechanism to notify each other about a memory management transaction targeted to a SEPID used by the device (see 8.5.2 *Transaction life cycle*).

2. Devices that cannot act as initiators and recipients of memory management transactions. These devices are called *dependent* peripheral devices. They rely on a PE endpoint to initiate and receive memory management transactions on their behalf. The PE endpoint is called a *proxy endpoint*.

   A dependent device could be *assigned* to a PE endpoint. This implies,

   - Access to its MMIO regions is assigned to the endpoint during boot (see 2.10 *System resource management* & Table 3.3).
   - The endpoint manages the association between Stream IDs generated by the device and stage 1 translations in the SMMU that the device is upstream of (see Table 3.3).

   The device could be either assigned to its *proxy endpoint* or a different PE endpoint.

   When assigned to its proxy endpoint, this version of the Framework assumes that all the Stream IDs generated by the device have the same visibility of the physical address space as the endpoint. The stage 2 translations in the SMMU for these Stream IDs are the same as those maintained by the partition manager on behalf of the endpoint. They are not assigned a SEPID. The partition ID of the *proxy endpoint* is used instead. All memory management transactions with this partition ID effect both sets of translations.

   When assigned to a different endpoint, the partition manifest of the *proxy endpoint* (see 3.2.1 *Manifest for isolated partitions*) must specify the following information to enable the partition manager to create an association between a device and a SEPID at boot time.

   - The SMMU ID that the device is upstream of.
   - Each Stream ID the device can generate.
   - The SEPID corresponding to each Stream ID.

   The partition ID of the proxy endpoint must be distinct from the SEPID allocated to manage the preceding association. The SEPID must be specified in the partition manifest of the proxy endpoint (see Table 3.1).

   The stage 2 translations corresponding to the SEPID are configured at boot time with no access to the physical address space.

   A memory management transaction targeted to the SEPID must be allowed to complete only if it is either initiated or authorized by the *proxy endpoint* for the device (see 8.5.2 *Transaction life cycle*).

The SEPIDs used by an *independent* device must be distinct from the SEPIDs used by a *dependent* device. This constraint avoids the scenario where a memory management transaction is allowed to change the stage 2 translations before the *proxy endpoint* has authorized it.

8.3 Address translation regimes

Memory management relies on the two fundamental operations of mapping and un-mapping a memory region from the stage of a translation regime managed by a partition manager on behalf of a partition. The translation regime and the stage depend on the type of partition as follows.

1. The Hypervisor creates and manages stage 2 translations on behalf of a EL1 PE endpoint, in the Non-secure EL1&0 translation regime, when EL2 is enabled.
2. The Hypervisor creates and manages stage 2 translations for a Non-secure Stream ID assigned to an independent or dependent peripheral device, in the Non-secure EL1&0 translation regime in the SMMU. A SEPID is used to identify the stage 2 translation tables (see 8.2.1 *Stream endpoint*).
3. The SPMC creates and manages stage 2 translations on behalf of a S-EL1 PE endpoint in the Secure EL1&0 translation regime, when S-EL2 is enabled.
4. The SPMC creates and manages stage 1 translations on behalf of a S-EL0 PE endpoint in the Secure EL2&0 translation regime, when S-EL2 is enabled.
5. The SPMC creates and manages stage 1 translations on behalf of a S-EL0 PE endpoint in the Secure EL1&0 translation regime, when S-EL2 is disabled.
6. The SPMC creates and manages stage 2 translations for a Secure Stream ID assigned to an independent or dependent peripheral device in the Secure EL1&0 translation regime in the SMMU. A SEPID is used to identify the stage 2 translation tables (see 8.2.1 *Stream endpoint*).

8.4 Ownership and access attributes

The Hypervisor, SPM, and all endpoints have *access* and *ownership* attributes associated with every memory region in the physical address space.

Access determines the data and instruction access permissions to the memory region. A component can have the following access permissions to a memory region.

- No access.
- Read-only, Execute-never.
- Read-only, Executable.
- Read/write, Execute-never.

Access control must be enforced through an IMPLEMENTATION DEFINED mechanism and/or by encoding these permissions in the translation regime of an endpoint managed by the partition manager (see 8.3 *Address translation regimes*).

Ownership is a software attribute that determines if a component can grant access to a memory region to another component. A component that has access to a memory region without ownership is called the *Borrower*. A component that lends access to a memory region it owns is called the *Lender*.

Ownership of a memory region is initially assigned to the component that it is allocated to. At boot time all memory regions are owned by Secure firmware. A memory region could be configured as Secure or normal memory either statically at reset, or by Secure firmware during boot. Secure firmware transfers ownership of normal memory to Normal world software. It sub-divides Secure memory such that:

- It owns and has exclusive access to some memory regions.
- It owns but grants access to some memory regions to SPs.
- It transfers ownership of some memory regions to SPs.

If virtualization is enabled in the Normal world, the Hypervisor divides a subset of normal memory among VMs and transfers ownership to them. In the absence of virtualization, all normal memory donated by the Secure world is owned by the OS kernel.

An endpoint requests access to and/or ownership of a memory region through its partition manifest (also see Table 3.2).

8.4.1 Ownership and access rules

The SPM and Hypervisor must enforce the following general ownership and access rules to memory regions.

1. The size of a memory region to which ownership and access rules apply must be a multiple of the smallest translation granule size supported on the system.
   - It is 4K in the AArch32 Execution state.
   - A EL1 or S-EL1 partition must discover this by reading the ID_AA64MMFR0_EL1 System register in the AArch64 Execution state.
   - A S-EL0 SP must determine this through an IMPLEMENTATION DEFINED discovery mechanism for example, DT or ACPI tables.
2. A normal memory region must be mapped with the Non-secure security attribute in any component that is granted access to it.
3. A Secure memory region must be mapped with the Secure security attribute in any component that is granted access to it.
4. Each memory region in the physical address space must have a single Owner.
5. A FF-A component must have access to a memory region it owns unless it has granted exclusive access to the region to another FF-A component.

6. Only the Owner of a memory region can grant access to it to one or more Borrowers in the system.
7. Only the Owner of a memory region can transfer its ownership to another endpoint in the system.
8. If an SP is terminated because of a fatal error condition, access to the memory regions of the SP is transferred to the SPM.
9. If a VM is terminated because of a fatal error condition, access to the memory regions of the VM and their ownership are transferred to the Hypervisor.
10. If the Hypervisor or OS kernel are terminated because of a fatal error condition, access to their memory regions and ownership are transferred to the SPM.
11. The number of distinct components to whom an Owner can grant access to a memory region is IMPLEMENTATION DEFINED.
12. The Owner of a memory region must not be able to change its ownership or access attributes until all Borrowers have relinquished access to it.

8.4.2 Ownership and access states

Table 8.1 describes the ownership states applicable to an FF-A component for a memory region.

Table 8.1: Ownership states

No.	Ownership state	Acronym	Description
1	Owner	Owner	Component owns the memory region.
2	Not Owner	!Owner	Component does not own the memory region.

Table 8.2 describes the access states applicable to an FF-A component for a memory region.

Table 8.2: Access states

No.	Access state	Acronym	Description
1	No access	NA	A component has *no* access to a memory region. It is not mapped in its translation regime.
2	Exclusive access	EA	A component has exclusive access to a memory region. It is mapped only in its translation regime.
3	Shared access	SA	A component has shared access to a memory. It is mapped in its translation regime and the translation regime of at least one other component

Table 8.3 describes the valid combination of access and ownership states applicable to an FF-A component for a memory region.

Table 8.3: Valid combinations of ownership and access states

No.	Ownership state	Access state	Acronym	Description
1	Not Owner	No access	!Owner-NA	Component has neither ownership nor access to the memory region.
2	Not Owner	Exclusive access	!Owner-EA	Component has exclusive access without ownership of the memory region.
3	Not Owner	Shared access	!Owner-SA	Component has shared access with one or more components without ownership of the memory region.
4	Owner	No access	Owner-NA	Component owns the memory region and has: • Either granted exclusive access to the memory region to another component. • Or shared access to the memory region among other components.
5	Owner	Exclusive access	Owner-EA	Component owns the memory region and has exclusive access to it.
6	Owner	Shared access	Owner-SA	Component owns the memory region and shares access to it with one or more components.

For two FF-A components *A* and *B* and a memory region, valid combinations of states defined in Table 8.3 are described in Table 8.4. Other combinations of states are considered invalid.

Table 8.4: Valid combinations of ownership and access states between two components

No.	Component A state	Component B state	Description
1	Owner-EA	!Owner-NA	Component A has exclusive access and ownership of a memory region that is inaccessible from component B.
2	Owner-NA	!Owner-NA	Component A has granted exclusive access to a memory region it owns to another component. It is inaccessible from component B.
3	Owner-NA	!Owner-EA	Component A has granted exclusive access to a memory region it owns to component B.
4	Owner-NA	!Owner-SA	Component A has relinquished access to a memory region it owns. Access to the memory region is shared between component B and at least one other component
5	Owner-SA	!Owner-NA	Component A shares access to a region of memory it owns with another component. Component B cannot access the memory region.

No.	Component A state	Component B state	Description
6	Owner-SA	!Owner-SA	Component A shares access to a region of memory it owns with component B and possibly other components.

U **Implementation Note**

To fulfill the use cases and enforce the rules listed earlier, FF-A components should track the state of a memory region. This could be done as follows,

- An Owner tracks the level of access it has to a memory region.
- An Owner tracks the level of access that Borrowers have to a memory region along with the identity of the Borrowers.
- A Borrower tracks the level of access the Owner has to a memory region along with the identity of the Owner.
- A Borrower tracks the level of access it has to a memory region.
- A Borrower tracks the level of access that other Borrowers have to a memory region along with the identity of the Borrowers.
- For each memory region, the SPM and Hypervisor track the following.
  - The identity of each Borrower.
  - The identity of the Owner.
  - The level of access of each Borrower.
  - The level of access of the Owner.

8.5 Memory management transactions

This version of the Framework describes transactions that enable endpoints to manage access and ownership of physical memory regions.

- Transitions between states described in 8.4.2 *Ownership and access states* happen in response to transactions described in Table 8.5. Each transaction involves exchange of one or more Framework and partition messages.
- Each transition is described as a transaction involving two endpoints (*A* and *B*) and a memory region. Endpoint A is the Owner of the memory region.

Table 8.5: Memory region transactions

No.	Transaction	Description
1.	Donate	• Endpoint A transfers ownership of a memory region it owns to endpoint B. See 8.6 *Donate memory transaction*.
2.	Lend	• Endpoint A relinquishes access to a memory region and grants it to only endpoint B. Endpoint B gains exclusive access to the memory region. • Endpoint A relinquishes access to a memory region and grants it to endpoint B and at least one other endpoint simultaneously. Endpoint B gains shared access to the memory region. • See 8.7 *Lend memory transaction*.
3.	Share	• Endpoint A grants access to a memory region to endpoint B and optionally to other endpoints simultaneously. See 8.8 *Share memory transaction*.
4.	Relinquish	• Endpoint B relinquishes access to a memory region granted to it by Endpoint A. Endpoint A reclaims exclusive access to the memory region. See 8.9 *Relinquish memory transaction*.

8.5.1 Component roles

In this version of the Framework, endpoints can fulfill the role of an Owner, Lender or Borrower (see 8.4 *Ownership and access attributes*).

The Hypervisor and SPM participate in memory management transactions to validate and transmit them from a *Sender* endpoint to a *Receiver* endpoint. They are also responsible for managing the translation regime of an endpoint and tracking the ownership and access attributes of a memory region. This collective role is termed as a *Relayer*.

Table 8.6 specifies the roles each FF-A component can play in a memory management transaction.

In the absence of the Hypervisor, the OS Kernel subsumes the role of the Relayer. Its roles as the Relayer, Owner, Lender and Borrower are considered to be logically separate from each other. The interface used by internal components that implement these roles to exchange memory management transactions is IMPLEMENTATION DEFINED.

The roles of the SPMD and SPMC components of the SPM (see 2.2 *SPM architecture*) as Relayers are as follows.

- In SPM configurations where the SPMD and SPMC reside in separate Exception levels (see Table 2.1 & Table 2.2)):

  - The SPMD component must forward memory management transactions between the Secure and Non-secure physical FF-A instances.
  - The SPMC component must handle outbound and inbound transactions on behalf of the Sender and Receiver.
- In the SPM configuration where the SPMC coexists with an SP at S-EL1 or Secure Supervisor mode (see Table 2.3), the roles of the SPMC as the Relayer and the SP as the Owner, Borrower or Lender are considered to be logically separate. The interface used by internal components that implement these roles to exchange memory management transactions is IMPLEMENTATION DEFINED. The SP and SPM must still appear as separate FF-A components to software in the Normal world and SPs at the Secure virtual FF-A instance. Also see 8.5.2 *Transaction life cycle*.

Table 8.6: FF-A component roles in a memory management transaction

Config No.	FF-A component	Owner	Lender	Borrower	Relayer
1.	NS-Endpoint	Yes	Yes	Yes	No
2.	S-Endpoint	Yes	Yes	Yes	No
3.	SEPID	Yes	Yes	Yes	No
4.	Hypervisor	No	No	No	Yes
5.	SPM	No	No	No	Yes

In all transactions, an endpoint must be a Sender or Receiver. This depends on the type of transaction as follows.

- In a transaction to donate ownership of a memory region, the Sender is the current Owner, and the Receiver is the new Owner.
- In a transaction to lend or share access to a memory region, the Sender is the Lender, and the Receiver is the Borrower.
- In a transaction to relinquish access to a memory region, the Sender is the Borrower, and the Receiver is the Lender.

Valid combinations of component roles in a transaction to donate, lend or share memory are listed in Table 8.7. A FF-A component can use one or more combinations in a memory management transaction as the Sender.

Valid combinations of component roles in a transaction to relinquish memory are listed in Table 8.8.

Table 8.7: Valid role combinations in donate, lend or share memory transactions

Config No.	Sender	Receiver	Relayer
1.	VM	VM	Hypervisor
2.	VM	NS SEPID	Hypervisor

Config No.	Sender	Receiver	Relayer
3.	NS-Endpoint	Secure SEPID	Hypervisor (if present) and SPM
4.	NS-Endpoint	SP	Hypervisor (if present) and SPM
5.	SP	Secure SEPID	SPM
6.	SP	SP	SPM

Table 8.8: Valid role combinations in relinquish memory transactions

Config No.	Sender	Receiver	Relayer
1.	VM	VM	Hypervisor
2.	NS-SEPID	VM	Hypervisor
3.	Secure SEPID	NS-Endpoint	Hypervisor (if present) and SPM
4.	SP	NS-Endpoint	Hypervisor (if present) and SPM
5.	Secure SEPID	SP	SPM
6.	SP	SP	SPM

8.5.2 Transaction life cycle

Each transaction described in Table 8.5 takes place in three steps as follows and illustrated in Figure 8.1.

1. The Sender sends a Framework message to the Relayer to start a transaction involving one or more Receivers.
2. The Sender sends a Partition message requesting each Receiver to complete the transaction.
3. Each Receiver sends a Framework message to the Relayer to complete the transaction.

A transaction could be targeted to a *dependent* peripheral device identified by a SEPID (see 8.2.1 *Stream endpoint*). In this case, the partition message in *step 2* is sent to the *proxy* endpoint of the device. The proxy endpoint sends a Framework message in *step 3* to validate, authorize and complete the transaction of behalf of the device.

A transaction could be targeted to an *independent* peripheral device identified by a SEPID (see 8.2.1 *Stream endpoint*). In this case, an IMPLEMENTATION DEFINED message is sent to this device in *step 2*. The device uses an IMPLEMENTATION DEFINED mechanism to communicate with the Relayer to complete the transaction in *step 3*.

In the SPM configuration where the SPMC coexists with an SP at S-EL1 or Secure Supervisor mode (see Table 2.3), the Relayer, Sender and Receiver components are implemented in the same Exception level and software image. The transaction life-cycle for this configuration is as follows.

- When the SP is the Sender, it must use an IMPLEMENTATION DEFINED interface to deliver the Framework message to the SPMC in step 1.
- When the SP is the Receiver, the Framework message sent in step 1 is received by the SPMC. It must be delivered to the SP at the Secure physical FF-A instance through an IMPLEMENTATION DEFINED interface between them.
  - A successful completion of the interface used in step 1 by the SPMC must indicate completion of the entire transaction to the Sender.
  - Steps 2 & 3 are not required as the SP is made aware of the ongoing transaction in Step 1.
  - The following aspects of the memory management transaction in this scenario are IMPLEMENTATION DEFINED.
    * How the Sender discovers the presence of this SPM configuration.
    * How the SPMC delivers the Framework message to the Receiver.
    * How the Receiver interacts with the SPMC to complete the transaction.

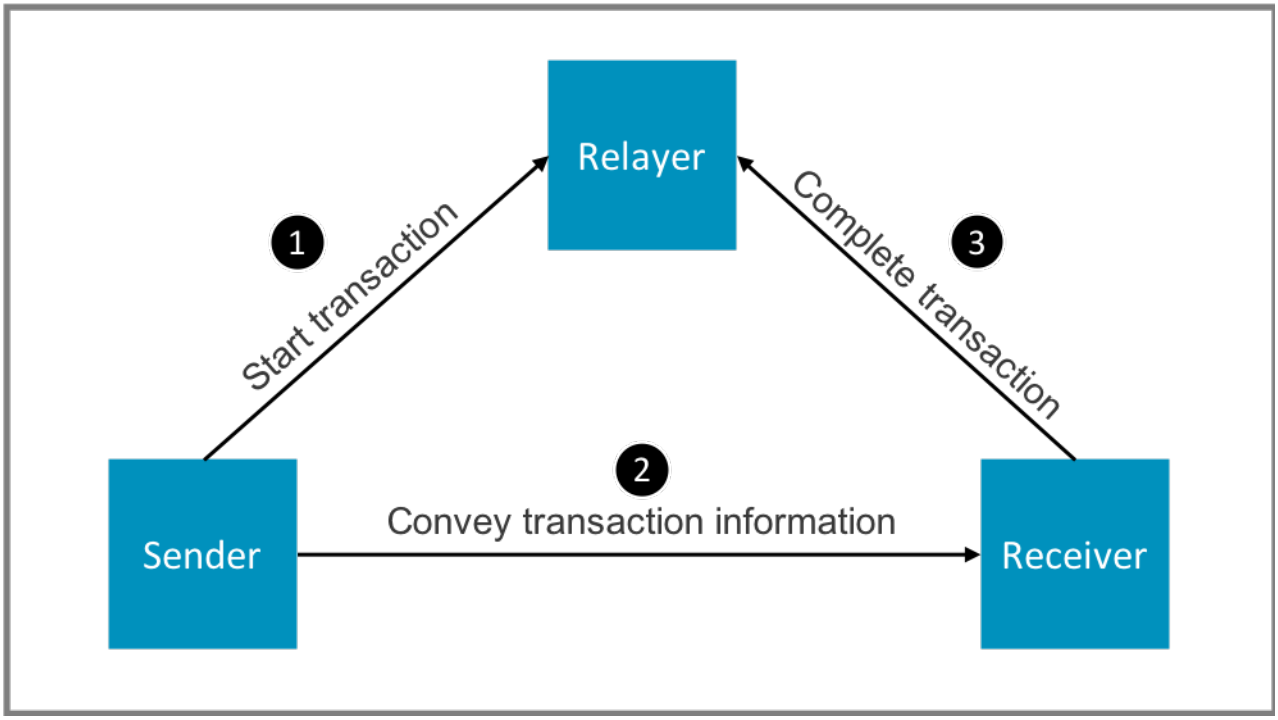

Figure 8.1: Memory management transaction lifecycle

8.6 Donate memory transaction

This transaction is used to transfer the ownership of a memory region from the endpoint that owns it to another endpoint. A list of valid combinations of roles played by various FF-A components in this transaction is specified in Table 8.7.

8.6.1 Donate memory state machine

Table 8.9 describes the state machine for donating a memory region from the perspective of two endpoints *A* & *B*. *A* owns the memory region. It attempts to donate the memory region to *B*. Valid and invalid state transitions in response to this transaction have been listed.

In each valid transition,

- *A* loses both ownership and access to the memory region and enters the *!Owner-NA* state.
- *B* gains ownership and exclusive access to the memory region and enters the *Owner-EA* state.

Table 8.9: Donate memory transaction state machine

No.	Current Endpoint A state	Current Endpoint B state	Next Endpoint A state	Next Endpoint B state	Description
1	Owner-EA	!Owner-NA	!Owner-NA	Owner-EA	• Owner has exclusive access to the memory region and transfers ownership to endpoint B.
2	Owner-NA	!Owner-NA	Error	–	• Owner does not have exclusive access to the memory region. It cannot transfer its ownership.
3	Owner-NA	!Owner-SA	Error	–	• Owner has lent access to the memory region to endpoint B and possibly other endpoints. It cannot transfer its ownership.
4	Owner-SA	!Owner-NA	Error	–	• Owner has shared access to the memory region with one or more endpoints. It cannot transfer its ownership.
5	Owner-SA	!Owner-SA	Error	–	• Owner has shared access to the memory region with endpoint B and possibly other endpoints. It cannot transfer its ownership.

8.6.2 Donate memory transaction lifecycle

This transaction takes place as follows (also see 8.5.2 *Transaction life cycle*).

1. The Owner uses the *FFA_MEM_DONATE* interface to describe the memory region and convey the identity of the Receiver to the Relayer as specified in Table 8.19. This interface is described in 14.1 *FFA_MEM_DONATE*.

2. If the Receiver is a *PE endpoint* or a *SEPID* associated with a dependent peripheral device,
    1. The Owner uses a Partition message to request the Receiver to retrieve the donated memory region. This message contains a description of the memory region relevant to the Receiver.
    2. The Receiver uses the *FFA_MEM_RETRIEVE_REQ* and *FFA_MEM_RETRIEVE_RESP* interfaces to map the memory region in its translation regime and complete the transaction. These interfaces are described in 14.4 *FFA_MEM_RETRIEVE_REQ* & 14.5 *FFA_MEM_RETRIEVE_RESP* respectively.

       In case of an error, the Sender can abort the transaction before the Receiver retrieves the memory region by calling the *FFA_MEM_RECLAIM* ABI (see 14.7 *FFA_MEM_RECLAIM*).
3. If the Receiver is a *SEPID* associated with an independent peripheral device, an IMPLEMENTATION DEFINED mechanism is used by the Sender and Relayer to map and describe the memory region to the Receiver (see 14.1.1 *Component responsibilities for FFA_MEM_DONATE*).

8.7 Lend memory transaction

This transaction is used by an Owner to relinquish its access to a memory region and grant access to it to one or more Borrowers.

- If the region is lent to a single Borrower, it is granted exclusive access to it.
- If the region is lent to more than one Borrower, they are granted shared access to it.

A list of valid combinations of roles played by various FF-A components in this transaction is specified in Table 8.7.

8.7.1 Lend memory transaction state machine

Table 8.10 describes the state machine for lending a memory region from the perspective of two components *A* & *B*. *A* owns the memory region. It attempts to relinquish its access to the memory region and grant shared or exclusive access to it to *B*. Valid and invalid state transitions in response to this transaction have been listed.

Table 8.10: Lend memory transaction state machine

No.	Current Endpoint A state	Current Endpoint B state	Next Endpoint A state	Next Endpoint B state	Description
1	Owner-EA	!Owner-NA	Owner-NA	!Owner-EA or !Owner-SA	• Owner has exclusive access to the memory region and relinquishes access to it to one or more Borrowers including endpoint B.
2	Owner-NA	!Owner-NA	Error	–	• Owner has already lent the memory region to one or more endpoints. It cannot lend it to endpoint B.
3	Owner-NA	!Owner-EA	Error	–	• Owner has already lent the memory region to endpoint B with exclusive access.
4	Owner-NA	!Owner-SA	Error	–	• Owner has already lent the memory region to endpoint B and other endpoints.
5	Owner-SA	!Owner-NA	Error	–	• Owner has already shared the memory region with one or more endpoints. It cannot lend it to endpoint B.
6	Owner-SA	!Owner-SA	Error	–	• Owner has already shared the memory region with endpoint B and possibly other endpoints.

8.7.2 Lend memory transaction lifecycle

This transaction takes place as follows (also see 8.5.2 *Transaction life cycle*).

1. The Lender uses the *FFA_MEM_LEND* interface to describe the memory region and convey the identities of the Borrowers to the Relayer as specified in Table 8.19. This interface is described in 14.2 *FFA_MEM_LEND*.
2. If a Borrower is a *PE endpoint* or a *SEPID* associated with a dependent peripheral device,
    1. The Lender uses a Partition message to request each Borrower to retrieve the lent memory region. This message contains a description of the memory region relevant to the Borrower.
    2. Each Borrower uses the *FFA_MEM_RETRIEVE_REQ* and *FFA_MEM_RETRIEVE_RESP* interfaces to map the memory region in its translation regime and complete the transaction. These interfaces are described in 14.4 *FFA_MEM_RETRIEVE_REQ* & 14.5 *FFA_MEM_RETRIEVE_RESP* respectively.
3. If the Borrower is a *SEPID* associated with an independent peripheral device, an IMPLEMENTATION DEFINED mechanism is used by the Lender and Relayer to map and describe the memory region to the Borrower (see 14.2.1 *Component responsibilities for FFA_MEM_LEND*).
4. In case of an error, the Lender can abort the transaction before the Borrower retrieves the memory region by calling the *FFA_MEM_RECLAIM* ABI (see 14.7 *FFA_MEM_RECLAIM*).

8.8 Share memory transaction

This transaction is used by an Owner of a memory region to share access to it with one or more Borrowers.

A list of valid combinations of roles played by various FF-A components in this transaction is specified in Table 8.7.

8.8.1 Share memory transaction state machine

Table 8.11 describes the state machine for sharing a memory region from the perspective of two components *A* & *B*. *A* owns the memory region. It attempts to share the memory region with *B*. Valid and invalid state transitions in response to this transaction have been listed.

Table 8.11: Share memory transaction state machine

No.	Current Endpoint A state	Current Endpoint B state	Next Endpoint A state	Next Endpoint B state	Description
1	Owner-EA	!Owner-NA	Owner-SA	!Owner-SA	• Owner has exclusive access to the memory region and grants access to it to one or more Borrowers including endpoint B.
2	Owner-NA	!Owner-NA	Error	–	• Owner has already lent the memory region to one or more endpoints. It cannot share it with endpoint B.
3	Owner-NA	!Owner-EA	Error	–	• Owner has already lent the memory region to endpoint B with exclusive access.
4	Owner-NA	!Owner-SA	Error	–	• Owner has already lent the memory region to endpoint B and other endpoints.
5	Owner-SA	!Owner-NA	Error	–	• Owner has already shared the memory region with one or more endpoints. It cannot share it with endpoint B.
6	Owner-SA	!Owner-SA	Error	–	• Owner has already shared the memory region with endpoint B and possibly other endpoints.

8.8.2 Share memory transaction lifecycle

This transaction takes place as follows (also see 8.5.2 *Transaction life cycle*).

1. The Lender uses the *FFA_MEM_SHARE* interface to describe the memory region and convey the identities of the Borrowers to the Relayer as specified in Table 8.19. This interface is described in 14.2 *FFA_MEM_LEND*.
2. If a Borrower is a *PE endpoint* or a *SEPID* associated with a dependent peripheral device,

1. The Lender uses a Partition message to request each Borrower to retrieve the shared memory region. This message contains a description of the memory region relevant to the Borrower.
2. Each Borrower uses the *FFA_MEM_RETRIEVE_REQ* and *FFA_MEM_RETRIEVE_RESP* interfaces to map the memory region in its translation regime and complete the transaction. These interfaces are described in 14.4 *FFA_MEM_RETRIEVE_REQ* & 14.5 *FFA_MEM_RETRIEVE_RESP* respectively.

3. If the Borrower is a *SEPID* associated with an independent peripheral device, an IMPLEMENTATION DEFINED mechanism is used by the Lender and Relayer to map and describe the memory region to the Borrower (see 14.3.1 *Component responsibilities for FFA_MEM_SHARE*).
4. In case of an error, the Lender can abort the transaction before the Borrower retrieves the memory region by calling the *FFA_MEM_RECLAIM* ABI (see 14.7 *FFA_MEM_RECLAIM*).

8.9 Relinquish memory transaction

This transaction is used by one or more Borrowers to relinquish their access to a memory region so that the Lender can reclaim exclusive access to it. The Lender starts this transaction by requesting each Borrower through a partition message to relinquish access. It reclaims access once all Borrowers have done so.

A list of valid combinations of roles played by various FF-A components in this transaction is specified in Table 8.8.

8.9.1 Relinquish memory access state machine

Table 8.12 describes the state machine for relinquishing a memory region from the perspective of two components *A & B*. *A* owns the memory region and could have lent or shared it with *B*. Alternatively, *B* might not have access to the memory region. *B* attempts to relinquish access to this memory region. Valid and invalid state transitions in response to this transaction have been listed.

Table 8.12: Relinquish and reclaim memory state machine

No.	Current Endpoint A state	Current Endpoint B state	Next Endpoint A state	Next Endpoint B state	Description
1	Owner-EA	!Owner-NA	Error	–	• Endpoint B tries to relinquish access to a memory region that the Owner has exclusive access to.
2	Owner-NA	!Owner-NA	Error	–	• Endpoint B tries to relinquish access to a memory region that the Owner has granted shared or exclusive access to one or more other Borrowers.
3	Owner-NA	!Owner-EA	Owner-EA	!Owner-NA	• Endpoint B relinquishes exclusive access to the memory region and transfers it back to the Owner.
4	Owner-NA	!Owner-SA	Owner-EA	!Owner-NA	• Endpoint B relinquishes access to the memory region that it shares with other Borrowers. Owner reclaims exclusive access once all Borrowers have relinquished access.
5	Owner-SA	!Owner-NA	Error	–	• Endpoint B tries to give up access to a memory region that the Owner shares with one or more other Borrowers.

No.	Current Endpoint A state	Current Endpoint B state	Next Endpoint A state	Next Endpoint B state	Description
6	Owner-SA	!Owner-SA	Owner-EA	!Owner-NA	• Endpoint B relinquishes access to the memory region that it shares with the Owner and possibly other Borrowers. Owner reclaims exclusive access once all Borrowers have relinquished access.

8.9.2 Relinquish memory transaction lifecycle

This transaction takes place as follows (also see 8.5.2 *Transaction life cycle*). It is assumed that the memory region was originally lent or shared by the Lender to the Borrowers. This transaction must not be used on a memory region owned by an endpoint.

1. If a Borrower is a *PE endpoint* or a *SEPID* associated with a dependent peripheral device,
    1. The Lender could use a Partition message to request each Borrower to relinquish access to the memory region. This message contains a description of the memory region relevant to the Borrower.
    2. Each Borrower uses the *FFA_MEM_RELINQUISH* interface (see 14.6 *FFA_MEM_RELINQUISH*) to unmap the memory region from its translation regime. This could be done in response to the message from the Lender or independently.
    3. Each Borrower uses a Partition message to inform the Lender that it has relinquished access to the memory region.

    In case of an error, the Borrower can abort the transaction before the Lender reclaims the memory region by calling the *FFA_MEM_RETRIEVE_REQ* ABI (see 14.4 *FFA_MEM_RETRIEVE_REQ*).
2. If the Borrower is a *SEPID* associated with an independent peripheral device,
    1. The Lender could use an IMPLEMENTATION DEFINED mechanism to request each Borrower to relinquish access to the memory region.
    2. Each Borrower uses an IMPLEMENTATION DEFINED mechanism to request the Relayer to unmap the memory region from its translation regime (see 14.7.1 *Component responsibilities for FFA_MEM_RECLAIM*). This could be done in response to the message from the Lender or independently.
    3. Each Borrower uses an IMPLEMENTATION DEFINED mechanism to inform the Lender that it has relinquished access to the memory region.
3. Once all Borrowers have relinquished access to the memory region, the Lender uses the *FFA_MEM_RECLAIM* interface to reclaim exclusive access to the memory region. This interface is described in 14.7 *FFA_MEM_RECLAIM*.

8.10 Memory region description

A memory region is described in a memory management transaction either through a *Composite memory region descriptor* (see 8.10.1 *Composite memory region descriptor*) or a globally unique *Handle* (see 8.10.2 *Memory region handle*).

The former is used to describe a memory region when a transaction to share, lend or donate memory is initiated by the Owner and when the memory region is retrieved by the Receiver.

The latter is used to describe a memory region when it is retrieved by a Receiver, relinquished by a Borrower and reclaimed by the Owner.

8.10.1 Composite memory region descriptor

A memory region is described in a memory management transaction by specifying the list and count of 4K sized pages that constitute it (see Table 8.13).

Table 8.13: Composite memory region descriptor

Field	Byte length	Byte offset	Description
Total page count	4	0	• Size of the memory region described as the count of 4K pages. • Must be equal to the sum of page counts specified in each constituent memory region descriptor. See Table 8.14.
Address range count	4	4	• Count of address ranges specified using constituent memory region descriptors.
Reserved (MBZ)	8	8	
Address range array	–	16	• Array of address ranges specified using constituent memory region descriptors.

The list is specified by using one or more constituent memory region descriptors (see Table 8.14). Each descriptor specifies the base address and size of a virtually or physically contiguous memory region.

Table 8.14: Constituent memory region descriptor

Field	Byte length	Byte offset	Description
Address	8	–	• Base VA, PA or IPA of constituent memory region aligned to the page size (4K) granularity.
Page count	4	8	• Number of 4K pages in constituent memory region.
Reserved (MBZ)	4	12	

The pages are addressed using VAs, IPAs or PAs depending on the FF-A instance at which the transaction is taking place. This is as follows.

- VAs are used at a Secure virtual FF-A instance if the partition runs in Secure EL0 or Secure User mode.
- IPAs are used at a virtual FF-A instance if the partition runs in one of the following Exception levels.
  - Secure EL1.
  - Secure Supervisor mode.
  - EL1.
  - Supervisor mode.
- PAs are used at all physical FF-A instances.

Figure 8.2 describes a virtually contiguous memory region range *VA_0* of size 16K through its composite memory region descriptors at the virtual and physical FF-A instances. *VA_0* was allocated through a dynamic memory management mechanism inside an endpoint for example, malloc. It is composed of:

- Two constituent IPA regions *IPA_0* and *IPA_1* of size 8K each at the virtual FF-A instance.
- *IPA_0* is comprised of two PA regions *PA_0* and *PA_1* at the physical FF-A instance. Each PA region is of size 4K.
- *IPA_1* is comprised of two PA regions *PA_2* and *PA_3* at the physical FF-A instance. Each PA region is of size 4K.

