Arm Firmware Framework for Arm A-profile

Document number	DEN0077A
Document quality	EAC0
Document version	1.2
Document confidentiality	Non-confidential

Arm Firmware Framework for Arm A-profile

Release information

Date	Version	Changes
2024/Jul/31	v1.2 EAC0	• Introduced RX buffer ownership flag to FFA_MSG_WAIT • Clarified guidance for endpoints exposing multiple UUIDs • Fixed description of FFA_VERSION output version number • Clarified count of descriptors in an FF-A discovery ABI • Clarified language for notification support without Hypervisor • Clarified use of RX buffer full notification guidance • Language fixes based upon feedback
2024/May/03	v1.2 BET0	• Clarified SPMC and SPMD are both types of Relayers • Fixed reference to Receiver in description of FFA_NOTIFICATION_SET ABI • Dropped reference to SP Lifecycle supplement • Added additional restrictions to the usage of the FFA_RUN ABI • Added guidance when exiting an allocation mode with pending virtual interrupts • Clarified usage of FFA_RX_ACQUIRE during in-direct message delivery • Added example in-direct messaging flows between different endpoint types • Added example notification flows between different endpoint types • Clarified description of SRI and NPI usage • Relaxed VM and Hypevisor flags from MBZ to SBZ in FFA_NOTIFICATION_GET • Introduced UUID field in partition message header • Fixed duplicated register descriptions in FFA_CONSOLE_LOG ABI definition • Clarified description of partition message header encoding • Added Reserved (MBZ) condition of RXTX_MAP page_count • Fixed Reserved (MBZ) encoding description in FFA_FEATURES • Fixed Reserved (SBZ) encoding description in FFA_MSG_SEND_DIRECT_RESP
2023/Oct/04	v1.2 ALP1	• Updated all ABIs to reserve extended register contents and clarified Reserved register usage • Reverted changes to extended the IMPDEF register usage of the DIRECT_REQ and DIRECT_RESP ABIs • Added the DIRECT_RESP2 ABI as a counter part to the DIRECT_REQ2 ABI • Added clarification that the DIRECT_REQ2 ABI may be used with the Nil UUID • Dropped the ability for the timeout parameter in FFA_YIELD to be used by other endpoints • Moved memory management guidance into FF-A memory management protocol supplement • Enabled FFA_CONSOLE_LOG to utilize extended registers • Clarified SP runtime model for power management messages • Added implementation note to clarify partition information can be filtered by a callee • Clarified definition of SPM LSPs • Clarified FFA_YIELD can be used in call chains • Clarified diagram illustrating NS interrupt returning execution to the Nwld • Added constraint for RXTX buffer allocation on 32-bit systems • Aligned usage of the DENIED error code • Clarified FFA_VERSION is used to negotiate the version at an FF-A instance during initialization • Added reference to SP Lifecycle supplement • Added a maximum size value in FFA_RXTX_MAP properties of FFA_FEATURES • Added guidance to describe how notifications can be used by a Hypervisor • Added a preface to describe the relationship between the Base specification and its supplement specifications

Date	Version	Changes
2023/Mar/17	v1.2 ALP0	• Enabled platforms to optionally bypass multi-borrower checks • Added FFA_CONSOLE_LOG ABI • Added guidance to enable communication between EL3 Logical SPs and other SPs • Added register based discovery mechanism PARTITION_INFO_GET_REGS • Enabled storing IMPDEF information in EMAD • Enabled FFA_YIELD to be used with Direct messaging • Enabled an SP to be periodically woken up • Extended Direct messaging register usage • Introduced the DIRECT_REQ2 ABI • Enabled a partition to export multiple UUIDs • Allowed delegating of G0 interrupts in EL3 via FFA_EL3_INTR_HANDLE • Added a canonical UUID for SPs that speak the EFI MM communication protocol • Added a maximum size value in FFA_RXTX_MAP properties of FFA_FEATURES
2022/Nov/30	v1.1 REL0	• Language fixes based upon feedback • Clarified ABI return code if RX/TX buffers are used but not registered • Clarified valid instances and conduits for direct requests and responses • Restructured layout of FFA_MEM_PERM documentation • Clarified Relayer responsibilities for memory transactions with both VMs and SPs • Clarified terminology in FFA_MEM_LEND description to align with lender perspective • Clarified when an owner can reclaim access to a memory region • Clarified memory handle lifetime • Added diagrams to illustrate memory management transactions • Clarified allowed usage of FFA_SECONDARY_EP_REGISTER • Fixed the reference v1.0 memory descriptor format • Clarified run-time model manifest entry depreciation notice • Added IMPDEF mechanism example to prevent privilege escalation with FFA_MEM_DONATE
2022/Mar/29	v1.1 EAC0	• Clarified guidance on partition ID and UUID usage • Revised protocol to pass boot information to the SPMC or an SP • Added compliance requirements for various features supported by the Framework • Clarified guidance on interrupt management in Secure world • Updated guidance on direct message passing to be used with logical partitions • Clarified usage of FFA_INTERRUPT ABI depending upon endpoint state and FF-A instance • Clarified usage of FFA_VERSION to enable negotiation of a compatible Framework version • Clarified responsibilities of FF-A components in memory management transactions involving multiple borrowers • Added guidance to enable a Hypervisor signal creation and destruction of VMs to an SP • Revised some data structures to make them forwards compatible • Added support for reporting execution state of an endpoint through FFA_PARTITION_INFO_GET • Miscellaneous clarifications to better describe ABI behaviour and features • Relaxed requirements around when the scheduler receiver interrupt is signalled by the SPMC • Constrained configuration of a Notification pending interrupt as an SGI or PPI

Date	Version	Changes
2021/Jul/05	v1.1 BET0	• Consolidated guidance on FF-A architecture in chapter 2 • Removed usage of AArch32 modes with usage of execution states • Added concept of logical partitions that could be co-resident with the SPMC or resident in a separate EL but not isolated from the SPMC • Replaced terms "isolated" and non-isolated" partitions with physical and logical partitions where applicable • Updated guidance on direct message passing to be used with logical partitions • Updated guidance on memory management to accommodate deployment of logical partitions • Added more generalised guidance on isolation of DMA capable devices in Section 4.2 • Extended guidance on memory management to allow a partition to lend memory that it cannot access • Reserved value -1 as an invalid memory handle • Added guidance on interrupt management in the Secure world • Added section on discovery of NS bit usage to allow v1.0 SPs to use the NS bit flag by discovering its presence and requesting this feature • Added guidance in FFA_PARITITON_INFO_GET to return the UUIDs and count of partitions in the system. • Updated terminology in guidance on interrupt management for consistency and readability • Added guidance for notification support without a Hypervisor • Replaced RETRY error code with new NO_DATA error code in FFA_NOTIFICATION_INFO_GET • Clarified guidance on Delay Schedule Receiver interrupt flag in FFA_NOTIFICATION_SET • Updated minor version of the spec to 1 • Added VM ID field to FFA_RX_ACQUIRE and RELEASE ABIs at NS physical FF-A instance • Allowed FFA_SPM_ID_GET to be forwarded by SPMD to SPMC • Added VM ID flag to FFA_MSG_SEND2 at NS physical FF-A instance • Clarified encoding of information returned by FFA_NOTIFICATION_INFO_GET ABI • Clarified that multiple StMM SPs are identified using a single UUID • Clarified guidance on SP to OS kernel indirect message passing • Restricted usage of FFA_MEM_PERM_GET/SET ABIs to primary PE and only during initialization
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

Date	Version	Changes
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
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

Arm Non-Confidential Document License (“License”)

This License is a legal agreement between you and Arm Limited (“**Arm**”) for the use of Arm’s intellectual property (including, without limitation, any copyright) embodied in the document accompanying this License (“**Document**”). Arm licenses its intellectual property in the Document to you on condition that you agree to the terms of this License. By using or copying the Document you indicate that you agree to be bound by the terms of this License.

“**Subsidiary**” means any company the majority of whose voting shares is now or hereafter owner or controlled, directly or indirectly, by you. A company shall be a Subsidiary only for the period during which such control exists.

This Document is **NON-CONFIDENTIAL** and any use by you and your Subsidiaries (“Licensee”) is subject to the terms of this License between you and Arm.

Subject to the terms and conditions of this License, Arm hereby grants to Licensee under the intellectual property in the Document owned or controlled by Arm, a non-exclusive, non-transferable, non-sub-licensable, royalty-free, worldwide License to:

(i) use and copy the Document for the purpose of designing and having designed products that comply with the Document;

(ii) manufacture and have manufactured products which have been created under the License granted in (i) above; and

(iii) sell, supply and distribute products which have been created under the License granted in (i) above.

Licensee hereby agrees that the Licenses granted above shall not extend to any portion or function of a product that is not itself compliant with part of the Document.

Except as expressly licensed above, Licensee acquires no right, title or interest in any Arm technology or any intellectual property embodied therein.

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm’s view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

Reference by Arm to any third party’s products or services within this document is not an express or implied approval or endorsement of the use thereof.

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

NOTWITHSTANDING ANYTHING TO THE CONTRARY CONTAINED IN THIS LICENSE, TO THE FULLEST EXTENT PERMITTED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, IN CONTRACT, TORT OR OTHERWISE, IN CONNECTION WITH THE SUBJECT MATTER OF THIS LICENSE (INCLUDING WITHOUT LIMITATION) (I) LICENSEE’S USE OF THE DOCUMENT; AND (II) THE IMPLEMENTATION OF THE DOCUMENT IN ANY PRODUCT CREATED BY LICENSEE UNDER THIS LICENSE). THE EXISTENCE OF MORE THAN ONE CLAIM OR SUIT WILL NOT ENLARGE OR EXTEND THE LIMIT. LICENSEE RELEASES ARM FROM ALL OBLIGATIONS, LIABILITY, CLAIMS OR DEMANDS IN EXCESS OF THIS LIMITATION.

This License shall remain in force until terminated by Licensee or by Arm. Without prejudice to any of its other rights, if Licensee is in breach of any of the terms and conditions of this License then Arm may terminate this License immediately upon giving written notice to Licensee. Licensee may terminate this License at any time. Upon termination of this License by Licensee or by Arm, Licensee shall stop using the Document and destroy all copies of the Document in its possession. Upon termination of this License, all terms shall survive except for the License grants.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

Arm document reference: PRE-21585

version 5.0, March 2024

Contents

Arm Firmware Framework for Arm A-profile

Preface

The guidance in this document constitutes the *Base* specification of the Firmware Framework. This document is accompanied by the following supplement specification:

1. FF-A memory management protocol [1]

The topics covered by the supplement specification are an extension to the Base specification. It has been separated from the Base specification for the sake of its brevity. The complete specification of the Firmware Framework is a combination of the Base specification and the supplement specification. This change is applicable from v1.2 ALP1 of the Base specification.

The supplement specification is not versioned independently. Instead, when a supplement specification undergoes a change, it adopts the latest version of the Base specification at the time of its release.

The guidance in this document can be at a different ALPHA quality level as compared to a supplement specification. E.g. this document could be at ALP3 while a supplement specification is at ALP1. To achieve alignment with the supplement specification, this document can be at the BETA quality level only if all supplement specifications are also at the same quality level. This approach allows this document to evolve somewhat independently of a supplement specification w.r.t quality levels, but also provides a point of alignment at the BETA quality level.

The reader is expected to use the guidance in this document in conjunction with the supplement specification.

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® FF-A memory management protocol version 1.2*. See https://developer.arm.com/documentation/den0140/c

[2] *Arm® System Memory Management Unit Architecture Specification*. See https://developer.arm.com/documentation/ihi0070/latest

[3] *Arm® System Memory Management Unit Architecture specification version 2.0*. See https://developer.arm.com/documentation/ihi0062/latest

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

[5] *SMC Calling Convention v1.2*. See https://developer.arm.com/documentation/den0028/c

[6] *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*. See https://developer.arm.com/documentation/ddi0487/latest

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

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

[9] *Management Mode Interface Specification*. See https://developer.arm.com/documentation/den0060/latest

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

[11] *Power State Coordination Interface*. See https://developer.arm.com/documentation/den0022/latest

Feedback

Arm welcomes feedback on its documentation.

If you have any comments or queries about our documentation, create a ticket at https://support.developer.arm.com/.

As part of the ticket, include:

- The title, Arm Firmware Framework for Arm A-profile.
- The document ID and version, DEN0077A 1.2.
- The section name to which your comments refer.
- The page number(s) to which your comments apply.
- The rule identifier(s) to which your comments apply, if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Chapter 1

Document organization

This document is organized as follows.

1. Chapter 2 *Introduction* provides an overview of the challenges being address by the Firmware Framework.
2. Chapter 3 *Software architecture* provides an overview of the Firmware Framework software architecture.
3. Chapter 4 *Concepts* describes some fundamental concepts that are used to define the Firmware Framework software architecture.
4. Chapter 5 *Setup* specifies the information contained in a partition manifest and how it is used to initialize a partition by a partition manager.
5. Chapter 6 *Identification and Discovery* describes how FF-A components are identified and can be discovered by other components in the system.
6. Chapter 7 *Message passing* describes the mechanisms that partitions can use for message passing.
7. Chapter 8 *Partition runtime models* describes the state transitions partitions are permitted make in the run-time models that their partition manager implements.
8. Chapter 9 *Interrupt management* specifies guidance on interrupt management in the Secure world.
9. Chapter 10 *Notifications* describes support for notifications. This is a mechanism that one partition can use to ring a *doorbell* of another partition.
10. Chapter 11 *Interface overview* provides an overview of the ABIs defined by the Firmware Framework.
11. ABIs used in the Firmware Framework for status reporting, setup and discovery of partitions, scheduling, messaging, notifications and interrupt management are specified in the following sections.
    - Chapter 12 *Status reporting interfaces*.
    - Chapter 13 *Setup and discovery interfaces*.

- Chapter 14 *CPU cycle management interfaces*.
- Chapter 15 *Messaging interfaces*.
- Chapter 16 *Notification interfaces*.
- Chapter 17 *Interrupt management interfaces*.

12. Chapter 18 *Appendix* provides guidance on the following additional topics.

    - 18.1 *S-EL0 partitions*.
    - 18.2 *Power Management*.
    - 18.3 *VM availability signaling*.
    - 18.4 *Legacy Indirect messaging usage*.

13. The FF-A memory management protocol [1] describes the mechanisms and ABIs that partitions can use for memory management.

Chapter 2
Introduction

The Armv8.4 architecture introduces the Virtualization extension in the Secure state. The Arm® SMMU v3.2 architecture [2] 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 v1.0 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 [3]). 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* [4].