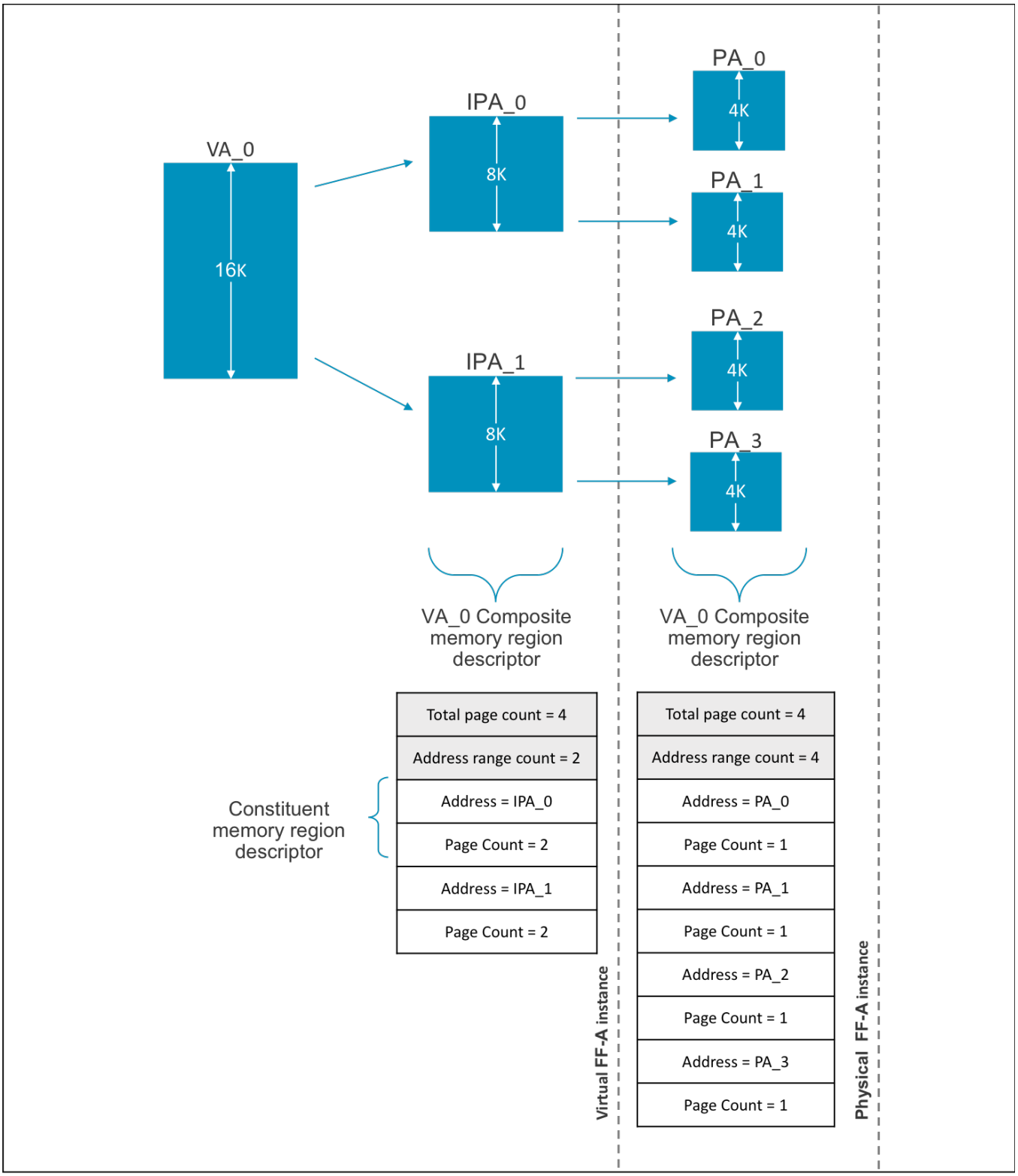

Figure 8.2: Example memory region description

8.10.2 Memory region handle

- A 64-bit *Handle* is used to identify a composite memory region description for example, *VA_0* described in Figure 8.2
- The *Handle* is allocated by the *Relayer* as follows.
  - The SPM must allocate the Handle if every Receiver participating in the memory management transaction is an SP or SEPID associated with a Secure Stream ID in the SMMU.
  - The Hypervisor must allocate the Handle if every Receiver participating in the memory management transaction is a VM or SEPID associated with a Non-secure Stream ID in the SMMU.
  - Either the Hypervisor or the SPM could allocate the Handle in all other cases (see 8.12.1 *Handle usage*).
- A *Handle* is allocated once a transaction to lend, share or donate memory is successfuly initiated by the *Owner*.
- Each *Handle* identifies a single unique composite memory region description that is, there is a 1:1 mapping between the two.
- A *Handle* is freed by the *Relayer* after it has been reclaimed by its *Owner* at the end of a successful transaction to relinquish the corresponding memory region description.
- Encoding of a *Handle* is as follows.
  - Bit[63]: Handle allocator.
    - b'0: Allocated by SPM.
    - b'1: Allocated by Hypervisor.
  - Bit[62:0]: IMPLEMENTATION DEFINED.
- A *Handle* must be encoded as a register parameter in any ABI that requires it as follows.
  - Two 32-bit general-purpose registers must be used such that if *Rx* and *Ry* are used, such that $x < y$,
    - *Rx = Handle[31:0].*
    - *Ry = Handle[63:32].*

8.11 Memory region properties

The properties of a memory region are as follows.

- *Instruction and data access permissions* describe the type of access permitted on the memory region.
- *One or more endpoint IDs* that have access to the memory region specified by a combination of access permissions and memory region attributes.
- *Memory region attributes* control the memory type, accesses to the caches, and whether the memory region is Shareable and therefore is coherent.

There is a 1:1 association between an endpoint and the permissions with which it can access a memory region. This is specified in Table 8.15.

Table 8.15: Memory access permissions descriptor

Field	Byte length	Byte offset	Description
Endpoint ID	2	–	• 16-bit ID of endpoint to which the memory access permissions apply.
Memory access permissions	1	2	• Permissions used to access a memory region. – bits[7:4]: Reserved (MBZ). – bits[3:2]: Instruction access permission. ∗ b'00: Not specified and must be ignored. ∗ b'01: Not executable. ∗ b'10: Executable. ∗ b'11: Reserved. Must not be used. – bits[1:0]: Data access permission. ∗ b'00: Not specified and must be ignored. ∗ b'01: Read-only. ∗ b'10: Read-write. ∗ b'11: Reserved. Must not be used.
Flags	1	3	• ABI specific flags as described in 8.11.1 *ABI-specific flags usage*.

Table 8.16 specifies the data structure that is used in memory management transactions to create an association between an endpoint, memory access permissions and a composite memory region description.

This data structure must be included in other data structures that are used in memory management transactions instead of being used as a stand alone data structure (see 8.12 *Lend, donate, and share transaction descriptor*). A composite memory region description is referenced by specifying an offset to it as described in Table 8.16. This enables one or more endpoints to be associated with the same memory region but with different memory access permissions for example, SP0 could have *RO* data access permission and SP1 could have *RW* data access permission to the same memory region.

Table 8.16: Endpoint memory access descriptor

Field	Byte length	Byte offset	Description
Memory access permissions descriptor	4	–	• Memory access permissions descriptor as specified in Table 8.15.
Composite memory region descriptor offset	4	4	• Offset to the composite memory region descriptor to which the endpoint access permissions apply (see Table 8.13). • Offset must be calculated from the base address of the data structure this descriptor is included in. • An offset value of 0 indicates that the endpoint access permissions apply to a memory region description identified by the *Handle* parameter specified in the data structure that includes this one.
Reserved (MBZ)	8	8	

8.11.1 ABI-specific flags usage

An endpoint can specify properties specific to the memory management ABI being invoked through this field.

In this version of the Framework, the *Flags* field MBZ and is reserved in an invocation of the following ABIs.

- FFA_MEM_DONATE.
- FFA_MEM_LEND.
- FFA_MEM_SHARE.

The *Flags* field must be encoded by the Receiver and the Relayer as specified in Table 8.17 in an invocation of the following ABIs.

- FFA_MEM_RETRIEVE_REQ.
- FFA_MEM_RETRIEVE_RESP.

The Relayer must return *INVALID_PARAMETERS* if the *Flags* field has been incorrectly encoded.

Table 8.17: Flags usage in FFA_MEM_RETRIEVE_REQ and FFA_MEM_RETRIEVE_RESP ABIs

Field	Description
Bit[0]	• Non-retrieval Borrower flag. – In a memory management transaction with multiple Borrowers, during the retrieval of the memory region, this flag specifies if the memory region must be or was retrieved on behalf of this endpoint or if the endpoint is another Borrower. ∗ b'0: Memory region must be or was retrieved on behalf of this endpoint. ∗ b'1: Memory region must not be or was not retrieved on behalf of this endpoint. It is another Borrower of the memory region. – This field MBZ if this endpoint: ∗ Is the only PE endpoint Borrower/Receiver in the transaction. ∗ Is a *Stream endpoint* and the caller of the *FFA_MEM_RETRIEVE_REQ* ABI is its *proxy endpoint*.
Bit[7:1]	• Reserved (MBZ).

8.11.2 Data access permissions usage

An endpoint could have either *Read-only* or *Read-write* data access permission to a memory region from the highest Exception level it runs in.

- *Read-write* permission is more permissive than *Read-only* permission.
- Data access permission is specified by setting *Bits[1:0]* in Table 8.15 to the appropriate value.

This access control is used in memory management transactions as follows.

1. In a transaction to lend or share memory,
   - The Lender must specify the level of access that the Borrower is permitted to have on the memory region. This is done while invoking the *FFA_MEM_SHARE* or *FFA_MEM_LEND* ABIs.
   - The Relayer must validate the permission specified by the Lender as follows. This is done in response to an invocation of the *FFA_MEM_SHARE* or *FFA_MEM_LEND* ABIs. The Relayer must return the *DENIED* error code if the validation fails.
     – At the Non-secure physical FF-A instance, an IMPLEMENTATION DEFINED mechanism is used to perform validation.
     – At any virtual FF-A instance, if the endpoint is running in EL1 or S-EL1 in either Execution state, the permission specified by the Lender is considered valid only if it is the same or less permissive than the permission used by the Relayer in the *S2AP* field in the stage 2 translation table descriptors for the memory region in one of the following translation regimes:
       ∗ Secure EL1&0 translation regime, when S-EL2 is enabled.
       ∗ Non-secure EL1&0 translation regime, when EL2 is enabled.
     – At the Secure virtual FF-A instance, if the endpoint is running in S-EL0 in either Execution state, the permission specified by the Lender is considered valid only if it is the same or less permissive than the permission used by the Relayer in the *AP[1]* field in the stage 1 translation table descriptors for the memory region in one of the following translation regimes:
       ∗ Secure EL1&0 translation regime, when EL2 is disabled.
       ∗ Secure PL1&0 translation regime, when EL2 is disabled.
       ∗ Secure EL2&0 translation regime, when Armv8.1-VHE is enabled.

If the Borrower is an independent peripheral device, then the validated permission is used to map the memory region into the address space of the device.

- The Borrower (if a PE or Proxy endpoint) should specify the level of access that it would like to have on the memory region.

  In a transaction to share or lend memory with more than one Borrower, each Borrower (if a PE or Proxy endpoint) could also specify the level of access that other Borrowers have on the memory region.

  This is done while invoking the *FFA_MEM_RETRIEVE_REQ* ABI.

- The Relayer must validate the permissions, if specified by the Borrower (if a PE or Proxy endpoint) in response to an invocation of the *FFA_MEM_RETRIEVE_REQ* ABI.

  It must ensure that the permission of the Borrower is the same or less permissive than the permission that was specified by the Lender and validated by the Relayer.

  It must ensure that the permissions for other Borrowers are the same as those specified by the Lender and validated by the Relayer.

  The Relayer must return the *DENIED* error code if the validation fails.

2. In a transaction to donate memory,
   - Whether the Owner is allowed to specify the level of access that the Receiver is permitted to have on the memory region depends on the type of Receiver.
     - If the Receiver is a PE or Proxy endpoint, the Owner must not specify the level of access.
     - If the Receiver is an independent peripheral device, the Owner could specify the level of access.

     The Owner must specify its choice in an invocation of the *FFA_MEM_DONATE* ABI.
   - The value of data access permission field specified by the Owner must be interpreted by the Relayer as follows. This is done in response to an invocation of the *FFA_MEM_DONATE* ABI.
     - If the Receiver is a PE or Proxy endpoint, the Relayer must return *INVALID_PARAMETERS* if the value is not *b'00*.
     - If the Receiver is an independent peripheral device and the value is not *b'00*, the Relayer must take one of the following actions.
       * Return *DENIED* if the permission is determined to be invalid through an IMPLEMENTATION DEFINED mechanism.
       * Use the permission specified by the Owner to map the memory region into the address space of the device.
     - If the Receiver is an independent peripheral device and the value is *b'00*, the Relayer must determine the permission value through an IMPLEMENTATION DEFINED mechanism.
   - The Receiver (if a PE or Proxy endpoint) should specify the level of access that it would like to have on the memory region. This is done while invoking the *FFA_MEM_RETRIEVE_REQ* ABI.
   - The Relayer must validate the permission specified by the Receiver to ensure that it is the same or less permissive than the permission determined by the Relayer through an IMPLEMENTATION DEFINED mechanism. This is done in response to an invocation of the *FFA_MEM_RETRIEVE_REQ* ABI. The Relayer must return the *DENIED* error code if the validation fails.
3. The Relayer must specify the permission that was used to map the memory region in the translation regime of the Receiver or Borrower. This must be done in an invocation of the *FFA_MEM_RETRIEVE_RESP* ABI.
4. In a transaction to relinquish memory that was lent to one or more Borrowers, the memory region must be mapped back into the translation regime of the Lender with the same data access permission that was used at the start of the transaction to lend the memory region. This is done in response to an invocation of the *FFA_MEM_RECLAIM* ABI.

8.11.3 Instruction access permissions usage

An endpoint could have either *Execute (X)* or *Execute-never (XN)* instruction access permission to a memory region from the highest Exception level it runs in.

- *Execute* permission is more permissive than *Execute-never* permission.
- Instruction access permission is specified by setting *Bits[3:2]* in Table 8.15 to the appropriate value.

This access control is used in memory management transactions as follows.

1. Only XN permission must be used in the following transactions.

   - In a transaction to share memory with one or more Borrowers.
   - In a transaction to lend memory to more than one Borrower.

   Bits[3:2] in Table 8.15 must be set to *b'00* as follows.

   - By the Lender in an invocation of *FFA_MEM_SHARE* or *FFA_MEM_LEND* ABIs.
   - By the Borrower in an invocation of the *FFA_MEM_RETRIEVE_REQ* ABI.

   The Relayer must set *Bits[3:2]* in Table 8.15 to *b'01* while invoking the *FFA_MEM_RETRIEVE_RESP* ABI.

2. In a transaction to donate memory or lend memory to a single Borrower,

   - Whether the Owner/Lender is allowed to specify the level of access that the Receiver is permitted to have on the memory region depends on the type of Receiver.
     - If the Receiver is a PE or Proxy endpoint, the Owner must not specify the level of access.
     - If the Receiver is an independent peripheral device, the Owner could specify the level of access.

     The Owner must specify its choice in an invocation of the *FFA_MEM_DONATE* or *FFA_MEM_LEND* ABIs.
   - The value of instruction access permission field specified by the Owner/Lender must be interpreted by the Relayer as follows. This is done in response to an invocation of the *FFA_MEM_DONATE* or *FFA_MEM_LEND* ABIs.
     - If the Receiver is a PE or Proxy endpoint, the Relayer must return *INVALID_PARAMETERS* if the value is not *b'00*.
     - If the Receiver is an independent peripheral device and the value is not *b'00*, the Relayer must take one of the following actions.
       - Return *DENIED* if the permission is determined to be invalid through an IMPLEMENTATION DEFINED mechanism.
       - Use the permission specified by the Owner to map the memory region into the address space of the device.
     - If the Receiver is an independent peripheral device and the value is *b'00*, the Relayer must determine the permission value through an IMPLEMENTATION DEFINED mechanism.
   - The Receiver (if a PE or Proxy endpoint) should specify the level of access that it would like to have on the memory region. This must be done in an invocation of the *FFA_MEM_RETRIEVE_REQ* ABI.
   - The Relayer must validate the permission specified by the Receiver (if a PE or Proxy endpoint) to ensure that it is the same or less permissive than the permission determined by the Relayer through an IMPLEMENTATION DEFINED mechanism.
     - For example, the Relayer could deny executable access to a Borrower on a memory region of Device memory type.

     This is done in response to an invocation of the *FFA_MEM_RETRIEVE_REQ* ABI. The Relayer must return the *DENIED* error code if the validation fails.

If the invocation of *FFA_MEM_RETRIEVE_REQ* succeeds, the Relayer must set *Bits[3:2]* in Table 8.15 to either *b'01* or *b'10'* while invoking the *FFA_MEM_RETRIEVE_RESP* ABI.

3. In a transaction to relinquish memory that was lent to one or more Borrowers, the memory region must be mapped back into the translation regime of the Lender with the same instruction access permission that was used at the start of the transaction to lend the memory region. This is done in response to an invocation of the *FFA_MEM_RECLAIM* ABI.

8.11.4 Memory region attributes usage

An endpoint can access a memory region by specifying attributes as follows.

- *Memory security state*. This could be Secure or Non-secure.
- *Memory type*. This could be Device or Normal. Device memory type could be one of the following types.
  - Device-nGnRnE.
  - Device-nGnRE.
  - Device-nGRE.
  - Device-GRE.

  The precedence rules for memory types are as follows. < should be read as *is less permissive than*.
  - Device-nGnRnE < Device-nGnRE < Device-nGRE < Device-GRE < Normal.
- *Cacheability attribute*. This could be one of the following types.
  - Non-cacheable.
  - Write-Back Cacheable.

  These attributes are used to specify both inner and outer cacheability. The precedence rules are as follows.
  - Non-cacheable < Write-Back Cacheable.
- *Shareability attribute*. This could be one of the following types.
  - Non-shareable.
  - Outer Shareable.
  - Inner Shareable.

  The precedence rules are as follows.
  - Non-Shareable < Inner Shareable < Outer shareable.

The data structure to encode memory region attributes is specified in Table 8.18.

The security state of a memory region is specified by setting *Bit[6]* in Table 8.18 to an appropriate value. The usage is described in 8.11.4.1 *Usage of NS bit*.

Other memory region attributes are specified by an endpoint by setting *Bits[5:0]* in Table 8.18 to appropriate values. The usage is described in 8.11.4.2 *Usage of other memory region attributes*.

Table 8.18: Memory region attributes descriptor

Field	Byte length	Byte offset	Description
Memory region attributes	1	–	• Attributes used to access a memory region. – bits[7]: Reserved (MBZ). – bits[6]: NS-bit. ∗ b'0: Secure memory. ∗ b'1: Non-secure memory. – bits[5:4]: Memory type. ∗ b'00: Not specified and must be ignored. ∗ b'01: Device memory. ∗ b'10: Normal memory. ∗ b'11: Reserved. Must not be used. – bits[3:2]: ∗ Cacheability attribute if bit[5:4] = b'10. · b'00: Reserved. Must not be used. · b'01: Non-cacheable. · b'10: Reserved. Must not be used. · b'11: Write-Back. ∗ Device memory attributes if bit[5:4] = b'01. · b'00: Device-nGnRnE. · b'01: Device-nGnRE. · b'10: Device-nGRE. · b'11: Device-GRE. – bits[1:0]: ∗ Shareability attribute if bit[5:4] = b'10. · b'00: Non-shareable. · b'01: Reserved. Must not be used. · b'10: Outer Shareable. · b'11: Inner Shareable. ∗ Reserved & MBZ if bit[5:4] = b'01.

8.11.4.1 Usage of NS bit

The *NS bit* is used by the SPMC to specify the security state of a memory region retrieved by a SP. The following rules govern the usage of this bit.

1. The *NS bit* is reserved and MBZ in an invocation of the following ABIs.
    1. FFA_MEM_DONATE
    2. FFA_MEM_LEND
    3. FFA_MEM_SHARE
    4. FFA_MEM_RETRIEVE_REQ

    The callee at any FF-A instance must return INVALID_PARAMETERS if the bit is set by the caller.
2. The *NS bit* is set to *b'1* by the Hypervisor in an invocation of the FFA_MEM_RETRIEVE_RESP ABI at the Non-secure virtual FF-A instance.
3. The *NS bit* is used by the SPMC in an invocation of the FFA_MEM_RETRIEVE_RESP ABI at the Secure virtual FF-A instance as described below.
    1. In response to an invocation of the FFA_MEM_RETRIEVE_REQ ABI to retrieve shared memory,
        1. *NS bit = b'1* in the following scenarios.

1. Owner of the memory region is a NS-Endpoint.
2. Owner of the memory region is a SP, and the region is mapped as Non-secure in the Owner's translation regime.

2. *NS bit = b'0* in the following scenario.
    1. Owner of the memory region is a SP, and the region is mapped as Secure in the Owner's translation regime.

2. In response to an invocation of the FFA_MEM_RETRIEVE_REQ ABI to retrieve lent memory,
    1. *NS bit = b'1* in the following scenarios.
        1. At least one Borrower of the memory region is a NS-Endpoint.
        2. Owner of the memory region is a NS-Endpoint, no Borrower of the memory region is a NS-Endpoint but the SPMC cannot change the security state of the memory region.
        3. Owner of the memory region is a SP, and the memory region is mapped as Non-secure in its translation regime.
    2. *NS bit = b'0* in any scenario where *NS bit != b'1*.
3. In response to an invocation of the FFA_MEM_RETRIEVE_REQ ABI to retrieve donated memory,
    1. *NS bit = b'1* in the following scenarios.
        1. Owner of the memory region is a NS-Endpoint, but the SPMC cannot change the security state of the memory region.
        2. Owner of the memory region is a SP, and the memory region is mapped as Non-secure in its translation regime.
    2. *NS bit = b'0* in any scenario where *NS bit != b'1*.
4. If the SPMC changes the security state of the memory region, then it must restore it back to its original security state before allowing the Owner of the memory region to successfuly reclaim the memory region through an invocation of the FFA_MEM_RECLAIM ABI.

8.11.4.2 Usage of other memory region attributes

Memory region attributes are used in memory management transactions as follows.

1. In a transaction to share memory with one or more Borrowers and to lend memory to more than one Borrower,
    - The Lender specifies the attributes that each Borrower must access the memory region with. This is done by invoking the *FFA_MEM_SHARE* or *FFA_MEM_LEND* ABIs. The same attributes are used for all Borrowers.
    - The Relayer validates the attributes specified by the Lender as follows. This is done in response to an invocation of the *FFA_MEM_SHARE* or *FFA_MEM_LEND* ABIs. The Relayer must return the *DENIED* error code if the validation fails.
        - At the Non-secure physical FF-A instance, an IMPLEMENTATION DEFINED mechanism is used.
        - At any virtual FF-A instance, if the endpoint is running in EL1 or S-EL1 in either Execution state, the attributes specified by the Lender are considered valid only if they are the same or less permissive than the attributes used by the Relayer in the stage 2 translation table descriptors for the memory region in one of the following translation regimes:
            * Secure EL1&0 translation regime, when S-EL2 is enabled.
            * Non-secure EL1&0 translation regime, when EL2 is enabled.

  - At the Secure virtual FF-A instance, if the endpoint is running in S-EL0 in either Execution state, the attributes specified by the Lender are considered valid only if they are either the same or less permissive than the attributes used by the Relayer in the stage 1 translation table descriptors for the memory region in one of the following translation regimes:
    * Secure EL1&0 translation regime, when EL2 is disabled.
    * Secure PL1&0 translation regime, when EL2 is disabled.
    * Secure EL2&0 translation regime, when Armv8.1-VHE is enabled.

  If the Borrower is an independent peripheral device, then the validated attributes are used to map the memory region into the address space of the device.

- The Borrower (if a PE or Proxy endpoint) should specify the attributes that it would like to access the memory region with. This is done by invoking the *FFA_MEM_RETRIEVE_REQ* ABI.
- The Relayer must validate the attributes specified by the Borrower (if a PE or Proxy endpoint) to ensure that they are the same or less permissive than the attributes that were specified by the Lender and validated by the Relayer. This is done in response to an invocation of the *FFA_MEM_RETRIEVE_REQ* ABI. The Relayer must return the *DENIED* error code if the validation fails.

2. In a transaction to donate memory or lend memory to a single Borrower,
   - Whether the Owner/Lender is allowed to specify the memory region attributes that the Receiver must use to access the memory region depends on the type of Receiver.
     - If the Receiver is a PE or Proxy endpoint, the Owner must not specify the attributes.
     - If the Receiver is an independent peripheral device, the Owner could specify the attributes.

     The Owner must specify its choice in an invocation of the *FFA_MEM_DONATE* or *FFA_MEM_LEND* ABIs.
   - The values in the memory region attributes field specified by the Owner/Lender must be interpreted by the Relayer as follows. This is done in response to an invocation of the *FFA_MEM_DONATE* or *FFA_MEM_LEND* ABIs.
     - If the Receiver is a PE or Proxy endpoint, the Relayer must return *INVALID_PARAMETERS* if the value in *bits[5:4]* != *b'00*.
     - If the Receiver is an independent peripheral device and the value is not *b'00*, the Relayer must take one of the following actions.
       * Return *DENIED* if the attributes are determined to be invalid through an IMPLEMENTATION DEFINED mechanism.
       * Use the attributes specified by the Owner to map the memory region into the address space of the device.
     - If the Receiver is an independent peripheral device and the value is *b'00*, the Relayer must determine the attributes through an IMPLEMENTATION DEFINED mechanism.
   - The Receiver (if a PE or Proxy endpoint) should specify the memory region attributes it would like to use to access the memory region. This is done while invoking the *FFA_MEM_RETRIEVE_REQ* ABI.
   - The Relayer must validate the attributes specified by the Receiver (if a PE or Proxy endpoint) to ensure that they are the same or less permissive than the attributes determined by the Relayer through an IMPLEMENTATION DEFINED mechanism.

     This is done in response to an invocation of the *FFA_MEM_RETRIEVE_REQ* ABI. The Relayer must return the *DENIED* error code if the validation fails.

3. The Relayer must specify the memory region attributes that were used to map the memory region in the translation regime of the Receiver or Borrower. This must be done while invoking the *FFA_MEM_RETRIEVE_RESP* ABI.
4. In a transaction to relinquish memory that was lent to one or more Borrowers, the memory region must be mapped back into the translation regime of the Lender with the same attributes that were used at the start of the transaction to lend the memory region. This is done in response to an invocation of the *FFA_MEM_RECLAIM* ABI.

8.12 Lend, donate, and share transaction descriptor

Table 8.19 specifies the data structure that must be used by the Owner/Lender and a Borrower/Receiver in a transaction to donate, lend or share a memory region. It specifies the memory region description (see 8.10 *Memory region description*), properties (see 8.11 *Memory region properties*) and other transaction attributes in an invocation of the following ABIs.

- FFA_MEM_DONATE.
- FFA_MEM_LEND.
- FFA_MEM_SHARE.
- FFA_MEM_RETRIEVE_REQ.
- FFA_MEM_RETRIEVE_RESP.

The interpretation of some fields in Table 8.19 depends on the ABI this table is used with. This variance in behavior is also specified in Table 8.19.

Table 8.19: Lend, donate or share memory transaction descriptor

Field	Byte length	Byte offset	Description
Sender endpoint ID	2	0	• ID of the Owner endpoint.
Memory region attributes	1	2	• Attributes must be encoded as specified in 8.11.4 *Memory region attributes usage*. • Attribute usage is subject to validation at the virtual and physical FF-A instances as specified in 8.11.4 *Memory region attributes usage*.
Reserved (MBZ)	1	3	
Flags	4	4	• Flags must be encoded as specified in in 8.12.4 *Flags usage*.
Handle	8	8	• Memory region handle in ABI invocations specified in 8.12.1 *Handle usage*.
Tag	8	16	• This field must be encoded as specified in 8.12.2 *Tag usage*.
Reserved (MBZ)	4	24	
Endpoint memory access descriptor count	4	28	• Count of endpoint memory access descriptors.
Endpoint memory access descriptor array	–	32	• Each entry in the array must be encoded as specified in 8.12.3 *Endpoint memory access descriptor array usage*. See Table 8.16 for the encoding of the endpoint memory access descriptor.

8.12.1 Handle usage

- This field must be zero (MBZ) in an invocation of the following ABIs at a virtual FF-A instance.
  - FFA_MEM_DONATE.
  - FFA_MEM_LEND.
  - FFA_MEM_SHARE.
- The Hypervisor could allocate the *Handle* and populate it in this field in the scenarios described in 8.10.2 *Memory region handle*. This is applicable in an invocation of the following ABIs at a Non-secure physical FF-A instance.
  - FFA_MEM_DONATE.
  - FFA_MEM_LEND.
  - FFA_MEM_SHARE.

  If the SPM cannot use the *Handle* allocated by the Hypervisor, it must return INVALID_PARAMETERS.
- A successful invocation of each of the preceding ABIs returns a *Handle* (see 8.10.2 *Memory region handle*) to identify the memory region in the transaction.

  This is also applicable to an invocation of these ABIs at the Non-secure physical FF-A instance where the Hypervisor allocates the *Handle*. The SPMC must return the allocated *Handle* if it can use it.
- The Sender must convey the *Handle* to the Receiver through a Partition message.
- This field must be used by the Receiver to encode this *Handle* in an invocation of the FFA_MEM_RETRIEVE_REQ ABI.
- A Relayer must validate this field in an invocation of the FFA_MEM_RETRIEVE_REQ ABI as follows.
  - Ensure that it holds a *Handle* value that was previously allocated and has not been reclaimed by the Owner.
  - Ensure that the *Handle* identifies a memory region that was shared, lent or donated to the Receiver.
  - Ensure that the *Handle* was allocated to the Owner specified in the *Sender endpoint ID* field of the transaction descriptor.

  It must return *INVALID_PARAMETERS* if the validation fails.
- This field must be used by the Relayer to encode the *Handle* in an invocation of the FFA_MEM_RETRIEVE_RESP ABI.

8.12.2 Tag usage

- This 64-bit field must be used to specify an IMPLEMENTATION DEFINED value associated with the transaction and known to participating endpoints.
- The Sender must specify this field to the Relayer in an invocation of the following ABIs.
  - FFA_MEM_DONATE.
  - FFA_MEM_LEND.
  - FFA_MEM_SHARE.
- The Sender must convey the *Tag* to the Receiver through a Partition message.
- This field must be used by the Receiver to encode the *Tag* in an invocation of the FFA_MEM_RETRIEVE_REQ ABI.
- The Relayer must ensure the *Tag* value specified by the Receiver is equal to the value that was specified by the Sender. It must return *INVALID_PARAMETERS* if the validation fails.
- This field must be used by the Relayer to encode the *Tag* value in an invocation of the FFA_MEM_RETRIEVE_RESP ABI.

8.12.3 Endpoint memory access descriptor array usage

8.12.3.1 Sender usage

A Sender must use this field to specify one or more Receivers and the access permissions each should have on the memory region it is donating, lending or sharing through an invocation of one of the following ABIs.

- FFA_MEM_DONATE.
- FFA_MEM_LEND.
- FFA_MEM_SHARE.

The access permissions and flags are subject to validation at the virtual and physical FF-A instances as specified in 8.11.3 *Instruction access permissions usage*, 8.11.2 *Data access permissions usage* and 8.11.1 *ABI-specific flags usage*.

In an FFA_MEM_SHARE ABI invocation, the Sender could request the memory region to be mapped with different data access permission in its own translation regime. It must specify these permissions and its endpoint ID in a separate Endpoint memory access descriptor.

A Sender must describe the memory region in a composite memory region descriptor (see Table 8.13) with the following non-exhaustive list of checks.

- Ensure that the address ranges specified in the composite memory region descriptor do not overlap each other.
- *Total page count* is equal to the sum of the *Page count* fields in each *Constituent memory region descriptor*.

The offset to this descriptor from the base of Table 8.19 must be specified in the *Offset* field of the Endpoint memory access descriptor as follows.

- In an FFA_MEM_DONATE ABI invocation,
  - The *Endpoint memory access descriptor count* field in the transaction descriptor must be set to 1. This implies that the Owner must specify a single Receiver endpoint in a transaction to donate memory.
  - The *Offset* field of the Endpoint memory access descriptor must be set to the offset of the composite memory region descriptor
- In an FFA_MEM_LEND and FFA_MEM_SHARE ABI invocation,
  - The *Endpoint memory access descriptor count* field in the transaction descriptor must be set to a non-zero value. This implies that the Owner must specify at least a single Borrower endpoint in a transaction to lend or share memory.
  - The *Offset* field in the Endpoint memory access descriptor of each Borrower must be set to the offset of the composite memory region descriptor. This implies that all values of the *Offset* field must be equal.

8.12.3.2 Receiver usage

A Receiver must use this field to specify the access permissions it should have on the memory region being donated, lent or shared in an invocation of the FFA_MEM_RETRIEVE_REQ ABI. This is specified in 8.11.3 *Instruction access permissions usage* and 8.11.2 *Data access permissions usage*.

- A Receiver could do this on its behalf if it is a PE endpoint.
- A Receiver could do this on the behalf of its dependent peripheral devices if it is a proxy endpoint.

A Receiver could specify the address ranges that must be used to map the memory region in its translation regime by describing them in a composite memory region descriptor. The Receiver must perform the same checks as a Sender. These checks are described in 8.12.3.1 *Sender usage*.

The offset to this descriptor from the base of Table 8.19 must be specified in the *Offset* field of the corresponding endpoint memory access descriptor in the array. This implies that all values of the *Offset* field could be different from each other.

A Receiver could let the Relayer allocate the address ranges that must be used to map the memory region in its translation regime and optionally provide an alignment hint (see *Address range alignment hint* in Table 8.21).

The value 0 must be specified in the *Offset* field of the corresponding endpoint memory access descriptor in the array. This implies that the *Handle* specified in Table 8.19 must be used to identify the memory region (see 8.12.1 *Handle usage*).

A memory management transaction could be to lend or share memory with multiple Borrowers. The Receiver must use this field to specify:

- The SEPIDs and data access permissions of any dependent peripheral devices (if any) that the Receiver is a *proxy endpoint* for.

  If the Relayer must allocate the address ranges to map the memory region in the *Stream endpoints*, the value 0 must be specified in the *Offset* field of the corresponding endpoint memory access descriptor in the array.

  If the Receiver specifies the address ranges to map the memory region in the *Stream endpoints*, then it must follow the preceding guidance to specify the address ranges that must be used to map the memory region in its translation regime.

- The identity of any other Borrowers and their data access permissions on the memory region (see 8.11.2 *Data access permissions usage* and 8.11.1 *ABI-specific flags usage*).

  The value 0 must be specified in the *Offset* field of the corresponding endpoint memory access descriptor in the array.

8.12.3.3 Relayer usage

A Relayer must validate the *Endpoint memory access descriptor count* and each entry in the *Endpoint memory access descriptor array* as follows.

- The Relayer could support memory management transactions targeted to only a single Receiver endpoint.

  It must return *INVALID_PARAMETERS* if the Sender or Receiver specifies an *Endpoint memory access descriptor count != 1*.

- It must ensure that these fields have been populated by the Sender as specified in 8.12.3.1 *Sender usage* in an invocation of any of the following ABIs.

  – FFA_MEM_DONATE.
  – FFA_MEM_LEND.
  – FFA_MEM_SHARE.

  The Relayer must return *INVALID_PARAMETERS* in case of an error.

- It must ensure that the *Endpoint ID* field in each Memory access permissions descriptor specifies a valid endpoint that it manages. The Relayer must return *INVALID_PARAMETERS* in case of an error.

- In an invocation of the FFA_MEM_RETRIEVE_REQ ABI,

  – It must ensure that these fields have been populated by the Receiver as specified in 8.12.3.2 *Receiver usage*.

  – If the memory region has been lent or shared with multiple Borrowers, the Relayer must ensure that the identity of each Borrower specified by the Receiver is the same as that specified by the Sender.

  – If one or more Borrowers are dependent peripheral devices, the Relayer must ensure that the Receiver is their proxy endpoint.

  – If the Receiver specifies the address ranges that must be used to map the memory region in its translation regime, the Relayer must ensure that the size of the memory region is equal to that specified by the Sender.

  The Relayer must return *INVALID_PARAMETERS* in case of an error.

- It must validate the access permissions in the *Memory access permissions descriptor* in each *Endpoint memory access descriptor* as specified in 8.11.2 *Data access permissions usage* and 8.11.3 *Instruction access permissions usage*.

A Relayer must use this field in an invocation of the FFA_MEM_RETRIEVE_RESP ABI in response to successful validation of an FFA_MEM_RETRIEVE_REQ ABI invocation as follows.

- To specify the access permissions with which the memory region has been mapped in the translation regime of the Receiver.
- A Receiver could let the Relayer allocate the address ranges to map the memory region. In this case, the Relayer must describe the address ranges in a composite memory region descriptor. The Relayer must perform the same checks as a Sender. These checks are described in 8.12.3.1 *Sender usage*.

  The offset to this descriptor from the base of Table 8.19 must be specified in the *Offset* field of the corresponding endpoint memory access descriptor in the array. This implies that all values of the *Offset* field could be different from each other.
- A Receiver could specify the address ranges that must be used to map the memory region in its translation regime. The Relayer must specify the value 0 in the *Offset* field of the corresponding endpoint memory access descriptor in the array.