Chapter 3

Software architecture

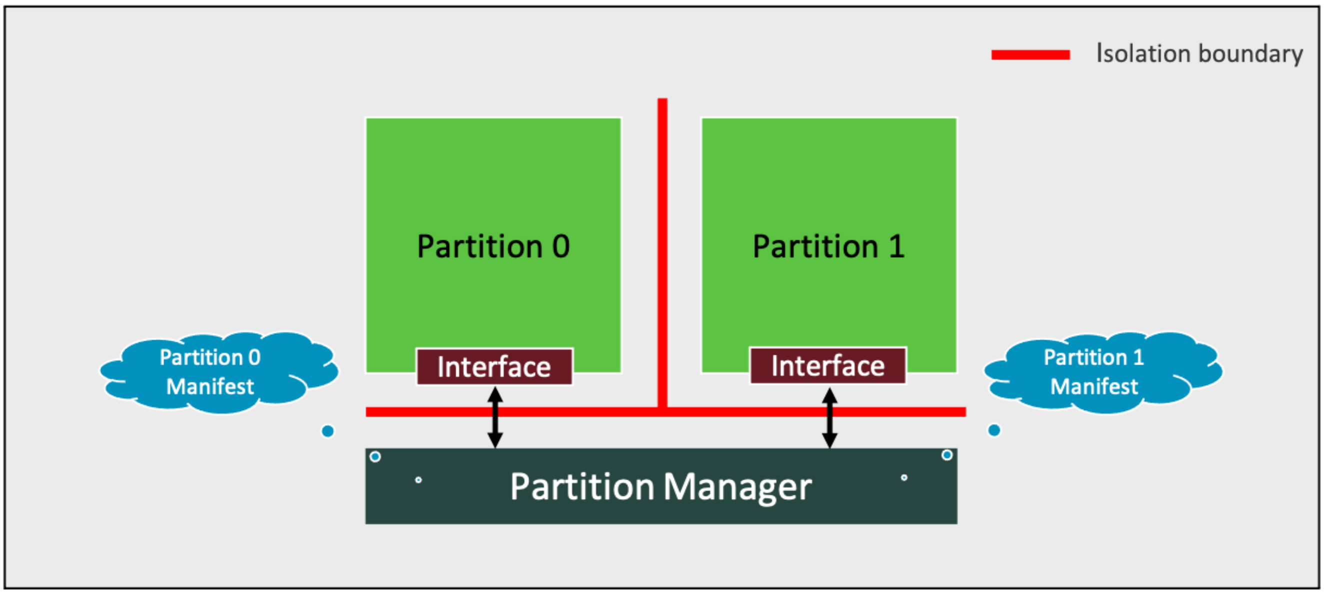

Figure 3.1: FF-A software architecture

The Firmware Framework is made up of the following building blocks as illustrated in Figure 3.1.

1. Isolation boundaries.
2. Partition interfaces.

3. Partitions.
4. Partition manifest.
5. Partition manager.

The following sub-sections describe these building blocks in more detail.

3.1 Isolation boundaries

The Framework defines two types of isolation boundaries.

1. A *Logical isolation boundary* that can be used to,
    1. Isolate one software module e.g. a library or a device driver from another within a software image in an exception level through an IMPLEMENTATION DEFINED mechanism. One or more services implemented inside a module are accessed through a IMPLEMENTATION DEFINED application programming interface (API).
    2. Isolate one software image from another by,
        1. Deploying them in separate exception levels.
        2. Deploying them in the same exception level. The images are temporally isolated from each other on a given PE.

        One or more services implemented inside a software image are accessed through an application binary interface (ABI).
2. A *Physical isolation boundary* that can be used to spatially isolate the physical address space of one software image from another through the following mechanisms.
    1. *The Arm® TrustZone Security extension*. It is used to protect the Secure physical address space ranges assigned to software images in the Secure state from software images in the Non-secure state.
    2. *Virtual memory-based memory protection provided by the Arm A-profile VMSA*. It is used to protect the physical address space ranges assigned to a software image from other software images in the same security state.

The Framework assumes that a physically isolated software image is logically isolated as well. For example,

- A Guest OS running inside a VM is both physically and logically isolated from a Guest OS in another VM.
- A Hypervisor running in EL2 is both physically and logically isolated from all VMs it manages.
- Firmware in EL3 is physically and logically isolated from any software image in the Normal world.

The Framework does not assume that a logically isolated software image is physically isolated as well. For example, a Trusted OS in S-EL1 is logically but not physically isolated from firmware in EL3 when any of the following scenarios apply.

1. S-EL2 is not present i.e. *ID_AA64PFR0_EL1.SEL2=0*.
2. S-EL2 is not enabled on the system by setting *SCR_EL3.EEL2=1*.
3. S-EL2 is present and enabled but Stage 2 address translation in the Secure EL1&0 translation regime is disabled i.e. *HCR_EL2.VM=0*.

The two images are not physically isolated since software in S-EL1 can access the physical address space of software in EL3.

The Framework defines ABIs that enable communication between software images across an exception level boundary. The images are logically isolated and could be physically isolated as well.

3.2 Partitions

A *partition* is defined as a software module or image that implements one or more services within an isolation boundary such that a service is accessible across the boundary only via well defined interfaces. If the partition is a software image, then the well defined interface is an FF-A ABI. If the partition is a software module, the well defined interface is an IMPLEMENTATION DEFINED API.

The Framework defines ABIs 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. Synchronous and asynchronous message passing between partitions.
3. Memory management between partitions.

A partition that is logically isolated but not physically isolated is called a *logical partition*. A partition that is both physically and logically isolated is called a *physical partition*. The term *partition* is used when it is not required to distinguish between a logical and physical partition. The term *endpoint* is used interchangeably with the term *partition*.

1. A VM (when the virtualization extension is enabled) or the OS kernel (when the virtualization extension is disabled or unavailable) is a physical or logical endpoint that runs in EL1 in the Non-secure security state. These endpoints are called *NS-Endpoints* in scenarios where it is not necessary to distinguish between them.

2. A partition in the Secure security state is called a *Secure Partition* (SP) and could be,

    1. A logical partition that runs in EL3, S-EL2 or S-EL1. A logical partition in the Secure security state is called a *Logical Secure Partition (LSP)*.
    2. A physical partition that runs in S-EL1 or S-EL0.

    SPs are called *S-Endpoints* in scenarios where it is not necessary to distinguish between them on the basis of the exception level they run in.

A *partition manifest* describes the physical address space ranges and system resources a partition needs, identity of partition services to enable their discovery and other attributes of the partition that govern its run-time behavior (also see Chapter 5 *Setup*).

3.3 Partition manager

A *partition manager* is responsible for creating and managing the physical isolation boundary of a partition. It uses a partition manifest to assign physical address space ranges and system resources to a partition, initialize it as per the specified attributes and enable discovery of its services. The partition manager also implements FF-A ABIs to enable inter-partition communication for access to partition services.

1. In the Secure world, this component is called the *Secure Partition Manager* (SPM).
2. In the Normal world it is a Hypervisor[1] (if the virtualization extension is enabled).

A partition manager is physically isolated from physical partitions and logically isolated from logical partitions it manages. All partitions managed by a partition manager reside at the same or a numerically lower exception level than the partition manager.

The term *partition manager* is used in the rest of this specification to collectively refer to the SPM and Hypervisor in scenarios where they have the same responsibilities, and it is not necessary to distinguish between them.

The Hypervisor uses the virtualization extension in the Arm A-profile VMSA to create physical isolation boundaries as follows.

- The *EL1&0 stage 2 translation regime, when EL2 is enabled* in a PE in the Non-secure state, is used to restrict visibility of the Non-secure physical address space from a VM to only those regions that have been assigned to the VM.

See 4.1 *SPM architecture* for a description of how the SPM creates and manages isolation boundaries for SPs.

See 4.2 *DMA isolation* for a description of how a partition manager creates and manages isolation boundaries for DMA capable devices.

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 trusts the Hypervisor to protect its resources from other VMs by creating and maintaining the correct physical isolation boundaries in the Non-secure physical address space.
- Every endpoint trusts the SPM to protect its resources from other endpoints by creating and maintaining the correct physical isolation boundaries in both the Secure and Non-secure physical address spaces.
- An SP does not trust the state of any Non-secure resource it has access to. Therefore, it does not trust the Hypervisor or a NS-Endpoint that could also access the same resource.

The term *FF-A component* is used to collectively refer to partitions and partition managers.

[1] 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.

3.4 Example configurations

The Non-secure and Secure security states in the Arm A-profile architecture typically adopt a client-server model where a partition in the Non-secure state is a client of services implemented by a partition in the Secure state.

Partitions within a security state could adopt the client-server model as well. Furthermore, a partition can be both a consumer of another partition's services and provider of its own services.

The FF-A software architecture generalizes the programming model to access a partition's services within and between the Non-secure and Secure security states.

Some example deployment scenarios of the FF-A software architecture on various configurations of an Arm A-profile system are listed in the following sub-sections.

3.4.1 FF-A deployment without S-EL2

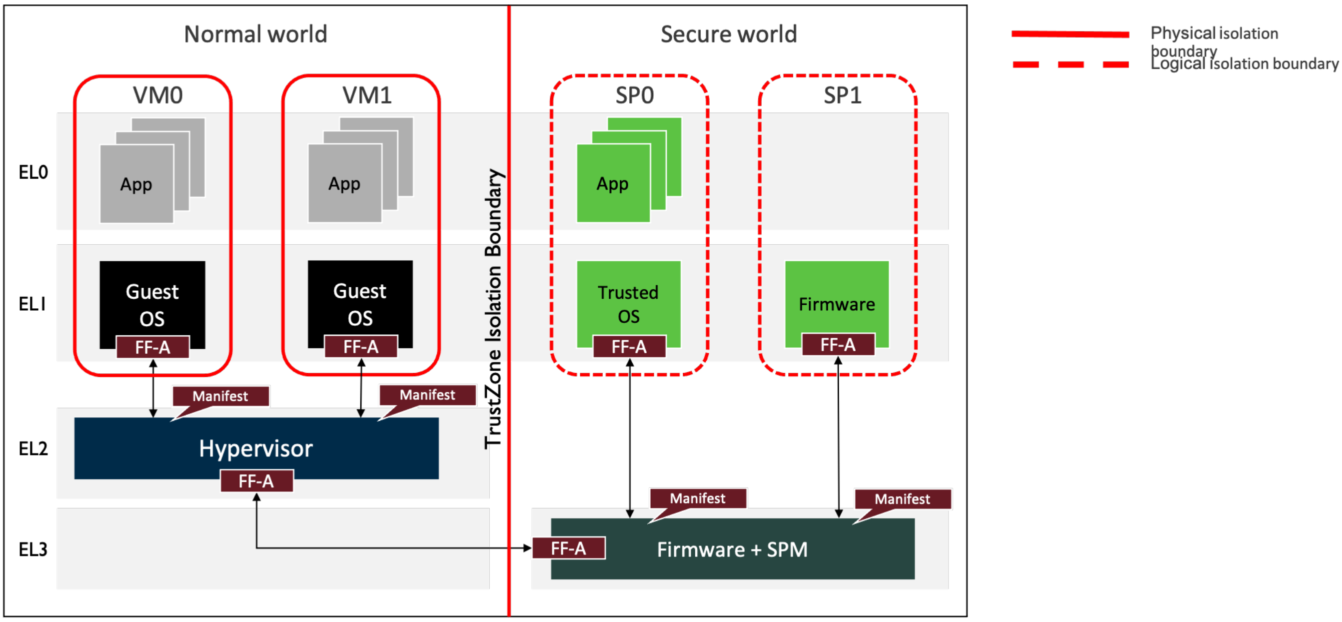

Figure 3.2: Example FF-A deployment without S-EL2

In Figure 3.2, the virtualization extension is enabled in the Non-secure state. It is either unavailable or disabled in the Secure state.

Both VM0 and VM1 implement an FF-A driver in EL1 to access services in S-Endpoints. They could use the same driver to access each other's services as well.

The Hypervisor facilitates access to services in S-Endpoints from VM0 and VM1 by implementing an FF-A driver in EL2. It could use the same driver to enable them to access each other's services.

The following software images are deployed in the Secure world.

1. A firmware image in EL3. It implements the SPM.
2. A firmware image in S-EL1 (SP1).
3. A Trusted OS image in S-EL1 (SP0).

SP0 and SP1 are temporally isolated logical partitions and could access each other's services via the SPM. The SPM is logically isolated from SP0 and SP1.

3.4.2 FF-A deployment with S-EL2

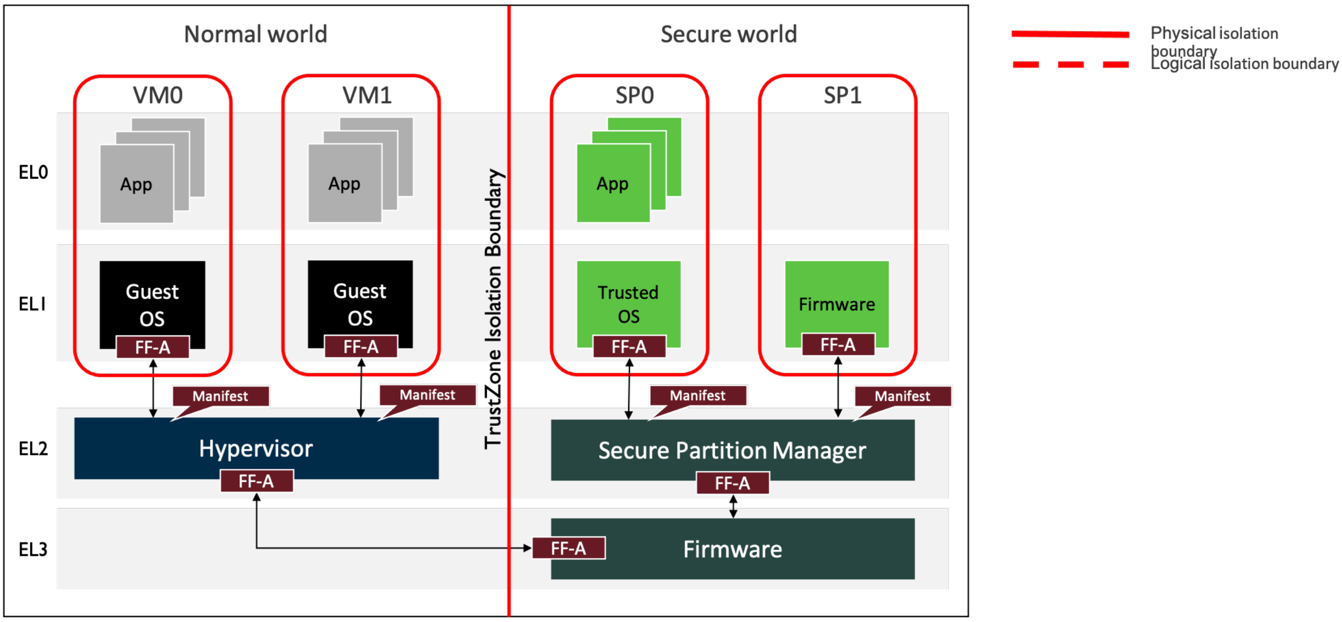

Figure 3.3: Example FF-A deployment with S-EL2

In Figure 3.3, the virtualization extension is enabled in both security states. The Normal world software stack is unchanged from Figure 3.2.

The following software images are deployed in the Secure world.

1. A firmware image in EL3.
2. An SPM image in S-EL2.
3. A firmware image in S-EL1 (SP1).
4. A Trusted OS image in S-EL1 (SP0).

SP0 and SP1 are physical partitions and could access each other's services via the SPM. The SPM is physically isolated from SP0 and SP1.

3.4.3 FF-A deployment with S-EL2 and Armv8.1-VHE

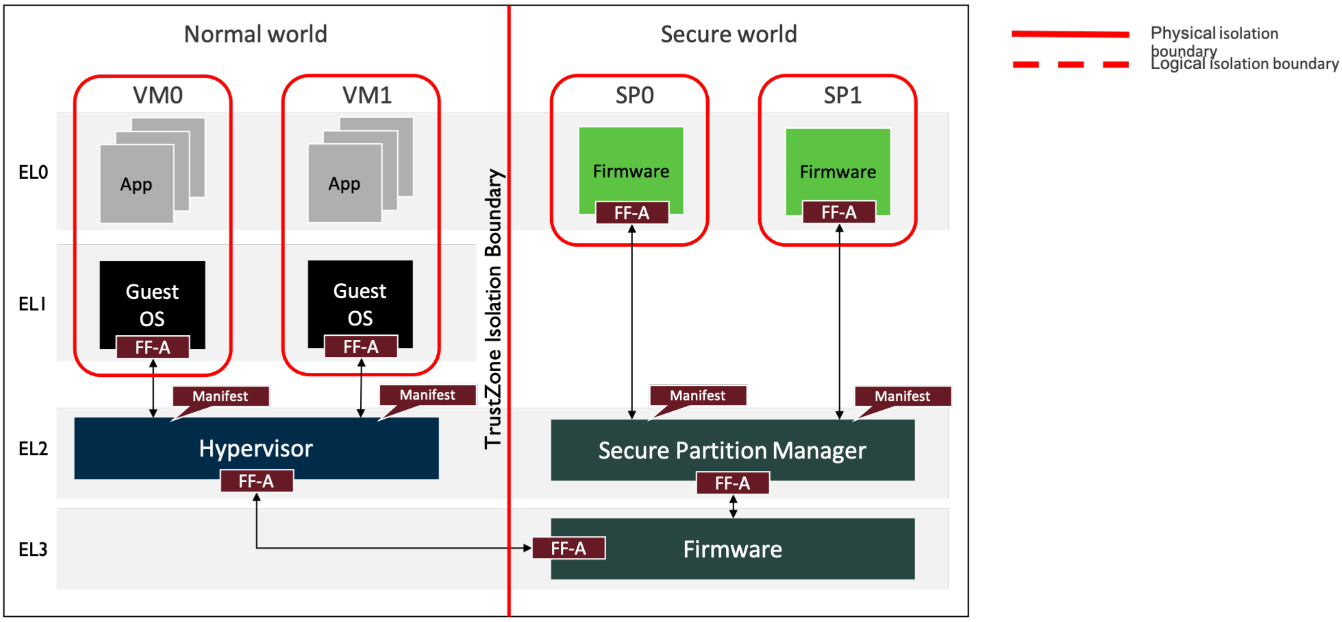

Figure 3.4: Example FF-A deployment with S-EL2 and Armv8.1-VHE

In Figure 3.4, the virtualization extension is enabled in both security states. Additionally, Armv8.1 VHE is enabled in the Secure world to manage S-EL0 SPs. The Normal world software stack is unchanged from Figure 3.2.

The following software images are deployed in the Secure world.

1. A firmware image in EL3.
2. An SPM image in S-EL2.
3. A firmware image in S-EL0 (SP1).
4. A firmware image in S-EL0 (SP0).

SP0 and SP1 are physical partitions that could access each other's services. The SPM is physically isolated from SP0 and SP1.

Chapter 4
Concepts

4.1 SPM architecture

The responsibilities of the SPM are split between two components: the *SPM Dispatcher* (SPMD) and *SPM Core* (SPMC). Both components have access to the entire physical address space and are a part of the *Trusted computing base*. The term SPM is used when it is not necessary to distinguish between these two components. The responsibilities of these components are listed below.

1. The SPMD resides in EL3 and runs in either the AArch64 or AArch32 execution state. It 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.
2. The SPMC either co-resides with the SPMD in EL3 or in an adjacent exception level i.e. S-EL1 or S-EL2. It is responsible for:
   - SP 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.

The SPM initializes EL3 LSPs at boot time and facilitates communication between them and other endpoints during runtime.

Table 4.1 lists the SPMC and SPMD configurations supported by the Framework vis-a-vis the exception levels they can reside in and the execution states they can run in.

Table 4.1: Valid SPM configurations in AArch64 and AArch32 Execution state

SPM config number	SPMD EL and Execution state	SPMC EL and Execution state	Name of configuration
1.	EL3 (AArch64)	EL3 (AArch64)	EL3 SPMC
2.	EL3 (AArch32)	EL3 (AArch32)	EL3 SPMC
3.	EL3 (AArch64)	S-EL1 (AArch64)	S-EL1 SPMC
4.	EL3 (AArch64)	S-EL1 (AArch32)	S-EL1 SPMC
5.	EL3 (AArch64)	S-EL2 (AArch64)	S-EL2 SPMC

In SPM configurations where the SPMD and SPMC reside in adjacent exception levels,

- They implement and report a mutually compatible version of the Firmware Framework. See 13.2.3 *SPM usage* for details.
- The mechanism used by the SPMD to initialize the SPMC is IMPLEMENTATION DEFINED. The guidance provided in Chapter 5 *Setup* could be used by the implementation.
- They use the ABIs defined in this specification for communication.

A description of each SPM configuration is provided in the following sections.

- 4.1.1 *Secure EL2 SPM core component*.
- 4.1.3 *EL3 SPM core component*.
- 4.1.2 *S-EL1 SPM core component*.

The SPM configurations without S-EL2 are used in the following scenarios.

- Reduce the size of the TCB by migrating EL3 & S-EL1 firmware components, that should not be a part of the TCB, to one or more physically isolated S-EL0 SPs.
- Make the TCB implementation more robust by migrating its components from EL3 & S-EL1 to one or more physically isolated S-EL0 SPs.
- Adopt the generalized programming model specified by the Framework to ease the migration of the Secure world software stack to an Arm A-profile system with S-EL2 enabled.
- Adopt the generalized programming model specified by the Framework for accessing services in S-Endpoints from NS-Endpoints irrespective of whether S-EL2 is used in the Secure world.

4.1.1 Secure EL2 SPM core component

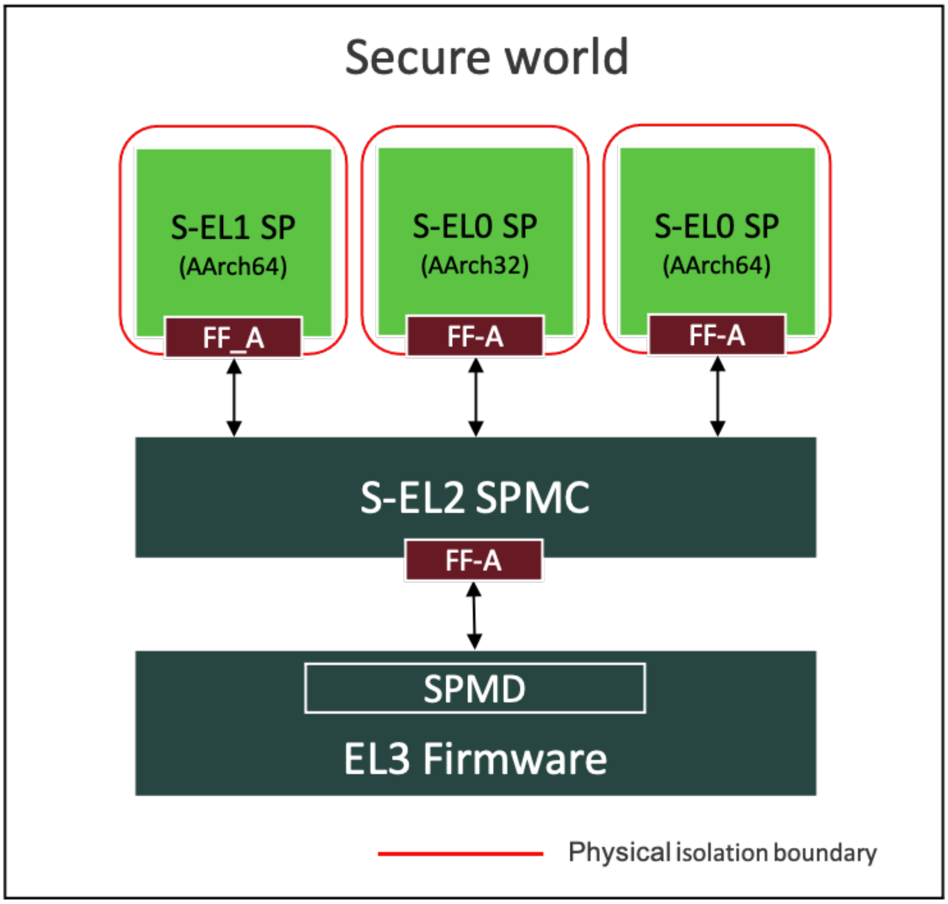

Figure 4.1: Example S-EL2 SPM Core and SP configuration

The S-EL2 SPMC is fundamental to enforcing the principle of least privilege in the Secure state on Armv8.4 or later systems as described in Chapter 2 *Introduction*. It supports one or more of the following SP configurations.

1. The SPMC uses *Armv8.1 VHE* to manage one or more physical SPs that run in S-EL0. Each SP runs in either the AArch32 or AArch64 execution state.

   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 physical SPs that run in S-EL1. Each SP runs in either the AArch32 or AArch64 execution state.

   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 4.1. Additionally, in each of the above configurations, the following LSP configurations can exist with the S-EL2 SPMC:

1. EL3 LSPs that are managed by the SPMD.
2. S-EL2 LSPs that are managed by the S-EL2 SPMC.

4.1.2 S-EL1 SPM core component

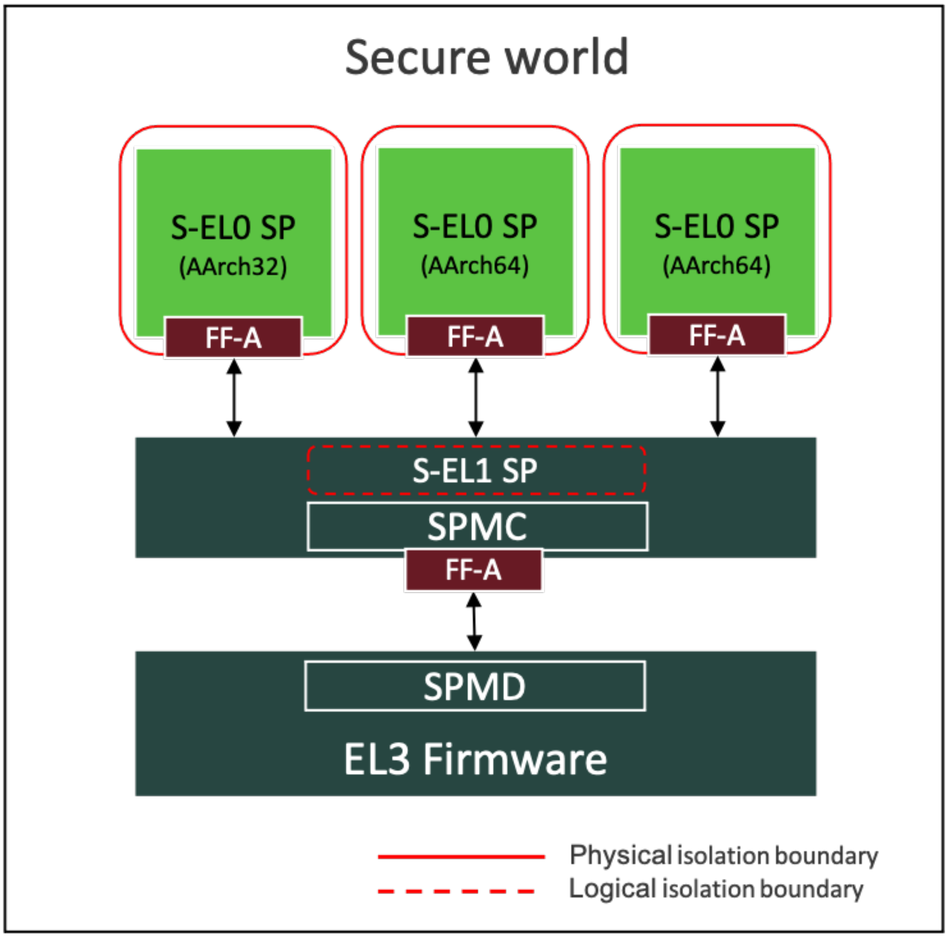

Figure 4.2: Example S-EL1 SPM Core and SP configuration

A S-EL1 SPMC runs in either the AArch64 or AArch32 execution state. It supports one or more of the following SP configurations.

1. The SPMC manages one or more physical SPs that run in S-EL0. Each SP runs in either the AArch32 or AArch64 (only if S-EL1 SPMC runs in AArch64 too) execution state.

   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* in either execution state.

2. The SPMC manages a single LSP that also runs in S-EL1. The SPMC and LSP are packaged in the same software image and logically isolated from each other.

   In this configuration:

   - 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 SP component through the IMPLEMENTATION DEFINED interface.
   - The SPMC and SP are initialized through an IMPLEMENTATION DEFINED mechanism. See Chapter 5 *Setup* for more information.

Figure 4.2 illustrates a combination of these configurations. Additionally, in each of the above configurations, EL3 LSPs that are managed by the SPMD can exist with the S-EL1 SPMC.

4.1.3 EL3 SPM core component

The EL3 SPMC co-exists with the SPMD in either the AArch64 or AArch32 execution state. It supports one of the following mutually exclusive SP configurations.