8.12.4 Flags usage

- Flags are used to govern the behavior of a memory management transaction.
- Usage of the Flags field in an invocation of the following ABIs is specified in Table 8.20.
  - FFA_MEM_DONATE.
  - FFA_MEM_LEND.
  - FFA_MEM_SHARE.
- Usage of the Flags field in an invocation of the FFA_MEM_RETRIEVE_REQ ABI is specified in Table 8.21.
- Usage of the Flags field in an invocation of the FFA_MEM_RETRIEVE_RESP ABI is specified in Table 8.22.

8.12.4.1 Zero memory flag

In some ABI invocations, the caller could set a flag to request the Relayer to zero a memory region. To do this, the Relayer must:

- Map the memory region in its translation regime once it is not mapped in the translation regime of any other FF-A component.

  The caller must ensure that the memory region fulfills the size and alignment requirements listed in 2.7 *Memory granularity and alignment* to allow the Relayer to map this memory region. It must discover these requirements by invoking the *FFA_FEATURES* interface with the function ID of the *FFA_RXTX_MAP* interface (see 11.2 *FFA_FEATURES*).

  The Relayer must return *INVALID_PARAMETERS* if the memory region does not meet these requirements.
- Zero the memory region and perform cache maintenance such that the memory regions contents are coherent between any PE caches, system caches and system memory.
- Unmap the memory region from its translation regime before it is mapped in the translation regime of any other FF-A component.

Table 8.20: Flags usage in FFA_MEM_DONATE, FFA_MEM_LEND and FFA_MEM_SHARE ABIs

Field	Description
Bit[0]	• Zero memory flag. – In an invocation of FFA_MEM_DONATE or FFA_MEM_LEND, this flag specifies if the memory region contents must be zeroed by the Relayer after the memory region has been unmapped from the translation regime of the Owner. ∗ b'0: Relayer must not zero the memory region contents. ∗ b'1: Relayer must zero the memory region contents. – MBZ in an invocation of FFA_MEM_SHARE, else the Relayer must return *INVALID_PARAMETERS*. – MBZ if the Owner has Read-only access to the memory region , else the Relayer must return *DENIED*.
Bit[1]	• Operation time slicing flag. – In an invocation of FFA_MEM_DONATE, FFA_MEM_LEND or FFA_MEM_SHARE, this flag specifies if the Relayer can time slice this operation. ∗ b'0: Relayer must not time slice this operation. ∗ b'1: Relayer can time slice this operation. – MBZ if the Relayer does not support time slicing of memory management operations (see 16.2.3 *Time slicing of memory management operations*), else the Relayer must return *INVALID_PARAMETERS*.
Bit[31:2]	• Reserved (MBZ).

Table 8.21: Flags usage in FFA_MEM_RETRIEVE_REQ ABI

Field	Description
Bit[0]	• Zero memory before retrieval flag. – In an invocation of FFA_MEM_RETRIEVE_REQ, during a transaction to lend or donate memory, this flag is used by the Receiver to specify whether the memory region must be retrieved with or without zeroing its contents first. ∗ b'0: Retrieve the memory region irrespective of whether the Sender requested the Relayer to zero its contents prior to retrieval. ∗ b'1: Retrieve the memory region only if the Sender requested the Relayer to zero its contents prior to retrieval by setting the *Bit[0]* in Table 8.20). – MBZ in a transaction to share a memory region, else the Relayer must return *INVALID_PARAMETER*. – If the Sender has Read-only access to the memory region and the Receiver sets Bit[0], the Relayer must return *DENIED*. – MBZ if the Receiver has previously retrieved this memory region, else the Relayer must return *INVALID_PARAMETERS* (see 14.4.2 *Support for multiple retrievals by a Borrower*).

Field	Description
Bit[1]	• Operation time slicing flag. – In an invocation of FFA_MEM_RETRIEVE_REQ, this flag specifies if the Relayer can time slice this operation. ∗ b'0: Relayer must not time slice this operation. ∗ b'1: Relayer can time slice this operation. – MBZ if the Relayer does not support time slicing of memory management operations (see 16.2.3 *Time slicing of memory management operations*), else the Relayer must return *INVALID_PARAMETERS*.
Bit[2]	• Zero memory after relinquish flag. – In an invocation of FFA_MEM_RETRIEVE_REQ, this flag specifies whether the Relayer must zero the memory region contents after unmapping it from the translation regime of the Borrower or if the Borrower crashes. ∗ b'0: Relayer must not zero the memory region contents. ∗ b'1: Relayer must zero the memory region contents. – If the memory region is lent to multiple Borrowers, the Relayer must clear memory region contents after unmapping it from the translation regime of each Borrower, if any Borrower including the caller sets this flag. – This flag could be overridden by the Receiver in an invocation of FFA_MEM_RELINQUISH (see *Flags* field in Table 14.25). – MBZ if the Receiver has Read-only access to the memory region, else the Relayer must return *DENIED*. The Receiver could be a PE endpoint or a dependent peripheral device. – MBZ in a transaction to share a memory region, else the Relayer must return *INVALID_PARAMETER*.
Bit[4:3]	• Memory management transaction type flag. – In an invocation of FFA_MEM_RETRIEVE_REQ, this flag is used by the Receiver to either specify the memory management transaction it is participating in or indicate that it will discover this information in the invocation of FFA_MEM_RETRIEVE_RESP corresponding to this request. ∗ b'00: Relayer must specify the transaction type in FFA_MEM_RETRIEVE_RESP. ∗ b'01: Share memory transaction. ∗ b'10: Lend memory transaction. ∗ b'11: Donate memory transaction. – Relayer must return *INVALID_PARAMETERS* if the transaction type specified by the Receiver is not the same as that specified by the Sender for the memory region identified by the *Handle* value specified in the transaction descriptor.

Field	Description
Bit[9:5]	• Address range alignment hint. – In an invocation of FFA_MEM_RETRIEVE_REQ, this flag is used by the Receiver to specify the boundary, expressed as multiples of 4KB, to which the address ranges allocated by the Relayer to map the memory region must be aligned. – Bit[9]: Hint valid flag. ∗ b'0: Relayer must choose the alignment boundary. Bits[8:5] are reserved and MBZ. ∗ b'1: Relayer must use the alignment boundary specified in Bits[8:5]. – Bit[8:5]: Alignment hint. ∗ If the value in this field is *n*, then the address ranges must be aligned to the *2*n x 4KB* boundary. – MBZ if the Receiver specifies the IPA or VA address ranges that must be used by the Relayer to map the memory region, else the Relayer must return *INVALID_PARAMETERS*. – Relayer must return *DENIED* if it is not possible to allocate the address ranges at the alignment boundary specified by the Receiver. – Relayer must return *INVALID_PARAMETERS* if a reserved value is specified by the Receiver.
Bit[31:10]	• Reserved (MBZ).

Table 8.22: Flags usage in FFA_MEM_RETRIEVE_RESP ABI

Field	Description
Bit[0]	• Zero memory before retrieval flag. – In an invocation of FFA_MEM_RETRIEVE_RESP during a transaction to lend or donate memory, this flag is used by the Relayer to specify whether the memory region was retrieved with or without zeroing its contents first. ∗ b'0: Memory region was retrieved without zeroing its contents. ∗ b'1: Memory region was retrieved after zeroing its contents. – MBZ in a transaction to share a memory region. – MBZ if the Sender has Read-only access to the memory region.
Bit[2:1]	• Reserved (MBZ).
Bit[4:3]	• Memory management transaction type flag. – In an invocation of FFA_MEM_RETRIEVE_RESP, this flag is used by the Relayer to specify the memory management transaction the Receiver is participating in. ∗ b'00: Reserved. ∗ b'01: Share memory transaction. ∗ b'10: Lend memory transaction. ∗ b'11: Donate memory transaction.
Bit[31:5]	• Reserved (MBZ).

Chapter 9
Interface overview

The interfaces used by FF-A components for communication at an FF-A instance are described in the following sections.

- Interfaces for reporting status of execution of other interfaces are described in Chapter 10 *Status reporting interfaces*.
- Interfaces for partition setup and discovery using Framework messages are described in Chapter 11 *Setup and discovery interfaces*.
- Interfaces to implement memory management transactions using Framework messages are described in Chapter 14 *Memory management interfaces*.
- Interfaces to manage CPU cycles allocated to an endpoint are described in Chapter 12 *CPU cycle management interfaces*.
- Interfaces to implement exchange of direct and indirect Partition messages between endpoints are described in Chapter 13 *Messaging interfaces*.
- Additional interfaces for memory management and interfaces pertaining to power management are described in Chapter 16 *Appendix*.

The following common rules govern the definition and behavior of FF-A ABIs.

1. Each interface is invoked using one more conduits described in 2.4 *Conduits*.
2. Each interface is based on the AArch64 and AArch32 SMC calling convention described in [4] apart from the divergence described in 9.1 *Divergence from SMC calling convention*.
3. Usage of only those architectural registers that are relevant to an interface is specified. The values of all other architectural registers must be ignored.

4. The following standard Secure service call identifier ranges have been reserved for FF-A interfaces in the SMCCC [4].
    1. **0x84000060-0x840000FF**: FF-A 32-bit calls.
        - A caller in the AArch32 Execution state, uses the function identifiers for 32-bit calls.
    2. **0xC4000060-0xC40000FF**: FF-A 64-bit calls.
        - A caller in the AArch64 Execution state, can use the function identifiers for 32-bit or 64-bit calls.
5. An invocation of any interface is completed by invoking the *FFA_ERROR* interface with the *NOT_SUPPORTED* error code in the following scenarios.
    - The interface was invoked at an FF-A instance where it cannot be invoked through any conduit.
    - The interface was invoked through an invalid conduit at an FF-A instance where it can be invoked.

    An FF-A component at the lower EL at an FF-A instance uses the *FFA_FEATURES* interface (see 11.2 *FFA_FEATURES*) to discover if an FF-A ABI is implemented by the FF-A component at the higher EL.

9.1 Divergence from SMC calling convention

The SMC calling convention describes the concept of *fast* and *yielding* SMC calls. The type of call is specified in *bit[31]* of the *Function ID* parameter of an SMC. The function ID range for yielding calls is reserved for legacy SMC interfaces.

FF-A interfaces fall in both categories. Furthermore, the *yielding* nature of some FF-A ABIs depends entirely upon the protocol between a service and its clients.

For example, a Receiver endpoint that is allocated CPU cycles through the FFA_MSG_SEND_DIRECT_REQ ABI could be preempted by a Non-secure interrupt or perform a managed exit. In the latter case, the endpoint could complete the requested operation before relinquishing control to the Normal world.

From the scheduler's perspective, the invocation of FFA_MSG_SEND_DIRECT_REQ completes with FFA_INTERRUPT in the former case and FFA_MSG_SEND_DIRECT_RESP in the latter case. In the latter case, whether the requested operation is preempted or completed depends upon the service level protocol between the Receiver and Scheduler endpoints. This is not visible to the Framework. The call runs to completion from the Framework's perspective.

On the other hand, *hypcall* interfaces are not preempted by Non-secure interrupts and run to completion from the caller's perspective.

It is not possible to consistently categorize FF-A ABIs as *fast* or *yielding*. Furthermore, function IDs for yielding calls cannot be allocated for FF-A ABIs as they lie in the reserved range. Hence, function IDs for FF-A ABIs are allocated from the *fast call* range. However, *bit[31]* of the *Function ID* parameter in an FF-A ABI is ignored by the Framework for the purpose defined in the SMC calling convention specification.

Chapter 10
Status reporting interfaces

10.1 Overview

Interfaces described in this section are used to report the status of a previous FF-A ABI invocation. The status indicates successful or unsuccessful completion or preemption of the ABI invocation. This ABI must be one that is listed in the following sections.

- Interfaces for partition setup and discovery[1] in Chapter 11 *Setup and discovery interfaces*.
- Interfaces to implement memory management transactions in Chapter 14 *Memory management interfaces*.
- Interfaces to manage CPU cycles in Chapter 12 *CPU cycle management interfaces*.
- Interfaces to implement messaging between endpoints in Chapter 13 *Messaging interfaces*.

[1] The *FFA_VERSION* interface (see 11.1 *FFA_VERSION*) is used for discovering the presence of a Framework implementation. It does not use the status reporting interfaces.

10.2 FFA_ERROR

Description

- Returns error code in response to a previous invocation of an FF-A function.
- Table 10.2 defines the values for status codes used with FF-A functions. All values are considered to be 32-bit signed integers.
- Valid FF-A instances and conduits are listed in Table 10.3.
- Syntax of this function is described in Table 10.4.
- Figure 10.1 illustrates example usage of this function with the following assumptions.
  - Component A makes an invalid request to Component B through an FF-A function described in this specification.
  - Component B uses the FFA_ERROR function to return the error code to Component A.
  - The FF-A function used by component A can be invoked through the SMC and ERET conduits.
  - Both components could be interacting at any FF-A instance support by the FF-A function. The two possible scenarios have been considered.
    - Component A is at a lower EL than component B at the FF-A instance.
    - Component A is at a higher EL than component B at the FF-A instance.

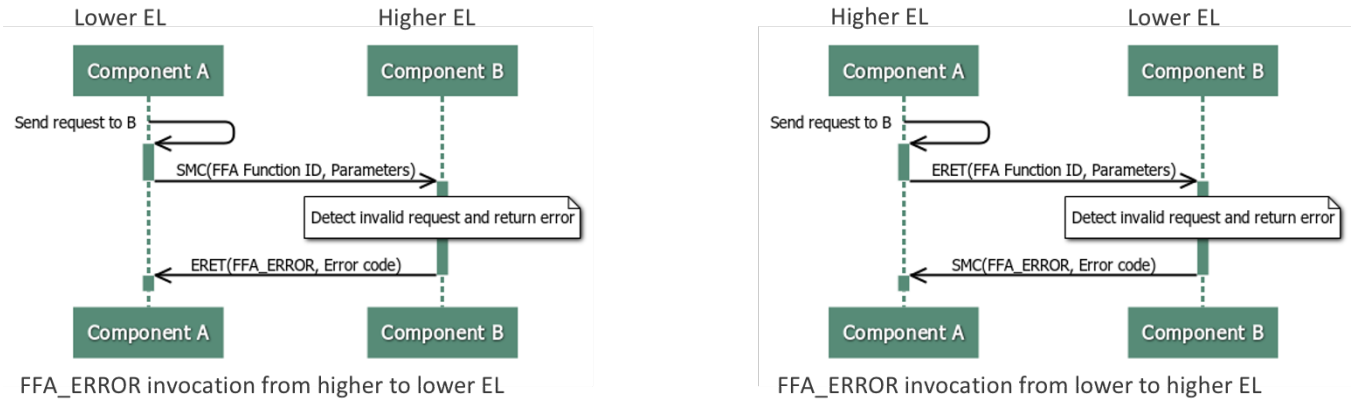

Figure 10.1: Example usage of FFA_ERROR

Table 10.2: Error status codes

Status code	Description
-1	NOT_SUPPORTED
-2	INVALID_PARAMETERS
-3	NO_MEMORY
-4	BUSY
-5	INTERRUPTED
-6	DENIED
-7	RETRY

Status code	Description
-8	ABORTED

Table 10.3: FFA_ERROR instances and conduits

Config No.	FF-A instance	Valid conduits
1	Secure and Non-secure physical	SMC, ERET
2	Non-secure virtual	SMC, HVC, ERET
3	Secure virtual	SMC, ERET

Table 10.4: FFA_ERROR function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000060.
uint32 Target information	w1	• Information to identify target SP/VM. – Valid only when SMC conduit is used at the Non-secure virtual FF-A instance. MBZ otherwise. – Bits[31:16]: ID of SP/VM. – Bits[15:0]: ID of vCPU of SP/VM to deliver error to.
int32 Error code	w2	• FF-A function specific error code. See function definition for applicable error codes .
Other Parameter registers	w3-w7 x3-x7	• Reserved (MBZ).

10.3 FFA_SUCCESS

Description

- Returns results on successful completion of a previous invocation of an FF-A function.
- Valid FF-A instances and conduits are listed in Table 10.6.
- Syntax of this function is described in Table 10.7.
- Figure 10.2 illustrates example usage of this function with the following assumptions.
  - Component A makes an valid request to Component B through an FF-A function described in this specification.
  - Component B uses the FFA_SUCCESS function to return the results to Component A.
  - The FF-A function used by component A can be invoked through the SMC and ERET conduits.
  - Both components could be interacting at any FF-A instance support by the FF-A function. The two possible scenarios have been considered.
    - Component A is at a lower EL than component B at the FF-A instance.
    - Component A is at a higher EL than component B at the FF-A instance.

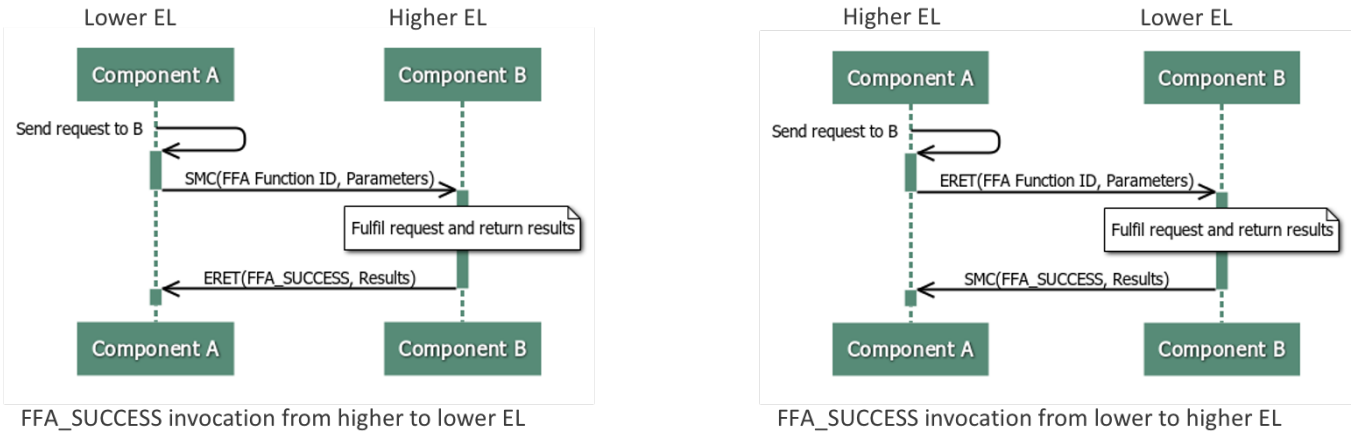

Figure 10.2: Example usage of FFA_SUCCESS

Table 10.6: FFA_SUCCESS instances and conduits

Config No.	FF-A instance	Valid conduits
1	Secure and Non-secure physical	SMC, ERET
2	Non-secure virtual FF-A	SMC, HVC, ERET
3	Secure virtual FF-A	SMC, ERET

Table 10.7: FFA_SUCCESS function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000061. • 0xC4000061.
uint32 Target information	w1	• Information to identify target SP/VM. – Valid only when SMC conduit is used at the Non-secure virtual FF-A instance. MBZ otherwise. – Bits[31:16]: ID of SP/VM. – Bits[15:0]: ID of vCPU of SP/VM to deliver results to.
uint32/uint64 Result registers	w2-w7 x2-x7	• FF-A function specific return results. See function definition for result encoding. MBZ if not explicitly specified.

10.4 FFA_INTERRUPT

Description

- Returns control from the caller to the callee in response to an interrupt that must be:
  - Either handled by the callee.
  - Or handled by another FF-A component reachable only through the callee.
- Valid FF-A instances and conduits are listed in Table 10.9.
- Syntax of this function is described in Table 10.10.
- Example usage of this interface is illustrated in Figure 6.1.

Table 10.9: FFA_INTERRUPT instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure physical	ERET
2	Secure and Non-secure virtual	ERET
3	Secure physical	SMC

Table 10.10: FFA_INTERRUPT function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000062.
uint32 Endpoint/vCPU IDs	w1	• Endpoint and vCPU IDs of the caller. Only valid at a physical FF-A instance. Else MBZ – Bits[31:16]: Endpoint ID. – Bits[15:0]: vCPU ID.
uint32 Interrupt ID	w2	• Interrupt ID. Only valid at the Secure virtual FF-A instance.
Other parameter registers	w3-w7 x3-x7	• Reserved (MBZ).

Chapter 11
Setup and discovery interfaces

11.1 FFA_VERSION

Description

- Returns version of the Firmware Framework implementation at an FF-A instance as described in 11.1.1 *Overview*.
- Valid FF-A instances and conduits are listed in Table 11.2.
- Syntax of this function is described in Table 11.3.
- Encoding of a version number in return parameters is described in Table 11.4.
- Encoding of error codes in return parameters is described in Table 11.5.

Table 11.2: FFA_VERSION instances and conduits

Config No.	FF-A instance	Valid conduits
1	Secure and Non-secure physical	SMC
2	Secure and Non-secure virtual	SMC, HVC, SVC

Table 11.3: FFA_VERSION function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000063.
uint32 Input version number	w1	• Version number specified by the caller as follows. – Bit[31]: Must be 0. – Bit[30:16] Major Version number. – Bit[15:0] Minor Version number.
Other Parameter registers	w2-w7 x2-x7	• Reserved (MBZ).

Table 11.4: Encoding of a version number

Parameter	Register	Value
int32 Output version number	w0	• On a successful return, the format of the value is as follows. – Bit[31]: Must be 0. – Bit[30:16] Major Version: Must be 1 for this revision of FF-A. – Bit[15:0] Minor Version: Must be 0 for this revision of FF-A.

Parameter	Register	Value
Other Result registers	w1-w7 x1-x7	• Reserved (MBZ).

Table 11.5: Encoding of error codes

Parameter	Register	Value
int32 Error code	w0	• NOT_SUPPORTED: A Firmware Framework implementation does not exist at this FF-A instance.

11.1.1 Overview

The version number of a Firmware Framework implementation is a 31-bit unsigned integer, with the upper 15 bits denoting the major revision, and the lower 16 bits denoting the minor revision.

If this function returns a valid version number:

- All the functions that are described in this specification must be implemented, unless it is explicitly stated that a function is optional.
- A partition manager could implement an optional interface and make it available to a subset of endpoints it manages.

The following rules apply to the version numbering.

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

In an invocation of this function, the compatibility of the version number *(x.y)* of the caller with the version number *(a.b)* of the callee can also be as follows.

1. If x != a, then the versions are incompatible.
   - The caller cannot inter-operate with the callee.
2. If $x == a$ and $y > b$, then the versions are incompatible.
   - The caller can inter-operate with the callee only if it downgrades its minor revision such that $y <= b$.
3. If $x == a$ and $y <= b$, then the versions are compatible.

A version number *(x.y)* is less than a version number *(a.b)* if one of the following conditions is true.

- $x < a$.
- $y < b$ if $x == a$.

11.1.2 Usage

This function enables the caller to determine if the callee implements the Firmware Framework and the version number of the implementation. The function must be invoked as follows.

- The caller must specify a version number in the *Input version number* parameter.
- The callee must take one of the following actions.
  - If it supports a Firmware Framework implementation that is compatible with the version number specified by the caller, it must return the version number of the implementation.

  - If it only supports a Firmware Framework implementation that is incompatible with and at a greater version number than specified by the caller, it must either return the version number of this implementation or the NOT_SUPPORTED error code.
  - If it supports a Firmware Framework implementation that is incompatible with and at a lesser version number than specified by the caller, it must return the highest version number of this implementation.
  - If it does not support any version of the Firmware Framework, it must return the NOT_SUPPORTED error code.
- The caller must use the preceding compatibility rules to determine if it can inter-operate with the version number returned by the callee.

Each FF-A instance must support this call and return its version number. For this revision of FF-A, the major version is 1 and the minor version is 1.

This interface returns a version number of the Framework at the FF-A instance where it is invoked. It is possible that version numbers of the Framework at different FF-A instances differ. These versions must be supported in accordance with the preceding major and minor version number compatibility rules.

11.1.3 SPM usage

In SPM configurations where the SPMD and SPMC reside in separate Exception levels (see Table 2.1 & Table 2.2), the versions of these two components could differ. The following constraints must be met to avoid a version mismatch.

- The SPMC must specify the version that it implements to the SPMD through an IMPLEMENTATION DEFINED mechanism.
- The SPMD must compare the version specified by the SPMC with the version it implements.
  - If the versions are not compatible as per the preceding compatibility rules, the SPMD must not initialize the SPMC.
  - If the versions are compatible, the SPMD must report the lowest compatible version in response to an invocation of FFA_VERSION at either physical FF-A instance.

11.2 FFA_FEATURES

Description

- This interface is used by an FF-A component at the lower EL at an FF-A instance to query:
  - The presence, properties and implementation of optional features of an FF-A interface.
  - The presence and properties of a feature supported by the Framework and not specific to an FF-A interface.
- This interface can be invoked at the FF-A instances through the conduits listed in Table 11.7.
- Syntax of this function is described in Table 11.8.
- If the FF-A interface or feature that was queried is implemented, the callee completes this call with an invocation of the *FFA_SUCCESS* interface as described in Table 11.9.
- If the FF-A interface or feature that was queried is not implemented or invalid, the callee completes this call with an invocation of the *FFA_ERROR* interface with the *NOT_SUPPORTED* error code.

Table 11.7: FFA_FEATURES instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure physical	SMC
2	Secure physical	ERET, SMC
3	Secure and Non-secure virtual	SMC, HVC, SVC

Table 11.8: FFA_FEATURES function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000064.
uint32 FF-A function ID or Feature ID	w1	• Bit[31] = b'1: Function ID of the FF-A interface whose implementation must be queried. – If an interface defines both SMC32 and SMC64 FIDs, then either FID could be used. • Bit[31] = b'0: ID of a feature supported by the Framework at this FF-A instance. IDs of supported features are listed in Table 11.10. – Bit[30:8]: Reserved (MBZ). – Bit[7:0]: Feature ID.
Other Parameter registers	w2-w7 x2-x7	• Reserved (MBZ).

Table 11.9: FFA_SUCCESS encoding

Parameter	Register	Value
uint32 Interface properties	w2-w3	• Used to encode any optional features implemented or any properties exported by the queried interface or feature. – FF-A interfaces that use these parameters and the encodings of their properties are listed in Table 11.11. – Feature IDs and encodings of their properties are listed in Table 11.10. • MBZ if no optional features are implemented or no implementation details are exported by the queried interface.
Other Result registers	w4-w7 x4-x7	• Reserved (MBZ).

Table 11.10: Feature IDs and properties table

FF-A Feature Name	FF-A Feature ID	Encoding of feature in return parameters
Notification pending interrupt	0x1	• *w2* : Interrupt ID. .
Schedule Receiver interrupt	0x2	• *w2* : Interrupt ID. .
Managed exit interrupt	0x3	• *w2* : Interrupt ID.

Table 11.11: Encoding of interface properties parameters

FF-A Function ID	Return parameters
FFA_RXTX_MAP	• *w2* : Bits[31:2] are reserved (MBZ) . – Bit[1:0]: Minimum buffer size and alignment boundary (see 4.2.2.3 *Buffer attributes*). ∗ b'00: 4K. ∗ b'01: 64K. ∗ b'10: 16K. ∗ b'11: Reserved. • *w3/x3* : Reserved (MBZ).

FF-A Function ID	Return parameters
FFA_MEM_DONATE	• *w2* : Bits[31:1] are reserved (MBZ) . – Bit[0]: Dynamically allocated buffer support. See 16.2.1 *Transmission of transaction descriptor in dynamically allocated buffers*. ∗ b'0: Partition manager does not support transmission of a memory transaction descriptor in a buffer dynamically allocated by the endpoint. ∗ b'1: Partition manager supports transmission of a memory transaction descriptor in a buffer dynamically allocated by the endpoint. • *w3/x3* : Reserved (MBZ).
FFA_MEM_LEND	• Same as FFA_MEM_DONATE.
FFA_MEM_SHARE	• Same as FFA_MEM_DONATE.
FFA_MEM_RETRIEVE_REQ	• *w2* : Bits[31:2] are reserved (MBZ) . – Bit[0]: Dynamically allocated buffer support. See 16.2.1 *Transmission of transaction descriptor in dynamically allocated buffers*. ∗ b'0: Partition manager does not support transmission of a memory transaction descriptor in a buffer dynamically allocated by the endpoint. ∗ b'1: Partition manager supports transmission of a memory transaction descriptor in a buffer dynamically allocated by the endpoint. – Bit[1]: Reserved for IMPLEMENTATION DEFINED usage. • *w3* : Outstanding retrievals field. – Bit[31:8]: Reserved MBZ. – Bit[7:0]: Number of times a Receiver is allowed to retrieve a memory region before relinquishing it. The value specified is interpreted as *((1U << (value + 1)) - 1*.

11.3 FFA_RX_ACQUIRE

Description

- Acquire ownership of a RX buffer before writing a message to it (see 4.2.2.4.3 *Management of buffer ownership between Hypervisor and SPMC*).
- Valid FF-A instances and conduits are listed in Table 11.13.
- Syntax of this function is described in Table 11.14.
- Returns FFA_SUCCESS without any further parameters on successful completion.
- Encoding of error code in the FFA_ERROR function is described in Table 11.15.

Table 11.13: FFA_RX_ACQUIRE instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure Physical	SMC
2	Secure Physical	ERET

Table 11.14: FFA_RX_ACQUIRE function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000084.
Other Parameter registers	w1-w7 x1-x7	• Reserved (MBZ).

Table 11.15: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• DENIED: Callee cannot relinquish ownership of the RX buffer. • NOT_SUPPORTED: This function is not implemented at this FF-A instance.

11.4 FFA_RX_RELEASE

Description

- Relinquish ownership of a RX buffer after reading a message from it (see 4.2.2.4 *Buffer synchronization*).
- Valid FF-A instances and conduits are listed in Table 11.17.
- Syntax of this function is described in Table 11.18.
- Returns FFA_SUCCESS without any further parameters on successful completion.
- Encoding of error code in the FFA_ERROR function is described in Table 11.19.

Table 11.17: FFA_RX_RELEASE instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure Physical	SMC
2	Secure Physical	ERET
3	Secure virtual	SMC, HVC, SVC
4	Non-secure virtual	SMC, HVC, SVC, ERET

Table 11.18: FFA_RX_RELEASE function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000065.
Other Parameter registers	w1-w7 x1-x7	• Reserved (MBZ).

Table 11.19: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• DENIED: Caller did not have ownership of the RX buffer. • NOT_SUPPORTED: This function is not implemented at this FF-A instance.

11.5 FFA_RXTX_MAP

Description

- Maps the RX/TX buffer pair in the translation regime of the callee on behalf of an endpoint or Hypervisor.
  - A SP describes the VA or IPA contiguous pages allocated for each buffer in the pair to the SPM.
  - A VM describes the VA or IPA contiguous pages allocated for each buffer in the pair to the Hypervisor.
  - Hypervisor or OS Kernel describe the physically contiguous pages allocated for each buffer in the pair to the SPM.
  - Hypervisor forwards the description of pages allocated for each buffer in the pair by a VM to the SPM.
    - Description of buffer pair is populated in the TX buffer of the Hypervisor as described in Table 11.24.
  - Both Hypervisor and SPM must ensure the caller has exclusive access and ownership of the RX/TX buffer memory regions.
- Valid FF-A instances and conduits are listed in Table 11.21.
- Syntax of this function is described in Table 11.22.
- Returns FFA_SUCCESS without any further parameters on successful completion.
- Encoding of error code in the FFA_ERROR function is described in Table 11.23.

Table 11.21: FFA_RXTX_MAP instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure physical	SMC
2	Secure physical	ERET
3	Virtual	SMC, HVC, SVC

Table 11.22: FFA_RXTX_MAP function syntax

Parameter	Register	Value
uint32 Function ID	w0/x0	• 0x84000066. • 0xC4000066.
uint32/uint64 TX address	w1/x1	• Base address of the TX buffer if invoked by an endpoint or Hypervisor to register its buffer pair. – Address is a IPA or VA at the virtual FF-A instance. – Address is a PA at the physical FF-A instance. • MBZ if Hypervisor is forwarding this call on behalf of an endpoint. – Description of RX/TX buffer and identity of endpoint is specified in the TX buffer of the Hypervisor.

Parameter	Register	Value
uint32/uint64 RX address	w2/x2	• Base address of the RX buffer. – Address is a IPA or VA at the virtual FF-A instance. – Address is a PA at the physical FF-A instance. • MBZ if Hypervisor is forwarding this call on behalf of an endpoint. – Description of RX/TX buffer and identity of endpoint is specified in the TX buffer of the Hypervisor.
uint32 RX/TX page count	w3/x3	• Bit[31:6]: Reserved (MBZ). • Bit[5:0]: Number of contiguous 4K pages allocated for each buffer.
Other Parameter registers	w4-w7 x4-x7	• Reserved (MBZ).

Table 11.23: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: One or more fields in input parameters is incorrectly encoded. • NO_MEMORY: – Not enough memory to map the buffers in the translation regime of the callee. – Not enough memory in TX buffer of Hypervisor to describe caller buffer pair to SPM. • DENIED: Buffer pair already registered for the FF-A component with specified ID. • NOT_SUPPORTED: This function is not implemented at this FF-A instance.

Table 11.24: Endpoint RX/TX descriptor

Field	Byte length	Byte offset	Description
Endpoint ID	2	0	• ID of endpoint that allocated the RX/TX buffer.
Reserved	2	2	• MBZ.
RX address range count	4	4	• Count of address ranges specified using constituent memory descriptors for the RX buffer.

Field	Byte length	Byte offset	Description
TX address range count	4	8	• Count of address ranges specified using constituent memory descriptors for the TX buffer.
RX address range array	–	12	• Array of address ranges allocated for the RX buffer that the callee must map in its translation regime. See Table 8.14 for how the address ranges are encoded.
TX address range array	–	–	• Array of address ranges allocated for the TX buffer that the callee must map in its translation regime. See Table 8.14 for how the address ranges are encoded.

11.6 FFA_RXTX_UNMAP

Description

- Unmaps the RX/TX buffer pair of an endpoint or Hypervisor from the translation regime of the callee.
  - A SP invokes this interface to unmap its buffer pair from the translation regime of the SPM.
  - A VM invokes this interface to unmap its buffer pair from the translation regime of the Hypervisor.
  - Hypervisor or OS Kernel invoke this interface to unmap their buffer pair from the translation regime of the SPM.
  - Hypervisor forwards an invocation of this interface by a VM to the SPM.
    - Identity of VM is specified in *w1*.
- Valid FF-A instances and conduits are listed in Table 11.26.
- Syntax of this function is described in Table 11.27.
- Returns FFA_SUCCESS without any further parameters on successful completion.
- Encoding of error code in the FFA_ERROR function is described in Table 11.28.

Table 11.26: FFA_RXTX_UNMAP instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure physical	SMC
2	Secure physical	ERET
3	Virtual	SMC, HVC, SVC

Table 11.27: FFA_RXTX_UNMAP function syntax

Parameter	Register	Value
uint32 Function ID	w0/x0	• 0x84000067.
uint32 ID	w1	• ID of FF-A component that allocated the RX/TX buffer. – Bit[31:16]: ID. – Bit[15:0]: Reserved MBZ.
Other Parameter registers	w2-w7 x2-x7	• Reserved (MBZ).

Table 11.28: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: There is no buffer pair registered on behalf of the caller. • NOT_SUPPORTED: This function is not implemented at this FF-A instance.

11.7 FFA_PARTITION_INFO_GET

Description

- Returns information about FF-A components implemented in the system as described in 11.7.1 *Overview*.
- Valid FF-A instances and conduits are listed in Table 11.30.
- Syntax of this function is described in Table 11.31.
- Encoding of result parameters in the FFA_SUCCESS function is described in Table 11.32.
- Encoding of error code in the FFA_ERROR function is described in Table 11.33.

Table 11.30: FFA_PARTITION_INFO_GET instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure physical	SMC
2	Secure physical	SMC, ERET
3	Non-secure virtual	SMC, HVC
4	Secure virtual	SMC, HVC, SVC

Table 11.31: FFA_PARTITION_INFO_GET function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000068.
uint128 UUID	w1-w4	• Specified as described in Section 5.3 of [4].
Other Parameter registers	w5-w7 x5-x7	• Reserved (MBZ).