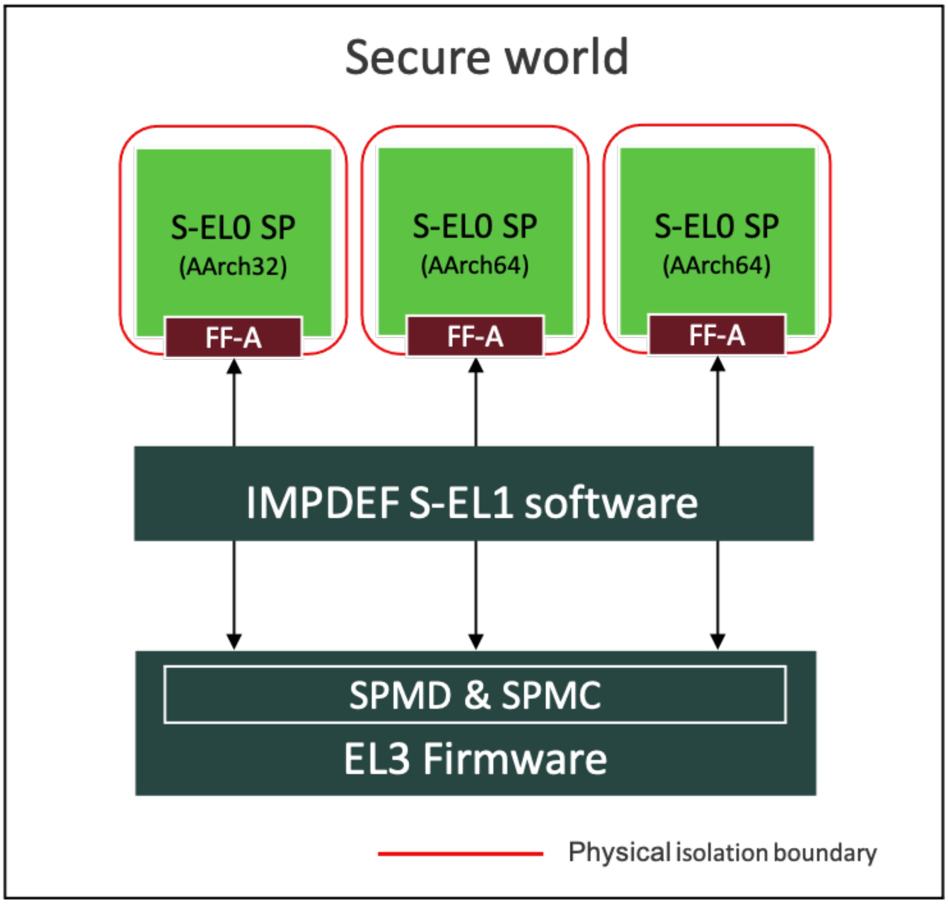

Figure 4.3: Example EL3 SPM Core and S-EL0 SP configuration

1. One or more physical SPs that run in S-EL0. Each SP runs in either the AArch32 or AArch64 (only if EL3 SPMC runs in AArch64 too) execution state.

   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*.

   This configuration is illustrated in Figure 4.3.

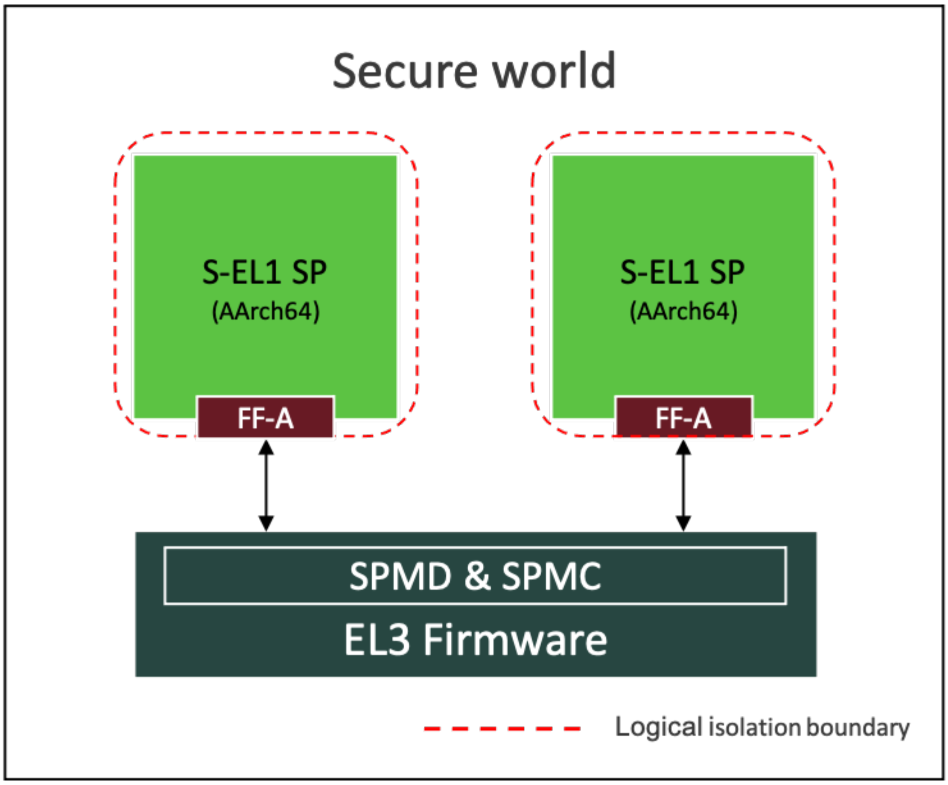

Figure 4.4: Example EL3 SPM Core and S-EL1 SP configuration

2. The SPMC and SPMD co-exist in EL3 in the AArch64 execution state. One or more logical SPs reside in S-EL1. Each SP runs in either the AArch32 or AArch64 execution state. The SPs are temporally isolated from each other by the SPMC.

   This configuration is illustrated in Figure 4.4.

Additionally, in each of the above configurations, EL3 LSPs can exist that are managed by the SPM. The division of responsibilities of the SPMD and EL3 SPMC in managing EL3 LSPs is IMPLEMENTATION DEFINED.

4.2 DMA isolation

The Framework enables the partition manager to control the visibility of the physical address space from a DMA capable device assigned to a partition. The Framework assumes that the system,

1. Implements an access control mechanism that can be programmed by a partition manager to limit accesses from DMA capable devices to specific ranges in the physical address space.
2. Guarantees that an access from a DMA capable device is allowed to complete only if,
    1. It is permitted by the access control mechanism.
    2. The access control mechanism is disabled by the partition manager.

The DMA capable device is said to reside *upstream* of the access control mechanism. Examples of access control mechanisms include an Arm® SMMU implementation or a vendor specific System MPU implementation.

If the system implements an Arm® SMMU, each access/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 security state of the transaction and the stage 1 and/or 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.

- The Hypervisor programs the SMMU to limit access to the Non-secure physical address space in response to transactions generated by a DMA capable device using a Non-secure Stream ID.
- The SPMC programs the SMMU to limit access to the Non-secure and Secure physical address spaces in response to transactions generated by a DMA capable device using a Secure Stream ID.

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 [1].

The Framework specifies the following programming models w.r.t DMA isolation.

1. Programming models that enable a partition to control the visibility that a DMA capable device, assigned to the partition has of the partition's physical memory regions. These models are described in 4.2.1 *Static DMA isolation* and 4.2.2 *Dynamic DMA isolation*. A partition uses one or the other model but never both.
2. Programming models that allow,
    1. A trusted DMA capable device to manage its access to the physical address space.
    2. A trusted partition to act on behalf of a DMA capable device to manage its access to the physical address space. The device is not assigned to the trusted partition.

    These models are described in 4.2.3 *Other DMA isolation models*.

On a system that does not implement an Arm® SMMU, the Framework assumes that the guidance in this specification can be applied to the IMPLEMENTATION DEFINED access control mechanism available on the system.

4.2.1 Static DMA isolation

In this model, a partition uses its manifest to specify the memory regions in its physical address space that must be visible to each DMA capable device assigned to it along with memory attributes such as read, write and execute permissions. A device cannot access the partition's memory regions unless access is explicitly granted in the partition manifest. The partition manager programs the access control mechanism to create the corresponding memory mappings before initializing the partition. These mappings remains in place for the lifetime of the partition. They cannot be changed during partition initialization and runtime through mechanisms defined by the Framework e.g. management transactions described in [1].

The static DMA isolation model is used on a system with an Arm® SMMU as described below.

1. The partition manager uses a single stage of address translation to enforce access control in the SMMU. This could be either the stage 1 or the stage 2 translation regime in the SMMU.
2. An EL1 partition in either security state uses this model to enable a DMA capable device to use a memory range in the partition's IPA space to access a memory range in the partition's PA space.
3. A S-EL0 partition uses this model to enable a DMA capable device to use a memory range in the partition's VA space to access a memory range in the partition's PA space.
4. The following properties of the device are specified in the memory region description (see Table 5.2) in the partition manifest (see 5.2.1 *Partition manifest*).
    1. The identity of the stream ID generated by the device that must have access to the memory region.
    2. The identity of the SMMU that the device is upstream of.
    3. The instruction and data access permissions on the memory region that must be used by transactions associated with the stream ID.

    The Framework assumes that the following attributes of the memory region are same irrespective of whether it is accessed by the partition or the device stream ID.

    - Memory type
    - Cacheability and shareability attributes.
    - Security state.
5. The specified device stream ID accesses the physical memory region with either the same IPA range used by an EL1 or S-EL1 partition or the same VA range used by a S-EL0 SP. The partition manager creates mappings for the memory region in either the Stage 1 or Stage 2 translation regime of the SMMU.

4.2.2 Dynamic DMA isolation

In this model, a partition controls the visibility that DMA capable devices assigned to it have of the partition's physical memory regions, during partition initialization and runtime. The partition manager programs the access control mechanism prior to partition initialization to ensure the devices cannot access the partition's physical address space. Memory mappings for the devices are created and destroyed by the partition manager on behalf of the partition.

The dynamic DMA isolation model is used on a system with an Arm® SMMU as described below.

1. The partition manager enables a single or both stages of address translation to enforce access control in the SMMU. This could be the stage 1, stage 2 or both translation regimes in the SMMU.
2. An EL1 partition in either security state uses this model to enable a DMA capable device to use a memory range in the partition's IPA or VA space to access a memory range in the partition's PA space.
3. A S-EL0 partition uses this model to enable a DMA capable device to use a memory range in the partition's VA space to access a memory range in the partition's PA space.
4. The partition uses an IMPLEMENTATION DEFINED mechanism during initialization and runtime to describe a memory region to the partition manager that must be *mapped* or *unmapped* from a device assigned to it. For example,
    1. A partition implements an SMMU driver to program Stage 1 translations so that memory regions in its VA space can be mapped or unmapped from a device.

        The partition manager *emulates* the SMMU accesses from the partition and ensures accesses from the device are restricted to the physical memory regions assigned to the partition.
    2. The partition manager exports a *para-virtualized* interface to the partition to program the SMMU so that memory regions in the partition's VA space can be mapped or unmapped from a device. The partition is able to specify the address, size and attributes of the memory region through the interface.

5. The use of a single or both stages of translation in the SMMU by the partition manager is IMPLEMENTATION DEFINED. The Framework specifies the following rule if the partition manager enables both stages of translations in the SMMU.

   Stage 2 translations for the partition are shared with the SMMU i.e. the Stream Table Entry (STE) in a Stream table selected by the stream ID references the same stage 2 translation tables used by the partition. The stream ID is generated by a device assigned to the partition. This also implies that any memory region,

   1. Shared, lent or donated to the partition through memory management transactions described in [1] is automatically mapped into the IPA space of a DMA capable devices assigned to the partition.

      The partition uses an IMPLEMENTATION DEFINED mechanism after the memory region is mapped in the stage 2 translation tables to program the Stage 1 translation tables in the SMMU to enable access to the memory region from a device.

   2. Relinquished by the partition is automatically unmapped from the IPA space of a DMA capable devices assigned to the partition.

      The partition uses an IMPLEMENTATION DEFINED mechanism before the memory region is unmapped from the stage 2 translation tables to program the Stage 1 translations in the SMMU to disable access to the memory region from a device.

4.2.3 Other DMA isolation models

The Arm® SMMU v3.2 architecture supports stage 1 and stage 2 translations in both security states. This enables a programming model in which each stage of translation corresponding to a stream ID of a DMA capable device is managed by a different FF-A component.

- The stage 1 translations are managed by the partition to which the device that generates the stream ID is assigned. This is done through an IMPLEMENTATION DEFINED interface between the partition and its partition manager.
- The stage 2 translations are managed by the device itself or another partition trusted by the device. In the former case, this is done through an IMPLEMENTATION DEFINED interface between the device and its partition manager. In the latter case, this is done through the mechanisms specified by the Framework.

This model enables separation of the partition that programs the device from the FF-A component that implements the policy for controlling the device's visibility of the physical address space. Physical memory regions are mapped and unmapped from the stage 2 translation tables by the partition manager in response to memory management transactions (see [1]) as per the DMA isolation policy for the device.

The Framework defines the following concepts to support management of stage 2 translations in this programming model.

1. *Stream endpoint*. It is a set of SMMU stage 2 translations maintained by a partition manager on behalf of a DMA capable device. 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.

   - 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*.

   In its simplest form, where a device generates a single stream ID and does not share access to the physical address space with stream IDs of other devices, the SEPID effectively identifies the device.

   SEPIDs are used in memory management transactions to (also see [1]):

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

   Each Stream endpoint is assigned a 16-bit ID called the *Stream endpoint ID* or *SEPID*.

A Stream endpoint is a physical endpoint since its physical address space can be spatially isolated from other FF-A components by its partition manager through stage 2 translations in the SMMU. Also see 3.1 *Isolation boundaries*.

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.

SEPID values must be distinct from those assigned to PE endpoints. A SEPID is discoverable via an FF-A partition discovery mechanism. Also see Chapter 6 *Identification and Discovery*.

2. *Independent peripheral device*. It is a DMA capable device that can initiate and receive memory management transactions. Each device specifies the following information in its partition manifest (see 5.2.3 *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 and an independent peripheral device use an IMPLEMENTATION DEFINED mechanism to notify each other about a memory management transaction targeted to a SEPID used by the device (see [1]).

3. *Dependent peripheral device*. It is a DMA capable device that cannot initiate and receive memory management transactions. It relies on a trusted PE endpoint to initiate and receive memory management transactions on its behalf. The PE endpoint is called a *proxy endpoint*.

   A dependent device is *assigned* to a PE endpoint that is distinct from its proxy endpoint. This implies,

   - Access to its MMIO regions is assigned to the endpoint during boot (see 4.8 *System resource management* & Table 5.3).

   - 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 5.3).

   The partition manifest of the *proxy endpoint* (see 5.2.1 *Partition manifest*) specifies 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 is distinct from the SEPID allocated to manage the preceding association. The SEPID is specified in the partition manifest of the proxy endpoint (see Table 5.1).

   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 is allowed to complete only if it is either initiated or authorized by the *proxy endpoint* for the device (see [1]).

   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.

4.3 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 4.4 *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*.
- Partitions are physically isolated at a virtual FF-A instance.
- Partitions are logically isolated at a physical FF-A instance.
- The instance between the SPMC and a logical SP is a Secure physical FF-A instance.
- The instance between the SPMC and a physical 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 4.2 lists the valid Secure FF-A instances. Table 4.3 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)*.