Table 11.32: FFA_SUCCESS encoding

Parameter	Register	Value
uint32 Count	w2	• Count of partition information descriptors populated in RX buffer of caller (see Table 11.34).
Other Result registers	w3-w7 x3-x7	• Reserved (MBZ).

Table 11.33: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• BUSY: RX buffer of the caller is not free. • INVALID_PARAMETERS: Unrecognized UUID. • NO_MEMORY: Results cannot fit in RX buffer of the caller. • DENIED: Callee is not in a state to handle this request. • NOT_SUPPORTED: This function is not implemented at this FF-A instance.

11.7.1 Overview

FFA_PARTITION_INFO_GET is used by FF-A components to discover the ID (see 2.8 *FF-A component identification and discovery*) and other properties of partitions. This information is,

- Requested by specifying a UUID as an input parameter as described in Table 11.31.
- Encoded in a *partition information descriptor* as described in Table 11.34.
- Returned in the RX buffer of the caller as an array of one or more partition information descriptors. The count of descriptors is returned in *w2* (see Table 11.32).

Table 11.34: Partition information descriptor

Field	Byte length	Byte offset	Description
Partition ID	2	0	• 16-bit ID of the partition, stream or auxiliary endpoint.
Execution context count or Proxy partition ID	2	2	• Number of execution contexts implemented by this partition (also see 2.9 *Execution context*) if *Bit[5:4] = b'00* in the *Partition properties* field. • ID of the proxy endpoint for a dependent peripheral device (see 8.2.1 *Stream endpoint* if *Bit[5:4] = b'10* in the *Partition properties* field. • Reserved and MBZ for all other encodings of the *Partition properties* field.

Field	Byte length	Byte offset	Description
Partition properties	4	4	• Flags to determine partition properties. • Bit[3:0] has the following encoding if *Bit[5:4] = b'00*. It is Reserved and MBZ otherwise. – Bit[0] has the following encoding: ∗ b'0: Does not support receipt of direct requests ∗ b'1: Supports receipt of direct requests. Count of execution contexts must be either 1 or equal to the number of PEs in the system (also see 4.4 *Direct messaging usage*). – bit[1] has the following encoding: ∗ b'0: Cannot send direct requests. ∗ b'1: Can send direct requests. – bit[2] has the following encoding: ∗ b'0: Cannot send and receive indirect messages. MBZ for an SP. ∗ b'1: Can send and receive indirect messages. – bit[3] has the following encoding: ∗ b'0: Does not support receipt of notifications. ∗ b'1: Supports receipt of notifications. • bit[5:4] has the following encoding: – b'00: Partition ID is a PE endpoint ID. – b'01: Partition ID is a SEPID for an independent peripheral device. – b'10: Partition ID is a SEPID for an dependent peripheral device. – b'11: Partition ID is an auxiliary ID 3.2.1 *Manifest for isolated partitions*. • bit[31:6]: Reserved (MBZ).

11.7.2 Usage

The result of an invocation of this ABI depends upon the version of the Framework, specified UUID and the FF-A instance where the ABI is invoked. This is described below.

- In both v1.0 and v1.1 of the Framework,
  - If the Nil UUID is specified at the Non-secure virtual FF-A instance, information for all partitions (including the caller) in the system in either security state is returned.
  - If the Nil UUID is specified at the Non-secure physical FF-A instance, information for all partitions in the Secure state is returned.
  - If the Nil UUID is specified at the Secure virtual FF-A instance, information for all partitions (including the caller) in the Secure state is returned. This ABI cannot be used to discover the IDs and properties of NS-Endpoints.
  - If the Nil UUID is specified at the Secure physical FF-A instance, FFA_ERROR is returned with NOT_SUPPORTED as the error code.
  - If a non-Nil UUID is specified at a Non-secure FF-A instance, information for all partitions in the system, corresponding to the UUID, in either security state is returned.
  - If a non-Nil UUID is specified at a Secure virtual FF-A instance, information for all partitions in the Secure state in the system, corresponding to the UUID is returned.

- If a non-Nil UUID is specified at a Secure physical FF-A instance, FFA_ERROR is returned with NOT_SUPPORTED as the error code. This ABI cannot be used to discover the IDs and properties of a NS-Endpoint.

The caller transfers ownership of the RX buffer back to the producer through a mechanism described in 4.2.2.4.2 *Transfer of buffer ownership*.

11.8 FFA_ID_GET

Description

- Returns 16-bit ID of calling FF-A component.
  - ID value *0* must be returned at the Non-secure physical FF-A instance (see 2.8 *FF-A component identification and discovery*).
- Valid FF-A instances and conduits are listed in Table 11.36.
- Syntax of this function is described in Table 11.37.
- Encoding of result parameters in the FFA_SUCCESS function is described in Table 11.38.
- Encoding of error code in the FFA_ERROR function is described in Table 11.39.

Table 11.36: FFA_ID_GET instances and conduits

Config No.	FF-A instance	Valid conduits
1	Physical FF-A instance	SMC
2	Virtual FF-A instance	SMC, HVC, SVC

Table 11.37: FFA_ID_GET function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000069.
Other Parameter registers	w1-w7 x1-x7	• Reserved (MBZ).

Table 11.38: FFA_SUCCESS encoding

Parameter	Register	Value
uint32 ID	w2	• ID of the caller. – Bit[31:16]: Reserved (MBZ). – Bit[15:0]: ID.
Other Result registers	w3-w7 x3-x7	• Reserved (MBZ).

Table 11.39: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• NOT_SUPPORTED: This function is not implemented at this FF-A instance.

11.9 FFA_SPM_ID_GET

Description

- Returns the 16-bit ID of the SPMC or SPMD depending upon the FF-A instance where this function is invoked. See 11.9.1 *Overview* for details.
- Valid FF-A instances and conduits are listed in Table 11.41.
- Syntax of this function is described in Table 11.42.
- Encoding of result parameters in the FFA_SUCCESS function is described in Table 11.43.
- Encoding of error code in the FFA_ERROR function is described in Table 11.44.

Table 11.41: FFA_SPM_ID_GET instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure physical	SMC
2	Secure physical	SMC, ERET
3	Non-secure virtual	SMC, HVC
4	Secure virtual	SMC, HVC, SVC

Table 11.42: FFA_SPM_ID_GET function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000085.
Other Parameter registers	w1-w7 x1-x7	• Reserved (MBZ).

Table 11.43: FFA_SUCCESS encoding

Parameter	Register	Value
uint32 ID	w2	• ID of the SPMD or SPMC as described in 11.9.2 *Usage*. – Bit[31:16]: Reserved (MBZ). – Bit[15:0]: ID.
Other Result registers	w3-w7 x3-x7	• Reserved (MBZ).

Table 11.44: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• NOT_SUPPORTED: This function is not implemented at this FF-A instance.

11.9.1 Overview

v1.1 of the Framework mandates that the SPMC and SPMD components must be assigned unique IMPLEMENTATION DEFINED 16-bit IDs (see 2.8 *FF-A component identification and discovery*).

The FFA_SPM_ID_GET ABI enables,

- Endpoints and the Hypervisor to discover the ID of the SPMC.
- The SPMC to discover the ID of the SPMD.

The ID returned depends upon the FF-A instance where the ABI is invoked. This is described in 11.9.2 *Usage*.

The Framework assumes that no FF-A component apart from the SPMC needs to discover and use the SPMD ID.

11.9.2 Usage

- An invocation of this ABI at a Non-secure virtual or physical FF-A instance returns the ID of the SPMC.
  - If the SPMC and SPMD are implemented at different exception levels (see 2.2 *SPM architecture*), the SPMD must forward the ABI invocation to the SPMC through the ERET conduit at the Secure physical FF-A instance.
- An invocation of this ABI at a Secure virtual FF-A instance returns the ID of the SPMC. This is irrespective of whether the SPMC and SPMD are implemented in the same or separate exception levels.
- An invocation of this ABI at the Secure physical FF-A instance returns the ID of the SPMD.

Chapter 12
CPU cycle management interfaces

12.1 FFA_MSG_WAIT

Description

- An invocation of this ABI at a virtual FF-A instance with the SMC, HVC or SVC conduits transitions the state of the calling execution context from *running* to *waiting* in the following runtime models.
  - 5.5 *Runtime model for SP initialization*.
  - 5.4 *Runtime model for Secure interrupt handling*.
  - 5.2 *Runtime model for FFA_RUN*.
- An invocation of this ABI at a physical FF-A instance with a valid conduit is used to inform the scheduler of the calling execution context about this state transition.
- An invocation of this ABI at the Non-secure virtual FF-A instance with the ERET conduit is used by the Hypervisor to inform the primary or a secondary scheduler about this state transition.
  - An optional 64-bit timeout could be specified by the Hypervisor if the calling execution context is a VM vCPU.
  - The scheduler runs the VM vCPU after the timeout expires.
  - Syntax of this function in this scenario is described in Table 12.5.
- An invocation of this ABI at a virtual FF-A instance with the SMC, HVC or SVC conduits completes when the calling execution context is allocated CPU cycles as described in Chapter 5 *Partition runtime models*.
- An invocation of this ABI at the Secure physical FF-A instance completes with an invocation of any FF-A ABI.
- Valid FF-A instances and conduits are listed in Table 12.2.
- Syntax of this function is described in Table 12.3.
- Encoding of error codes in the FFA_ERROR function is described in Table 12.4.

Table 12.2: FFA_MSG_WAIT instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure physical	ERET
2	Secure physical	SMC
3	Non-secure virtual	SMC, HVC, ERET
4	Secure virtual	SMC, HVC, SVC

Table 12.3: FFA_MSG_WAIT function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x8400006B.
Other Parameter registers	w1-w7 x1-x7	• Reserved (MBZ).

Table 12.4: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: Unrecognized endpoint or vCPU ID specified at Non-secure physical or virtual FF-A instance. • DENIED: Callee is not in a state to handle this request. • NOT_SUPPORTED: This function is not implemented at this FF-A instance.

Table 12.5: FFA_MSG_WAIT function syntax with the ERET conduit at NS virtual FF-A instance

Parameter	Register	Value
uint32 Function ID	w0	• 0x8400006B.
uint32 Endpoint/vCPU IDs	w1	• Endpoint and vCPU IDs of the caller. – Bit[31:16]: Endpoint ID. – Bit[15:0]: vCPU ID.
uint32 TimeoutLo	w2	• Bits[31:0] of an interval measured in nanoseconds after which vCPU of the endpoint specified in *w1* must be run. • This parameter MBZ if the caller does not specify a timeout.
uint32 TimeoutHi	w3	• Bits[63:32] of an interval measured in nanoseconds after which vCPU of the endpoint specified in *w1* must be run. • This parameter MBZ if the caller does not specify a timeout.
Other Parameter registers	w4-w7 x4-x7	• Reserved (MBZ).

12.2 FFA_YIELD

Description

- An invocation of this ABI at a virtual FF-A instance with the SMC, HVC or SVC conduits transitions the state of the calling execution context from *running* to *blocked* in the following runtime models.
  - 5.2 *Runtime model for FFA_RUN*.
- An invocation of this ABI at a physical FF-A instance with an allowed conduit, is used to inform the scheduler of the calling execution context, about this state transition.
- An invocation of this ABI at the Non-secure virtual FF-A instance with the ERET conduit is used by the Hypervisor to inform the primary or a secondary scheduler about this state transition.
  - An optional 64-bit timeout could be specified by the Hypervisor if the calling execution context is a VM vCPU.
  - The scheduler runs the VM vCPU after the timeout expires.
  - Syntax of this function in this scenario is described in Table 12.10.
- An invocation of this ABI at a virtual FF-A instance with the SMC, HVC or SVC conduits completes when the calling execution context is unblocked through the following transitions as described in 5.2 *Runtime model for FFA_RUN*.
  - eret(FFA_RUN).
  - eret(FFA_INTERRUPT).
- An invocation of this ABI at the Secure physical FF-A instance completes with an invocation of any FF-A ABI.
- Valid FF-A instances and conduits are listed in Table 12.7.
- Syntax of this function is described in Table 12.8.
- Encoding of error codes in the FFA_ERROR function is described in Table 12.9.

Table 12.7: FFA_YIELD instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure physical	ERET
2	Secure physical	SMC
3	Non-secure virtual	SMC, HVC, ERET
4	Secure virtual	SMC, HVC, SVC

Table 12.8: FFA_YIELD function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x8400006C.
Other Parameter registers	w1-w7 x1-x7	• Reserved (MBZ).

Table 12.9: FFA_ERROR encoding

Parameter	Register	Value
int32 Status	w2	• INVALID_PARAMETERS: Unrecognized endpoint or vCPU ID. Only valid with the ERET conduit. • DENIED: Callee is not in a state to handle this request. • NOT_SUPPORTED: This function is not implemented at this FF-A instance.

Table 12.10: FFA_YIELD function syntax with the ERET conduit at NS virtual FF-A instance

Parameter	Register	Value
uint32 Function ID	w0	• 0x8400006C.
uint32 Endpoint/vCPU IDs	w1	• Endpoint and vCPU IDs of the caller. – Bit[31:16]: Endpoint ID. – Bit[15:0]: vCPU ID.
uint32 TimeoutLo	w2	• Bits[31:0] of an interval measured in nanoseconds after which vCPU of the endpoint specified in *w1* must be run. • This parameter MBZ if the caller does not specify a timeout.
uint32 TimeoutHi	w3	• Bits[63:32] of an interval measured in nanoseconds after which vCPU of the endpoint specified in *w1* must be run. • This parameter MBZ if the caller does not specify a timeout.
Other Parameter registers	w4-w7 x4-x7	• Reserved (MBZ).

12.3 FFA_RUN

Description

- This ABI is used by a scheduler (see 2.11 *Primary scheduler*) to allocate CPU cycles to a target endpoint execution context specified in the *Target information* parameter.
- An invocation of this ABI at a virtual FF-A instance with the SMC, HVC or SVC conduits transitions the state of the calling execution context from *running* to *blocked* in the following runtime models.
  - 5.2 *Runtime model for FFA_RUN*.
  - 5.3 *Runtime model for FFA_MSG_SEND_DIRECT_REQ*.
- An invocation of this ABI at a virtual FF-A instance with the ERET conduit results in a state transition of the target endpoint execution context as described below.
  - If the endpoint execution context is in the *waiting* state, it transitions to the *running* state with the following runtime model.
    - 5.2 *Runtime model for FFA_RUN*.
  - If the endpoint execution context is in the *blocked* state, it transitions to the *running* state in the following runtime models.
    - 5.2 *Runtime model for FFA_RUN*.
    - 5.3 *Runtime model for FFA_MSG_SEND_DIRECT_REQ*.
- If the target endpoint execution context is in the *preempted* state, it transitions to *running* state in response to an invocation of this ABI. The partition manager of the execution context changes the state through the *eret()* transition. This transition is applicable if the execution context is using the following runtime models.
  - 5.2 *Runtime model for FFA_RUN*.
  - 5.3 *Runtime model for FFA_MSG_SEND_DIRECT_REQ*.
- An invocation of this ABI at a physical FF-A instance with a valid conduit, is used to request the partition manager of the target execution context, to perform the applicable state transition listed above.
- An invocation of this ABI at a virtual FF-A instance with the SMC, HVC and SVC conduits and, at the Non-secure physical FF-A instance with the SMC conduit, completes and transitions the state of calling execution context from *blocked* to *running* through the following transitions.
  - eret(FFA_INTERRUPT).
  - eret(FFA_MSG_WAIT).
  - eret(FFA_YIELD).
  - eret(FFA_MSG_SEND_DIRECT_RESP).
- An invocation of this ABI at the Secure physical FF-A instance with the ERET conduit completes with invocations of the following ABIs.
  - smc(FFA_INTERRUPT).
  - smc(FFA_MSG_WAIT).
  - smc(FFA_YIELD).
  - smc(FFA_MSG_SEND_DIRECT_RESP).
- Valid FF-A instances and conduits are listed in Table 12.12.
- Syntax of this function is described in Table 12.13.
- Encoding of error code in the FFA_ERROR function is described in Table 12.14.

Table 12.12: FFA_RUN instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure physical	SMC
2	Secure physical	ERET

Config No.	FF-A instance	Valid conduits
3	Non-secure virtual	SMC, HVC, ERET
4	Secure virtual	SMC, HVC, SVC, ERET

Table 12.13: FFA_RUN function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x8400006D.
uint32 Target information	w1	• Information to identify target SP/VM. – Bits[31:16]: ID of SP/VM. – Bits[15:0]: ID of vCPU of SP/VM to run.
Other Parameter registers	w2-w7 x2-x7	• Reserved (MBZ).

Table 12.14: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: Unrecognized endpoint or vCPU ID. • NOT_SUPPORTED: This function is not implemented at this FF-A instance. • DENIED: Callee is not in a state to handle this request. • BUSY: vCPU is busy and caller must retry later. • ABORTED: vCPU or VM ran into an unexpected error and has aborted.

12.4 FFA_NORMAL_WORLD_RESUME

Description

- Request SPMD to resume execution of Normal world on current PE. See 12.4.1 *Overview* for details.
- Valid FF-A instances and conduits are listed in Table 12.16.
- Syntax of this function is described in Table 12.17.
- Successful completion of this function is indicated through the invocation of any FF-A function.
- Encoding of error code in the FFA_ERROR function is described in Table 12.18.

Table 12.16: FFA_NORMAL_WORLD_RESUME instances and conduits

Config No.	FF-A instance	Valid conduits
1	Secure physical	SMC

Table 12.17: FFA_NORMAL_WORLD_RESUME function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x8400007C.
Other Parameter registers	w1-w7 x1-x7	• Reserved (MBZ).

Table 12.18: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• DENIED: Callee is not in a state to handle this request. • NOT_SUPPORTED: This function is not implemented at this FF-A instance.

12.4.1 Overview

Execution in Normal world could be preempted in response to an exception for example, a Secure physical interrupt. As per the Armv8-A architecture, the exception will be delivered to EL3 in the AArch64 Execution state or Monitor mode in the AArch32 Execution state. The exception could be handled in the Secure state at a lower Exception level than EL3 or Monitor mode.

This function must be used by the SPMC in S-EL2 (see 2.2.1 *SPM architecture with Secure EL2*), S-EL1 (see 2.2.2.1 *S-EL1 SPM core component*) or Secure Supervisor mode (see 2.2.2.2 *Secure Supervisor mode SPM core component*) to request the SPMD to resume Normal world execution once the exception has been handled.

The SPMD must ensure that the Normal world execution is resumed with exactly the same PE state that was saved when it was preempted.

The SPMD must return *DENIED* if this function is invoked at the Secure physical FF-A instance and the Normal world execution was not preempted.

The partition manager must return *NOT_SUPPORTED* if this function is invoked at any other FF-A instance.

An invocation of this function at the Secure physical FF-A instance could be completed through a valid invocation of any FF-A function through the ERET conduit.

Chapter 13
Messaging interfaces

13.1 FFA_MSG_SEND2

Overview

- This ABI is invoked at a virtual FF-A instance with the SMC, HVC or SVC conduits to,
  - Transmit a partition message from the TX buffer of the caller endpoint to the RX buffer of the Receiver endpoint as described in 4.2.2.1.1 *Transmission of partition messages*.
  - Notify the Receiver's scheduler that the Receiver endpoint must be run to process the partition message as described in 7.8.1 *RX buffer full notification*.
- An invocation of this ABI at a physical FF-A instance with a valid conduit is used to request the SPMC to transmit the message to a SP.
- A partition message is encoded as described in Table 4.2.
- Valid FF-A instances and conduits are listed in Table 13.2.
- Syntax of this function is described in Table 13.3.
- Returns FFA_SUCCESS without any further parameters on successful completion.
- Encoding of error code in the FFA_ERROR function is described in Table 13.4.

Table 13.2: FFA_MSG_SEND2 instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure physical	SMC
2	Secure physical	ERET
3	Non-secure virtual	SMC, HVC
4	Secure virtual	SMC, HVC, SVC

Table 13.3: FFA_MSG_SEND2 function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000086.
uint32 Flags	w1	• Message flags. – Must be ignored by callee when SVC conduit is used. – Bit[0]: Reserved. MBZ and ignored. – Bit[1]: Delay *Schedule Receiver* interrupt flag. Guidance in. 15.5.1 *Delay Schedule Receiver interrupt flag* applies to the FFA_MSG_SEND2 ABI. – Bit[31:2]: Reserved. MBZ and ignored.
Other Parameter registers	w2-w7 x2-x7	• Reserved (MBZ).

Table 13.4: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: A field in input parameters is incorrectly encoded. • BUSY: Receiver RX buffer is not free. • DENIED: – Callee is not in a state to handle this request. – Receiver endpoint does not support receipt of partition messages through indirect messaging. • NO_MEMORY: Insufficient memory to handle this request. • NOT_SUPPORTED: This function is not implemented at this FF-A instance.

13.2 FFA_MSG_SEND_DIRECT_REQ

Description

- Send a Partition or Framework message in parameter registers as a request to a target endpoint, run the endpoint and block until a response is available.
- Valid FF-A instances and conduits are listed in Table 13.6.
- Syntax of this function is described in Table 13.7.
- Successful completion of this function is indicated through an invocation of the following interfaces by the callee:
  - *FFA_MSG_SEND_DIRECT_RESP* to provide a response to the direct request.
  - *FFA_INTERRUPT* to indicate that the direct request was interrupted and must be resumed through the *FFA_RUN* interface.
  - *FFA_SUCCESS* to indicate completion of the direct request without a corresponding direct response. All other parameter registers MBZ.
- Encoding of error code in the FFA_ERROR function is described in Table 13.8.

Table 13.6: FFA_MSG_SEND_DIRECT_REQ instances and conduits

Config No.	FF-A instance	Valid conduits
1	Physical	SMC, ERET
2	Non-secure virtual	SMC, HVC, ERET
3	Secure virtual	SMC, HVC, SVC, ERET

Table 13.7: FFA_MSG_SEND_DIRECT_REQ function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x8400006F. • 0xC400006F.
uint32 Source/Destination IDs	w1	• Source and destination endpoint IDs. – Bit[31:16]: Source endpoint ID. – Bit[15:0]: Destination endpoint ID.

Parameter	Register	Value
uint32 Flags	w2	• Message flags. – Bit[31]: Message type. ∗ b'0: Message encoded in parameter registers is a partition message. ∗ b'1: Message encoded in parameter registers is a framework message. – Bit[30:8]: Reserved (MBZ). – Bit[7:0]: ∗ Reserved (MBZ) if bit[31] = b'0. ∗ Framework message type if bit[31] = b'1. · b'00000000: Message for a power management operation initiated by a PSCI function. See 16.3.4 *Power Management messages* and Table 16.30. · b'00000001: Message for a warm boot. See 16.3.4 *Power Management messages* and Table 16.31.
Other Parameter registers	w3-w7 x3-x7	• IMPLEMENTATION DEFINED values.

Table 13.8: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: Invalid endpoint ID or message flags. • DENIED: Callee is not in a state to handle this request. • NOT_SUPPORTED: This function is not implemented at this FF-A instance. • BUSY: Message target is busy. • ABORTED: Message target ran into an unexpected error and has aborted.

13.2.1 Component responsibilities for FFA_MSG_SEND_DIRECT_REQ

This section describes the common responsibilities that the participating FF-A components must fulfill during transmission of Partition and Framework messages between endpoints through the *FFA_MSG_SEND_DIRECT_REQ* interface. This interface is used in the scenarios listed in Table 4.7.

13.2.1.1 Sender responsibilities

13.2.1.1.1 Send from NS-Endpoint to S-Endpoint

1. Must write message payload to parameter registers.
2. Must specify Sender and Receiver endpoint IDs.
3. Must implement support for handling all error status codes that can be returned on completion of this interface.

4. See 13.2.1.2.2 *Relay from VM to S-Endpoint* & 13.2.1.3.3 *Relay from NS-Endpoint to S-Endpoint* for Relayer responsibilities in this message transmission.

13.2.1.1.2 Send from VM to VM

1. Same as Sender responsibilities while sending message from NS-Endpoint to S-Endpoint as listed in 13.2.1.1.1 *Send from NS-Endpoint to S-Endpoint*.
2. See 13.2.1.2.1 *Relay from VM to VM* for Relayer responsibilities in this message transmission.

13.2.1.1.3 Send from SP to SP

1. Same as Sender responsibilities while sending message from NS-Endpoint to S-Endpoint as listed in 13.2.1.1.1 *Send from NS-Endpoint to S-Endpoint*.
2. See 13.2.1.3.1 *Relay from SP to SP* for Relayer responsibilities in this message transmission.

13.2.1.1.4 Send from S-Endpoint to NS-Endpoint

1. Same as Sender responsibilities while sending message from NS-Endpoint to S-Endpoint as listed in 13.2.1.1.1 *Send from NS-Endpoint to S-Endpoint*.
2. See 13.2.1.3.2 *Relay from S-Endpoint to NS-Endpoint* for Relayer responsibilities in this message transmission.

13.2.1.2 Hypervisor responsibilities

13.2.1.2.1 Relay from VM to VM

1. Must validate that the Sender is allowed to send direct messages. Invoke *FFA_ERROR* with *NOT_SUPPORTED* as status if this is not the case.
2. Must validate Sender and Receiver endpoint IDs. Invoke *FFA_ERROR* with *INVALID PARAMETER* as status if either is invalid.
3. Must check that,
    1. Reserved parameter registers are 0 for a Partition and Framework message.
    2. Parameters are correctly encoded for a Framework message.

    FFA_ERROR with *INVALID PARAMETER* must be invoked as status if either is invalid.
4. Must ensure that target endpoint supports receipt of direct messages. Invoke *FFA_ERROR* with *DENIED* as status if this is not the case.
5. Must determine availability of an idle target endpoint execution context on this PE. Invoke *FFA_ERROR* with *BUSY* as status if not available.
6. Must ensure invocation of this interface by the Sender is completed only in response to an invocation of the *FFA_MSG_SEND_DIRECT_RESP* interface.
7. Must copy parameter registers from *Sender* execution context to *Receiver* execution context.
8. Must complete the invocation of the interface the Receiver had used to enter the idle state with an invocation of *FFA_MSG_SEND_DIRECT_REQ* through the *ERET* conduit.

13.2.1.2.2 Relay from VM to S-Endpoint

1. Same as 1-3 in 13.2.1.2.1 *Relay from VM to VM*.
2. Invoke *FFA_MSG_SEND_DIRECT_REQ* at physical FF-A instance through the *SMC* conduit with the same parameters as specified by the *Sender*. See 13.2.1.3.3 *Relay from NS-Endpoint to S-Endpoint* for responsibilities of the SPM as the Relayer.

13.2.1.2.3 Relay from S-Endpoint to VM

1. Same as 1-8 in 13.2.1.2.1 *Relay from VM to VM*.

13.2.1.3 SPM responsibilities

13.2.1.3.1 Relay from SP to SP

1. Same as 1-8 in 13.2.1.2.1 *Relay from VM to VM*.

13.2.1.3.2 Relay from S-Endpoint to NS-Endpoint

1. Same as 1-3 in 13.2.1.2.1 *Relay from VM to VM*.
2. Invoke *FFA_MSG_SEND_DIRECT_REQ* at physical FF-A instance through the *ERET* conduit with the same parameters as specified by the *Sender*. See 13.2.1.2.3 *Relay from S-Endpoint to VM* for responsibilities as the Relayer.

13.2.1.3.3 Relay from NS-Endpoint to S-Endpoint

1. Same as 1-8 in 13.2.1.2.1 *Relay from VM to VM*.

13.2.1.4 Receiver responsibilities

All Receivers have the same responsibilities irrespective of the origin of the message and the role of the Relayers in transmitting the message. These are as follows.

1. Copy message from parameter registers and process it.
2. Use the *FFA_MSG_SEND_DIRECT_RESP* interface to return the results of message processing.

13.3 FFA_MSG_SEND_DIRECT_RESP

Description

- Send a Partition or Framework message in parameter registers as a response to a target endpoint, run the endpoint and block until a new message is available.
- Valid FF-A instances and conduits are listed in Table 13.10.
- Syntax of this function is described in Table 13.11.
- Successful completion of this function is indicated in the same manner as that of the *FFA_MSG_WAIT* function (also see 12.1 *FFA_MSG_WAIT*).
- Encoding of error code in the FFA_ERROR function is described in Table 13.12.

Table 13.10: FFA_MSG_SEND_DIRECT_RESP instances and conduits

Config No.	FF-A instance	Valid conduits
1	Physical	SMC, ERET
2	Non-secure virtual	SMC, HVC, ERET
3	Secure virtual	SMC, HVC, SVC, ERET

Table 13.11: FFA_MSG_SEND_DIRECT_RESP function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000070. • 0xC4000070.
uint32 Source/Destination IDs	w1	• Source and destination endpoint IDs. – Bit[31:16]: Source endpoint ID. – Bit[15:0]: Destination endpoint ID.

Parameter	Register	Value
uint32 Flags	w2	• Message flags. – Bit[31]: Message type. ∗ b'0: Message encoded in parameter registers is a partition message. ∗ b'1: Message encoded in parameter registers is a framework message. – Bit[30:1]: Reserved (MBZ). – Bit[7:0]: ∗ Reserved (MBZ) if bit[31] = b'0. ∗ Framework message type if bit[31] = b'1. · b'00000000: Message for a power management operation initiated by a PSCI function. See 16.3.4 *Power Management messages* and Table 16.30. · b'00000001: Message for a warm boot. See 16.3.4 *Power Management messages* and Table 16.31.
Other Parameter registers	w3-w7 x3-x7	• IMPLEMENTATION DEFINED values.

Table 13.12: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: Unrecognized endpoint or message flags. • DENIED: Callee is not in a state to handle this request. • NOT_SUPPORTED: This function is not implemented at this FF-A instance. • BUSY: Message target is busy.

13.3.1 Component responsibilities for FFA_MSG_SEND_DIRECT_RESP

This section describes the common responsibilities that the participating FF-A components must fulfill during transmission of Partition and Framework messages between endpoints through the *FFA_MSG_SEND_DIRECT_RESP* interface. This interface is used in the scenarios listed in Table 4.7.

13.3.1.1 Sender responsibilities

13.3.1.1.1 Send from NS-Endpoint to S-Endpoint

1. Must write message payload to parameter registers.
2. Must specify Sender and Receiver endpoint IDs.
3. Must implement support for handling all error status codes that can be returned on completion of this interface.
4. See 13.3.1.2.2 *Relay from VM to S-Endpoint* & 13.3.1.3.3 *Relay from NS-Endpoint to S-Endpoint* for Relayer responsibilities in this message transmission.

13.3.1.1.2 Send from VM to VM

1. Same as Sender responsibilities while sending message from NS-Endpoint to S-Endpoint as listed in 13.3.1.1.1 *Send from NS-Endpoint to S-Endpoint*.
2. See 13.3.1.2.1 *Relay from VM to VM* for Relayer responsibilities in this message transmission.

13.3.1.1.3 Send from SP to SP

1. Same as Sender responsibilities while sending message from NS-Endpoint to S-Endpoint as listed in 13.3.1.1.1 *Send from NS-Endpoint to S-Endpoint*.
2. See 13.3.1.3.1 *Relay from SP to SP* for Relayer responsibilities in this message transmission.

13.3.1.1.4 Send from S-Endpoint to NS-Endpoint

1. Same as Sender responsibilities while sending message from NS-Endpoint to S-Endpoint as listed in 13.3.1.1.1 *Send from NS-Endpoint to S-Endpoint*.
2. See 13.3.1.3.2 *Relay from S-Endpoint to NS-Endpoint* for Relayer responsibilities in this message transmission.

13.3.1.2 Hypervisor responsibilities

13.3.1.2.1 Relay from VM to VM

1. Must validate that the Sender is allowed to send direct messages. Invoke *FFA_ERROR* with *NOT_SUPPORTED* as status if this is not the case.
2. Must validate Sender and Receiver endpoint IDs. Invoke *FFA_ERROR* with *INVALID PARAMETER* as status if either is invalid.
3. Must check that,
    1. Reserved parameter registers are 0 for a Partition and Framework message.
    2. Parameters are correctly encoded for a Framework message.

    FFA_ERROR with *INVALID PARAMETER* must be invoked as status if either is invalid.
4. Must ensure that target endpoint supports receipt of direct messages. Invoke *FFA_ERROR* with *DENIED* as status if this is not the case.
5. Must determine availability of an idle target endpoint execution context on this PE. Invoke *FFA_ERROR* with *BUSY* as status if not available.
6. Must ensure invocation of this interface by the Sender is completed only in response to a previous invocation of the *FFA_MSG_SEND_DIRECT_REQ* interface.
7. Must copy parameter registers from *Sender* execution context to *Receiver* execution context.
8. Must complete the invocation of the *FFA_MSG_SEND_DIRECT_REQ* interface that the Receiver had used to send the request to which the response is being provided.

13.3.1.2.2 Relay from VM to S-Endpoint

1. Same as 1-3 in 13.3.1.2.1 *Relay from VM to VM*.
2. Invoke *FFA_MSG_SEND_DIRECT_RESP* at physical FF-A instance through the *SMC* conduit with the same parameters as specified by the *Sender*. See 13.3.1.3.3 *Relay from NS-Endpoint to S-Endpoint* for responsibilities of the SPM as the Relayer.

13.3.1.2.3 Relay from S-Endpoint to VM

1. Same as 1-8 in 13.3.1.2.1 *Relay from VM to VM*.

13.3.1.3 SPM responsibilities

13.3.1.3.1 Relay from SP to SP

1. Same as 1-7 in 13.3.1.2.1 *Relay from VM to VM*.

13.3.1.3.2 Relay from S-Endpoint to NS-Endpoint

1. Same as 1-3 in 13.3.1.2.1 *Relay from VM to VM*.
2. Invoke *FFA_MSG_SEND_DIRECT_RESP* at physical FF-A instance through the *ERET* conduit with the same parameters as specified by the *Sender*. See 13.3.1.2.3 *Relay from S-Endpoint to VM* for responsibilities as the Relayer.

13.3.1.3.3 Relay from NS-Endpoint to S-Endpoint

1. Same as 1-8 in 13.3.1.2.1 *Relay from VM to VM*.

13.3.1.4 Receiver responsibilities

All Receivers have the same responsibilities irrespective of the origin of the message and the role of the Relayers in transmitting the message. These are as follows.

1. Copy response from parameter registers and process it.

Chapter 14
Memory management interfaces

14.1 FFA_MEM_DONATE

Description

- Starts a transaction to transfer of ownership of a memory region from a Sender endpoint to a Receiver endpoint.
- Transaction details are described in a memory transaction descriptor (see Table 8.19).
- Descriptor is populated in the TX buffer of the Owner by default.
- Valid FF-A instances and conduits are listed in Table 14.2.
- Syntax of this function is described in Table 14.3.
- Encoding of result parameters in the FFA_SUCCESS function is described in Table 14.4.
- Encoding of error code in the FFA_ERROR function is described in Table 14.5.

Table 14.2: FFA_MEM_DONATE instances and conduits

Config No.	FF-A instance	Valid conduits
1	Secure and Non-secure physical	SMC, ERET
2	Secure and Non-secure virtual	SMC, HVC, SVC

Table 14.3: FFA_MEM_DONATE function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000071. • 0xC4000071.
uint32 Total length	w1	• Total length of the memory transaction descriptor in bytes.
uint32 Fragment length	w2	• Length in bytes of the memory transaction descriptor passed in this ABI invocation. • *Fragment length* must be <= *Total length*. • If *Fragment length* < *Total length* then see 16.2.2 *Transmission of transaction descriptor in fragments* about how the remainder of the descriptor will be transmitted.
uint32/uint64 Address	w3/x3	• Base address of a buffer allocated by the Owner and distinct from the TX buffer. See 16.2.1 *Transmission of transaction descriptor in dynamically allocated buffers*. • MBZ if the TX buffer is used.

Parameter	Register	Value
uint32 Page count	w4	• Number of 4K pages in the buffer allocated by the Owner and distinct from the TX buffer. See 16.2.1 *Transmission of transaction descriptor in dynamically allocated buffers*. • MBZ if the TX buffer is used.
Other Parameter registers	w5-w7 x5-x7	• Reserved (MBZ).

Table 14.4: FFA_SUCCESS encoding

Parameter	Register	Value
uint64 Handle	w2/w3	• Globally unique Handle to identify the memory region on successful transmission of the transaction descriptor.
Other Result registers	w4-w7 x4-x7	• Reserved (MBZ).

Table 14.5: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS. • DENIED. • NO_MEMORY. • BUSY. • ABORTED.

14.1.1 Component responsibilities for FFA_MEM_DONATE

This interface is used to initiate a transaction to donate a memory region to a single Receiver endpoint (also see 8.6.2 *Donate memory transaction lifecycle*). Only the Owner and Relayer participate in this stage of the transaction. Responsibilities of the:

- Owner are listed in 14.1.1.1 *Owner responsibilities*.
- Relayer are listed in 14.1.1.2 *Relayer responsibilities*.

The transaction descriptor could be populated in a buffer dynamically allocated by the Owner as specified in 16.2.1 *Transmission of transaction descriptor in dynamically allocated buffers*.

Transmission of the transaction descriptor in fragments must be implemented by the Owner and Relayer as specified in 16.2.2 *Transmission of transaction descriptor in fragments*.

Time slicing of this ABI invocation must be implemented by the Owner and Relayer as specified in 16.2.3 *Time slicing of memory management operations*.

14.1.1.1 Owner responsibilities

1. Must ensure it is a *PE endpoint* and Owner of the memory region.
2. Must ensure the memory region is in an access state suitable for donation (see Table 8.9).
3. Must ensure the memory region fulfills the applicable rules stated in 8.4.1 *Ownership and access rules*.
4. Must describe memory region in the descriptor specified in Table 8.19 with a single endpoint memory access descriptor (also see 8.12.3.1 *Sender usage*).
5. Must implement support for handling all error status codes that can be returned on completion of this interface.
6. If the invocation of this interface completes successfuly, then must send at least the following information to the Receiver in a Partition message:
   1. Globally unique Handle returned by the Relayer.
   2. Owner endpoint ID.

   If the Receiver specified in the memory transaction descriptor is a SEPID, then the message must be sent to:

   - Either the *proxy endpoint* for the SEPID (see 8.2 *Direct memory access*) through a Partition message.
   - Or the *independent* peripheral device associated with the SEPID through an IMPLEMENTATION DEFINED mechanism.

   Provision of any other information from the transaction descriptor is IMPLEMENTATION DEFINED.
7. If the Receiver rejects the request in step 6, the Sender should use the *FFA_MEM_RECLAIM* interface with the Handle returned by the Relayer to reclaim ownership of the memory region. It must treat the memory region as being inaccessible until the *FFA_MEM_RECLAIM* invocation completes.

14.1.1.2 Relayer responsibilities

1. Must validate the *Total length* input parameter to ensure that the length of the transaction descriptor does not exceed the size of the buffer it has been populated in. Must return *INVALID_PARAMETERS* in case of an error.
2. Must validate the *Sender endpoint ID* field in the transaction descriptor to ensure that the Sender is the Owner of the memory region and a *PE endpoint*. Must return *DENIED* in case of an error.
3. Must ensure that a request by an SP to donate Secure memory to a NS-Endpoint is rejected by returning the *DENIED* error code.
4. Must ensure that the memory region is in the *Owner-EA* state for the Owner (see Table 8.9). It must return *DENIED* in case of an error.
5. Must validate that the *Endpoint memory access descriptor count* & *Endpoint memory access descriptor array* fields in the transaction descriptor as specified in 8.12.3.3 *Relayer usage*.
6. Must validate the *Memory region attributes* field in the transaction descriptor as specified in 8.11.4 *Memory region attributes usage*.
7. Must validate the *Flags* field specified in the transaction descriptor as specified in 8.12.4 *Flags usage*.
8. Must validate the *Handle* field specified in the transaction descriptor as specified in 8.12.1 *Handle usage*.
9. Unmap the memory region from the translation regime of the Owner, if managed by the Relayer as specified in 8.3 *Address translation regimes*.
10. If the Receiver is a *PE endpoint* or a *Stream endpoint* with a *proxy endpoint* managed by the Relayer, then the Relayer must:
    1. Save the transaction descriptor information so that it can be validated when retrieved through invocations of the *FFA_MEM_RETRIEVE_REQ* & *FFA_MEM_RETRIEVE_RESP* interfaces.
    2. Return *NO_MEMORY* if there is not enough memory to complete this operation.

11. If the Receiver is a *Stream endpoint* associated with an *independent* device managed by the Relayer, then the Relayer must:
    1. Allocate an IPA range and map the memory region in the translation regime of the Receiver managed by the Relayer as specified in 8.3 *Address translation regimes*.

       The mapping must be created with the memory region attributes and permissions specified in the transaction descriptor.
    2. Describe the memory region to the device using the SEPID through an IMPLEMENTATION DEFINED mechanism.
12. If the call executes successfuly, the Relayer must:
    1. Ensure that the state of the memory region in the participating FF-A components is observed as follows:
       1. If the Receiver is a *PE endpoint* or a *SEPID* associated with a dependent peripheral device, then:
          - *Owner-NA* for the Owner.
          - *!Owner-NA* for the Receiver.
       2. If the Receiver is a *SEPID* associated with an independent peripheral device, then:
          - *!Owner-NA* for the Owner.
          - *Owner-EA* for the Receiver.
    2. Allocate and return a *Handle* as described in 8.10.2 *Memory region handle*.
13. If the Owner is a VM and the Receiver is an SP or SEPID associated with a Secure Stream ID, the Hypervisor must forward the memory transaction descriptor to the SPM. This must be done by invoking this interface at the Non-secure physical FF-A instance as follows.
    1. The fields of the transaction descriptor must be unchanged apart from the following exception.
       1. The memory region must be described as composed of physically addressed constituent 4K pages in one or more *Constituent memory region descriptors*.

          This must be done by performing the VA or IPA to PA translation of the memory region described by the Owner at the non-secure virtual FF-A instance.

          The order in which the address ranges are specified by the Owner must be preserved by the Hypervisor.
       2. The *Constituent memory region descriptors* must be described in a *Composite memory region descriptor* which must be referenced by the *Endpoint memory access descriptor* included in the transaction descriptor.
    2. The updated transaction descriptor must be copied into the TX buffer shared between the Hypervisor and SPM.

       If the TX buffer is busy, the Hypervisor must return *BUSY*.

       If the TX buffer is too small and it is not possible to use the optional features to transmit the descriptor listed in 16.2.2 *Transmission of transaction descriptor in fragments* and 16.2.1 *Transmission of transaction descriptor in dynamically allocated buffers*, the Hypervisor must return *NO_MEMORY*

    The SPM must fulfill the Relayer responsibilities listed in this section.

14.2 FFA_MEM_LEND

Description

- Starts a transaction to transfer access to a memory region from its Owner to one or more Borrowers.
- Transaction details are described in a memory transaction descriptor (see Table 8.19).
- Descriptor is populated in the TX buffer of the Owner by default.
- Valid FF-A instances and conduits are listed in Table 14.7.
- Syntax of this function is described in Table 14.8.
- Encoding of result parameters in the FFA_SUCCESS function is described in Table 14.9.
- Encoding of error code in the FFA_ERROR function is described in Table 14.10.

Table 14.7: FFA_MEM_LEND instances and conduits

Config No.	FF-A instance	Valid conduits
1	Secure and Non-secure physical	SMC, ERET
2	Secure and Non-secure virtual	SMC, HVC, SVC

Table 14.8: FFA_MEM_LEND function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000072. • 0xC4000072.
uint32 Total length	w1	• Total length of the memory transaction descriptor in bytes.
uint32 Fragment length	w2	• Length in bytes of the memory transaction descriptor passed in this ABI invocation. • *Fragment length* must be <= *Total length*. • If *Fragment length* < *Total length* then see 16.2.2 *Transmission of transaction descriptor in fragments* about how the remainder of the descriptor will be transmitted.
uint32/uint64 Address	w3/x3	• Base address of a buffer allocated by the Owner and distinct from the TX buffer. See 16.2.1 *Transmission of transaction descriptor in dynamically allocated buffers*. • MBZ if the TX buffer is used.

Parameter	Register	Value
uint32 Page count	w4	• Number of 4K pages in the buffer allocated by the Owner and distinct from the TX buffer. See 16.2.1 *Transmission of transaction descriptor in dynamically allocated buffers*. • MBZ if the TX buffer is used.
Other Parameter registers	w5-w7 x5-x7	• Reserved (MBZ).

Table 14.9: FFA_SUCCESS encoding

Parameter	Register	Value
uint64 Handle	w2/w3	• Globally unique Handle to identify the memory region on successful transmission of the transaction descriptor. MBZ otherwise (see 8.10.2 *Memory region handle*).
Other Result registers	w4-w7 x4-x7	• Reserved (MBZ).

Table 14.10: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS. • DENIED. • NO_MEMORY. • BUSY. • ABORTED.

14.2.1 Component responsibilities for FFA_MEM_LEND

This interface is used to initiate a transaction to lend a memory region to one or more Borrower endpoints (also see 8.7.2 *Lend memory transaction lifecycle*). Only the Lender and Relayer participate in this stage of the transaction. Responsibilities of the:

- Lender are listed in 14.2.1.1 *Lender responsibilities*.
- Relayer are listed in 14.2.1.2 *Relayer responsibilities*.

The transaction descriptor could be populated in a buffer dynamically allocated by the Lender as specified in 16.2.1 *Transmission of transaction descriptor in dynamically allocated buffers*.

Transmission of the transaction descriptor in fragments must be implemented by the Lender and Relayer as specified in 16.2.2 *Transmission of transaction descriptor in fragments*.

Time slicing of this ABI invocation must be implemented by the Lender and Relayer as specified in 16.2.3 *Time slicing of memory management operations*.

14.2.1.1 Lender responsibilities

1. Must ensure it is a *PE endpoint* and Owner of the memory region.
2. Must ensure the memory region is in an access state suitable for lending (see Table 8.10).
3. Must ensure the memory region fulfills the applicable rules stated in 8.4.1 *Ownership and access rules*.
4. Must describe memory region in the descriptor specified in Table 8.19 with an endpoint memory access descriptor for each Borrower (also see 8.12.3.1 *Sender usage*).
5. Must implement support for handling all error status codes that can be returned on completion of this interface.
6. If the invocation of this interface completes successfuly, then must send at least the following information to each Borrower in a Partition message:
    1. Globally unique Handle returned by the Relayer.
    2. Lender endpoint ID.

    If the Borrower specified in the memory transaction descriptor is a SEPID, then the message must be sent to:

    - Either the *proxy endpoint* for the SEPID (see 8.2 *Direct memory access*) through a Partition message.
    - Or the *independent* peripheral device associated with the SEPID through an IMPLEMENTATION DEFINED mechanism.

    Provision of any other information from the transaction descriptor is IMPLEMENTATION DEFINED.
7. If the Borrower rejects the request in step 6, the Lender should use the *FFA_MEM_RECLAIM* interface with the Handle returned by the Relayer to reclaim ownership of the memory region. It must treat the memory region as being inaccessible until the *FFA_MEM_RECLAIM* invocation completes.

14.2.1.2 Relayer responsibilities

1. Must validate the *Total length* input parameter to ensure that the length of the transaction descriptor does not exceed the size of the buffer it has been populated in. Must return *INVALID_PARAMETERS* in case of an error.
2. Must validate the *Sender endpoint ID* field in the transaction descriptor to ensure that the Lender is the Owner of the memory region and a *PE endpoint*. Must return *DENIED* in case of an error.
3. Must ensure that a request by an SP to lend Secure memory to a NS-Endpoint is rejected by returning the *DENIED* error code.
4. Must validate that the memory region is in the *Owner-EA* state for the Lender (see Table 8.10). It must return *DENIED* in case of an error.
5. Must validate that the *Endpoint memory access descriptor count* & *Endpoint memory access descriptor array* fields in the transaction descriptor as specified in 8.12.3.3 *Relayer usage*.
6. Must validate the *Memory region attributes* field in the transaction descriptor as specified in 8.11.4 *Memory region attributes usage*.
7. Must validate the *Flags* field specified in the transaction descriptor as specified in 8.12.4 *Flags usage*.
8. Must validate the *Handle* field specified in the transaction descriptor as specified in 8.12.1 *Handle usage*.
9. Unmap the memory region from the translation regime of the Lender, if managed by the Relayer as specified in 8.3 *Address translation regimes*.
10. If the Borrower is a *PE endpoint* or a *Stream endpoint* with a *proxy endpoint* managed by the Relayer, then the Relayer must:
    1. Save the transaction descriptor information so that it can be validated when retrieved through invocations of the *FFA_MEM_RETRIEVE_REQ* & *FFA_MEM_RETRIEVE_RESP* interfaces.
    2. Return *NO_MEMORY* if there is not enough memory to complete this operation.

11. If the Borrower is a *Stream endpoint* associated with an *independent* device managed by the Relayer, then the Relayer must:
    1. Allocate an IPA range and map the memory region in the translation regime of the Borrower managed by the Relayer as specified in 8.3 *Address translation regimes*.

       The mapping must be done with the memory region attributes and permissions specified in the transaction descriptor.
    2. Describe the memory region to the device using the SEPID through an IMPLEMENTATION DEFINED mechanism.
12. If the call executes successfuly, the Relayer must:
    1. Ensure that the state of the memory region in the participating FF-A components is observed as follows:
       1. If a Borrower is a *PE endpoint* or a *SEPID* associated with a dependent peripheral device, then:
          - *Owner-NA* for the Lender.
          - *!Owner-NA* for the Borrower.
       2. If a Borrower is a *SEPID* associated with an independent peripheral device, then:
          - *Owner-NA* for the Lender.
          - *!Owner-EA* for the Borrower, if the count of Borrowers in the transaction is = *1*.
          - *Owner-SA* for the Borrower, if the count of Borrowers in the transaction is > *1*.
    2. Allocate and return a *Handle* as described in 8.10.2 *Memory region handle*.
13. If the Lender is a VM and the Borrower is an SP or SEPID associated with a Secure Stream ID, the Hypervisor must forward the memory transaction descriptor to the SPM. This must be done by invoking this interface at the Non-secure physical FF-A instance as follows.
    1. The fields of the transaction descriptor must be unchanged apart from the following exception.
       1. The memory region must be described as composed of physically addressed constituent 4K pages in one or more *Constituent memory region descriptors*.

          This must be done by performing the VA or IPA to PA translation of the memory region described by the Owner at the non-secure virtual FF-A instance.

          The order in which the address ranges are specified by the Lender must be preserved by the Hypervisor.
       2. The *Constituent memory region descriptors* must be described in a *Composite memory region descriptor* which must be referenced by the *Endpoint memory access descriptor* included in the transaction descriptor.
    2. The updated transaction descriptor must be copied into the TX buffer shared between the Hypervisor and SPM.

       If the TX buffer is busy, the Hypervisor must return *BUSY*.

       If the TX buffer is too small and it is not possible to use the optional features to transmit the descriptor listed in 16.2.2 *Transmission of transaction descriptor in fragments* and 16.2.1 *Transmission of transaction descriptor in dynamically allocated buffers*, the Hypervisor must return *NO_MEMORY*

    The SPM must fulfill the Relayer responsibilities listed in this section.

14.3 FFA_MEM_SHARE

Description

- Starts a transaction to grant access to a memory region to one or more Borrowers.
- Transaction details are described in a memory transaction descriptor (see Table 8.19).
- Descriptor is populated in the TX buffer of the Owner by default.
- Valid FF-A instances and conduits are listed in Table 14.12.
- Syntax of this function is described in Table 14.13.
- Encoding of result parameters in the FFA_SUCCESS function is described in Table 14.14.
- Encoding of error code in the FFA_ERROR function is described in Table 14.15.

Table 14.12: FFA_MEM_SHARE instances and conduits

Config No.	FF-A instance	Valid conduits
1	Secure and Non-secure physical	SMC, ERET
2	Secure and Non-secure virtual	SMC, HVC, SVC

Table 14.13: FFA_MEM_SHARE function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000073. • 0xC4000073.
uint32 Total length	w1	• Total length of the memory transaction descriptor in bytes.
uint32 Fragment length	w2	• Length in bytes of the memory transaction descriptor passed in this ABI invocation. • *Fragment length* must be <= *Total length*. • If *Fragment length* < *Total length* then see 16.2.2 *Transmission of transaction descriptor in fragments* about how the remainder of the descriptor will be transmitted.
uint32/uint64 Address	w3/x3	• Base address of a buffer allocated by the Owner and distinct from the TX buffer. See 16.2.1 *Transmission of transaction descriptor in dynamically allocated buffers*. • MBZ if the TX buffer is used.

Parameter	Register	Value
uint32 Page count	w4	• Number of 4K pages in the buffer allocated by the Owner and distinct from the TX buffer. See 16.2.1 *Transmission of transaction descriptor in dynamically allocated buffers*. • MBZ if the TX buffer is used.
Other Parameter registers	w5-w7 x5-x7	• Reserved (MBZ).

Table 14.14: FFA_SUCCESS encoding

Parameter	Register	Value
uint64 Handle	w2/w3	• Globally unique Handle to identify the memory region on successful transmission of the transaction descriptor. MBZ otherwise (see 8.10.2 *Memory region handle*).
Other Result registers	w4-w7 x4-x7	• Reserved (MBZ).

Table 14.15: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS. • DENIED. • NO_MEMORY. • BUSY. • ABORTED.

14.3.1 Component responsibilities for FFA_MEM_SHARE

This interface is used to initiate a transaction to share a memory region with one or more Receiver endpoints (also see 8.8.2 *Share memory transaction lifecycle*). Only the Owner and Relayer participate in this stage of the transaction. Responsibilities of the:

- Owner are listed in 14.3.1.1 *Owner responsibilities*.
- Relayer are listed in 14.3.1.2 *Relayer responsibilities*.

The transaction descriptor could be populated in a buffer dynamically allocated by the Lender as specified in 16.2.1 *Transmission of transaction descriptor in dynamically allocated buffers*.

Transmission of the transaction descriptor in fragments must be implemented by the Lender and Relayer as specified in 16.2.2 *Transmission of transaction descriptor in fragments*.

Time slicing of this ABI invocation must be implemented by the Lender and Relayer as specified in 16.2.3 *Time slicing of memory management operations*.

14.3.1.1 Owner responsibilities

1. Must ensure it is a *PE endpoint* and Owner of the memory region.
2. Must ensure the memory region is in an access state suitable for sharing (see Table 8.11).
3. Must ensure the memory region fulfills the applicable rules stated in 8.4.1 *Ownership and access rules*.
4. Must describe memory region in the descriptor specified in Table 8.19 with an endpoint memory access descriptor for each Borrower (also see 8.12.3.1 *Sender usage*).
5. Must implement support for handling all error status codes that can be returned on completion of this interface.
6. If the invocation of this interface completes successfuly, then must send at least the following information to each Borrower in a Partition message:
    1. Globally unique Handle returned by the Relayer.
    2. Owner endpoint ID.

    If the Borrower specified in the memory transaction descriptor is a SEPID, then the request must be sent to:

    - Either the *proxy endpoint* for the SEPID (see 8.2 *Direct memory access*) through a Partition message.
    - Or the *independent* peripheral device associated with the SEPID through an IMPLEMENTATION DEFINED mechanism.

    Provision of any other information from the transaction descriptor is IMPLEMENTATION DEFINED.
7. If the Receiver rejects the request in step 6, the Sender should use the *FFA_MEM_RECLAIM* interface with the Handle returned by the Relayer to reclaim ownership of the memory region. It must treat the memory region as being inaccessible until the *FFA_MEM_RECLAIM* invocation completes.

14.3.1.2 Relayer responsibilities

1. Must validate the *Total length* input parameter to ensure that the length of the transaction descriptor does not exceed the size of the buffer it has been populated in. Must return *INVALID_PARAMETERS* in case of an error.
2. Must validate the *Sender endpoint ID* field in the transaction descriptor to ensure that the Lender is the Owner of the memory region and a *PE endpoint*. Must return *DENIED* in case of an error.
3. Must ensure that a request by an SP to share Secure memory to a NS-Endpoint is rejected by returning the *DENIED* error code.
4. Must validate that the memory region is in an access state suitable for sharing (see Table 8.11) and return *DENIED* in case of an error.
5. Must validate that the *Endpoint memory access descriptor count* & *Endpoint memory access descriptor array* fields in the transaction descriptor as specified in 8.12.3.3 *Relayer usage*.
6. Must validate the *Memory region attributes* field in the transaction descriptor as specified in 8.11.4 *Memory region attributes usage*.
7. Must validate the *Flags* field specified in the transaction descriptor as specified in 8.12.4 *Flags usage*.
8. Must validate the *Handle* field specified in the transaction descriptor as specified in 8.12.1 *Handle usage*.
9. If the Lender has specified a different data access permission to access the memory region in its translation regime, then the Relayer must validate the permission as specified in 8.11.2 *Data access permissions usage*, save the current permission and update the translation tables to reflect the new permission.
10. If the Borrower is a *PE endpoint* or a *Stream endpoint* with a *proxy endpoint* managed by the Relayer, then the Relayer must:
    1. Save the transaction descriptor information so that it can be validated when retrieved through invocations of the *FFA_MEM_RETRIEVE_REQ* & *FFA_MEM_RETRIEVE_RESP* interfaces.

2. Return *NO_MEMORY* if there is not enough memory to complete this operation.

11. If the Borrower is a *Stream endpoint* associated with an *independent* device managed by the Relayer, then the Relayer must:
    1. Allocate an IPA range and map the memory region in the translation regime of the Borrower managed by the Relayer as specified in 8.3 *Address translation regimes*.

       The mapping must be done with the memory region attributes and permissions specified in the transaction descriptor.
    2. Describe the memory region to the device using the SEPID through an IMPLEMENTATION DEFINED mechanism.
12. If the call executes successfuly, the Relayer must:
    1. Ensure that the state of the memory region in the participating FF-A components is observed as follows:
       1. If a Borrower is a *PE endpoint* or a *SEPID* associated with a dependent peripheral device, then:
          - *Owner-SA* for the Lender.
          - *!Owner-NA* for the Borrower.
       2. If a Borrower is a *SEPID* associated with an independent peripheral device, then:
          - *Owner-SA* for the Lender.
          - *!Owner-SA* for the Borrower.
    2. Allocate and return a *Handle* as described in 8.10.2 *Memory region handle*.
13. If the Lender is a VM and the Borrower is an SP or SEPID associated with a Secure Stream ID, the Hypervisor must forward the memory transaction descriptor to the SPM. This must be done by invoking this interface at the Non-secure physical FF-A instance as follows.
    1. The fields of the transaction descriptor must be unchanged apart from the following exception.
       1. The memory region must be described as composed of physically addressed constituent 4K pages in one or more *Constituent memory region descriptors*.

          This must be done by performing the VA or IPA to PA translation of the memory region described by the Owner at the non-secure virtual FF-A instance.

          The order in which the address ranges are specified by the Lender must be preserved by the Hypervisor.
       2. The *Constituent memory region descriptors* must be described in a *Composite memory region descriptor* which must be referenced by the *Endpoint memory access descriptor* included in the transaction descriptor.
    2. The updated transaction descriptor must be copied into the TX buffer shared between the Hypervisor and SPM.

       If the TX buffer is busy, the Hypervisor must return *BUSY*.

       If the TX buffer is too small and it is not possible to use the optional features to transmit the descriptor listed in 16.2.2 *Transmission of transaction descriptor in fragments* and 16.2.1 *Transmission of transaction descriptor in dynamically allocated buffers*, the Hypervisor must return *NO_MEMORY*.

The SPM must fulfill the Relayer responsibilities listed in this section.

14.4 FFA_MEM_RETRIEVE_REQ

Description

- Requests completion of a donate, lend or share memory management transaction.
- Transaction details are described in a memory transaction descriptor (see Table 8.19).
- Descriptor is populated in the TX buffer of the Receiver by default.
- Valid FF-A instances and conduits are listed in Table 14.17.
- Syntax of this function is described in Table 14.18.
- Encoding of error code in the FFA_ERROR function is described in Table 14.19.
- Successful transmission of the transaction descriptor is indicated by an invocation of the *FFA_MEM_RETRIEVE_RESP* function (see 14.5 *FFA_MEM_RETRIEVE_RESP*).

Table 14.17: FFA_MEM_RETRIEVE_REQ instances and conduits

Config No.	FF-A instance	Valid conduits
1	Secure and Non-secure physical	SMC, ERET
2	Secure and Non-secure virtual	SMC, HVC, SVC

Table 14.18: FFA_MEM_RETRIEVE_REQ function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000074. • 0xC4000074.
uint32 Total length	w1	• Total length of the memory transaction descriptor in bytes.
uint32 Fragment length	w2	• Length in bytes of the memory transaction descriptor passed in this ABI invocation. • *Fragment length* must be <= *Total length*. • If *Fragment length* < *Total length* then see 16.2.2 *Transmission of transaction descriptor in fragments* about how the remainder of the descriptor will be transmitted.
uint32/uint64 Address	w3/x3	• Base address of a buffer allocated by the Owner and distinct from the TX buffer. See 16.2.1 *Transmission of transaction descriptor in dynamically allocated buffers*. • MBZ if the TX buffer is used.

Parameter	Register	Value
uint32 Page count	w4	• Number of 4K pages in the buffer allocated by the Owner and distinct from the TX buffer. See 16.2.1 *Transmission of transaction descriptor in dynamically allocated buffers*. • MBZ if the TX buffer is used.
Other Parameter registers	w5-w7 x5-x7	• Reserved (MBZ).

Table 14.19: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS. • DENIED. • NO_MEMORY. • BUSY. • ABORTED.

14.4.1 Component responsibilities for FFA_MEM_RETRIEVE_REQ

This ABI is used by a Receiver to retrieve a memory region. Retrieval implies a request to the Relayer to map the memory region in the translation regime of the Receiver. The Receiver must use the transaction descriptor (see Table 8.19) to identify the memory region and specify its properties. The Receiver could:

1. Retrieve a memory region that was shared, lent or donated by an Owner. For this scenario, responsibilities of the:
   - Receiver are listed in 14.4.1.1 *Receiver responsibilities*.
   - Relayer are listed in 14.4.1.2 *Relayer responsibilities*.
2. Retrieve a memory region that it had relinquished but has not been reclaimed by the Owner yet (see 14.4.2 *Support for multiple retrievals by a Borrower*).

It is also possible for a Hypervisor to use this interface to retrieve a memory region description on its behalf. This scenario is described in 14.4.3 *Support for retrieval by the Hypervisor*.

In all cases, a successful retrieval is indicated by an invocation of the *FFA_MEM_RETRIEVE_RESP* ABI by the Relayer.

The transaction descriptor could be populated in a buffer dynamically allocated by the Receiver as specified in 16.2.1 *Transmission of transaction descriptor in dynamically allocated buffers*.

Transmission of the transaction descriptor in fragments must be implemented by the caller and Relayer as specified in 16.2.2 *Transmission of transaction descriptor in fragments*.

Time slicing of this ABI invocation must be implemented by the Receiver and Relayer as specified in 16.2.3 *Time slicing of memory management operations*.

14.4.1.1 Receiver responsibilities

1. Must populate a transaction descriptor with the *Handle* (see 8.12.1 *Handle usage*) that identifies the memory region, the *Endpoint ID* that identifies the Owner and the *Tag* (see 8.12.2 *Tag usage*) associated with the

transaction.

Could populate other fields of the transaction descriptor as follows. This depends on how much information the Sender shares with the Receiver about the memory management transaction.

- See 8.12.3.2 *Receiver usage* for usage of the *Endpoint memory access descriptor array* field.
- See 8.12.4 *Flags usage* for usage of the *Flags* field.
- See 8.11.4 *Memory region attributes usage* for usage of the *Memory region attributes* field.

The Relayer must validate the information provided by the Sender. It must reject the transaction by sending a Partition message to the Sender if the validation fails.

The protocol between the Sender and Receiver to convey transaction information and to reject the transaction is IMPLEMENTATION DEFINED.

2. Must implement support for handling all error status codes that can be returned on completion of this interface.

14.4.1.2 Relayer responsibilities

1. Must validate the *Total length* input parameter to ensure that the length of the transaction descriptor does not exceed the size of the buffer it has been populated in. Must return *INVALID_PARAMETERS* in case of an error.
2. Must validate the Sender endpoint ID field in the transaction descriptor to ensure that the Sender is the Owner of the memory region or the proxy endpoint acting on behalf of a Stream endpoint. Must return *DENIED* in case of an error.
3. Must validate the *Memory region attributes* field in the transaction descriptor as specified in 8.11.4 *Memory region attributes usage*.
4. Must validate the *Flags* field specified in the transaction descriptor as specified in 8.12.4 *Flags usage*.
5. Must validate the *Handle* field specified in the transaction descriptor as specified in 8.12.1 *Handle usage*.
6. Must validate the *Tag* field specified in the transaction descriptor as specified in 8.12.2 *Tag usage*.
7. Must validate that the *Endpoint memory access descriptor count* & *Endpoint memory access descriptor array* fields in the transaction descriptor as specified in 8.12.3.3 *Relayer usage*.
8. Must map the memory region in the translation regime of the Receiver managed by the Relayer as specified in 8.3 *Address translation regimes*.

   If the Receiver is a proxy endpoint for one or more Stream endpoints then the memory region must be mapped in the stage 2 translation tables corresponding to each SEPID. The memory region must not be mapped in the translation regime of the proxy endpoint.

   The order in which the address ranges are specified by the Lender must be preserved by the Hypervisor.

   The Relayer must return *NO_MEMORY* if there is not enough memory to map the memory region.
9. Must return *BUSY*, if *FFA_MEM_RETRIEVE_RESP* cannot be invoked because the Receiver RX buffer is busy.
10. Must return *NO_MEMORY* if *FFA_MEM_RETRIEVE_RESP* cannot be invoked because there is not enough memory to allocate a transaction descriptor to describe the memory region.
11. If the call executes successfuly, the Relayer must ensure that the state of the memory region in the participating FF-A components is observed as follows:
    - If the transaction type is FFA_MEM_DONATE,
      - *!Owner-NA* for the Owner.
      - *Owner-EA* for the Receiver.

- If the transaction type is FFA_MEM_LEND, and the count of Borrowers in the transaction is = *1*,
  - *Owner-NA* for the Lender.
  - *!Owner-EA* for the Borrower.
- If the transaction type is FFA_MEM_LEND, and the count of Borrowers in the transaction is > *1*,
  - *Owner-SA* for the Lender.
  - *!Owner-SA* for the Borrower.
- If the transaction type is FFA_MEM_SHARE,
  - *Owner-SA* for the Lender.
  - *!Owner-SA* for the Borrower.

14.4.2 Support for multiple retrievals by a Borrower

After a Receiver relinquishes access to a memory region (see 14.6 *FFA_MEM_RELINQUISH*) that was lent or shared, a Relayer must allow the Receiver to retrieve the memory region again as long as it has not been reclaimed by its Owner. To support this mechanism, it must:

1. Allow the Owner to reclaim the memory region only if all Borrowers have relinquished it as many times as they have retrieved it.
2. Unmap the memory region from the translation regime of the Borrower only after it has been relinquished as many times as it was retrieved.
3. Ensure that the address ranges used to describe the memory region on each retrieval are the same if the memory region is already mapped in the translation regime of the Receiver.

The number of times a Receiver is allowed to retrieve a memory region without relinquishing it first is *1* by default. A Receiver must use the FFA_FEATURES ABI (see 11.2 *FFA_FEATURES*) to determine the number of outstanding retrievals supported by the Relayer. The Relayer must return *DENIED* if a Receiver exceeds the retrieval count.

14.4.3 Support for retrieval by the Hypervisor

In a transaction to donate, share or lend a memory region between an Owner VM and a Receiver SP, the SPM is responsible for allocating the *Handle* to identify the memory region (see 8.10.2 *Memory region handle*).

The Hypervisor implementation could maintain an association between the Handle and the memory region. For example, to map the memory region back into the translation regime of the Owner in response to an FFA_MEM_RECLAIM ABI.

A Hypervisor implementation could choose to rely on the SPM to manage the association between the Handle and the memory region. For example, to avoid memory costs associated with tracking this state over a period of time.

In this case, the Hypervisor could use the FFA_MEM_RETRIEVE_REQ ABI to obtain the memory region description by specifying its Handle. It would use this description to map or unmap the memory region depending on the operation requested by a VM. For example, an operation to reclaim a memory region would follow these steps.

1. Lender VM calls FFA_MEM_RECLAIM.
2. Hypervisor uses the Handle to call FFA_MEM_RETRIEVE_REQ and obtain the memory region description.
3. Hypervisor forwards FFA_MEM_RECLAIM to the SPM to ensure all Borrowers have stopped using the memory region.
4. On a successful return from the SPM, the Hypervisor uses the memory region description to map the region in the translation regime of the Lender VM.
5. Hypervisor completes the invocation of FFA_MEM_RECLAIM from the Lender VM successfuly.

If it chooses to use this mechanism, the Hypervisor must populate the transaction descriptor as follows.

1. It must specify the Handle specified by the VM in the *Handle* field.
2. It must ensure that all other fields in the transaction descriptor are zeroed.

From the perspective of an SPM, an invocation of the FFA_MEM_RETRIEVE_REQ ABI at the Non-secure physical FF-A instance could,

- Either originate from the Hypervisor as described above.
- Or originate from a Borrower VM. It was forwarded by the Hypervisor.

In the former case, the SPM must not update the ownership and access state associated with the memory region as it would do in the latter case (see 14.4.1.2 *Relayer responsibilities*). To do this, the SPM must distinguish between the two types of invocation as follows.

- In the former case, the *Endpoint memory access descriptor count* in the transaction descriptor must be *0*.
- In the latter case, the *Endpoint memory access descriptor count* in the transaction descriptor must be >= *1*.

In the former case, the SPM must also validate the *Handle* field specified in the transaction descriptor as follows.

- Ensure that it identifies a memory region that was either shared or lent to at least a single VM or is owned by a VM.
- Ensure that it was previously allocated and has not been reclaimed by the Owner.

The SPM must provide the memory region description to the Hypervisor through an invocation of the FFA_MEM_RETRIEVE_RESP ABI as follows.

- The memory region must be described as composed of physically addressed constituent 4K pages in one or more *Constituent memory region descriptors*.
- The *Constituent memory region descriptors* must be described in a *Composite memory region descriptor*.
- The *Composite memory region descriptor* must be referenced by a single *Endpoint memory access descriptor* included in the transaction descriptor.
- The *Sender endpoint ID* field must be set to the Lender or Owner VM ID in the transaction descriptor.
- The *Handle* field must be set to the input *Handle*.

U **Implementation Note**

This feature allows the Hypervisor to retrieve the physical address ranges of a memory region that must be either mapped or unmapped from the stage 2 translation descriptors of a VM.

It is possible that the Hypervisor implementation maintains mappings in the stage 2 translation descriptors for a VM such that a *IPA != PA*. In this case, it must track the original IPA ranges through an IMPLEMENTATION DEFINED mechanism to be able to correctly map or unmap the retrieved memory region.

Furthermore, in the case where the Hypervisor must map the memory region in the stage 2 translation descriptors for a VM, it must track the original memory access permissions and attributes of the memory region through an IMPLEMENTATION DEFINED mechanism.

14.5 FFA_MEM_RETRIEVE_RESP

Description

- A Relayer uses this interface to describe a memory region and its properties in response to the latest successful invocation of the *FFA_MEM_RETRIEVE_REQ* interface by an endpoint or Hypervisor.
- Transaction details are described in a transaction descriptor specified in Table 8.19.
- Descriptor is populated in the RX buffer of the Receiver by default.
- Valid FF-A instances and conduits are listed in Table 14.21.
- Syntax of this function is described in Table 14.22.
- Encoding of error code in the FFA_ERROR function is described in Table 14.23.
- Successful transmission of the transaction descriptor is indicated by an invocation of any FF-A function by the Receiver.