Table 4.2: Secure FF-A instances

EL boundary	EL3 (AArch64)	EL3 (AArch32)	S-EL2	S-EL1 (AArch64)	S-EL1 (AArch32)
S-EL2	Secure physical	NA	NA	NA	NA
S-EL1 (AArch64)	Secure physical	NA	Secure virtual	NA	NA
S-EL1 (AArch32)	Secure physical	Secure physical	Secure virtual	NA	NA
S-EL0 (AArch64)	Secure virtual	NA	Secure virtual	Secure virtual	NA
S-EL0 (AArch32)	Secure virtual	Secure virtual	Secure virtual	Secure virtual	Secure virtual

Table 4.3: Non-secure FF-A instances

EL boundary	EL3 (AArch64)	EL3 (AArch32)	EL2 (AArch64)	EL2 (AArch32)
EL2 (AArch64)	Non-secure physical	NA	NA	NA
EL2 (AArch32)	Non-secure physical	Non-secure physical	NA	NA
EL1 (AArch64)	Non-secure physical	NA	Non-secure virtual	Non-secure virtual

EL boundary	EL3 (AArch64)	EL3 (AArch32)	EL2 (AArch64)	EL2 (AArch32)
EL1 (AArch32)	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 IMPLEMENTATION DEFINED. This is applicable to the SPM configurations described in 4.1.3 *EL3 SPM core component* and 4.1.2 *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.

4.4 Conduits

The Framework defines interfaces to enable communication between various FF-A components (see Chapter 11 *Interface overview*). Each interface is accessible through one or more conduits as follows.

The SMC conduit as described in [5] 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 11 *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 12.2 *FFA_ERROR*).
- Results of the request. The *FFA_SUCCESS* function is used to return the results (see 12.3 *FFA_SUCCESS*).
- An invocation of another interface described in Chapter 11 *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.

4.5 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 FF-A component is called *Normal memory*.
- Memory that is accessed with the NS bit cleared in an FF-A component translation regime is called *Secure memory*.

4.6 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 7.2.2.3 *Buffer attributes* & the FF-A memory management protocol [1]). 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 5.2.1 *Partition manifest*. 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 13.3 *FFA_FEATURES*).

4.7 Execution context

Each endpoint has one or more *execution contexts* depending on its implementation. An execution context comprises 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 4.9 *Primary scheduler*). Figure 4.5 illustrates this relationship.

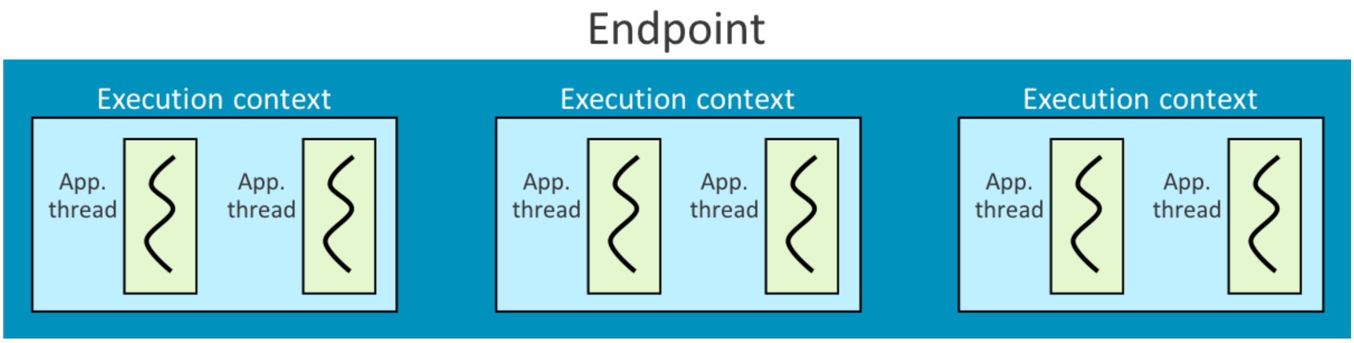

Figure 4.5: Example endpoint with execution contexts and threads

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 5.2.1 *Partition manifest*).
- 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 4.8 *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.

4.8 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 the FF-A memory management protocol [1].

Rules associated with ownership and access of CPU cycles are described in 4.9 *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 4.2 *DMA isolation*).

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 5.3). 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 5 *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 assumes that the following actions pertaining to the MMIO region of a device are performed through an IMPLEMENTATION DEFINED mechanism:

- Transfer of ownership of a device MMIO region to another partition during run-time.
- Grant of access to a device MMIO region to another partition during run-time.
- Revocation of access to a device MMIO region from a partition during run-time.

4.9 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 [6]).
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 4.6 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.

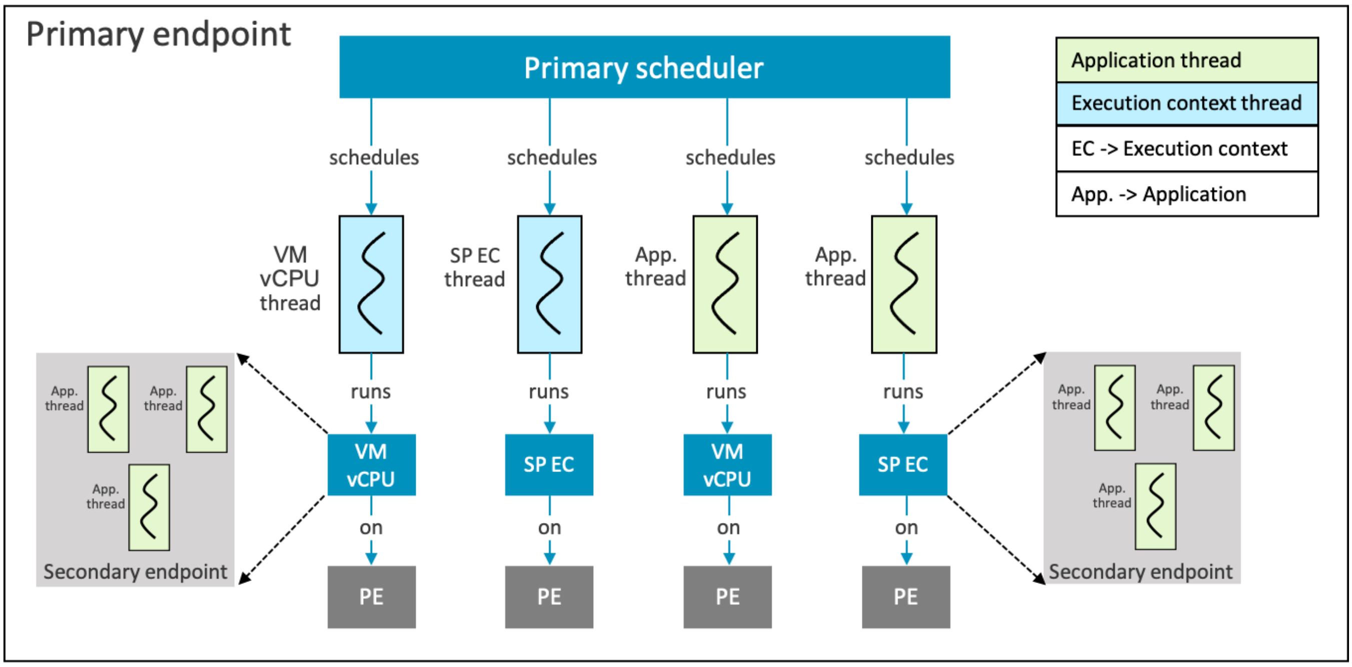

Figure 4.6: Example primary endpoint configuration

Figure 4.7 illustrates this example of a secondary endpoint. The secondary scheduler manages application threads, that could in turn, run execution contexts of SPs.

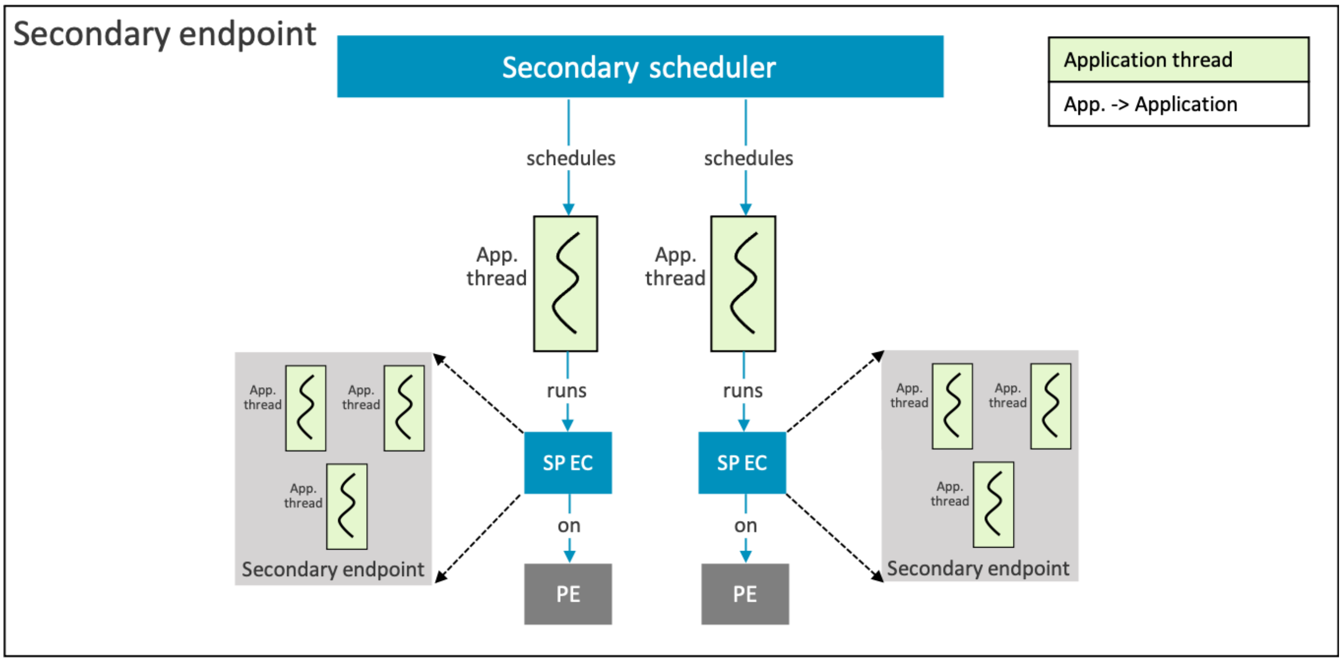

Figure 4.7: Example secondary endpoint configuration

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 the following ABIs to endpoints to allocate CPU cycles to other endpoints. These are,

1. FFA_MSG_SEND_DIRECT_REQ. See 15.2 *FFA_MSG_SEND_DIRECT_REQ*.
2. FFA_MSG_SEND_DIRECT_REQ2. See 15.4 *FFA_MSG_SEND_DIRECT_REQ2*.
3. FFA_RUN. See 14.3 *FFA_RUN*.

4.10 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 ABIs.

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 interrupted by an interrupt while doing work[1].
- *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 4.11 *Run-time state transitions*.

[1] A partition manager could either run another execution context in place of the interrupted execution context or resume the interrupted execution context. The Framework treats the interrupted execution context as being in the *preempted* state irrespective of whether it is resumed immediately or subsequently by the partition manager.

4.11 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 4.4 *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 4.10 *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_REQ2
   3. FFA_MSG_SEND_DIRECT_RESP
   4. FFA_MSG_SEND_DIRECT_RESP2
   5. FFA_RUN
   6. FFA_MSG_WAIT
   7. 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 13 *Setup and discovery interfaces*.
   - FFA_SECONDARY_EP_REGISTER interface in 18.2.2 *Secondary boot protocol*.
   - FFA_MSG_SEND2 messaging interface in Chapter 15 *Messaging interfaces*.
   - Memory management interfaces in [1].

   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 9 *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 18.2.4 *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 8 *Partition runtime models*.

Chapter 5
Setup

5.1 Overview

The Firmware Framework is responsible for partition and partition manager setup during a cold and warm boot. This chapter describes how the Framework initializes the execution context of these components on the primary PE during a cold boot. See 18.2 *Power Management* for the role of the Framework in partition and partition manager setup during a cold boot of a secondary PE or a warm boot of any PE.

- In the Secure world,
  - The SPMD initializes the SPMC (also see 4.1 *SPM architecture*). If they reside in the same exception level, initialization is done in an IMPLEMENTATION DEFINED manner.

    If they reside in separate exception levels, initialization is done either in an IMPLEMENTATION DEFINED manner or by using the following guidance.

    * The SPMC manifest (see 5.2.2 *SPMC manifest*) to determine information such as the entry point address, execution state and Framework version of the SPMC.
    * Guidance on programming general-purpose and system registers prior to invoking the SPMC entry point (see 5.3 *Register state*).
    * Protocol for passing any boot information to the SPMC (see 5.4 *Boot information protocol*).
    * Protocol for indicating completion of initialization (see 5.5 *Protocol for completing execution context initialization*).
  - The SPMC initializes each SP. If they reside in the same exception level (see 4.1.2 *S-EL1 SPM core component* and 4.1.3 *EL3 SPM core component*), initialization is done in an IMPLEMENTATION

DEFINED manner. For example, the information required to initialize the logical SP in the S-EL1 SPMC configuration could be encoded in the SPMC manifest.

If they reside in separate exception levels, the SPMC uses the SP manifest (see 5.2.1 *Partition manifest*) to initialize the SP as described below.

1. Validates the contents of the manifest.
2. Configures the partition as per the properties described in the manifest.
3. Assigns the requested physical address space ranges and system resources to the partition.
4. Isolates a physical SP as per the mechanism described for the SPMC configuration in 4.1 *SPM architecture*).
5. Programs the general-purpose and system register prior to invoking the SP entry point as described in 5.3 *Register state*.
6. Uses the protocol described in 5.4 *Boot information protocol* for passing any boot information to the SP.
7. Uses the runtime model described in 8.5 *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.

5.2 Manifests

5.2.1 Partition manifest

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

- Partition properties as described in Table 5.1.
- Memory regions as described in Table 5.2 (for more information see also [1]).
- Devices as described in Table 5.3.
- Partition boot protocol as described in Table 5.10.

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 5.1: 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.
Partition ID	No	• Pre-allocated partition ID.
UUIDs	Yes	• List of UUIDs associated with the partition. Also see *6.2.3 Partition UUID usage*.
Messaging method	Yes	• This field specifies which messaging methods are supported by the partition for each UUID exported by it. This I could be one or both of Direct and Indirect messaging. These methods are described in Chapter 7 *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. *7.4.1 Discovery and setup* specifies the information that must be provided. • If the partition is associated with multiple UUIDs and a subset of UUIDs are accessible by a specific messaging method, a mapping between a UUID and messaging method can be provided. For more information see *6.2.3 Partition UUID usage*.
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.
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.