Table 14.21: FFA_MEM_RETRIEVE_RESP instances and conduits

Config No.	FF-A instance	Valid conduits
1	Secure and Non-secure physical	SMC, ERET
2	Secure and Non-secure virtual	ERET

Table 14.22: FFA_MEM_RETRIEVE_RESP function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000075.
uint32 Total length	w1	• Total length of the memory transaction descriptor in bytes.
uint32 Fragment length	w2	• Length in bytes of the memory transaction descriptor passed in this ABI invocation. • *Fragment length* must be <= *Total length*. • If *Fragment length* < *Total length* then see 16.2.2 *Transmission of transaction descriptor in fragments* about how the remainder of the descriptor will be transmitted.
uint32/uint64 Parameter	w3/x3	• Reserved (MBZ).
uint32/uint64 Parameter	w4/x4	• Reserved (MBZ).
Other Parameter registers	w5-w7 x5-x7	• Reserved (MBZ).

Table 14.23: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS. • NO_MEMORY.

14.5.1 Component responsibilities for FFA_MEM_RETRIEVE_RESP

A Relayer invokes this interface as the caller with an endpoint as the callee in the following scenarios. It must fulfill the responsibilities listed in 14.5.1.1 *Relayer responsibilities*.

- The endpoint calls *FFA_MEM_RETRIEVE_REQ* to retrieve a memory region that was donated, lent or shared with it by the Owner. The Relayer completes the transaction by invoking the *FFA_MEM_RETRIEVE_RESP* interface.
- The endpoint calls *FFA_MEM_RETRIEVE_REQ* to retrieve a memory region it had relinquished earlier. The Relayer fulfills the request by invoking the *FFA_MEM_RETRIEVE_RESP* interface (see 14.4.2 *Support for multiple retrievals by a Borrower*).

The SPM invokes this interface as the caller with the Hypervisor as the callee in the scenario described in 14.4.3 *Support for retrieval by the Hypervisor*.

In all scenarios, the Relayer must populate a transaction descriptor specified in Table 8.19 to describe the memory region and its properties. This must be done in one of the following buffers.

- The RX buffer of the callee must be used if the callee used its TX buffer in the counterpart invocation of the *FFA_MEM_RETRIEVE_REQ* ABI earlier.
- If the callee used a dynamically allocated buffer in the counterpart invocation of the *FFA_MEM_RETRIEVE_REQ* ABI earlier, then the same buffer must be used (see 16.2.1 *Transmission of transaction descriptor in dynamically allocated buffers*).

Transmission of the transaction descriptor in fragments in all scenarios must be implemented by the Relayer and callee as specified in 16.2.2 *Transmission of transaction descriptor in fragments*.

In all scenarios, the callee (endpoint or Hypervisor) must fulfill the responsibilities listed in 14.5.1.2 *Callee responsibilities*. It must use the error codes listed in Table 14.23 to report an error back to the Relayer.

14.5.1.1 Relayer responsibilities

1. Must populate the Sender endpoint ID field in the transaction descriptor with the endpoint ID of the Owner.
2. Must populate the *Memory region attributes* field in the transaction descriptor as specified in 8.11.4 *Memory region attributes usage*.
3. Must populate the *Flags* field specified in the transaction descriptor as specified in 8.12.4 *Flags usage*.
4. Must populate the *Handle* field specified in the transaction descriptor as specified in 8.12.1 *Handle usage*.
5. Must populate the *Tag* field specified in the transaction descriptor as specified in 8.12.2 *Tag usage*.
6. Must populate the *Endpoint memory access descriptor count* & *Endpoint memory access descriptor array* fields in the transaction descriptor as specified in 8.12.3.3 *Relayer usage*.

14.5.1.2 Callee responsibilities

1. Must return *INVALID_PARAMETERS* if any field in the transaction descriptor has been incorrectly encoded.
2. Must return *NO_MEMORY* if there is not enough memory to use the memory region description provided by the Relayer.

3. Must transfer ownership of the RX buffer back to the producer if it was used to transmit the transaction descriptor by the Relayer. Also see 4.2.2.4.2 *Transfer of buffer ownership*.

14.6 FFA_MEM_RELINQUISH

Description

- Starts a transaction to transfer access to a shared or lent memory region from a Borrower back to its Owner.
- Valid FF-A instances and conduits are listed in Table 14.26.
- Syntax of this function is described in Table 14.27.
- Successful completion of this function is indicated through the invocation of the FFA_SUCCESS function by the callee without any further parameters.
- Encoding of error code in the FFA_ERROR function is described in Table 14.28.

Table 14.25: Descriptor to relinquish a memory region

Field	Byte length	Byte offset	Description
Handle	8	0	• Globally unique Handle to identify a memory region.
Flags	4	8	• Bit[0]: Zero memory after relinquish flag. – This flag specifies if the Relayer must clear memory region contents after unmapping it from the translation regime of the Borrower. ∗ b'0: Relayer must not zero the memory region contents. ∗ b'1: Relayer must zero the memory region contents. – If the memory region was lent to multiple Borrowers, the Relayer must clear memory region contents after unmapping it from the translation regime of each Borrower, if any Borrower including the caller sets this flag. – MBZ if the memory region was shared, else the Relayer must return *INVALID_PARAMETERS*. – MBZ if the Borrower has Read-only access to the memory region, else the Relayer must return *DENIED*. – Relayer must fulfill memory zeroing requirements listed in 8.12.4 *Flags usage*.

Field	Byte length	Byte offset	Description
			• Bit[1]: Operation time slicing flag. – This flag specifies if the Relayer can time slice this operation. ∗ b'0: Relayer must not time slice this operation . ∗ b'1: Relayer can time slice this operation. • MBZ if the Relayer does not support time slicing of memory management operations (see 16.2.3 *Time slicing of memory management operations*).
			• Bit[31:2]: Reserved (MBZ).
Endpoint count	4	12	• Count of endpoints.
Endpoint array	–	16	• Array of endpoint IDs. Each entry contains a 16-bit ID.

Table 14.26: FFA_MEM_RELINQUISH instances and conduits

Config No.	FF-A instance	Valid conduits
1	Secure and Non-secure physical	SMC, ERET
2	Secure and Non-secure virtual	SMC, HVC, SVC

Table 14.27: FFA_MEM_RELINQUISH function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000076.
Other Parameter registers	w1-w7 x1-x7	• Reserved (MBZ).

Table 14.28: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS. • DENIED. • NO_MEMORY. • ABORTED.

14.6.1 Component responsibilities for FFA_MEM_RELINQUISH

This interface is used by a Borrower endpoint to inform the Relayer that it is relinquishing access to a memory region that was lent or shared with it earlier. The memory region is identified by its *Handle*.

Transaction details are populated in the descriptor specified in Table 14.25 as follows.

- The Handle and list of Borrower endpoints is populated in the descriptor described in Table 14.25 in the TX buffer of the caller.

  If the caller is a *proxy endpoint*, then the identity and count of the *Stream* endpoints on whose behalf it is relinquishing the memory region must be specified in the *Endpoint count* and *Endpoint array* fields in the descriptor.

  If the caller is a *PE endpoint* Borrower, then it must specify its ID in the *Endpoint array* field in the descriptor.

- The caller could use the *Flags* field to request the Relayer to zero the memory region after it has been unmapped from its translation regime or time slice the unmapping operation.

Responsibilities of the:

- Borrower are listed in 14.6.1.1 *Borrower responsibilities*.
- Relayer are listed in 14.6.1.2 *Relayer responsibilities*.

Time slicing of this ABI invocation must be implemented by the Borrower and Relayer as specified in 16.2.3 *Time slicing of memory management operations*.

14.6.1.1 Borrower responsibilities

1. Must ensure it has access to the memory region identified by the *Handle* parameter.
2. Must ensure it is either the Borrower of the memory region or the proxy endpoint acting on behalf of one or more Stream endpoints who are the Borrowers instead.
3. Must implement support for handling all error status codes that can be returned on completion of this interface.

14.6.1.2 Relayer responsibilities

1. Must ensure that the *Handle* provided by the Borrower is valid and associated with a memory region it can access. Must return *INVALID_PARAMETERS* in case of an error.
2. Must ensure that the *Flags* parameter is correctly encoded in the descriptor and the identities of Borrower endpoints are valid. Must return *INVALID_PARAMETERS* in case of an error.
3. Must ensure that the *Endpoint count* field has a value > *0*. Must return *INVALID_PARAMETERS* in case of an error.
4. Must ensure that the memory region is in the *!Owner-SA* or *!Owner-EA* state (see Table 8.4) for all Borrower endpoints specified by the caller. Must return *DENIED* in case of an error.

5. Must ensure that the memory region is unmapped from the translation regime of the Borrower (that is, it enters the *!Owner-NA* state (see Table 8.4)) only if it has been relinquished as many times as it has been retrieved by the Borrower.

   The memory region must be unmapped from the translation regime of the Borrower managed by the Relayer as specified in 8.3 *Address translation regimes*.

   If the caller is a proxy endpoint for a Stream endpoint then the memory region must be unmapped from the stage 2 translation tables corresponding to the SEPID.

   The Relayer must update internal state of the Borrower associated with the memory region to *!Owner-NA*.

   The Relayer must return *NO_MEMORY* if there is not enough memory to unmap the memory region.

6. Must clear the contents of the memory region after unmapping it if *bit[0]* is set in the *Flags* parameter.

14.7 FFA_MEM_RECLAIM

Description

- Restores exclusive access to a memory region back to its Owner.
- Valid FF-A instances and conduits are listed in Table 14.30.
- Syntax of this function is described in Table 14.31.
- Successful completion of this function is indicated through the invocation of the FFA_SUCCESS function by the callee.
- Encoding of error code in the FFA_ERROR function is described in Table 14.32.

Table 14.30: FFA_MEM_RECLAIM instances and conduits

Config No.	FF-A instance	Valid conduits
1	Secure and Non-secure physical	SMC, ERET
2	Secure and Non-secure virtual	SMC, HVC, SVC

Table 14.31: FFA_MEM_RECLAIM function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000077.
uint64 Handle	w1/w2	• Globally unique Handle to identify the memory region (see 8.10.2 *Memory region handle*).
uint32 Flags	w3	• Bit[0]: Zero memory before reclaim flag. – This flag specifies if the Relayer must clear memory region contents before mapping it in the Owner translation regime. ∗ b'0: Relayer must not zero the memory region contents. ∗ b'1: Relayer must zero the memory region contents. – Relayer must fulfill memory zeroing requirements listed in in 8.12.4 *Flags usage*. – MBZ if the Owner has Read-only access to the memory region, else the Relayer must return *DENIED*.

Parameter	Register	Value
		• Bit[1]: Operation time slicing flag. – This flag specifies if the Relayer can time slice this operation. ∗ b'0: Relayer must not time slice this operation. ∗ b'1: Relayer can time slice this operation. • MBZ if the Relayer does not support time slicing of memory management operations (see 16.2.3 *Time slicing of memory management operations*) .
		• Bit[31:2]: Reserved (MBZ).
Other Parameter registers	w4-w7 x4-x7	• Reserved (MBZ).

Table 14.32: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS. • DENIED. • NO_MEMORY. • ABORTED.

14.7.1 Component responsibilities for FFA_MEM_RECLAIM

This interface is used in the following ways.

1. To complete a transaction to relinquish a memory region owned by the caller endpoint. Borrowers use the *FFA_MEM_RELINQUISH* interface to relinquish access to the memory region. The Owner uses this interface to reclaim exclusive access to the memory region.
2. To abort an in-progress transaction to donate, lend or share a memory region owned by the caller endpoint. If any Receiver endpoint is unable to accept the transaction and the memory region is not mapped into the translation regime of any other Receiver endpoint, the Owner can use this transaction to reclaim exclusive access to the memory region.

Responsibilities of the:

- Owner are listed in 14.7.1.1 *Owner responsibilities*.
- Relayer are listed in 14.7.1.2 *Relayer responsibilities*.

Time slicing of this ABI invocation must be implemented by the Owner and Relayer as specified in 16.2.3 *Time slicing of memory management operations*.

14.7.1.1 Owner responsibilities

1. Must ensure it is the Owner of the memory region identified by the *Handle* parameter.
2. Must ensure that access to the memory region has been relinquished by all Borrowers.

3. Must implement support for handling all error status codes that can be returned on completion of this interface.

14.7.1.2 Relayer responsibilities

1. Must ensure that the *Handle* provided by the Owner is valid and associated with a memory region it owns. Must return *INVALID_PARAMETERS* in case of an error.
2. Must ensure that the *Flags* parameter is correctly encoded. Must return *INVALID_PARAMETERS* in case of an error.
3. Must ensure that the memory region is in the *!Owner-NA* state (see Table 8.4) for all the Receiver endpoints managed by the Relayer and associated with the memory region.

   If one or more Borrowers are Stream endpoints associated with an *independent* peripheral device then in this case:

   1. Each Borrower must relinquish access to the memory region through an IMPLEMENTATION DEFINED mechanism.
   2. The Relayer must unmap the memory region from the stage 2 translation tables identified by the SEPID.

   Must return *DENIED* in case of an error.
4. Must clear the contents of the memory region if *bit[0]* is set in the *Flags* parameter.
5. If the state of the memory region for the Owner is *Owner-NA*, this implies that the region was lent. The Relayer must map the memory region in the translation regime of the Owner as specified in 8.3 *Address translation regimes*.

   The mapping must be created at the same address range and with the same memory region properties as those when the *FFA_MEM_LEND* interface was invoked.

   Must return *NO_MEMORY* in case there is not enough memory to create the mapping in the Owner translation regime.
6. If the state of the memory region for the Owner is *Owner-SA* this implies that the region was shared. The Relayer must map the memory region in the translation regime of the Owner as specified in 8.3 *Address translation regimes*.

   The mapping must be created at the same address range and with the same memory region properties as those when the *FFA_MEM_SHARE* interface was invoked.

   Must return *NO_MEMORY* in case there is not enough memory to change the mapping in the Owner translation regime.
7. If a VM is the Owner and the Borrower is an SP or SEPID associated with a Secure Stream ID, the Hypervisor must forward an invocation of this interface to the SPM.

   This must be done by invoking this interface at the Non-secure physical FF-A instance with the same parameter values specified by the Owner at the Non-secure virtual FF-A instance.
8. If the call executes successfuly, the state of the memory region for the Owner must transition to *Owner-EA*.

Chapter 15
Notification interfaces

15.1 FFA_NOTIFICATION_BITMAP_CREATE

Description

- This ABI is invoked by the Hypervisor at the Non-secure physical FF-A instance with the SMC conduit to request the SPMC to create the *SP* and *SPM framework* notifications bitmap for the VM specified in the *VM ID* input parameter. Also see 7.3 *Notification bitmap setup*.
- Valid FF-A instances and conduits are listed in Table 15.2.
- Syntax of this function is described in Table 15.3.
- Returns FFA_SUCCESS without any further parameters on successful completion.
- Encoding of error codes in the FFA_ERROR function is described in Table 15.4.

Table 15.2: FFA_NOTIFICATION_BITMAP_CREATE instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure physical	SMC
2	Secure physical	ERET

Table 15.3: FFA_NOTIFICATION_BITMAP_CREATE function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x8400007D.
uint32 VM ID	w1	• ID of VM for which a bitmap must be created in the Secure world to enable SPs to send notifications to this VM. – Bit[31:16]: Reserved and MBZ. – Bit[15:0]: VM ID.
uint32 vCPU count	w2	• Number of vCPUs implemented by the VM.
Other Parameter registers	w3-w7 x3-x7	• Reserved (MBZ).

Table 15.4: Encoding of return codes

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: Unrecognized VM ID. • NOT_SUPPORTED: This function is not implemented at this FF-A instance. • DENIED: Notification bitmap is already created. • NO_MEMORY: There is not enough memory to allocate notification bitmap.

15.2 FFA_NOTIFICATION_BITMAP_DESTROY

Description

- This ABI is invoked by the Hypervisor at the Non-secure physical FF-A instance with the SMC conduit to request the SPMC to destroy the *SP* and *SPM framework* notifications bitmap for the VM specified in the *VM ID* input parameter. Also see 7.3 *Notification bitmap setup*.
- Valid FF-A instances and conduits are listed in Table 15.6.
- Syntax of this function is described in Table 15.7.
- Returns FFA_SUCCESS without any further parameters on successful completion.
- Encoding of error codes in the FFA_ERROR function is described in Table 15.8.

Table 15.6: FFA_NOTIFICATION_BITMAP_DESTROY instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure physical	SMC
2	Secure physical	ERET

Table 15.7: FFA_NOTIFICATION_BITMAP_DESTROY function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x8400007E.
uint32 VM ID	w1	• ID of VM whose notification bitmap in the Secure world must be destroyed to prevent SPs to send notifications to this VM. – Bit[31:16]: Reserved and MBZ. – Bit[15:0]: VM ID.
Other Parameter registers	w2-w7 x2-x7	• Reserved (MBZ).

Table 15.8: Encoding of return codes

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: Unrecognized partition ID. • NOT_SUPPORTED: This function is not implemented at this FF-A instance. • DENIED: Notification bitmap is not registered or is registered but not in a masked and non-pending state.

15.3 FFA_NOTIFICATION_BIND

Description

- This ABI is invoked by an endpoint at a virtual FF-A instance with the SMC, HVC or SVC conduits to request the partition manager to bind notifications specified in the *Notification bitmap* parameter to the Sender endpoint. Also see 7.4.2 *Notification binding*.
- This ABI is invoked by the Hypervisor at the Non-secure physical FF-A instance with the SMC conduit to request the SPMC to bind SP notifications specified in the *Notification bitmap* parameter to the SP specified in the *Sender endpoint ID* parameter.
- Valid FF-A instances and conduits are listed in Table 15.10.
- Syntax of this function is described in Table 15.11.
- Returns FFA_SUCCESS without any further parameters on successful completion.
- Encoding of error codes in the FFA_ERROR function is described in Table 15.12.

Table 15.10: FFA_NOTIFICATION_BIND instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure virtual	SMC, HVC
2	Secure virtual	SMC, HVC, SVC
3	Non-secure physical	SMC
4	Secure physical	ERET

Table 15.11: FFA_NOTIFICATION_BIND function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x8400007F.
uint32 Sender/Receiver IDs	w1	• Sender and Receiver endpoint IDs. – Bit[31:16]: Sender endpoint ID. – Bit[15:0]: Receiver endpoint ID.
uint32 Flags	w2	• Notification flags. – Bit[1]: Per-vCPU notification flag (see 7.4.2 *Notification binding*). ∗ b'1: All notifications in the bitmap are per-vCPU notifications ∗ b'0: All notifications in the bitmap are global notifications – Bit[31:1]: Reserved (MBZ).

Parameter	Register	Value
uint32 Notification bitmap Lo	w3	• Bits[31:0] of a bitmap with one or more set bits to identify the notifications which the Sender endpoint is allowed to signal. • For each bit in the bitmap, if the value is: – b'1: The Sender endpoint can signal this notification. – b'0: Has no effect.
uint32 Notification bitmap Hi	w4	• Bits[63:32] of a bitmap with one or more set bits to identify the notifications which the Sender endpoint is allowed to signal. • For each bit in the bitmap, if the value is: – b'1: The Sender endpoint can signal this notification. – b'0: Has no effect.
Other Parameter registers	w5-w7 x5-x7	• Reserved (MBZ).

Table 15.12: Encoding of return codes

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: Unrecognized partition ID or invalid bitmap. • NOT_SUPPORTED: This function is not implemented at this FF-A instance. • DENIED: At least one notification is bound to another Sender or is currently pending. • ABORTED: Sender partition ran into an unexpected error and has aborted.

15.4 FFA_NOTIFICATION_UNBIND

Description

- This ABI is invoked by an endpoint at a virtual FF-A instance with the SMC, HVC or SVC conduits to request the partition manager to unbind notifications specified in the *Notification bitmap* parameter to the Sender endpoint. Also see 7.4.2 *Notification binding*.
- This ABI is invoked by the Hypervisor at the Non-secure physical FF-A instance with the SMC conduit to request the SPMC to unbind SP notifications specified in the *Notification bitmap* parameter to the SP specified in the *Sender endpoint ID* parameter.
- Valid FF-A instances and conduits are listed in Table 15.14.
- Syntax of this function is described in Table 15.15.
- Returns FFA_SUCCESS without any further parameters on successful completion.
- Encoding of error codes in the FFA_ERROR function is described in Table 15.16.

Table 15.14: FFA_NOTIFICATION_UNBIND instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure virtual	SMC, HVC
2	Secure virtual	SMC, HVC, SVC
3	Non-secure physical	SMC
4	Secure physical	ERET

Table 15.15: FFA_NOTIFICATION_UNBIND function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000080.
uint32 Sender/Receiver IDs	w1	• Sender and Receiver endpoint IDs. – Bit[31:16]: Sender endpoint ID. – Bit[15:0]: Receiver endpoint ID.
uint32/uint64 Reserved	w2/x2	• Reserved for future use (MBZ).
uint32 Notification bitmap Lo	w3	• Bits[31:0] of a bitmap with one or more set bits to identify the notifications which the Sender endpoint is not allowed to signal. • For each bit in the bitmap, if the value is: – b'1: The Sender endpoint cannot signal this notification. – b'0: Has no effect.

Parameter	Register	Value
uint32 Notification bitmap Hi	w4	• Bits[63:32] of a bitmap with one or more set bits to identify the notifications which the Sender endpoint is not allowed to signal. • For each bit in the bitmap, if the value is: – b'1: The Sender endpoint cannot signal this notification. – b'0: Has no effect.
Other Parameter registers	w5-w7 x5-x7	• Reserved (MBZ).

Table 15.16: Encoding of return codes

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: Unrecognized partition ID or invalid bitmap. • NOT_SUPPORTED: This function is not implemented at this FF-A instance. • DENIED: At least one notification is bound to another Sender or is currently pending. • ABORTED: Sender partition ran into an unexpected error and has aborted.

15.5 FFA_NOTIFICATION_SET

Description

- This ABI is invoked by an endpoint at a virtual FF-A instance with the SMC, HVC or SVC conduits to request the partition manager to signal notifications specified in the *Notification bitmap* parameter to the Sender endpoint. Also see 7.5 *Notification signaling*.
- This ABI is invoked by the Hypervisor at the Non-secure physical FF-A instance with the SMC conduit to request the SPMC to signal VM notifications specified in the *Notification bitmap* parameter to the SP specified in the *Receiver endpoint ID* parameter on behalf of the VM specified in the *Sender endpoint ID* parameter.
- Valid FF-A instances and conduits are listed in Table 15.18.
- Syntax of this function is described in Table 15.19.
- Returns FFA_SUCCESS without any further parameters on successful completion.
- Encoding of error codes in the FFA_ERROR function is described in Table 15.20.

Table 15.18: FFA_NOTIFICATION_SET instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure virtual	SMC, HVC
2	Secure virtual	SMC, HVC, SVC
3	Non-secure physical	SMC
4	Secure physical	ERET

Table 15.19: FFA_NOTIFICATION_SET function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000081.
uint32 Sender/Receiver IDs	w1	• Sender and Receiver endpoint IDs. – Bit[31:16]: Sender endpoint ID. – Bit[15:0]: Receiver endpoint ID.

Parameter	Register	Value
uint32 Flags	w2	• Flags. – Bit[0]: Receiver vCPU ID valid flag. ∗ b'0: The notifications in the bitmap are global notifications. Receiver vCPU ID field is not used and MBZ. ∗ b'1: The notifications in the bitmap are per-vCPU notifications and must be signaled to the specified Receiver vCPU ID. – Bit[1]: Delay *Schedule Receiver* interrupt flag. See 15.5.1 *Delay Schedule Receiver interrupt flag*. – Bit[15:2]: Reserved MBZ. – Bit[31:16]: Receiver vCPU ID.
uint32 Notification bitmap Lo	w3	• Bits[31:0] of a bitmap with one or more set bits to identify the notifications which must be signaled to the Receiver endpoint. • For each bit in the bitmap, if the value is: – b'1: The notification corresponding to this bit position must be signaled to the Receiver. – b'0: The notification corresponding to this bit position must not be signaled to the Receiver.
uint32 Notification bitmap Hi	w4	• Bits[63:32] of a bitmap with one or more set bits to identify the notifications which must be signaled to the Receiver endpoint. • For each bit in the bitmap, if the value is: – b'1: The notification corresponding to this bit position must be signaled to the Receiver. – b'0: The notification corresponding to this bit position must not be signaled to the Receiver.
Other Parameter registers	w5-w7 x5-x7	• Reserved (MBZ).

Table 15.20: Encoding of return codes

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: Unrecognized partition ID. • NOT_SUPPORTED: This function is not implemented at this FF-A instance. • DENIED: – Sender is not permitted to signal at least one notification to the Receiver. – Receiver does not support receipt of notifications. • ABORTED: Receiver partition ran into an unexpected error and has aborted.

15.5.1 Delay Schedule Receiver interrupt flag

It is possible that the *Schedule Receiver* interrupt either preempts or triggers a managed exit of the Sender execution context immediately upon the completion of an FFA_NOTIFICATION_SET invocation. This might be undesirable for the Sender.

The *Delay Schedule Receiver interrupt* flag enables the Sender execution context to instruct the partition manager that this interrupt must be pended when it next enters the *waiting* state.

If the Sender execution context does not set this flag, the partition manager can pend the *Schedule Receiver interrupt* as per its IMPLEMENTATION DEFINED policy.

15.6 FFA_NOTIFICATION_GET

Description

- This ABI is invoked by an endpoint at a virtual FF-A instance with the SMC, HVC or SVC conduits to request the partition manager to retrieve notifications pending in notification bitmaps specified in the *Flags* parameter. Also see 7.5 *Notification signaling*.
- This ABI is invoked by the Hypervisor at the Non-secure physical FF-A instance with the SMC conduit to request the SPMC to return pending SP or SPM Framework notifications as specified in the *Flags* parameter for the VM specified in the *Receiver endpoint ID* parameter. The *Receiver vCPU ID* parameter is used to return any pending per-vCPU notifications.
- Valid FF-A instances and conduits are listed in Table 15.22.
- Syntax of this function is described in Table 15.23.
- Encoding of result parameters in the FFA_SUCCESS function is described in Table 15.24.
- Returns FFA_SUCCESS without any further parameters on successful completion.
- Encoding of error codes in the FFA_ERROR function is described in Table 15.25.

Table 15.22: FFA_NOTIFICATION_GET instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure virtual	SMC, HVC
2	Secure virtual	SMC, HVC, SVC
3	Non-secure physical	SMC
4	Secure physical	ERET

Table 15.23: FFA_NOTIFICATION_GET function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000082.
uint32 Receiver ID	w1	• Receiver endpoint and vCPU ID. – Bit[31:16]: Receiver endpoint ID. – Bit[15:0]: Receiver vCPU ID.

Parameter	Register	Value
uint32 Flags	w2	• Bit[0]: Receiver's SP notifications bitmap identifier. – b'1: Return bitmap for notifications pended by SPs. – b'0: Do not return bitmap for notifications pended by SPs. • Bit[1]: Receiver's VM notifications bitmap identifier. This bit MBZ at the Non-secure physical FF-A instance. – b'1: Return bitmap for notifications pended by VMs. – b'0: Do not return bitmap for notifications pended by VMs. • Bit[2]: Receiver's SPM Framework notification bitmap identifier. – b'1: Return bitmap for notifications pended by the SPM. – b'0: Do not return bitmap for notifications pended by the SPM. • Bit[3]: Receiver's Hypervisor Framework notifications bitmap identifier. This bit MBZ at the Non-secure physical FF-A instance. – b'1: Return bitmap for notifications pended by the Hypervisor. – b'0: Do not return bitmap for notifications pended by the Hypervisor. • Bit[31:4]: Reserved (MBZ).
Other Parameter registers	w3-w7 x3-x7	• Reserved (MBZ).

Table 15.24: FFA_SUCCESS encoding

Parameter	Register	Value
uint32 SP Notifications bitmap Lo	w2	• Bits[31:0] of the SP notifications bitmap with one or more set bits to identify the notifications which are pending for the Receiver endpoint. • For each bit in the bitmap, if the value is: – b'1: The notification corresponding to this bit position is pending for the Receiver – b'0: The notification corresponding to this bit position is not pending for the Receiver. • Caller must ignore this field if *Bit[0]* in the *Flags* field was not set.

Parameter	Register	Value
uint32 SP Notifications bitmap Hi	w3	• Bits[63:32] of the SP notifications bitmap with one or more set bits to identify the notifications which are pending for the Receiver endpoint. • For each bit in the bitmap, if the value is: – b'1: The notification corresponding to this bit position is pending for the Receiver – b'0: The notification corresponding to this bit position is not pending for the Receiver. • Caller must ignore this field if *Bit[0]* in the *Flags* field was not set.
uint32 VM Notifications bitmap Lo	w4	• Bits[31:0] of the VM notifications bitmap with one or more set bits to identify the notifications which are pending for the Receiver endpoint. • For each bit in the bitmap, if the value is: – b'1: The notification corresponding to this bit position is pending for the Receiver – b'0: The notification corresponding to this bit position is not pending for the Receiver. • Caller must ignore this field if *Bit[1]* in the *Flags* field was not set.
uint32 VM Notifications bitmap Hi	w5	• Bits[63:32] of the VM notifications bitmap with one or more set bits to identify the notifications which are pending for the Receiver endpoint. • For each bit in the bitmap, if the value is: – b'1: The notification corresponding to this bit position is pending for the Receiver – b'0: The notification corresponding to this bit position is not pending for the Receiver. • Caller must ignore this field if *Bit[1]* in the *Flags* field was not set.
uint32 Framework Notifications bitmap Lo	w6	• Bits[31:0] of the Framework notifications bitmap with one or more set bits to identify the notifications which are pending for the Receiver endpoint as sent by the SPM. • These 32 bits will be set by the SPM and reflect notifications regarding events in the secure world. • Caller must ignore this field if *Bit[2]* in the *Flags* field was not set.

Parameter	Register	Value
uint32 Framework Notifications bitmap Hi	w7	• Bits[63:32] of the Framework notifications bitmap with one or more set bits to identify the notifications which are pending for the Receiver endpoint as sent by the Hypervisor. • These 32 bits will be set by the Hypervisor and reflect notifications regarding events in the normal world. • Caller must ignore this field if *Bit[3]* in the *Flags* field was not set.

Table 15.25: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: Unrecognized partition ID or incorrectly encoded Flags parameter. • NOT_SUPPORTED: This function is not implemented at this FF-A instance.

15.7 FFA_NOTIFICATION_INFO_GET

Description

- This ABI is invoked by an endpoint at a virtual FF-A instance with the SMC, HVC or SVC conduits to request the partition manager to return the list of endpoints that have pending notifications. Also see 7.5 *Notification signaling*.
  - The partition manager returns the list of those endpoints whose schedulers are implemented in the calling endpoint.
  - If an endpoint has a pending per-vCPU notification, the ID of the target vCPU is returned as well.
- This ABI is invoked by the Hypervisor at the Non-secure physical FF-A instance with the SMC conduit to request the SPMC to return the list of SPs that have pending notifications.
- Valid FF-A instances and conduits are listed in Table 15.27.
- Syntax of this function is described in Table 15.28.
- Encoding of result parameters in the FFA_SUCCESS function is described in Table 15.29.
- Encoding of error codes in the FFA_ERROR function is described in Table 15.30.

Table 15.27: FFA_NOTIFICATION_INFO_GET instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure physical	SMC
2	Secure physical	ERET
3	Non-secure virtual	SMC, HVC
4	Secure virtual	SMC, HVC, SVC

Table 15.28: FFA_NOTIFICATION_INFO_GET function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000083. • 0xC4000083.
Other Parameter registers	w1-w7 x1-x7	• Reserved (MBZ).

Table 15.29: FFA_SUCCESS encoding

Parameter	Register	Value
uint32/uint64 Pending notification flags	w2/x2	• See 15.7.1 *Parameter encoding*.
uint32/uint64 ID lists	w3-w7 x3-x7	• See 15.7.1 *Parameter encoding*.

Table 15.30: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• RETRY: There are no notifications for which information has not already been retrieved • NOT_SUPPORTED: This function is not implemented at this FF-A instance.

15.7.1 Parameter encoding

The following information is encoded in ID lists registers *w3/x3-w7/x7*

- A list of endpoint IDs. Each Endpoint has one or more pending notifications.
- Optionally, for a given endpoint ID, a list of vCPU IDs. Each vCPU has a pending per-vCPU notification. If no vCPU ID is specified, then the endpoint has pending global notifications.

A endpoint or vCPU ID is 16 bits in length.

- With the SMC32 calling convention, the ID registers can accommodate 10 IDs.
- With the SMC64 calling convention, the ID registers can accommodate 20 IDs.

IDs are returned in *w3/x3-w7/x7* in lists. The first ID in each list is a Receiver endpoint ID. If the Receiver endpoint has pending per-vCPU notifications, subsequent IDs in the list are vCPU IDs.

The ID lists are tightly packed in registers as follows.

- The start of the first list is encoded in w3/x3. Subsequent lists follow in the same or a higher numbered register.
- The IDs are encoded in the little-endian byte order.

Each list can have a maximum of 4 IDs. If an endpoint has pending per-vCPU notifications for more than 3 vCPUs, it creates more than 1 list to encode all the vCPU IDs.

The number of lists and the number of IDs (endpoint and vCPU) in each list is specified in the *Pending notification flags* parameter in *w2/x2*.

- Bit[0]: More pending notifications flag.
  - b'0: Caller has retrieved all ID lists of Receiver endpoints with pending notifications.
  - b'1: Caller has not retrieved all ID lists. It must invoke this interface again to retrieve the remaining lists.
- Bit[7:1]: Reserved MBZ.
- Bit[11:8]: Count of lists returned in ID lists registers.
- Bit[((2 x *i*) - 1) + *off*: (2 x (*i* - 1)) + *off*]: Count of IDs in list *i* where,
  - 1 <= *i* <= 10 if the SMC32 convention is used.
  - 1 <= *i* <= 20 if the SMC64 convention is used.
  - *off* is the starting bit offset = 12

  Count of IDs in unused lists MBZ.
- Bit[63:52]: Reserved (MBZ), if the SMC64 convention is used.

An ID list is provided by the partition manager only once i.e., a list retrieved in one invocation of this interface cannot be retrieved again in a subsequent invocation.

Chapter 16
Appendix

16.1 S-EL0 & User mode partitions

S-EL0 & Secure User mode partitions are used to achieve isolation among Secure services on Armv8.3 and earlier architecture versions. They could host one or more device drivers to control hardware that is only accessible from the Secure world. Normal world accesses these drivers through the message passing interfaces described in this specification. An example use case of S-EL0 partitions is described in 16.1.1 *UEFI PI Standalone Management Mode partitions*.

16.1.1 UEFI PI Standalone Management Mode partitions

Standalone management mode (STMM) is described in [9] as a processor architecture agnostic, sandboxed secure execution environment. It is meant to be used for device drivers that cannot be implemented in the OS kernel but are required during run-time.

On Armv8-A systems, STMM is implemented in a S-EL0 partition to constraint its visibility of the system address map and physical interrupts. This isolation enables a more robust Secure firmware implementation. This design is better from a security perspective than a design where STMM drivers are implemented in EL3.

Furthermore, execution in EL3 always runs to completion. Isolation of STMM drivers in an SP enables Secure firmware to transparently preempt them in response to OS Kernel interrupts and resume them once the interrupt has been handled. For some use cases, this prevents an adverse impact on OS responsiveness that could happen with a run to completion model.

16.1.1.1 FF-A usage to access STMM services

This section provides guidance around how services that would be typically implemented in EL3, can be implemented in multiple STMM S-EL0 partitions and accessed through FF-A interfaces. This guidance is based on certain assumptions about the Standalone management mode as follows.