Information fields	Mandatory	Description
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 7.2.2.3 *Buffer attributes*.
Notification support	No	• This field specifies if the partition supports receipt of notifications as described in Chapter 10 *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 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 4.10 *Run-time states*. – *Preemptible*. SP execution can be preempted. An execution context of this SP can transition between all states described in 4.10 *Run-time states*. This is the default run-time model for a S-EL0 SP if this field is not specified in the partition manifest. • This field is deprecated in v1.1 of the Framework. Please see 9.4 *Support for legacy run-time models* for more details.
Action in response to Non-secure interrupts	Yes	• This field specifies the action that the SPMC must take in response to a Non-secure physical interrupt as described in 9.3.1 *Actions for a Non-secure interrupt*. • This field supersedes the *Managed exit supported* field in the FF-A v1.0 specification.

Information fields	Mandatory	Description
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 4.2.3 *Other DMA isolation models*). – *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.
VM availability messages	No	• This field specifies the VM availability messages the SP is interested in receiving. See 18.3 *VM availability signaling*.
Power management messages	No	• This field specifies the power management messages the SP is interested in receiving. See 18.2.4 *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 18.2 *Power Management* and 18.2.2 *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. • The specified register must be distinct from the register used to carry the address of the boot information blob specified in Table 5.10. The partition is not initialized if there is a clash.

Table 5.2: Memory regions

Information fields	Mandatory	Description
Base address or Load address relative offset	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), IPA (for S-EL1 and EL1 partitions) or a positive offset (for S-EL0 partitions) relative to the *Load address* of the partition image. 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 an offset is specified, this must be indicated in the manifest through an IMPLEMENTATION DEFINED mechanism. • If present, the address or offset must be aligned to the Translation granule size.

Information fields	Mandatory	Description
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.
Stream & SMMU IDs	No	• Identity of the SMMU and stream IDs of a device upstream of the SMMU that can access this memory region with the access permissions specified in the stream ID access permissions field.

Stream ID access permissions	No	• Device access permissions for each Stream ID if the Stream and SMMU IDs field is present.
		  – Instruction access permission.
		  – Data access permission.

Table 5.3: 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.

Information fields	Mandatory	Description
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. ∗ This is a virtual priority value for a S-EL1 SP that runs under the S-EL2 SPMC. – 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.
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.

5.2.2 SPMC manifest

The following aspects of the SPMC 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 5.4: SPMC properties

Information fields	Mandatory	Description
FF-A version	Yes	• Version of Firmware Framework implemented by the SPMC component. See 13.2 *FFA_VERSION* for more information about the usage of this field.

Information fields	Mandatory	Description
SPMC ID	No	• Pre-allocated ID for the SPMC.
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 SPMC binary image.
FF-A boot protocol usage	No	• See Table 5.10.
Cold boot reason register	No	• Presence of this field indicates that the SPMC expects that the entry point offset field must be reused for a secondary cold boot (see 18.2 *Power Management* and 18.2.2 *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 SPMC manifest.

5.2.3 Independent peripheral device manifest

This manifest must be used by *independent peripheral devices* to describe their properties to a partition manager. See 4.2.3 *Other DMA isolation models* for more details.

Table 5.5: 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 5.6: 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 [1]).
		• 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 5.7: 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.
Stream IDs	Yes	• List of Stream IDs assigned to this device.
Name	No	• Name of the device region for example, for debugging purposes.

5.3 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 5.10.

5.4 Boot information protocol

An SP or SPMC could rely on boot information for their initialization e.g. a flattened device tree with nodes to describe the devices and memory regions assigned to the SP or SPMC. The Framework specifies a protocol that can be used by a *producer* to pass boot information to a *consumer* at a Secure FF-A instance. The Framework assumes that the boot information protocol is used by a producer and consumer pair that reside at adjacent exception levels as listed below.

1. SPMD (producer) and an SPMC (consumer) in either S-EL1 or S-EL2.
2. An SPMC (producer) and SP (consumer) pair listed below.
   1. EL3 SPMC and a Logical S-EL1 SP.
   2. S-EL2 SPMC and Physical S-EL1 SP.
   3. EL3 SPMC and a S-EL0 SP.
   4. S-EL2 SPMC and a S-EL0 SP.
   5. S-EL1 SPMC and a S-EL0 SP.

The boot information protocol used by a producer and consumer pair that reside at the same exception level is IMPLEMENTATION DEFINED.

The Framework also makes the following assumptions about the usage of the boot information protocol between a producer and consumer pair.

1. Boot information is passed only to the consumer execution context that is initialized on the primary PE by the producer.
2. Boot information is passed when the consumer execution context on the primary PE is first entered through an exception return from the producer.
3. Boot information is encoded in a format chosen by the consumer and the producer through an IMPLEMENTATION DEFINED mechanism e.g. a flattened device tree, a handover block list (HOB list) etc.
4. One or more distinct instances of boot information could be passed from the producer to the consumer.

A producer maintains backwards compatibility while using the boot information protocol described in this or an earlier version of the Framework. A consumer requests usage of the boot information protocol by specifying the corresponding field in their manifest (see Table 5.10). A consumer also specifies the version of the Framework it implements in its manifest. A producer parses the manifest of the consumer for this information and ensures it uses the boot information protocol specified in the version of the Framework specified in the manifest.

5.4.1 Boot information descriptor

The Framework defines a descriptor (see Table 5.8) to describe an distinct instance of boot information.

Table 5.8: Boot information descriptor

Field	Byte length	Byte offset	Description
Name	16	0	• Name of boot information passed to the consumer.

Field	Byte length	Byte offset	Description
Type	1	16	• Type of boot information passed to the consumer. – Bit[7]: Boot information type. ∗ b'0: Standard boot information. ∗ b'1: IMPLEMENTATION DEFINED boot information. – Bit[6:0]: Boot information identifier. ∗ Standard boot information (bit[7] = b'0). · 0: Flattened device tree (FDT). · 1: Hand-Off Block (HOB) List. · All other identifiers are reserved. ∗ IMPLEMENTATION DEFINED identifiers (bit[7] = b' 1). · Identifier is defined by the implementation.
Reserved	1	17	• Reserved (MBZ).
Flags	2	18	• Flags to describe properties of boot information associated with this descriptor. – Bits[15:4]: Reserved (MBZ). – Bits[3:2]: Format of *Contents* field. ∗ b'0: Address of boot information identified by the *Name* and *Type* fields. ∗ b'1: Value of boot information identified by the *Name* and *Type* fields. ∗ All other bit encodings are reserved for future use. – Bits[1:0]: Format of *Name* field. ∗ b'0: Null terminated string. ∗ b'1: UUID encoded in little-endian byte order. ∗ All other bit encodings are reserved for future use.
Size	4	20	• Size (in bytes) of boot information identified by the *Name* and *Type* fields.
Contents	8	24	• Value or address (see *Flags* field) of boot information identified by the *Name* and *Type* fields. – If in the *Flags* field, bit[3:2] = b'0, ∗ The address has the same attributes as the boot information blob address described in 5.4.3 *Boot information address*. ∗ *Size* field contains the length (in bytes) of boot information at the specified address. – If in the *Flags* field, bit[3:2] = b'1, ∗ *Size* field contains the exact size of the value specified in this field. ∗ Size is >= 1 bytes and <= 8 bytes.

The fields of the descriptor are described below.

1. The *Name* and *Type* fields uniquely identify the boot information. The *Type* field is the primary identification mechanism. The *Name* field can be used as a secondary identification mechanism. This is described in the following usage models.

1. The *Type* field identifies the format in which boot information is encoded and the *Name* field identifies the contents of the boot information. For example,
   1. An SP could consume two distinct FDTs during initialization. The *Type* field would be used to identify that boot information is encoded in the device tree format. The *Name* field could be a NULL terminated 15-byte string or a 16-byte UUID to identify what information is specified in an FDT.
2. The *Type* field identifies both the format and the contents of the boot information. For example, an SP could consume a HOB list during initialization.

   The *Type* field uniquely identifies the format and contents of the boot information in both examples. The *Name* field could be unused or a NULL terminated string for debugging purposes.

The *Flags* field is used to specify whether the *Name* field encodes a NULL terminated string or a UUID.

2. The *Contents* and *Size* fields allow boot information to be,
   1. Either referenced from the descriptor by populating the address of the boot information in the *Contents* field.
   2. Or encoded in the descriptor in the *Contents* field. This is subject to the width of the *Contents* field.

The *Flags* field is used to specify whether the *Contents* field encodes the boot information or a reference to it.

5.4.2 Boot information header

The producer passes one or more instances of boot information to a consumer as an array of boot information descriptors (see Table 5.8). The array is preceded by a boot information header defined in Table 5.9.

The combination of boot information referenced from the boot information descriptor array, the array itself and the boot information header is called the *Boot information blob*.

Table 5.9: Boot information header

Field	Byte length	Byte offset	Description
Signature	4	0	• Hexadecimal value *0x0FFA* to identify the header.
Version	4	4	• Version of the boot information blob encoded as per the *Input version number* field in Table 13.4.
Size of boot information blob	4	8	• Size of boot information blob spanning contiguous memory.
Boot information descriptor size	4	12	• Size of each boot information descriptor in the array (see Table 5.8).
Boot information descriptor count	4	16	• Count of boot information descriptors in the array.
Boot information descriptor array offset	4	20	• Offset to array of boot information descriptors.
Reserved	8	24	• Reserved (MBZ).
Optional padding	–	32	• This field is not a part of the boot information header. It has been included only for informational purposes.
Boot information descriptor array	–	–	• Array of boot information descriptors.

The fields of the boot information header are described below.

1. The *Signature* field is a magic number to let a consumer ensure that an FF-A boot information blob has been passed by the partition manager.
2. The *Version* field identifies the version of the boot information blob. This includes the boot information header and descriptor. This version is equal to the version of the Framework (see 13.2.2 *Usage*).

   The producer determines the version of the Framework implemented by a consumer through its manifest (see Table 5.1) or an IMPLEMENTATION DEFINED mechanism.

   The producer ensures that the version of the boot information blob passed to a consumer is the same as the version of the Framework implemented by the consumer.
3. The *Size of boot information blob* field specifies the size of the blob that spans one or more contiguous 4K pages used by the producer to populate it. It is calculated by adding the following values.
   1. *Boot information descriptor array offset*.
   2. Product of *Boot information descriptor count* and *Boot information descriptor size*.
   3. Total size of all boot information referenced by boot information descriptors.

      This is determined by adding the values in the *Size* field of each boot information descriptor whose *Contents* field contains an address.
   4. Any padding between,
      1. The boot information descriptor array and the boot information referenced from it.
      2. Distinct instances of boot information referenced from the boot information descriptor array.

   This field enables a consumer to map all of the boot information blob in its translation regime (not managed by the producer) or copy it to another memory location without parsing each element in the boot information descriptor array.
4. The *Boot information descriptor size* field contains the size of the descriptor. This enables a consumer to parse the array without relying on a static association between the Framework version it implements and the size of the boot information descriptor in that version of the Framework.
5. The *Boot information descriptor count* field contains the number of descriptors in the boot information descriptor array.
6. The *Boot information descriptor array offset* field is the offset from the base address of the *Boot information header* to the first element in the *Boot information descriptor array*. The offset must be aligned to the 8-byte boundary.

Figure 5.1 illustrates an example boot information array that includes,

1. A reference to a FDT whose name is identified by a UUID. The FDT blob is populated at the next available address after the boot information array.
2. IMPLEMENTATION DEFINED boot information encoded as a value in the boot information descriptor.

The boot information array and FDT blob fit in a single 4K page.

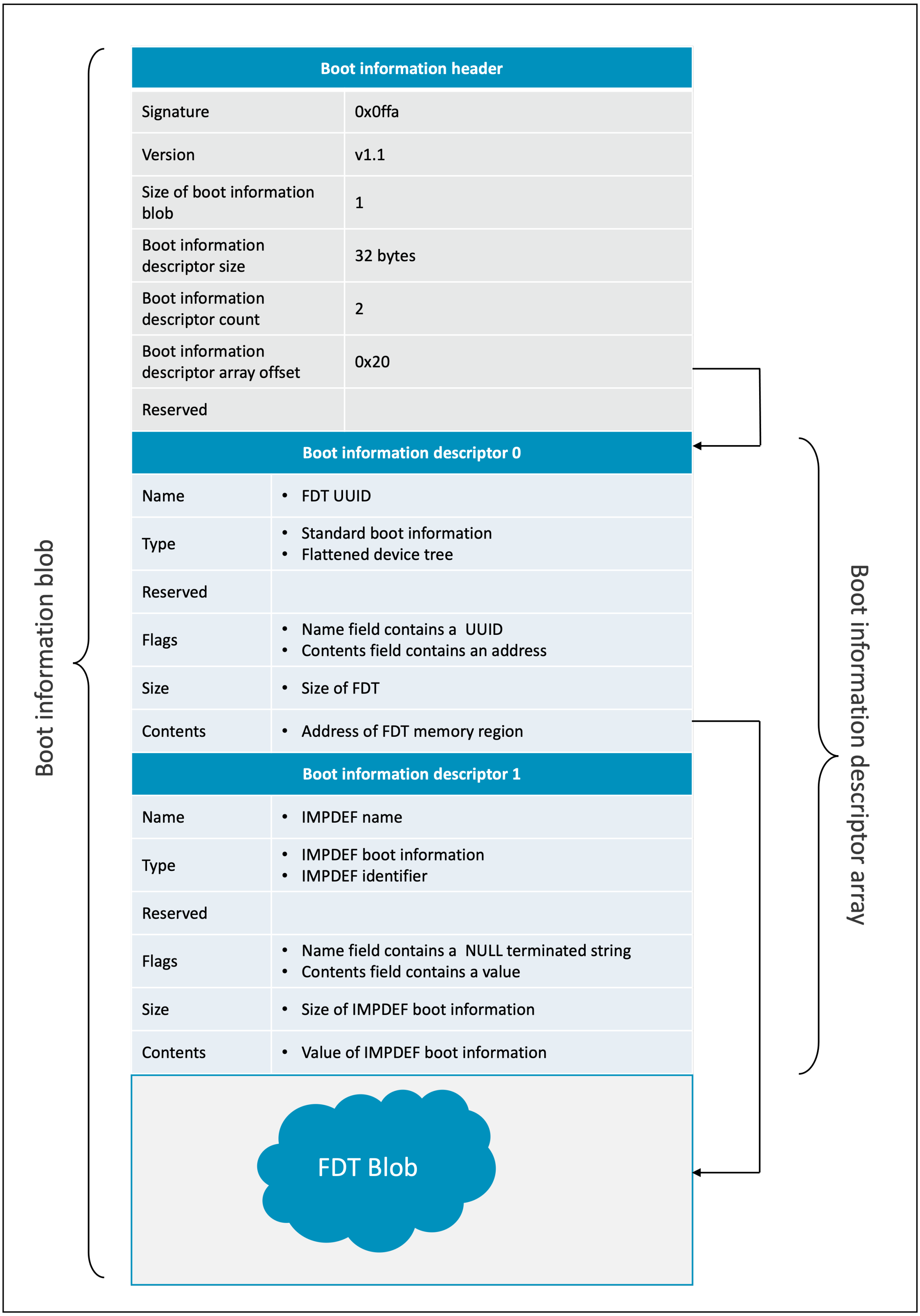

Figure 5.1: Example boot information array

5.4.3 Boot information address

The producer passes the address of the boot information blob to a consumer in a general purpose register specified in the consumer manifest (see Table 5.10). The presence of this field in the consumer manifest enables the producer to discover that the consumer expects the boot information blob to be passed through the FF-A boot information protocol. The boot information blob address is,

1. A VA for a S-EL0 SP at the Secure virtual FF-A instance.
2. An IPA for a Physical S-EL1 SP at the Secure virtual FF-A instance.
3. A PA for a Logical S-EL1 SP, S-EL2 SPMC and S-EL1 SPMC at the Secure physical FF-A instance.