- A STMM driver is neither reentrant nor thread safe but its single execution context can run on any PE in the system. Hence, a STMM S-EL0 partition is considered to be a *UP migrate capable* partition.
- STMM services are accessed from the UEFI runtime environment in the Normal world through direct Partition messages (see 4.4 *Direct messaging usage*). A component called the MM communication driver is used for this purpose.
- STMM services can be accessed in response to an interrupt targeted to EL3 apart from the UEFI runtime environment.
- There are no dependencies between STMM partitions. One partition does not access services of another partition.
- A STMM partition processes one request at a time and is incapable of having multiple outstanding requests at any point of time.

The MM interface specification [10] specifies the **MM_COMMUNICATE** interface that enables the Normal world to access driver services implemented in a single STMM S-EL0 partition.

The Framework enables deployment of multiple STMM S-EL0 SPs through the use of,

1. An appropriate run-time and scheduling model described in Chapter 5 *Partition runtime models* and 6.5 *SP scheduling models* respectively.
2. Interfaces to manage the instruction and data access permissions of memory regions accessible by a STMM S-EL0 SP. This management is typically required during partition initialization (also see [11]). These interfaces are described in the following sections.
    - 16.1.1.2 *FFA_MEM_PERM_GET*.
    - 16.1.1.3 *FFA_MEM_PERM_SET*.

Some example flows to illustrate common aspects of interaction with a STMM SP based on the preceding concepts are as follows.

- Figure 16.1 describes how the MM communication driver can discover presence of STMM SPs and their properties. It is assumed that:

  - All STMM SPs share a MM service UUID. This UUID is used by MM communication driver to discover all the STMM SPs.
  - Each STMM SP specifies this UUID, its run-time model, memory regions, devices etc. in its partition manifest.
  - The MM communication buffer for each STMM SP is allocated by the EFI MM communication driver.

- Figure 16.2 describes how the MM communication driver and a STMM SP can communicate using direct Partition messages and the communication buffer shared between them.
- Figure 16.3 describes how the STMM SP can be invoked in response to an interrupt.

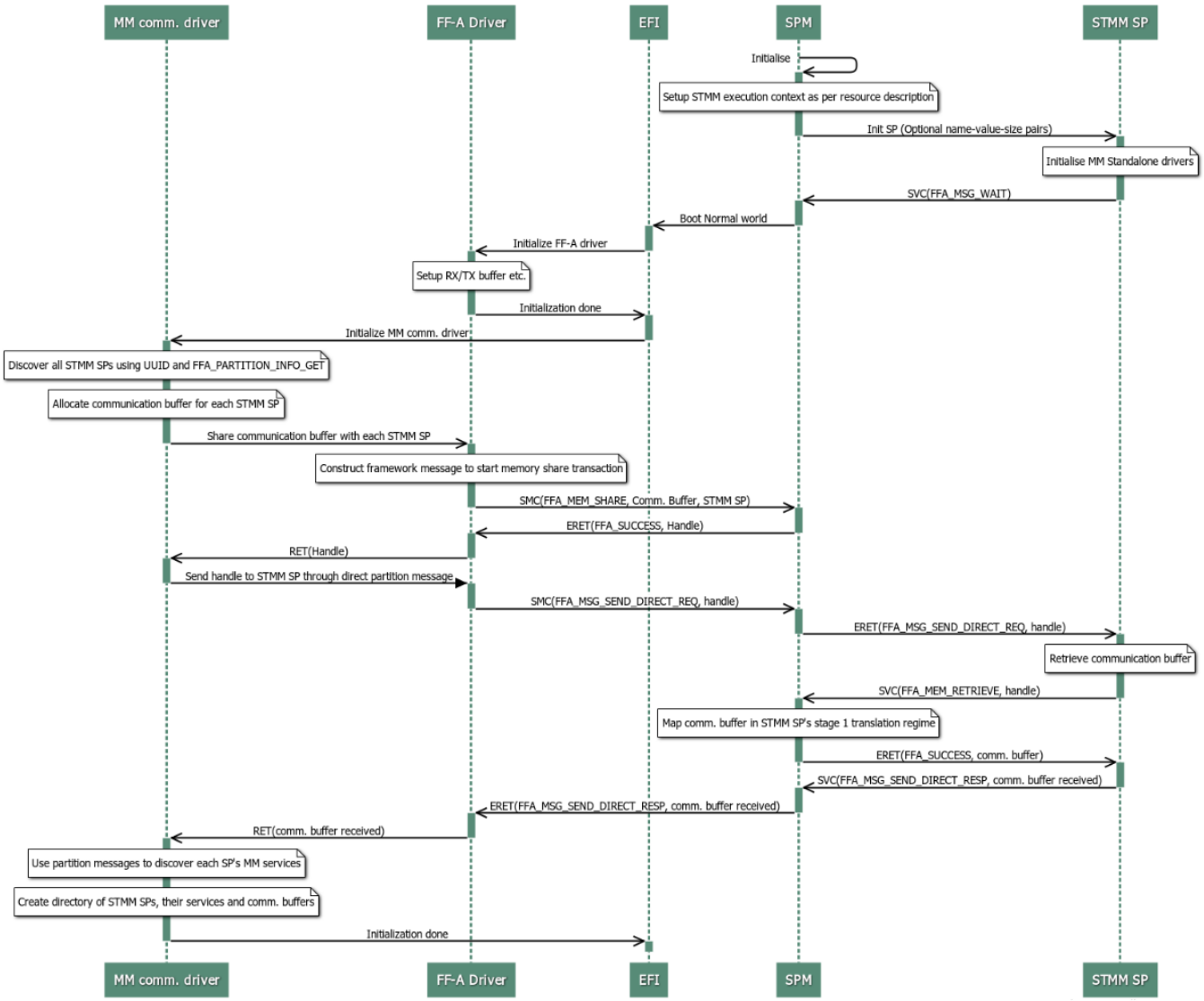

Figure 16.1: MM communication driver initialization

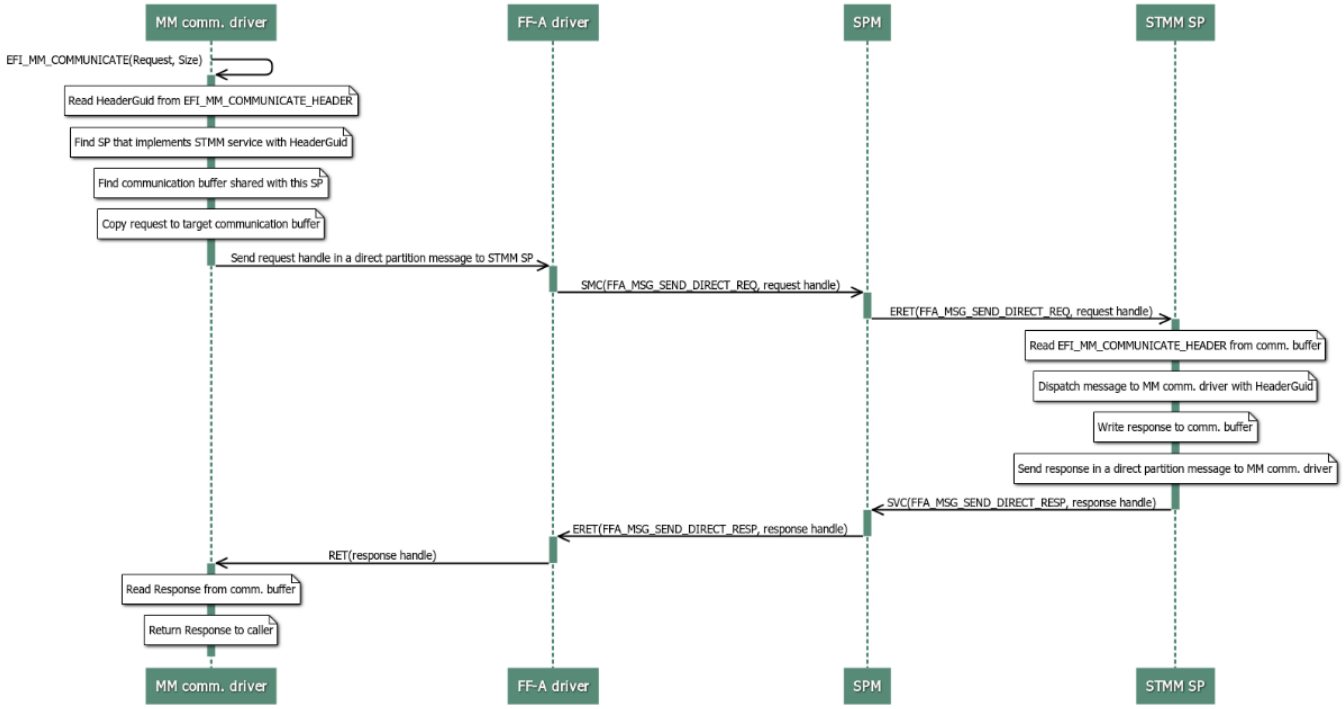

Figure 16.2: Message exchange between a STMM SP and MM communication driver

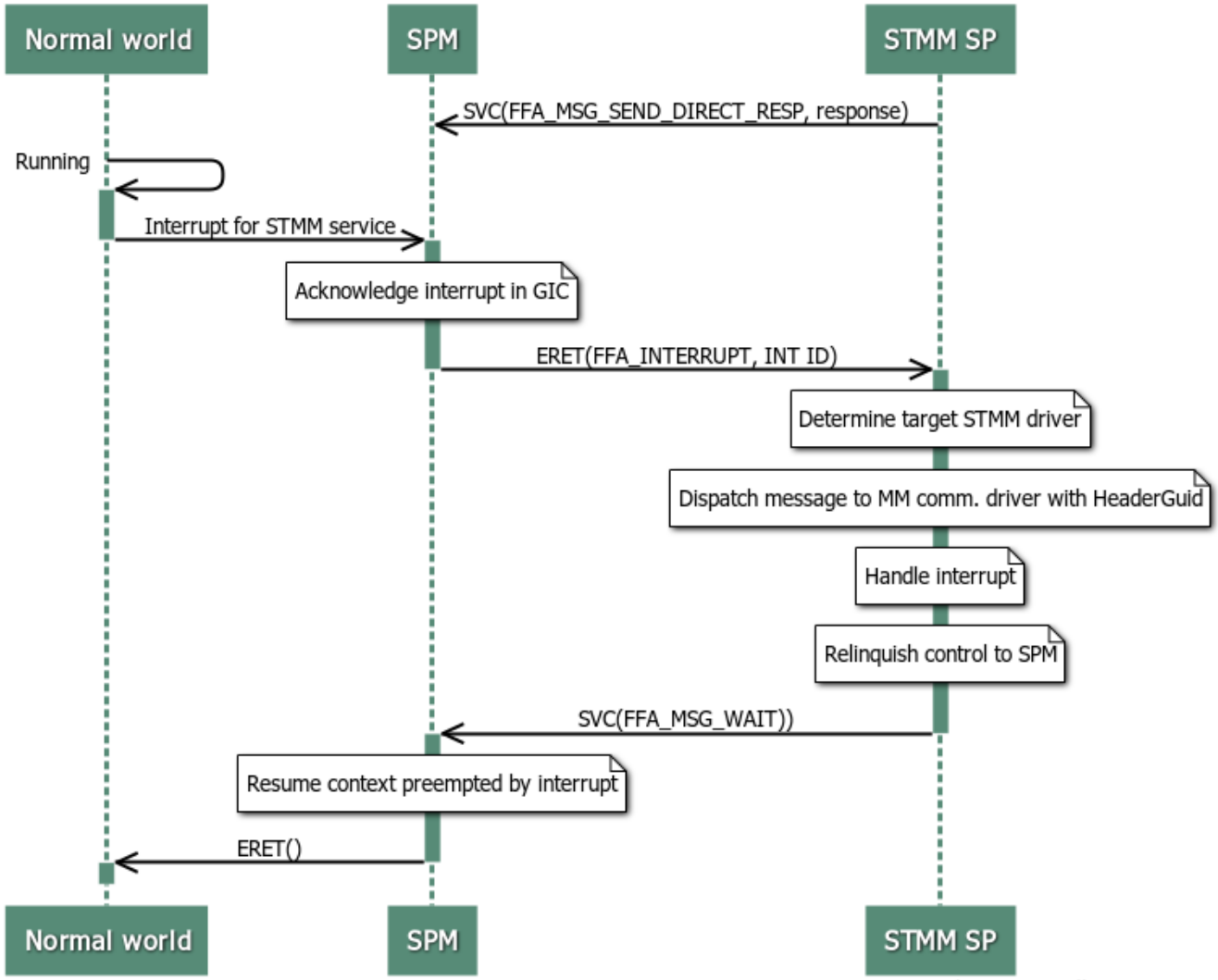

Figure 16.3: Invocation of a STMM SP in response to an interrupt

16.1.1.2 FFA_MEM_PERM_GET

Description

- This ABI is invoked at the Secure virtual FF-A instance with the SVC conduit to determine the permission attributes for a memory region accessible by the caller. Also see 16.1.1.2.1 *Overview*.
- Valid FF-A instances and conduits are listed in Table 16.2.
- Syntax of this function is described in Table 16.3.
- Encoding of result parameters in the FFA_SUCCESS function is described in Table 16.4.
- Encoding of error code in the FFA_ERROR function is described in Table 16.5.

Table 16.2: FFA_MEM_PERM_GET instances and conduits

Config No.	FF-A instance	Valid conduits
1	Secure virtual	SVC

Table 16.3: FFA_MEM_PERM_GET function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000088.
uint64 Base Address	x1	• Base VA of a translation granule whose permission attributes must be returned.
Other parameter registers	w2-w7 x2-x7	• Reserved (MBZ).

Table 16.4: FFA_SUCCESS encoding

Parameter	Register	Value
uint32 Memory permissions	w2	• Bit[1:0]: Data access permission. – b'00: No access. – b'01: Read-write access. – b'10: Reserved. – b'11: Read-only access. • Bit[2]: Instruction access permission. – b'0: Executable. – b'1: Non-executable. • Bit[31:3]: Reserved and MBZ.
Other Result registers	w3-w7 x3-x7	• Reserved (MBZ).

Table 16.5: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: Caller is not allowed to access the memory region the address lies in . • NOT_SUPPORTED: This function is not implemented at this FF-A instance.

16.1.1.2.1 Overview

FFA_MEM_PERM_GET is used to request the permission attributes of a memory region mapped in the following translation regimes of a S-EL0 SP.

- Stage 1 of the Secure EL1&0 translation regime.
- Single stage in the Secure EL2&0 translation regime.

The size of the memory region for which permission attributes are returned is equal to the translation granule size used in the applicable translation regime.

The VA specified in the *Base address* parameter is aligned to the size of the translation granule used in the translation regime. The permission attributes for this translation granule are returned by the callee. The caller determines the translation granule size through an IMPLEMENTATION DEFINED mechanism.

16.1.1.3 FFA_MEM_PERM_SET

Description

- This ABI is invoked at the Secure virtual FF-A instance with the SVC conduit to set the permission attributes for a memory region accessible by the caller. Also see 16.1.1.3.1 *Overview*.
- Valid FF-A instances and conduits are listed in Table 16.7.
- Syntax of this function is described in Table 16.8.
- Returns FFA_SUCCESS without any further parameters on successful completion.
- Encoding of error code in the FFA_ERROR function is described in Table 16.9.

Table 16.7: FFA_MEM_PERM_SET instances and conduits

Config No.	FF-A instance	Valid conduits
1	Secure virtual	SVC

Table 16.8: FFA_MEM_PERM_SET function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000089.
uint64 Base address	x1	• Base VA of a memory region whose permission attributes must be set.
uint32 Page count	w2	• Number of translation granule size pages starting from the *Base address* whose permissions must be set.
uint32 Memory permissions	w3	• Bit[1:0]: Data access permission. – b'00: No access. – b'01: Read-write access. – b'10: Reserved. – b'11: Read-only access. • Bit[2]: Instruction access permission. – b'0: Executable. – b'1: Non-executable. • Bit[31:3]: Reserved and MBZ.
Other parameter registers	w4-w7 x4-x7	• Reserved (MBZ).

Table 16.9: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: – Caller is not allowed to access the memory region the address lies in. – Memory permissions are incorrectly encoded. – Base address is not correctly aligned. • NOT_SUPPORTED: This function is not implemented at this FF-A instance.

16.1.1.3.1 Overview

FFA_MEM_PERM_SET is used to set the permission attributes of a memory region mapped in the following translation regimes of a S-EL0 SP.

- Stage 1 of the Secure EL1&0 translation regime.
- Single stage in the Secure EL2&0 translation regime.

The VA of the memory region specified in the *Base address* parameter is aligned to the size of the translation granule used in the translation regime.

The size of the memory region for which permission attributes are set is expressed as a count of pages. The size of each page is equal to the translation granule size in the applicable translation regime.

The *Memory permissions* parameter must be encoded as per the following rules.

1. A combination of attributes that mark the region with RW and Executable permissions is prohibited.
2. A request to mark a device memory region with Executable permissions is prohibited.

In case of an error, the callee preserves the original permissions of the memory regions.

16.2 Additional memory management features

16.2.1 Transmission of transaction descriptor in dynamically allocated buffers

16.2.1.1 Rationale

The transaction descriptor (see Table 8.19) is transmitted from the caller to the callee in an invocation of the following ABIs.

- *FFA_MEM_DONATE*. See 14.1 *FFA_MEM_DONATE*.
- *FFA_MEM_LEND*. See 14.2 *FFA_MEM_LEND*.
- *FFA_MEM_SHARE*. See 14.3 *FFA_MEM_SHARE*.
- *FFA_MEM_RETRIEVE_REQ*. See 14.4 *FFA_MEM_RETRIEVE_REQ*.
- *FFA_MEM_RETRIEVE_RESP*. See 14.5 *FFA_MEM_RETRIEVE_RESP*.

This version of the Framework assumes that by default, the transaction descriptor is populated in the RX/TX buffers of an endpoint, Hypervisor or SPM as follows.

- The TX buffer is used to transmit the descriptor from an endpoint to the Hypervisor or SPM.
- The TX buffer is used to transmit the descriptor from the Hypervisor to the SPM.
- The RX buffer is used to transmit the descriptor from the Hypervisor or SPM to an endpoint.
- The RX buffer is used to transmit the descriptor from the SPM to the Hypervisor.

It is possible that the size of the descriptor is larger than the RX or TX buffer. For example, an endpoint memory access descriptor entry (see Table 8.16) in the transaction descriptor could reference one or more composite memory region descriptors (see Table 8.13). The total size of the composite memory region descriptors could be larger than the RX or TX buffer.

Each FF-A component is allowed to share only a single RX/TX pair with another FF-A component (see 4.2.2 *RX/TX buffers*). It is possible that an endpoint or a partition manager cannot tolerate the latency in acquiring access to these buffers for a memory management operation on a busy system. It is also possible that other users cannot tolerate the latency to acquire access to these buffers due to an ongoing memory management operation.

16.2.1.2 Overview

This version of the Framework supports an optional feature that allows an endpoint to:

1. Dynamically allocate a separate buffer, instead of using the TX buffer, to transmit the transaction descriptor in an invocation of the following ABIs.
   - *FFA_MEM_DONATE*.
   - *FFA_MEM_LEND*.
   - *FFA_MEM_SHARE*.
   - *FFA_MEM_RETRIEVE_REQ*.
2. Use this buffer instead of the RX buffer to transmit the transaction descriptor in an invocation of the *FFA_MEM_RETRIEVE_RESP* ABI.

16.2.1.3 Description

The ability of an endpoint to use this feature depends on whether its partition manager implements support to map the dynamically allocated buffer into its translation regime. An endpoint can discover the availability of this support through the *FFA_FEATURES* interface (see 11.2 *FFA_FEATURES*).

An endpoint must follow these rules while allocating a buffer dynamically.

- The dynamically allocated buffer must use the same attributes as RX/TX buffers that are specified in 4.2.2.3 *Buffer attributes*.
- The dynamically allocated buffer must be contiguous in the address space where it is allocated.

- The dynamically allocated buffer must fulfill the size and alignment requirements listed in 2.7 *Memory granularity and alignment* to allow the partition manager to map it. The endpoint must discover these requirements by invoking the *FFA_FEATURES* interface with the function ID of the *FFA_RXTX_MAP* interface (see 11.2 *FFA_FEATURES*).

The address and size of a dynamically allocated buffer must be specified in an invocation of the following ABIs.

- *FFA_MEM_DONATE*.
- *FFA_MEM_LEND*.
- *FFA_MEM_SHARE*.
- *FFA_MEM_RETRIEVE_REQ*.

The syntax for specifying the address and size is as follows.

- The *w3/x3* register must be used to specify the VA, IPA or PA of the dynamically allocated buffer.

  A value of 0 must be specified to indicate that the TX buffer is being used.

- The *w4* register must be used to specify the size of the dynamically allocated buffer as a count of the contiguous 4K pages that constitute it.

  A value of 0 must be specified if the TX buffer is being used.

In an invocation of the *FFA_MEM_RETRIEVE_RESP* ABI:

- The partition manager must use the same buffer that was used in the counterpart *FFA_MEM_RETRIEVE_REQ* ABI invocation.
- A value of 0 must be specified in the *w3/x3* register since there is no need to specify which buffer is being used.
- A value of 0 must be specified in the *w4* register since there is no need to specify the size of the buffer is being used.

If dynamically allocated buffers are supported, a partition manager must map the dynamically allocated buffer in its translation regime on invocation, and unmap it on completion of the following ABIs.

- *FFA_MEM_DONATE*.
- *FFA_MEM_LEND*.
- *FFA_MEM_SHARE*.

The buffer must be mapped by the partition manager on invocation of the *FFA_MEM_RETRIEVE_REQ* ABI. It must be unmapped after the complete transaction descriptor has been transmitted through the invocation of the counterpart *FFA_MEM_RETRIEVE_RESP* ABI.

A partition manager must return:

- *INVALID_PARAMETERS* in the following scenarios:
  - It does not support this feature and an endpoint attempts to use it as described above.
  - The address or size of the dynamically allocated buffer is invalid.
- *NO_MEMORY* if it does not have enough memory to map the dynamically allocated buffer in its translation regime.

16.2.2 Transmission of transaction descriptor in fragments

16.2.2.1 Rationale

The size of a memory transaction descriptor (see Table 8.19) could exceed the size of the buffer used by an endpoint to transmit it. This is possible in the following scenarios.

1. An endpoint or partition manager does not implement support for dynamically allocated buffers (see 16.2.1 *Transmission of transaction descriptor in dynamically allocated buffers*). The RX/TX buffers must be used instead and cannot always accommodate the memory transaction descriptor.
2. An endpoint or partition manager do implement support for dynamically allocated buffers. In some memory management operations, the size of the memory transaction descriptor exceeds the size of the dynamically allocated buffer.

16.2.2.2 Overview

This version of the Framework supports an optional feature that:

- Allows the Sender of the transaction descriptor, to break the descriptor into equal or variable sized *fragments*, such that each fragment fits into the RX, TX or a dynamically allocated buffer.
- Adds support to transmit the first fragment of a transaction descriptor instead of the entire descriptor to the following ABIs.
  - *FFA_MEM_DONATE*. See 14.1 *FFA_MEM_DONATE*.
  - *FFA_MEM_LEND*. See 14.2 *FFA_MEM_LEND*.
  - *FFA_MEM_SHARE*. See 14.3 *FFA_MEM_SHARE*.
  - *FFA_MEM_RETRIEVE_REQ*. See 14.4 *FFA_MEM_RETRIEVE_REQ*.
  - *FFA_MEM_RETRIEVE_RESP*. See 14.5 *FFA_MEM_RETRIEVE_RESP*.
- Defines the following ABIs to transmit the remaining fragments from the Sender to the Receiver.
  - *FFA_MEM_FRAG_RX*. See 16.2.2.4 *FFA_MEM_FRAG_RX*.
  - *FFA_MEM_FRAG_TX*. See 16.2.2.5 *FFA_MEM_FRAG_TX*.

  A Sender can invoke these interfaces as many times as there are fragments to transmit the complete descriptor to the Receiver.

16.2.2.3 Description

The ability of an endpoint to use this feature depends on whether its partition manager implements support for receipt and transmission of fragments of the memory transaction descriptor. An endpoint can discover the availability of this support through the *FFA_FEATURES* interface (see 11.2 *FFA_FEATURES*). An endpoint must support this feature if its partition manager supports it.

It is strongly recommended that endpoint and partition manager implementations include support for this feature.

An endpoint and partition manager must implement the following protocol to use this feature.

1. An endpoint is the *Sender* and the partition manager is the *Receiver* of fragments in an invocation of the following ABIs.
   - *FFA_MEM_DONATE*.
   - *FFA_MEM_LEND*.
   - *FFA_MEM_SHARE*.
   - *FFA_MEM_RETRIEVE_REQ*.
2. The partition manager is the *Sender* and endpoint is the *Receiver* of fragments in an invocation of the *FFA_MEM_RETRIEVE_RESP* ABI.
3. A *Sender* must use these ABIs to transmit the first fragment of the memory transaction descriptor as follows.
   - The *w2* register must be used to specify the length the first fragment.

- The buffer used to transmit the first fragment depends on the ABI being invoked as follows.
  - The *Sender* must either use its TX buffer or a dynamically allocated buffer (if supported by the *Receiver*) in an invocation of the following ABIs.
    - *FFA_MEM_DONATE*.
    - *FFA_MEM_LEND*.
    - *FFA_MEM_SHARE*.
    - *FFA_MEM_RETRIEVE_REQ*.
  - The buffer used by the *Sender* in an invocation of the *FFA_MEM_RETRIEVE_RESP* ABI must be one of the following.
    - The RX buffer of the *Receiver* if it used its TX buffer in the earlier counterpart invocation of the *FFA_MEM_RETRIEVE_REQ* ABI.
    - The dynamically allocated buffer that was used by the *Receiver* in the earlier counterpart invocation of the *FFA_MEM_RETRIEVE_REQ* ABI.
- A partition manager as the *Receiver* must return *INVALID_PARAMETERS* if it does not support this feature or the length of the fragment is invalid.

4. After receiving the first fragment, a *Receiver* must allocate a *Handle* (see 8.10.2 *Memory region handle*) and use it to associate the remaining fragments with the current instance of the ABI invocation.

   The same *Handle* must be used to identify the memory region description once all the fragments have been received.

5. A *Receiver* must request the *Sender* to transmit the next fragment through an invocation of the *FFA_MEM_FRAG_RX* ABI. See 16.2.2.4 *FFA_MEM_FRAG_RX* for a description of this ABI and its parameters.

   The *Receiver* must use this interface to request retransmission of a fragment as well. This could happen if it was unable to receive the previous fragment due to an IMPLEMENTATION DEFINED reason.

   The *Receiver* must populate the *w4* parameter register at a physical FF-A instance as follows.

   1. With the endpoint ID of the *Owner* of the memory region, if the fragment is being transmitted in response to the following ABI invocations.
      - *FFA_MEM_DONATE*.
      - *FFA_MEM_LEND*.
      - *FFA_MEM_SHARE*.
      - *FFA_MEM_RETRIEVE_REQ* on behalf of the Hypervisor see 14.4.3 *Support for retrieval by the Hypervisor*.
      - *FFA_MEM_RETRIEVE_RESP* on behalf of the Hypervisor see 14.4.3 *Support for retrieval by the Hypervisor*.
   2. With the endpoint ID of the *Borrower* of the memory region, if the fragment is being transmitted in response to the following ABI invocations.
      - *FFA_MEM_RETRIEVE_REQ* by the Hypervisor on behalf of a Borrower VM.
      - *FFA_MEM_RETRIEVE_RESP* by the Hypervisor on behalf of a Borrower VM.

6. A *Sender* must transmit the next fragment to the *Receiver* through an invocation of the *FFA_MEM_FRAG_TX* ABI. See 16.2.2.5 *FFA_MEM_FRAG_TX* for a description of this ABI and its parameters.

   The buffer used to transmit the fragment must be the same as the one used to transmit the first fragment.

   The *Sender* must populate the *w4* parameter register at a physical FF-A instance with the endpoint ID that was populated in the same register in the counterpart invocation of *FFA_MEM_FRAG_RX* by the *Receiver*.

7. A *Receiver* must acknowledge receipt of the final fragment. It must do this by completing the invocation of the ABI that was invoked to transmit the first fragment. For example, *FFA_MEM_SHARE* must be completed with the *FFA_SUCCESS* function as described in 14.3 *FFA_MEM_SHARE*.

8. A *Receiver* could abort the memory management operation while transmission of fragments is in-progress due to IMPLEMENTATION DEFINED reasons. The operation is identified by the ABI used to transmit the first fragment. The invocation of this ABI must be completed to signal to the *Sender* that the operation has been aborted.

   The mechanism to do this depends on the type of *Receiver* and the FF-A instance it resides at as follows.

   - The *Receiver* is a partition manager at a virtual FF-A instance. It must invoke the *FFA_ERROR* function with the *ABORTED* error code.
   - The *Receiver* is a partition manager at a physical FF-A instance. It must invoke the *FFA_ERROR* function with the *ABORTED* error code.
   - The *Receiver* is an endpoint at a virtual FF-A instance. In this version of the Framework, this scenario is possible only during the invocation of the *FFA_MEM_RETRIEVE_RESP* ABI. The *Receiver* must invoke the *FFA_MEM_RELINQUISH* ABI (see 14.6 *FFA_MEM_RELINQUISH*) to abort the operation.

   In all cases, the *Receiver* must restore any globally observable state associated with the memory region described by the transaction descriptor to what it was prior to receipt of the first fragment.

   In all cases, the *Sender* must restore any globally observable state associated with the memory region described by the transaction descriptor to what it was prior to transmission of the first fragment.

9. A *Sender* at a virtual FF-A instance must not abort the memory management operation while transmission of fragments is in-progress.

   The Hypervisor could abort an operation as the *Sender* at the Non-secure physical FF-A instance. It must invoke the *FFA_ERROR* function with the *ABORTED* error code to do this.

Figure 16.4 illustrates an example where the *FFA_MEM_RETRIEVE_REQ* and *FFA_MEM_RETRIEVE_RESP* interfaces are used to retrieve a transaction descriptor in fragments at a virtual FF-A instance. The following assumptions have been made.

- The memory region is shared with only a single Borrower.
- The RX/TX buffers of the Borrower are used by these interfaces.
- In invocations of both *FFA_MEM_RETRIEVE_REQ* and *FFA_MEM_RETRIEVE_RESP*, the descriptor in Table 8.19 is split into two fragments to be delivered to the Relayer and Borrower respectively.
- In invocations of both *FFA_MEM_RETRIEVE_REQ* and *FFA_MEM_RETRIEVE_RESP*, only parameters relevant to fragment transmission have been illustrated.

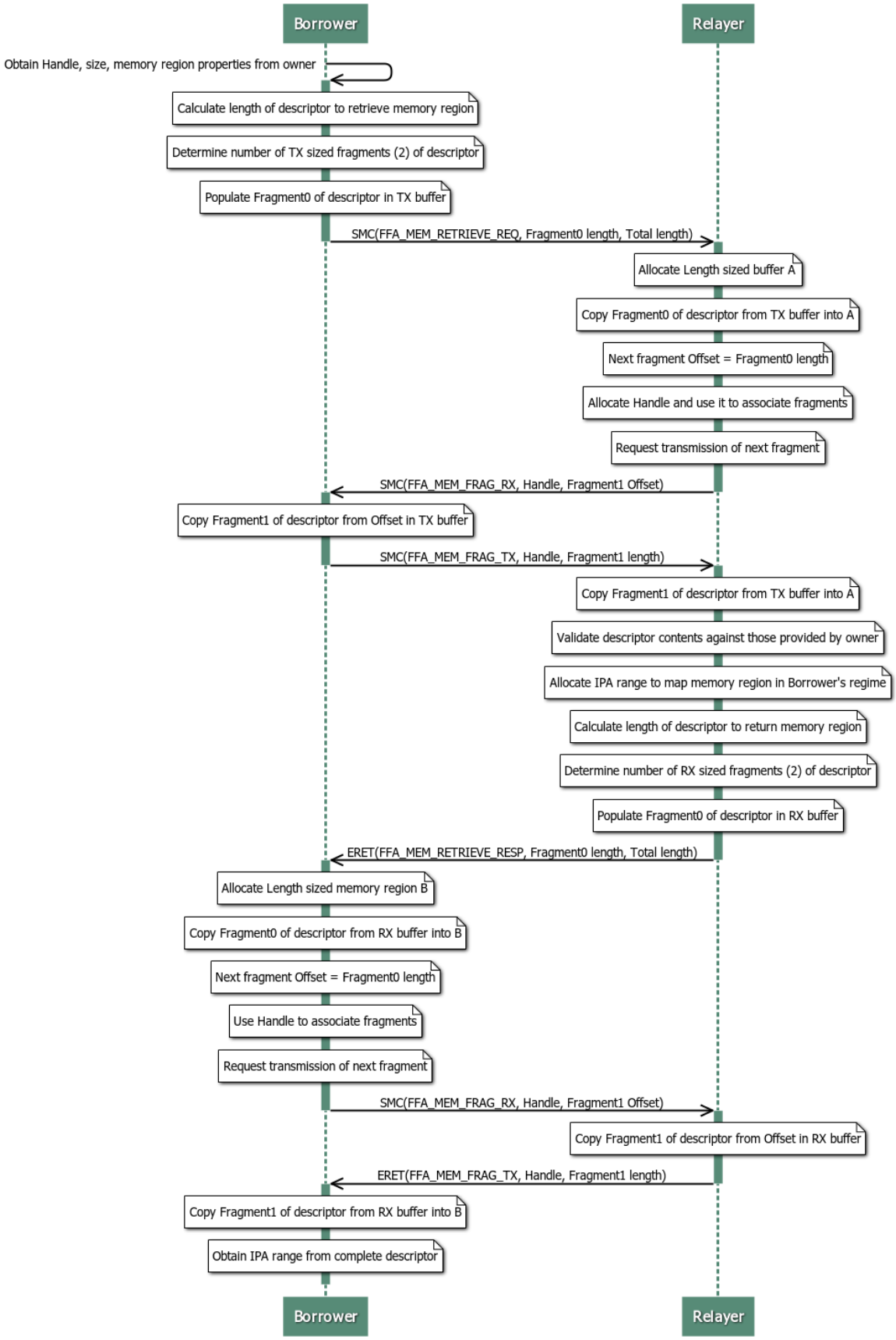

Figure 16.4: Example of fragment transmission while retrieving memory

16.2.2.4 FFA_MEM_FRAG_RX

Description

- A caller uses this interface to request the callee to transmit the next fragment of the memory transaction descriptor.
- Valid FF-A instances and conduits are listed in Table 16.11.
- Syntax of this function is described in Table 16.12.
- Successful completion of this function is indicated by an invocation of the *FFA_MEM_FRAG_TX* function (see 16.2.2.5 *FFA_MEM_FRAG_TX*).
- Encoding of error code in the FFA_ERROR function is described in Table 16.13.

Table 16.11: FFA_MEM_FRAG_RX instances and conduits

Config No.	FF-A instance	Valid conduits
1	Secure and Non-secure physical	SMC, ERET
2	Secure and Non-secure virtual	SMC, HVC, SVC, ERET

Table 16.12: FFA_MEM_FRAG_RX function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x8400007A.
uint64 Handle	w1/w2	• *Handle* value to associate the fragment with the transaction descriptor of the ongoing memory management transaction with the callee.
uint32 Fragment offset	w3	• Byte offset from where the next fragment to be transmitted must start. • Offset must be calculated from the base of the transaction descriptor being transmitted . • Offset must be equal to one of the following: – The number of bytes of the transaction descriptor transmitted prior to the invocation of this interface. – The offset used in the previous invocation of this interface. This allows the Sender to retransmit the previous fragment if the Receiver could not receive it due to an IMPLEMENTATION DEFINED reason.
uint32 Endpoint ID	w4	• ID of the Owner or Borrower endpoint. – Bit[31:16]: Endpoint ID. – Bit[15:0]: Reserved (MBZ). • Reserved (MBZ) at any virtual FF-A instance.

Parameter	Register	Value
Other parameter registers	w5-w7 w5-x7	• Reserved (MBZ).