Table 5.10: Boot protocol information

Information fields	Mandatory	Description
FF-A boot protocol usage	No	• The register in which the address of the boot information blob must be passed by the producer. Register must be between *w0/x0-w3/x3*. The width of the register is derived from its Execution state specified in the partition manifest.

5.4.4 Boot information memory requirements

The producer populates the boot information blob in a memory region that fulfill the following requirements.

1. Size of memory region is a multiple of the translation granule size used by the consumer.
2. Address of memory region is aligned to the translation granule size used by the consumer.
3. The memory region is mapped in the translation regime of the consumer that is managed by the producer (see [1]). The producer does not map the memory region in the translation regime of a consumer at the Secure physical FF-A instance.
4. The memory region is mapped in the producer and consumer's translation regimes with the same memory attributes as the RX/TX buffers as described in 7.2.2.3 *Buffer attributes*.
5. The memory region comprises of translation granule sized contiguous pages.
   1. The pages are physically contiguous at the Secure physical FF-A instance.
   2. The pages are virtually contiguous at the Secure virtual FF-A instance.

The boot information blob is populated at offset 0 in the memory region. The Framework uses the little-endian byte order to encode the boot information blob.

The memory region used to populate the boot information blob could be owned by the producer or consumer during the latter's initialization.

- In the latter case, the consumer specifies a memory region in its manifest or through an IMPLEMENTATION DEFINED mechanism. The producer uses this memory region for populating the boot information blob. The producer maps the memory region in its translation regime to access it. It unmaps the memory region from its translation regime and maps it in the translation regime of the consumer (if applicable) prior to handing control to the consumer for initialization. The consumer is the owner of the memory region from this stage onwards.
- In the former case, the memory region is owned by the producer and shared with a consumer for the duration of its initialization. The consumer should not assume access to the memory region post-initialization (see 5.5 *Protocol for completing execution context initialization*).
  - At the Secure virtual FF-A instance, the producer unmaps the memory region from the translation regime of the consumer that it manages ([1]).

- The Framework strongly recommends that the consumer at the Secure physical FF-A instance does not access the memory region post-initialization.

The producer could reuse the same memory region to pass boot information to multiple consumers. In this case, it ensures that boot information passed to one consumer is cleared in the memory region before boot information for another consumer is populated in the same memory region. The producer performs cache maintenance such that the memory region contents after clearing are coherent between any PE caches, system caches and system memory.

5.5 Protocol for completing execution context initialization

A partition must use the *FFA_MSG_WAIT* (also see 14.1 *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 12.2 *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 8.5 *Runtime model for SP initialization*.

Chapter 6

Identification and Discovery

6.1 Partition identification

Partitions are identified in the Framework by a globally unique 16-bit *ID*. This implies that no two partitions in the Framework can be assigned the same ID. This ID is used in FF-A ABIs to identify the partition e.g. the sender or receiver of a message, lender or borrower of shared memory.

An ID is assigned to the partition by its partition manager before the partition is initialized. A partition uses the *FFA_ID_GET* interface (also see 13.10 *FFA_ID_GET*) to discover its ID. The ID can be,

1. Specified in the manifest of the partition, validated and assigned by the partition manager. The partition manager does not boot a partition if the ID specified in the manifest cannot be assigned to the partition.
2. Allocated and assigned by the partition manager through an IMPLEMENTATION DEFINED mechanism.

The Hypervisor and SPM are collectively responsible for ensuring that an ID allocated to a partition is globally unique. This is done as described below.

1. Each partition manager ensures that it allocates unique IDs to the partitions it manages i.e. the Hypervisor allocates unique IDs to VMs and SPM does the same for SPs.
2. Both partition managers ensure that IDs are allocated without the risk of the same ID being allocated by both. This is done either through an IMPLEMENTATION DEFINED mechanism or one of the following mechanisms specified by the Framework.
    1. The 16-bit partition ID namespace is split into two parts for use by the Hypervisor and SPM as described below.
        - Bit[15]: Partition type identifier.
            - b’0: Bits[14:0] are reserved for use by the Hypervisor to identify a VM.

    - b'1: Bits[14:0] are reserved for use by the SPM to identify an SP.
  - Bit[14:0]: IMPLEMENTATION DEFINED value chosen by,
    - The Hypervisor if Bit[15] = b'0.
    - The SPM if Bit[15] = b'1.

2. The Framework assumes that the SPM always initializes SPs before the Hypervisor initializes VMs. Hence, the SPM allocates IDs for SPs before the Hypervisor allocates IDs for VMs.

   In this mechanism, the Hypervisor discovers the IDs assigned to SPs by invoking the *FFA_PARTITION_INFO_GET* or *FFA_PARTITION_INFO_GET_REGS* ABI at the Non-secure physical FF-A instance with the Nil UUID as the input. It assigns IDs to VMs that are not already used by the SPM for SPs.

The Framework assumes that the system integrator ensures that both the Hypervisor and SPM use the same mechanism.

In a configuration with the S-EL2 or S-EL1 SPMC, the Framework assumes that any EL3 LSPs have been initialized before the SPMC initializes its SPs. The SPMC must use one of the following mechanisms to ensure that IDs it allocates are unique.

- An IMPLEMENTATION DEFINED mechanism.
- The *FFA_PARTITION_INFO_REGS* interface is used with the Nil UUID to discover the IDs of EL3 LSPs.

6.2 Partition discovery

The identity and other properties of a partition are discovered by an FF-A component as described below.

1. Each partition is associated with one or more UUIDs (Unique Universal Identifier) (see [7]) that are specified in its manifest. Also see, Table 5.1 in 5.2.1 *Partition manifest*.
2. An FF-A component discovers the identity and properties of a partition by specifying a UUID exported by the partition as an input to the *FFA_PARTITION_INFO_GET* or *FFA_PARTITION_INFO_GET_REGS* interfaces (see 13.8 *FFA_PARTITION_INFO_GET* and 13.9 *FFA_PARTITION_INFO_GET_REGS*). The Nil UUID is used to discover all available partitions and their UUIDs at the FF-A instance where the ABI is invoked.

6.2.1 Partition information descriptor

The format of the partition information descriptor changed between Framework versions 1.0 and 1.1. The changes to the FF-A v1.0 descriptor (see Table 18.22) and implementation responsibilities to maintain backward compatibility are specified in 18.5 *Changes to FF-A v1.0 data structures for forward compatibility*.

Table 6.1: 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 4.7 *Execution context*) if *Bit[5:4] = b'00* in the *Partition properties* field. • ID of the proxy endpoint for a dependent peripheral device (see 4.2.3 *Other DMA isolation models* if *Bit[5:4] = b'10* in the *Partition properties* field. • Reserved (MBZ) for all other encodings of the *Partition properties* field.
Partition properties	4	4	• Flags to specify partition properties for the UUID described by this descriptor (see also Table 6.2).
Partition UUID	16	8	• A UUID associated with the partition, stream or auxiliary endpoint (see 6.2.3 *Partition UUID usage*) if the Nil UUID was specified as an input parameter. • This field is Reserved (MBZ) if a non-Nil UUID was. specified as an input parameter.

The properties of a partition returned via the discovery ABIs are encoded as shown in Table 6.2.

Table 6.2: Partition properties descriptor

Field	Description
Bits[3:0]	• Has the following encoding if Bits[5:4] = b'00. It is Reserved (MBZ) otherwise. – Bit[0] has the following encoding: ∗ b'0: Cannot receive Direct requests via the FFA_MSG_SEND_DIRECT_REQ ABI. ∗ b'1: Can receive Direct requests via the FFA_MSG_SEND_DIRECT_REQ ABI. – Bit[1] has the following encoding: ∗ b'0: Cannot send Direct requests via the FFA_MSG_SEND_DIRECT_REQ ABI. ∗ b'1: Can send Direct requests via the FFA_MSG_SEND_DIRECT_REQ ABI. – Bit[2] has the following encoding: ∗ b'0: Cannot send and receive Indirect messages. ∗ 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.
Bits[5:4]	• 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 5.2.1 *Partition manifest*.
Bit[6]	• b'0: Partition must not be informed about each VM that is created by the Hypervisor. • b'1: Partition must be informed about each VM that is created by the Hypervisor. • bit[6] is used only if the following conditions are true. It is Reserved (MBZ) in all other scenarios. – This ABI is invoked at the Non-secure physical FF-A instance. – The partition is an SP that supports receipt of Direct requests i.e. Bit[0] = b'1. • Also see 18.3 *VM availability signaling*.
Bit[7]	• b'0: Partition must not be informed about each VM that is destroyed by the Hypervisor. • b'1: Partition must be informed about each VM that is destroyed by the Hypervisor. • bit[7] is used only if the following conditions are true. It is Reserved (MBZ) in all other scenarios. – This ABI is invoked at the Non-secure physical FF-A instance. – The partition is an SP that supports receipt of Direct requests i.e. Bit[0] = b'1. • Also see 18.3 *VM availability signaling*.
Bit[8]	• b'0: Partition runs in the AArch32 execution state. • b'1: Partition runs in the AArch64 execution state.
Bits[10:9]	• Has the following encoding if *Bits[5:4] = b'00*. It is Reserved (MBZ) otherwise. – Bit[9] has the following encoding: ∗ b'0: Cannot receive Direct requests via the FFA_MSG_SEND_DIRECT_REQ2 ABI. ∗ b'1: Can receive Direct requests via the FFA_MSG_SEND_DIRECT_REQ2 ABI. – Bit[10] has the following encoding: ∗ b'0: Cannot send Direct requests via the FFA_MSG_SEND_DIRECT_REQ2 ABI. ∗ b'1: Can send Direct requests via the FFA_MSG_SEND_DIRECT_REQ2 ABI.
Bit[31:11]	Reserved (MBZ).

6.2.2 Partition discovery ABI usage

The type of partitions (see 3.2 *Partitions*) that can be discovered via the discovery ABIs depend upon the caller and callee in an ABI invocation. This is listed in Table 6.3.

The term *SPM LSP* is used when it is not necessary to distinguish between the SPM component the LSP is co-resident with.

Table 6.3: Table of discoverable partitions between two FF-A components

Caller	Callee	Available Partitions for Discovery
VM	Hypervisor	• VMs • SPs • SPM LSPs
OS Kernel / Hypervisor	SPM	• SPs • SPM LSPs
SP	SPMC	• SPs • SPM LSPs
SPMC	SPMD	• EL3 LSPs
SPMD	SPMC	• SPs • SPMC LSPs
SPM	Hypervisor/ OS Kernel	• N/A

The discovery ABIs can return the properties or count of discoverable partitions. This is governed by the input parameters specified by the caller (e.g. flags and type of UUID (see 6.2.3 *Partition UUID usage*)) in an ABI invocation and described below.

- If the *FFA_PARTITION_INFO_GET* ABI is invoked with the *Return information type flag* in *Flags* input parameter = *b'0* or the *FFA_PARTITION_INFO_GET_REGS* ABI is invoked and,
  - If the Nil UUID is specified, information for partitions (including the caller) as detailed in Table 6.3 is returned.
  - If a non-Nil UUID is specified, information for all partitions as detailed in Table 6.3 corresponding to the UUID, is returned.

  If the Nil UUID is specified at a valid FF-A instance and a partition exports multiple UUIDs in its manifest, then properties corresponding to each UUID are returned in a distinct partition information descriptor. The descriptors corresponding to this partition have the same value in their *Partition ID* field.
- If the *FFA_PARTITION_INFO_GET* ABI is invoked with the *Return information type flag* in *Flags* input parameter = *b'1* and,
  - If the Nil UUID is specified, count for partitions (including the caller) as detailed in Table 6.3 is returned.
  - If a non-Nil UUID is specified, count for all partitions as detailed in Table 6.3 corresponding to the UUID, is returned.

  If the Nil UUID is specified at a valid FF-A instance and one or more partitions export multiple UUIDs in their manifests, then the returned count corresponds to the sum of the products of the number of partitions and the number of UUIDs exported by each partition.

A callee is allowed to provide a subset of partition information in an invocation of the discovery ABIs.

6.2.3 Partition UUID usage

A partition implements one or more services. Each service is accessed by a communication protocol built on top of FF-A messaging mechanisms. A UUID exported by the partition identifies an implemented communication protocol. There is a *one-to-many* relationship between a communication protocol and UUIDs. Some example usage models are described below:

1. A partition implements *N* services. Each service is accessed via a unique communication protocol. It associates a UUID with each service and exports *N* UUIDs in its manifest. The communication protocol used to access each service is implicitly identified by its UUID. There is a *1:1* mapping between communication protocols and the UUIDs.

2. A partition implements *N* services. Each service is accessed via a common communication protocol. It associates a UUID with each service and exports *N* UUIDs in its manifest. The communication protocol used to access each service is implicitly identified by its UUID. There is a *1:N* mapping between the common communication protocol and the UUIDs.
3. A partition implements *N* services. Each service is accessed via a common communication protocol. It associates a UUID with the communication protocol and exports a single UUID in its manifest. The communication protocol provides an IMPLEMENTATION DEFINED mechanism to discover and access the *N* services. There is a *1:1* mapping between the common communication protocol and the UUID.

To support the scenarios where a partition exports multiple UUIDs, an entry corresponding to each UUID is encoded in the returned partition information. E.g. SP0 exports UUID_0 and UUID_1. The returned partition information for SP0 has two entries as follows,

- SP0 ID, properties, UUID_0.
- SP0 ID, properties, UUID_1.

If the Nil UUID is specified in an invocation of an FF-A discovery ABI and one or more partitions export multiple UUIDs in their manifests, then the returned count corresponds to the sum of the products of the number of partitions and the number of UUIDs exported by each partition.