Table 16.13: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: Invalid Handle, fragment offset or endpoint ID value. • NOT_SUPPORTED: This function is not implemented at this FF-A instance. • ABORTED. Sender aborted transmission of fragments.

16.2.2.5 FFA_MEM_FRAG_TX

Description

- A caller uses this interface to transmit the next fragment of the transaction descriptor to the callee.
- Valid FF-A instances and conduits are listed in Table 16.15.
- Syntax of this function is described in Table 16.16.
- Successful completion of this function is indicated by an invocation of the *FFA_MEM_FRAG_RX* function (see 16.2.2.4 *FFA_MEM_FRAG_RX*).
- Encoding of error code in the FFA_ERROR function is described in Table 16.17.

Table 16.15: FFA_MEM_FRAG_TX instances and conduits

Config No.	FF-A instance	Valid conduits
1	Secure and Non-secure physical	SMC, ERET
2	Secure and Non-secure virtual	SMC, HVC, SVC, ERET

Table 16.16: FFA_MEM_FRAG_TX function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x8400007B.
uint64 Handle	w1/w2	• *Handle* value to associate the fragment with the transaction descriptor of the ongoing memory management transaction with the callee.
uint32 Fragment length	w3	• Length of the fragment being transmitted.
uint32 Endpoint ID	w4	• ID of the Owner or Borrower endpoint. – Bit[31:16]: Endpoint ID. – Bit[15:0]: Reserved (MBZ). • Reserved (MBZ) at any virtual FF-A instance.
Other parameter registers	w5-w7 w5-x7	• Reserved (MBZ).

Table 16.17: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: Invalid Handle, fragment length or endpoint ID value. • NOT_SUPPORTED: This function is not implemented at this FF-A instance. • ABORTED. Receiver aborted transmission of fragments.

16.2.3 Time slicing of memory management operations

16.2.3.1 Rationale

In each FF-A memory management ABI as follows, the partition manager is responsible for mapping or unmapping a memory region from the translation regime of an endpoint that invokes the ABI.

- *FFA_MEM_DONATE*. This interface is described in 14.1 *FFA_MEM_DONATE*.
- *FFA_MEM_LEND*. This interface is described in 14.2 *FFA_MEM_LEND*.
- *FFA_MEM_SHARE*. This interface is described in 14.3 *FFA_MEM_SHARE*.
- *FFA_MEM_RETRIEVE_REQ*. This interface is described in 14.4 *FFA_MEM_RETRIEVE_REQ*.
- *FFA_MEM_RELINQUISH*. This interface is described in 14.6 *FFA_MEM_RELINQUISH*.
- *FFA_MEM_RECLAIM*. This interface is described in 14.7 *FFA_MEM_RECLAIM*.

The duration of a mapping or unmapping operation on a set of translation tables depends on factors such as the size of the memory region, number of translation table entries it requires, number of cache and TLB maintenance operations etc. The operation runs to completion in the partition manager. This could prevent progress of the endpoint that requested the operation. In some scenarios, an endpoint might not be able to tolerate this delay for example, if it is prevented from processing its pending interrupts.

16.2.3.2 Overview

This version of the Framework supports an optional feature that allows the partition manager to divide the translation table operations into *time slices*.

An operation runs for the duration of a time slice. Once the time slice is over, the partition manager relinquishes control back to the endpoint. The endpoint resumes the operation later. On resumption the partition manager runs the operation for another time slice. The process repeats itself until the operation completes. The duration of a time slice and its discovery by a partition manager is IMPLEMENTATION DEFINED.

This optional feature enables both the endpoint and the partition manager to make progress during a long running memory management ABI invocation.

16.2.3.3 Description

The ability of an endpoint to use this feature depends on whether its partition manager implements support for time-slicing memory management ABI invocations. An endpoint can discover the availability of this support through the *FFA_FEATURES* interface (see 11.2 *FFA_FEATURES*). This feature is only available to EL1 and S-EL1 endpoints.

An endpoint and its partition manager must implement the following protocol to use this feature.

1. An endpoint must request its partition manager to use time-slicing by setting the *Operation time slicing* flag in the *Flags* field as follows:
   - In the transaction descriptor (see 8.12.4 *Flags usage*) in an invocation of the following ABIs.
     - FFA_MEM_DONATE.
     - FFA_MEM_LEND.
     - FFA_MEM_SHARE.
     - FFA_MEM_RETRIEVE_REQ.
   - In Table 14.25 in an invocation of the FFA_MEM_RELINQUISH ABI.
   - In *w3* in an invocation of the FFA_MEM_RECLAIM ABI.
   - A partition manager must return *INVALID_PARAMETERS* if an endpoint sets the *Operation time slicing* flag and it does not support this feature.
2. A partition manager must divide the translation table operations required by the invoked ABI, if their duration is expected to exceed the time slice duration.

   This must be done only after the partition manager has received the entire transaction descriptor in an invocation of the following ABIs.

- FFA_MEM_DONATE.
- FFA_MEM_LEND.
- FFA_MEM_SHARE.
- FFA_MEM_RETRIEVE_REQ.

Once the time slice duration expires, the partition manager must:

- Save enough state to resume the ABI invocation at a later point of time and not prevent progress of other FF-A functions before the paused ABI invocation is resumed.
- Cater for the scenario where the ABI invocation is paused on one PE and resumed by the endpoint on another PE.
- Use the *Handle* (see 8.10.2 *Memory region handle*) to identify the current instance of the ABI invocation when it is resumed later. A new *Handle* must be allocated if this was not done previously.
- Invoke the *FFA_MEM_OP_PAUSE* interface (see 16.2.3.4 *FFA_MEM_OP_PAUSE*) to inform the endpoint that the current ABI invocation has been paused and must be resumed later.

3. An endpoint must use the *FFA_MEM_OP_RESUME* interface (see 16.2.3.5 *FFA_MEM_OP_RESUME*) to resume the paused ABI invocation identified by the *Handle*.

   The endpoint could invoke other FF-A functions before resuming the paused ABI invocation.

4. A partition manager could abort the ABI invocation when it is resumed later due to IMPLEMENTATION DEFINED reasons. It must signal to the endpoint that the ABI invocation has been aborted by invoking the *FFA_ERROR* function with the *ABORTED* error code.

Figure 16.5 illustrates an example where the *FFA_MEM_RETRIEVE_REQ* and *FFA_MEM_RETRIEVE_RESP* interfaces are used by an endpoint to retrieve a transaction descriptor at a virtual FF-A instance. The following assumptions have been made.

- The operation to map the memory region in the translation regime of the endpoint is expected to take longer than the time slice value known to the partition manager.
- The endpoint has requested time slicing by setting the *Operation time slicing* flag.
- The partition manager divides the *FFA_MEM_RETRIEVE_REQ* function invocation into 3 time slices through the *FFA_MEM_OP_PAUSE* and *FFA_MEM_OP_RESUME* interfaces.

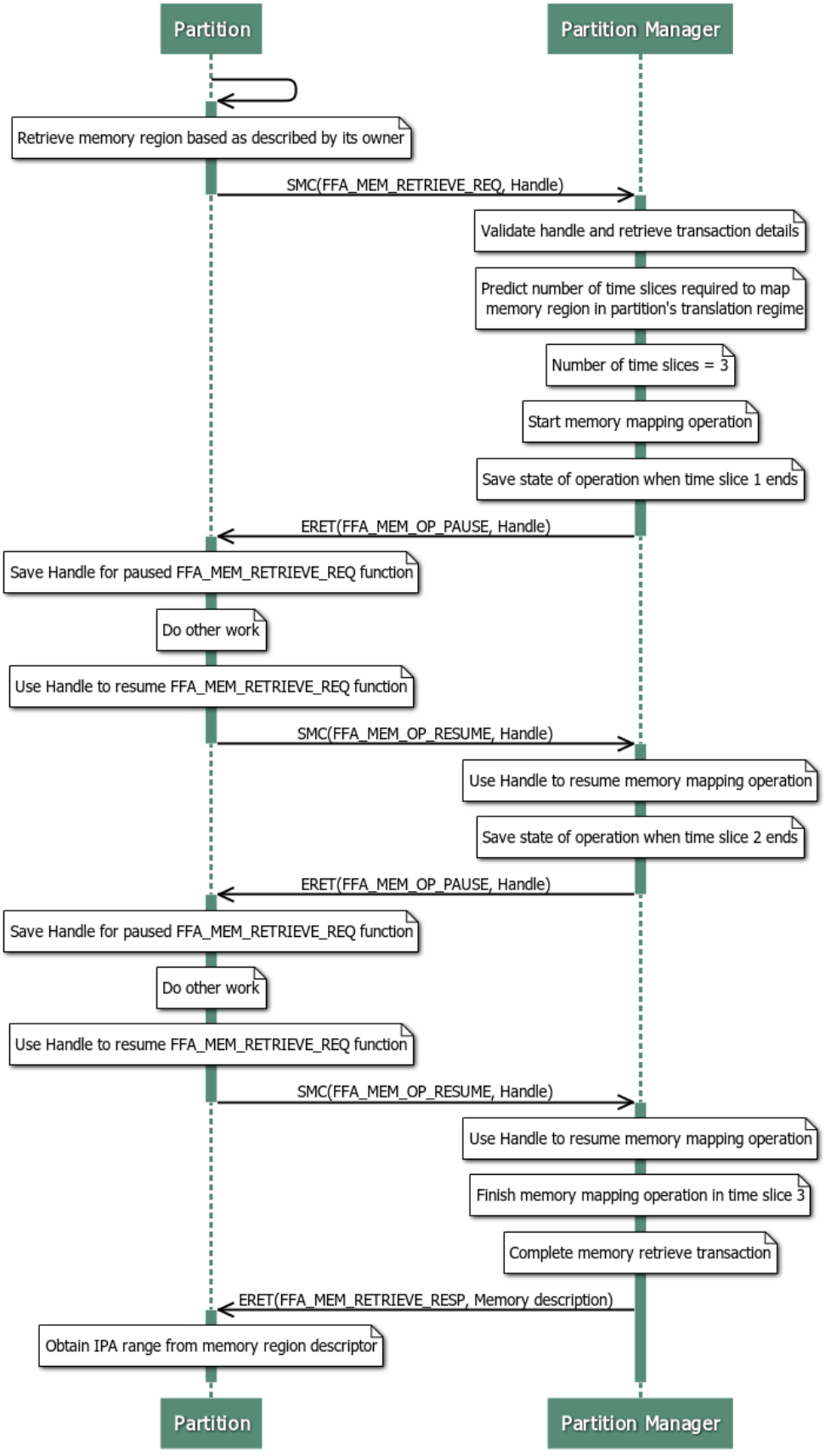

Figure 16.5: Example of time slicing during FFA_MEM_RETRIEVE_REQ

16.2.3.4 FFA_MEM_OP_PAUSE

Description

- A partition manager uses this interface to pause the execution of a memory management ABI invoked by an endpoint. Execution is returned to the endpoint.
- Valid FF-A instances and conduits are listed in Table 16.19.
- Syntax of this function is described in Table 16.20.
- Encoding of error code in the FFA_ERROR function is described in Table 16.21.

Table 16.19: FFA_MEM_OP_PAUSE instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure physical	ERET
2	Secure physical	SMC
3	Secure and Non-secure virtual	ERET

Table 16.20: FFA_MEM_OP_PAUSE function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000078.
uint64 Handle	w1/w2	• *Handle* value to identify the paused memory management operation.
Other parameter registers	w3-w7 x3-x7	• Reserved (MBZ).

Table 16.21: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: Invalid handle value. • NOT_SUPPORTED: This function is not implemented at this FF-A instance.

16.2.3.5 FFA_MEM_OP_RESUME

Description

- An endpoint uses this interface to request the partition manager to resume execution of a paused memory management ABI. The paused operation is identified by the supplied *Handle*.
- Valid FF-A instances and conduits are listed in Table 16.23.
- Syntax of this function is described in Table 16.24.
- Encoding of error code in the FFA_ERROR function is described in Table 16.25.

Table 16.23: FFA_MEM_OP_RESUME instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure physical	SMC
2	Secure physical	ERET
3	Secure and Non-secure virtual	SMC, HVC, SVC

Table 16.24: FFA_MEM_OP_RESUME function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000079.
uint64 Handle	w1/w2	• *Handle* value to identify the paused memory management operation.
Other parameter registers	w3-w7 x3-x7	• Reserved (MBZ).

Table 16.25: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: Invalid Handle value. • NOT_SUPPORTED: This function is not implemented at this FF-A instance.

16.3 Power Management

16.3.1 Overview

A PE could be released from reset from different low power or power down states. The states range from the system being fully switched off to only the PE being power-gated. Entry into and exit from these states is governed by OSPM policy implemented in NS-Endpoints and the Hypervisor. The policy is exercised through OSPM operations such as,

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

The PSCI specification [12] describes these states and OSPM operations. It also defines a standard interface that these FF-A components can use to initiate OSPM operations at the Non-secure physical and virtual FF-A instances.

The impact of OSPM operations on the Secure world are twofold.

1. When a PE is released from reset, execution contexts of the SPMC and SPs are initialized on the PE. The protocol to do this depends upon whether the PE is responsible for,

    1. Initializing the system (see Section 4.4 in [12]) after a system reset/shutdown through PSCI SYSTEM_OFF, SYSTEM_RESET, SYSTEM_RESET2 functions or a hardware power-cycle sequence. The PE is called the *primary PE* and performs a *cold boot* (see [12]). The protocol for initializing an execution context of both UP and MP SPs, and the SPMC during a cold boot on the primary PE is described in Chapter 3 *Setup*.

    2. Initializing the PE after exiting a power down state in response to an invocation of the PSCI CPU_ON function. The PE is called the *secondary PE* and performs a *cold boot*. The protocol for initializing an execution context of an MP SP and the SPMC during a cold boot on a secondary PE is described in 16.3.2 *Secondary boot protocol*.

    3. Restoring the system state after exiting the Suspend to RAM state in response to a wakeup event. The PE entered this state through an invocation of the PSCI SYSTEM_SUSPEND function.

        Restoring the PE state after exiting another low power state in response to a wakeup event. The PE entered this state through an invocation of the PSCI CPU_SUSPEND function.

        The PE performs a *warm boot*. The protocol for restoring an execution context of any SP and the SPMC and informing them about an exit from a low power state during a warm boot, is described in 16.3.3 *Warm boot protocol*.

2. FF-A components in the Secure world do not perform power management independently from the Normal world. Instead, the SPMD, SPMC and SPs are informed about OSPM operations initiated by the Normal world through PSCI functions. This allows them to take some action in response to a PSCI function invocation at EL3. For example, if CPU0 is being dynamically removed, the SPMC would re-target any physical interrupts targeted to CPU0 to another CPU.

    The Framework describes a mechanism to inform FF-A components in the Secure world about OSPM operations in 16.3.4 *Power Management messages*.

16.3.2 Secondary boot protocol

In order to initialize an execution context of a MP SP or SPMC during a cold boot on a secondary PE, the SPMD and SPMC must know the entry point address of the execution context. The Framework describes two mechanisms to determine the entry point.

1. The entry point specified in the manifest and used for initializing the execution context during a primary cold boot is reused (see Chapter 3 *Setup*). The distinction between a primary and secondary cold boot is made by encoding a value in a general-purpose register when the entry point is invoked in each boot phase. Also see,

- Table 3.1.
- Table 3.4.

2. The FFA_SECONDARY_EP_REGISTER function (see 16.3.2.1 *FFA_SECONDARY_EP_REGISTER*) enables a SP or SPMC to register this entry point with the SPMC and the SPMD respectively.

If both mechanisms are implemented and FFA_SECONDARY_EP_REGISTER is used by the SP or SPMC, then the registered entry point takes precedence over the one specified in the manifest.

The SPMC must use the runtime model described in 5.5 *Runtime model for SP initialization* to initialize the SP execution context.

16.3.2.1 FFA_SECONDARY_EP_REGISTER

Description

- Enables an MP SP or SPMC to register the entry point of their execution contexts for initialization during a secondary cold boot. Also see 16.3.2.1.1 *Usage*.
- Valid FF-A instances and conduits are listed in Table 16.27.
- Syntax of this function is described in Table 16.28.
- Returns FFA_SUCCESS without any further parameters on successful completion.
- Encoding of error code in the FFA_ERROR function is described in Table 16.29.

Table 16.27: FFA_SECONDARY_EP_REGISTER instances and conduits

Config No.	FF-A instance	Valid conduits
1	Secure physical	SMC
2	Secure virtual	SMC, HVC

Table 16.28: FFA_SECONDARY_EP_REGISTER function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000087. • 0xC4000087.
uint32/uint64 Entry point address	w1/x1	• Entry point address of a secondary execution context. – Address is a IPA at the Secure virtual FF-A instance with a S-EL2 SPMC. – Address is a PA at the Secure virtual FF-A instance with a EL3 SPMC and a S-EL1 SP. – Address is a PA at the Secure physical FF-A instance with a EL3 SPMD and S-EL1 SPMC.
Other Parameter registers	w2-w7 x2-x7	• Reserved (MBZ).

Table 16.29: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• NOT_SUPPORTED: This function is not implemented at this FF-A instance. • INVALID_PARAMETERS: An invalid entry point address was specified by the caller.

16.3.2.1.1 Usage

This function is invoked by a SP or the SPMC during the initialization of their execution context during a primary cold boot (see 16.3 *Power Management*).

The callee must return NOT_SUPPORTED if this function is invoked by a caller that implements version v1.0 of the Framework.

The entry point address must be in secure memory and accessible from the caller. The callee must return INVALID_PARAMETERS otherwise.

If this function is invoked multiple times, then the entry point address specified in the last valid invocation must be used by the callee.

The Framework does not provide an interface to unregister the entry point address. Once registered, the entry point is used by,

- The SPMD until the system is reset or shutdown
- The SPMC until,
  - The system is reset or shutdown or
  - The execution of the SP is terminated e.g., due to a fatal error.

For each SP and the SPMC, the Framework assumes that the same entry point address is used for initializing any execution context during a secondary cold boot.

At the time of invoking the entry point address, the general-purpose and system registers should be programmed as specified in 3.3 *Register state*.

16.3.3 Warm boot protocol

The key difference between a warm and cold boot is that in the former case, main memory contents are preserved. Hence, it is possible to resume software from the state it was in, prior to entry into the low power state. In the Secure world, this is contingent upon the following, before the PE enters, and after it exits the low power state.

- The SPMD saves and restores the execution context of the SPMC.
- The SPMC saves and restores the execution context of each SP.

The Framework assumes that both the SPMD and SPMC fulfil these responsibilities. Additionally, the Framework defines a power management message that can be used by,

- The SPMD to inform the SPMC about the warm boot.
- The SPMC to inform an SP about the warm boot.

The message is described in 16.3.4 *Power Management messages*.

16.3.4 Power Management messages

The Framework defines a set of framework messages that describe power management operations invoked at EL3. Two types of operations are considered in this specification.

1. Operations that result in a PE entering a low power or a power down state. These operations are requested through an invocation of the following PSCI functions.
    - CPU_OFF.
    - CPU_SUSPEND.
    - SYSTEM_OFF.
    - SYSTEM_RESET.
    - SYSTEM_RESET2.
    - SYSTEM_SUSPEND.
2. Warm boot of any PE as described in 16.3 *Power Management* and 16.3.3 *Warm boot protocol*.

These messages are used in the Secure world as follows.

- If the SPMD and SPMC are implemented in separate exception levels, the SPMD at EL3 uses these messages at the Secure physical FF-A instance, to inform the SPMC at S-EL1 or S-EL2 about the power management operation that was invoked.
- The SPMC at EL3 uses these messages at the Secure virtual FF-A instance, to inform one or more SPs at S-EL0 or a single SP at S-EL1 about the power management operation that was invoked.
- The SPMC at S-EL2 uses these messages at the Secure virtual FF-A instance, to inform one or more SPs at S-EL1 or S-EL0 about the power management operation that was invoked.
- The SPMC at S-EL1 uses these messages at the Secure virtual FF-A instance, to inform one or more SPs at S-EL0 about the power management operation that was invoked.

The Framework mandates that the SPMD must inform the SPMC about the invocation of every operation listed above.

The Framework enables an SP to specify to the SPMC, the power management operations it must be informed about. This interest is registered through the SP manifest. See Table 3.1 and Table 3.4.

- Operations that are requested by a PSCI function invocation are specified through their PSCI function IDs.
- The warm boot operation is specified in an IMPLEMENTATION DEFINED manner.

An SP could choose to not register for a message in response to a power management operation that powers down the PE it is invoked on. It is possible that an execution context of this SP is running on a PE on which the operation is invoked. Since the SPMC cannot notify the SP's execution context about the operation, this scenario must be handled in one of the following ways.

- If the execution context is not pinned to the PE, the SPMC must migrate it to another PE.
- It is possible that the execution context is pinned to the PE or the PE is the last one in the system to be powered off. In this case, the SP must be robust enough to cope with the power down of the PE.

Direct messaging is used to exchange these framework messages as described below (also see 4.4 *Direct messaging usage*).

- The Sender uses the FFA_MSG_SEND_DIRECT_REQ interface to send a request message to the Receiver.
- The Receiver uses the FFA_MSG_SEND_DIRECT_RESP interface to send the response message to the Sender.

The IDs of the SPMC and SPMD are used in the Sender and Receiver fields of these ABIs (also see 11.9 *FFA_SPM_ID_GET*).

Messages sent by the SPMD to the SPMC and the SPMC to an SP through the FFA_MSG_SEND_DIRECT_REQ interface are encoded in *w3/x3-w7/x7* as described in Table 16.30 and Table 16.31.

Table 16.30: Power management request message encoding for PSCI functions

Register	Parameter
w3	PSCI Function ID
w4/x4	Input parameter in w1/x1 in PSCI function invocation at EL3.
w5/x5	Input parameter in w2/x2 in PSCI function invocation at EL3.
w6/x6	Input parameter in w3/x3 in PSCI function invocation at EL3.
w7/x7	Reserved (MBZ).

Table 16.31: Power management message encoding for a warm boot

Register	Parameter
w3	• Bit[30:1]: Reserved (MBZ). • Bit[0]: Warm boot type. – b'0: Exit from a suspend to RAM state. – b'1: Exit from a low power state shallower than the suspend to RAM state.
w4-w7 x4-x7	Reserved (MBZ).

Messages sent by the SPMC to the SPMD and an SP to the SPMC through the FFA_MSG_SEND_DIRECT_RESP interface are encoded in *w3/x3-w7/x7* as described in Table 16.32.

Table 16.32: Power management response message encoding

Register	Parameter
w3	Return error code SUCCESS or DENIED as defined in [12].
w4-w7 x4-x7	Reserved (MBZ).

An SP or the SPMC must use the SUCCESS return error code to indicate successful processing of the request message.

An SP or the SPMC must use the DENIED return error code to indicate unsuccessful processing of the request message.

The SPMC must return DENIED to the SPMD even if a single SP returns this error code to the SPMC.

If the SPMC returns SUCCESS, the SPMD must facilitate completion of the power management operation.

If the SPMC returns DENIED, the action taken by the SPMD is IMPLEMENTATION DEFINED.

A power management message must be delivered to an SP or the SPMC execution context only if the message target is in the *waiting* state.

The following requirements must be fulfilled while processing a power management message.

- It must be processed on the same PE where it is delivered.
- A request from a SP to switch to the Normal world during message processing must be denied by the SPMC.
- A request from the SPMC to switch to the Normal world during message processing must be denied by the SPMD.

Figure 16.6 illustrates an example power management message exchange between the SPMD in EL3, SPMC in S-EL2 and a single SP in S-EL1, in response to a PSCI function invocation at EL3.

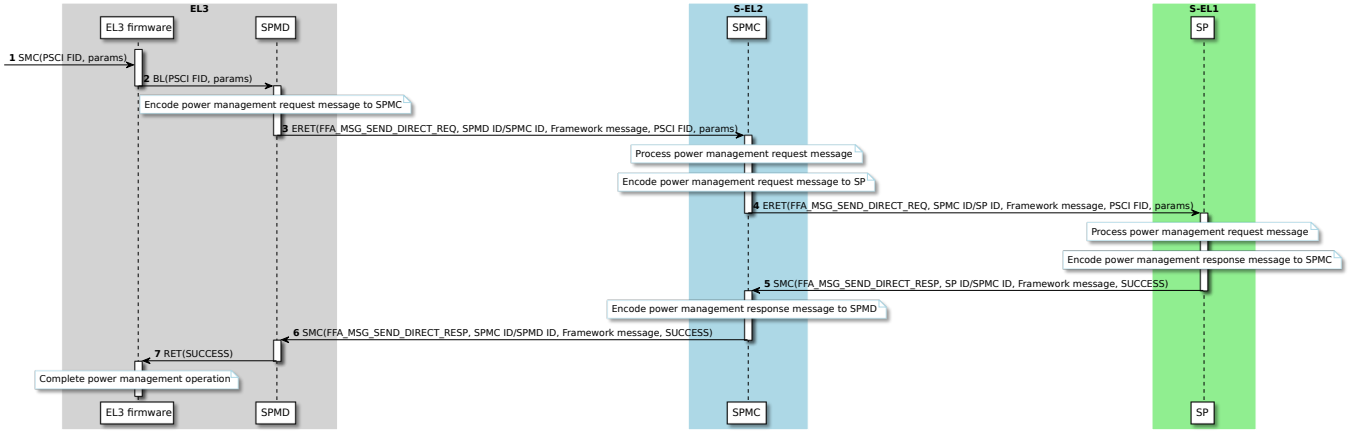

Figure 16.6: Example power management message usage

16.4 Legacy indirect messaging usage

In version 1.0 of the Framework, guidance on indirect messaging differs from the guidance in the current version of the Framework in the following ways.

1. Only VMs can exchange partition messages using indirect messaging. It is now possible to exchange partition messages between any pair of endpoints.
2. The identities of the Sender and Receiver endpoints and the length of a partition message are encoded in input parameter registers in an FFA_MSG_SEND ABI invocation. As a result, the Receiver endpoint could have to invoke the FFA_MSG_POLL ABI to determine this information. It is not available in the RX buffer.

   In this version of the framework, this information is encoded along with the partition message payload in the RX and TX buffers as described in Table 4.2. As a result, there is no need for the Receiver endpoint to call FFA_MSG_POLL.
3. Only the primary scheduler runs the Receiver VM. In this version of the framework, a Receiver endpoint can be run by a primary or a secondary scheduler. Also, the notification mechanism is used to inform the scheduler.

The guidance on indirect messaging in v1.0 of the Framework is deprecated. The FFA_MSG_SEND and FFA_MSG_POLL interfaces are described to maintain compatibility between v1.0 and the current version of the Framework. These interfaces could be removed in a future version of the framework.

16.4.1 FFA_MSG_SEND

Overview

- Send a Partition message to a VM through the RX/TX buffers by using indirect messaging.
  - Message is copied by Hypervisor from the TX buffer of Sender NS-Endpoint to the RX buffer of Receiver NS-endpoint.
  - The scheduler is informed about the pending message in the RX buffer of the Receiver.
  - Message will be read when the Receiver endpoint is scheduled to run.
  - See 16.4.1.2 *Component responsibilities for FFA_MSG_SEND* for caller and callee roles and responsibilities.
  - Must not be invoked when the caller is processing a direct request.
- Valid FF-A instances and conduits are listed in Table 16.34.
  - Is used with the ERET conduit in the following scenarios.
    - Inform an endpoint that a message is available in its RX buffer.
    - Inform the primary scheduler that the Receiver has a pending message in its RX buffer.
- Syntax of this function is described in Table 16.35.
- Successful completion of this function call is indicated as follows.
  - *w0* contains *FFA_SUCCESS* function ID.
  - *w1/x1-w7/x7* are reserved and MBZ.
  - Successful completion of this function does not imply that the message has been read by the Receiver endpoint.
- Encoding of error code in the FFA_ERROR function is described in Table 16.36.
  - See 16.4.1.1 *Target availability notification* for behavior when BUSY is returned and caller must be notified about availability of TX buffer.

Table 16.34: FFA_MSG_SEND instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure virtual	SMC, HVC, ERET

Table 16.35: FFA_MSG_SEND function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x8400006E.
uint32 Sender/Receiver IDs	w1	• Sender and Receiver endpoint IDs. – Bit[31:16]: Sender endpoint ID. – Bit[15:0]: Receiver endpoint ID.
uint32/uint64 Reserved	w2/x2	• Reserved for future use (MBZ).

Parameter	Register	Value
uint32 Message size	w3	• Length of message payload in the RX buffer. • This is an optional field when used with the *ERET* conduit at the Non-secure virtual FF-A instance and the callee is not the Receiver of the message. It MBZ in this case.
uint32 Flags	w4	• Message flags. – Must be ignored by callee when SVC conduit is used. – Bit[0]: Blocking behavior. ∗ b'0: Return BUSY if message cannot be delivered to Receiver. ∗ b'1: Return BUSY if message cannot be delivered to Receiver and notify when delivery is possible. – Bit[31:1]: Reserved (MBZ).
uint32 Sender vCPU ID	w5	• Information to identify execution context or vCPU of Sender endpoint. – Only valid when ERET conduit is used. MBZ and ignored by callee otherwise. – Bits[31:16]: Reserved (MBZ). – Bits[15:0]: vCPU ID of Sender endpoint.
Other Parameter registers	w6-w7 x6-x7	• Reserved (MBZ).

Table 16.36: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: A field in input parameters is incorrectly encoded. • BUSY: Receiver RX buffer is not free. • DENIED: Callee is not in a state to handle this request. • NO_MEMORY: Insufficient memory to handle this request. • NOT_SUPPORTED: This function is not implemented at this FF-A instance.

16.4.1.1 Target availability notification

When this interface is invoked, it is possible that the callee determines that the RX buffer of the Receiver VM cannot be written to. This can happen if either another instance of a Producer is writing to the RX buffer or the Receiver VM is reading from it as a Consumer (see 4.2.2.4 *Buffer synchronization*). The callee must complete the interface invocation with a *BUSY* error code in this case.

A VM running in EL1 in either Security state can request to be notified when the RX buffer becomes available again by setting *bit[0] = 1* in the *Flags* parameter. In this case, the Hypervisor must:

1. Determine when the RX buffer is available as per the ownership rules described in 4.2.2.4 *Buffer synchronization*.
2. Notify each caller about the RX buffer availability.

The Hypervisor must describe the interrupt to indicate availability of the Receiver VM RX buffer to each VM respectively through an IMPLEMENTATION DEFINED mechanism. This could be done through a platform discovery mechanism like ACPI or Device tree.

A Consumer that is, OS kernel or VM must indicate the availability of its RX buffer by using a mechanism listed in 4.2.2.4 *Buffer synchronization* for example, through the **FFA_RX_RELEASE** interface.

16.4.1.2 Component responsibilities for FFA_MSG_SEND

This section describes the common responsibilities that the participating FF-A components must fulfill during transmission of Partition messages between VMs through the *FFA_MSG_SEND* interface. This interface is used in the scenarios listed in 4.1.1 *Indirect messaging*.

16.4.1.2.1 Sender VM responsibilities

1. Must acquire ownership of empty TX buffer (see 4.2.2.4 *Buffer synchronization*).
2. Must write Partition message payload to TX buffer.
3. Must specify length of Partition message payload.
4. Must specify blocking behavior in *Flags* parameter.
5. Must specify Sender and Receiver VM IDs.
6. Must implement support for handling all error status codes that can be returned on completion of these interfaces.
7. See 16.4.1.2.2 *Hypervisor responsibilities* for Hypervisor responsibilities in this message transmission.

16.4.1.2.2 Hypervisor responsibilities

1. Must validate Sender and Receiver VM IDs and return *INVALID PARAMETER* if either is invalid.
2. Must check that reserved bits are 0 in *Flags* parameter. Return *INVALID PARAMETER* if this check fails.
3. Must check that reserved and unused parameter registers are 0. Return *INVALID PARAMETER* if this check fails.
4. Must check that the size of the *Receiver* RX buffer is large enough to accommodate the message. Must return *NO_MEMORY* if this is not true.
5. Must lock TX buffer of *Sender* from concurrent accesses before copying the message.
6. Must determine availability of RX buffer of *Receiver*.
    1. Return *BUSY* if RX buffer is not available.
        1. Save *Sender* ID if it wants the target availability interrupt when the RX buffer becomes free.
        2. Arrange for target availability interrupt to be delivered to Sender.
    2. Mark RX buffer as unavailable if it is available.
7. Must protect RX buffer of *Receiver* from concurrent accesses.
8. Must copy message from *Sender* TX buffer to *Receiver* RX buffer.
9. Must unlock TX buffer of *Sender* after copying the message.
10. Must unlock RX buffer of *Receiver* after copying the message.
11. Must inform primary scheduler that *Receiver* has a pending message as described in 16.4.1.3 *Legacy mechanism for scheduler notification*.
12. Must return *SUCCESS* to *Sender* if message is successfuly transmitted.
13. Must mark the RX buffer as available when the Receiver releases it.

16.4.1.2.3 Receiver VM responsibilities

1. Copy message from RX buffer.
2. Transfer ownership of the RX buffer by invoking the *FFA_RX_RELEASE* interface.

16.4.1.3 Legacy mechanism for scheduler notification

This section describes how the primary scheduler must be notified depending on its location relative to the message Sender.

1. A VM is the Sender. The primary scheduler and Hypervisor are co-resident. The Hypervisor must use an IMPLEMENTATION DEFINED mechanism to notify the primary scheduler in response to the *FFA_MSG_SEND* call.
2. A VM is the Sender.
    1. The primary scheduler is resident in another VM.
        1. The Hypervisor must forward the *FFA_MSG_SEND* call to the primary scheduler using the *ERET* conduit on the PE where the call is made.
        2. Primary scheduler must respond to the forwarded *FFA_MSG_SEND* call with either a *FFA_SUCCESS* or *FFA_ERROR* invocation through the SMC conduit.
3. The primary scheduler and Sender VM are co-resident. The Sender VM must use an IMPLEMENTATION DEFINED mechanism to notify the scheduler.

16.4.2 FFA_MSG_POLL

Description

- Poll if a message is available in the RX buffer of the caller. Execution is returned to the caller if no message is available.
  - Must not be invoked when the caller is processing a direct request.
- Valid FF-A instances and conduits are listed in Table 16.38.
- Syntax of this function is described in Table 16.39.
- Successful completion of this function is indicated through the invocation of the FFA_MSG_SEND interface (see 16.4.1 *FFA_MSG_SEND*).
- Encoding of error code in the FFA_ERROR function is described in Table 16.40.

Table 16.38: FFA_MSG_POLL instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure virtual	SMC, HVC

Table 16.39: FFA_MSG_POLL function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x8400006A.
Other Parameter registers	w1-w7 x1-x7	• Reserved (MBZ).

Table 16.40: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• RETRY: Message is not available in the caller's RX buffer. • DENIED: Callee is not in a state to handle this request. • NOT_SUPPORTED: This function is not implemented at this FF-A instance.

Terms and abbreviations

ABI

Application Binary Interface

DMA

Direct Memory Access

DSP

Digital Signal Processor

FF-A

Firmware Framework for A-profile

GIC

Generic Interrupt Controller

HVC

Hypervisor Call

MBP

Must be preserved

MBZ

Must be zero

MM

Management Mode

MMIO

Memory Mapped Input Output

MP

Multi-processing

OS

Operating System

OSPM

Operating System Power Management

PE

Processing Element

PPI

Private Peripheral Interrupt

PSA

Platform Security Architecture

SGI

Software Generated Interrupt

SMC

Secure Monitor Call

SMCCC

SMC Calling Convention

SMMU

System Memory Management Unit

SP

Secure Partition

SPCI

Secure Partition Client Interface

SPI

Shared Peripheral Interrupt

SPM

Secure Partition Manager

SPRT

Secure Partition Run Time

STMM

Standalone Management Mode

SVC

Supervisor Call

TCB

Trusted Computing Base

TEE

Trusted Execution Environment

UUID

Unique Universal Identifier

VCPU

Virtual CPU

VHE

Virtualization Host Extensions

VM

Virtual Machine

VMSA

Virtual Memory System Architecture