If the Nil UUID is specified in an invocation of the *FFA_MSG_SEND_DIRECT_REQ2* or in a partition message header (see Table 7.2) for indirect messages using the *FFA_MSG_SEND2* ABI then the communication protocol or service is determined by the callee through an IMPLEMENTATION DEFINED mechanism.

6.2.3.1 UUIDs and messaging methods

A partition could be associated with multiple UUIDs. It could implement messaging methods that take a UUID as an input parameter as well as those that do not. In such a partition, communication protocols corresponding to a subset of UUIDs are accessible via one or more of the following:

- Only via messaging methods that take a UUID as an input parameter.
- Only via messaging methods that do not take a UUID as an input parameter.
- Via any messaging method.

When a communication protocol corresponding to a UUID is accessible only via a specific messaging method, the Framework enables discovery of this property of the partition, by other partitions as follows:

- The partition specifies the mapping between the UUID and the messaging method in its partition manifest. See also 5.2.1 *Partition manifest*.
- The partition manager specifies only the messaging methods that the UUID is mapped to in the partition properties field of the partition information descriptor for that UUID.

Otherwise, in the partition properties in a partition information descriptor for that UUID, the partition manager specifies all the messaging methods implemented by the partition.

For example, SP0 has the following properties:

- It implements two communication protocols with UUID_0 and UUID_1.
- It implements the FFA_MSG_SEND_DIRECT_REQ & FFA_MSG_SEND_DIRECT_REQ2 ABIs.
- The communication protocol corresponding to UUID_0 is accessible only via FFA_MSG_SEND_DIRECT_REQ.
- The communication protocol corresponding to UUID_1 is accessible only via FFA_MSG_SEND_DIRECT_REQ2.

If the SP specifies the mapping between the UUIDs and the ABIs it implements, all of the following are true:

- An invocation of an FF-A discovery ABI with UUID_0 returns a partition information descriptor where Bits[0] == b’1 and Bits[9] == b’0 in the partition properties descriptor.
- An invocation of an FF-A discovery ABI with UUID_1 returns a partition information descriptor where Bits[0] == b’0 and Bits[9] == b’1 in the partition properties descriptor.

- An invocation of an FF-A discovery ABI with the Nil UUID returns a partition information descriptor for UUID_0 where Bits[0] == b'1 and Bits[9] == b'0 in the partition properties descriptor.
- An invocation of an FF-A discovery ABI with the Nil UUID returns a partition information descriptor for UUID_1 where Bits[0] == b'0 and Bits[9] == b'1 in the partition properties descriptor.

If the SP does not specify the mapping between the UUIDs and the ABIs it implements, all of the following are true:

- An invocation of an FF-A discovery ABI with UUID_0 returns a partition information descriptor where Bits[0] == b'1 and Bits[9] == b'1 in the partition properties descriptor.
- An invocation of an FF-A discovery ABI with UUID_1 returns a partition information descriptor where Bits[0] == b'1 and Bits[9] == b'1 in the partition properties descriptor.
- An invocation of an FF-A discovery ABI with the Nil UUID returns a partition information descriptor for UUID_0 where Bits[0] == b'1 and Bits[9] == b'1 in the partition properties descriptor.
- An invocation of an FF-A discovery ABI with the Nil UUID returns a partition information descriptor for UUID_1 where Bits[0] == b'1 and Bits[9] == b'1 in the partition properties descriptor.
- The SP uses an IMPLEMENTATION DEFINED mechanism to map the communication protocol associated with a UUID to a messaging method.

6.3 Partition manager identification

Partition managers are identified in the Framework as described below.

1. A partition manager is identified by a globally unique 16-bit *ID*. This ID is not assigned to any another FF-A component in the system.
2. The ID value *0* is reserved for the Hypervisor as described in [5].
3. The ID values assigned to the SPMC and SPMD components are IMPLEMENTATION DEFINED. From v1.1 of the Framework, the IDs assigned to the SPMC and SPMD can be discovered through the FFA_SPM_ID_GET interface (see 13.11 *FFA_SPM_ID_GET*).

The Framework can be deployed on an Arm A-profile system that does not implement EL2. The ID value *0* is reserved for the OS kernel in this configuration.

Chapter 7
Message passing

7.1 Overview

The Firmware Framework defines a set of ABIs that enable synchronous and asynchronous message passing between FF-A components. A message exchange comprises the following phases.

1. Transmission of the message payload from the Sender to the Receiver. The mechanisms used by the Framework to transmit messages are described in 7.2 *Message transmission*.
2. Allocation of CPU cycles to the Receiver to process the message on a PE in the system. The type of message passing (synchronous or asynchronous) is governed by when the Receiver is allocated CPU cycles.
    1. Synchronous message passing is described in 7.1.2 *Direct messaging*.
    2. Asynchronous message passing is described in 7.1.1 *Indirect messaging*.
3. Message processing by the Receiver using the allocated cycles. The role of the Framework during message processing is described in the following sections.
    1. Chapter 8 *Partition runtime models*.
    2. Chapter 9 *Interrupt management*.

In any message exchange between two FF-A components, 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 the absence of a Hypervisor, the SPMC and SPMD are the only Relayers in the Framework. FF-A components in the Secure world participate in message passing as described below.

1. The SPMD forwards a message from an NS-Endpoint to the SPMC and vice-versa.
2. The SPMC forwards a message from the SPMD to its target S-Endpoint and vice-versa.
3. The SPMC forwards a message between any two S-Endpoints.
4. FF-A ABIs are used between the SPMC and a SP for sending and receiving messages when the SP resides in a separate exception level from the SPMC.
5. An IMPLEMENTATION DEFINED mechanism is used between the SPMC and a SP for sending and receiving messages when the SP resides in the same exception level as the SPMC.

7.1.1 Indirect messaging

The asynchronous message passing method specified by the Framework is called Indirect messaging. In this method, the Sender does not relinquish control to the Receiver at the time of message transmission. Instead, the Relayer ensures that CPU cycles are subsequently allocated to the Receiver for message processing. Effectively, the Sender *Indirectly schedules* the Receiver on the same or a different PE via the Relayer. CPU cycle allocation is decoupled from message transmission. The Relayer requests the primary or secondary scheduler in a VM or Hypervisor to allocate CPU cycles to the Receiver for message processing on behalf of the Sender. The Sender could make progress concurrently with the Receiver if the latter is scheduled on a different PE. The Sender either polls for a response or is notified when a response from the Receiver is available.

In this version of the Framework, Indirect messaging is used only for message exchanges between endpoints. 7.3 *Indirect messaging usage* describes this method in detail. Figure 7.1 illustrates an example Indirect message exchange.

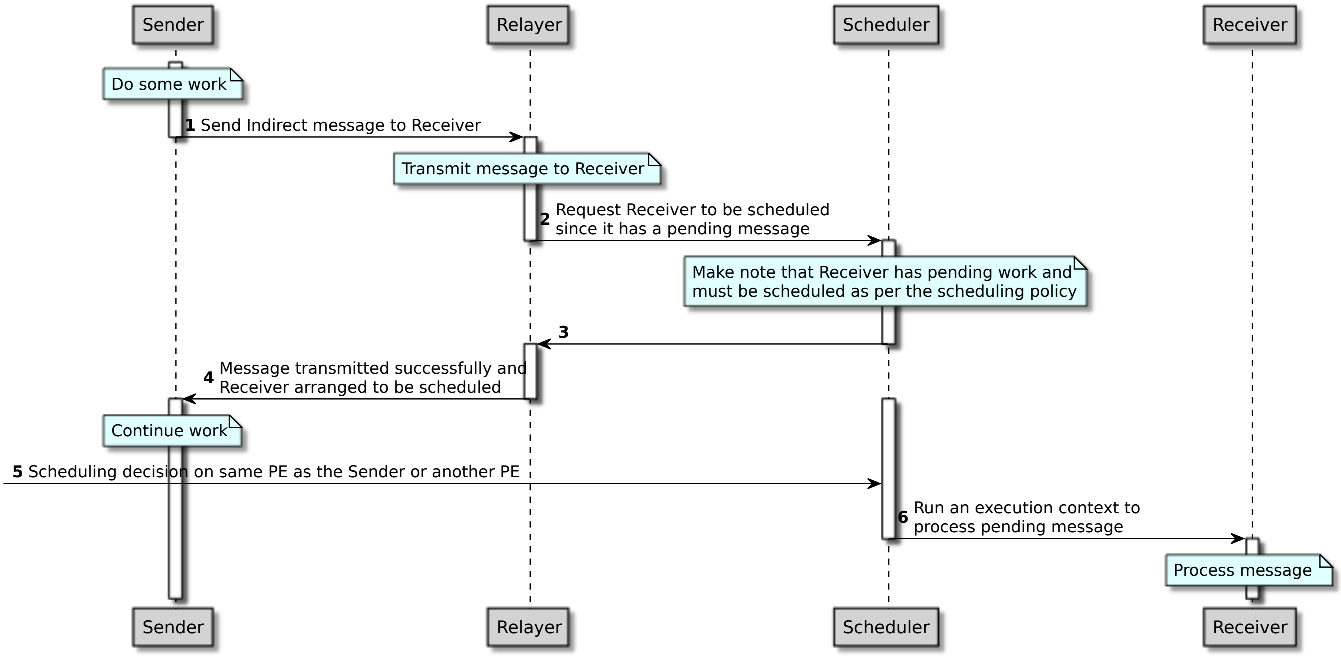

Figure 7.1: Example Indirect messaging flow

7.1.2 Direct messaging

The synchronous message passing method specified by the Framework is called Direct messaging. In this method, the Sender relinquishes control to the Receiver at the time of message transmission and blocks until its receives a response from the Receiver. Effectively, the Sender *directly schedules* the Receiver on the same PE where the message is transmitted by the Relayer. CPU cycle allocation is tightly coupled with message transmission.

This method is used for message exchanges between the following FF-A components.

1. An endpoint and a partition manager. These message exchanges are called *hypcalls* (also see 4.11 *Run-time state transitions*).
2. Between endpoints in the following non-exhaustive list of scenarios.
   1. The scheduler is not available. For example,
      - The system is booting and the primary scheduler has not been initialized yet.
      - It is not possible to communicate with the scheduler. For example, from EFI runtime services that run as a peer of the OS scheduler.
   2. The Receiver must be run on the same PE as the Sender.

7.4 *Direct messaging usage* describes this method in detail. Figure 7.2 illustrates this method.

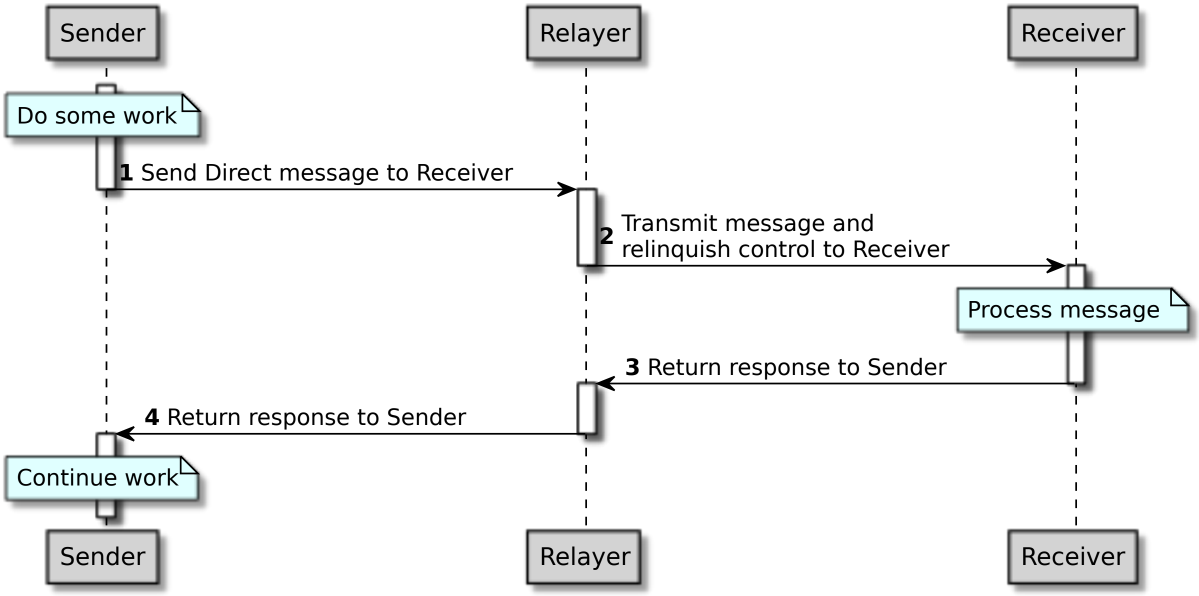

Figure 7.2: Example Direct messaging flow

7.2 Message transmission

7.2.1 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 7.2.2 *RX/TX buffers*).

- Direct messaging can use both these mechanisms along with the ABIs described in 7.1.2 *Direct messaging*.
- Indirect messaging must use only the RX/TX message buffers along with the ABIs described in 7.1.1 *Indirect messaging*.

The Framework defines a message as all information encoded in,

1. The input parameter registers (*w0-w7* or *x0-x17*) as per the applicable SMC calling convention 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 13.2 *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 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 validates the header information and uses it to route them correctly. Hence, by definition these messages are only exchanged between endpoints.

The relationship between *message types* (Framework and Partition), *messaging method* (Direct and Indirect) and *message payload location* (registers and RX/TX buffers) is summarized below and described in Table 7.1.

- Direct messaging is 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 is used to only transmit Partition messages in the RX/TX buffers.

Table 7.1: Combinations of messaging and message transmission mechanisms

Messaging method	Message type	Message payload location	FF-A ABI
Direct	Partition	Register	• FFA_MSG_SEND_DIRECT_REQ. • FFA_MSG_SEND_DIRECT_REQ2. • FFA_MSG_SEND_DIRECT_RESP. • FFA_MSG_SEND_DIRECT_RESP2.
Direct	Partition	RX/TX	• Invalid usage.
Direct	Framework	Register	• Hypcalls.
Direct	Framework	RX/TX	• Hypcalls.
Indirect	Partition	Register	• Invalid usage.
Indirect	Partition	RX/TX	• FFA_MSG_SEND2.
Indirect	Framework	Register	• Invalid usage.
Indirect	Framework	RX/TX	• Invalid usage.

7.2.2 RX/TX buffers

The guidance on this topic applies to physical partitions and logical partitions that reside in a different exception level from their partition manager. The use of RX/TX buffers between a logical partition that co-resides with its partition manager is IMPLEMENTATION DEFINED.

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 4.1 *SPM architecture*. In configurations where the SPMC resides in a separate Exception level from the SPMD (see Table 4.1), it is IMPLEMENTATION DEFINED whether the two SPM components share an RX/TX buffer pair.

These message buffer configurations are illustrated in Figure 7.3.

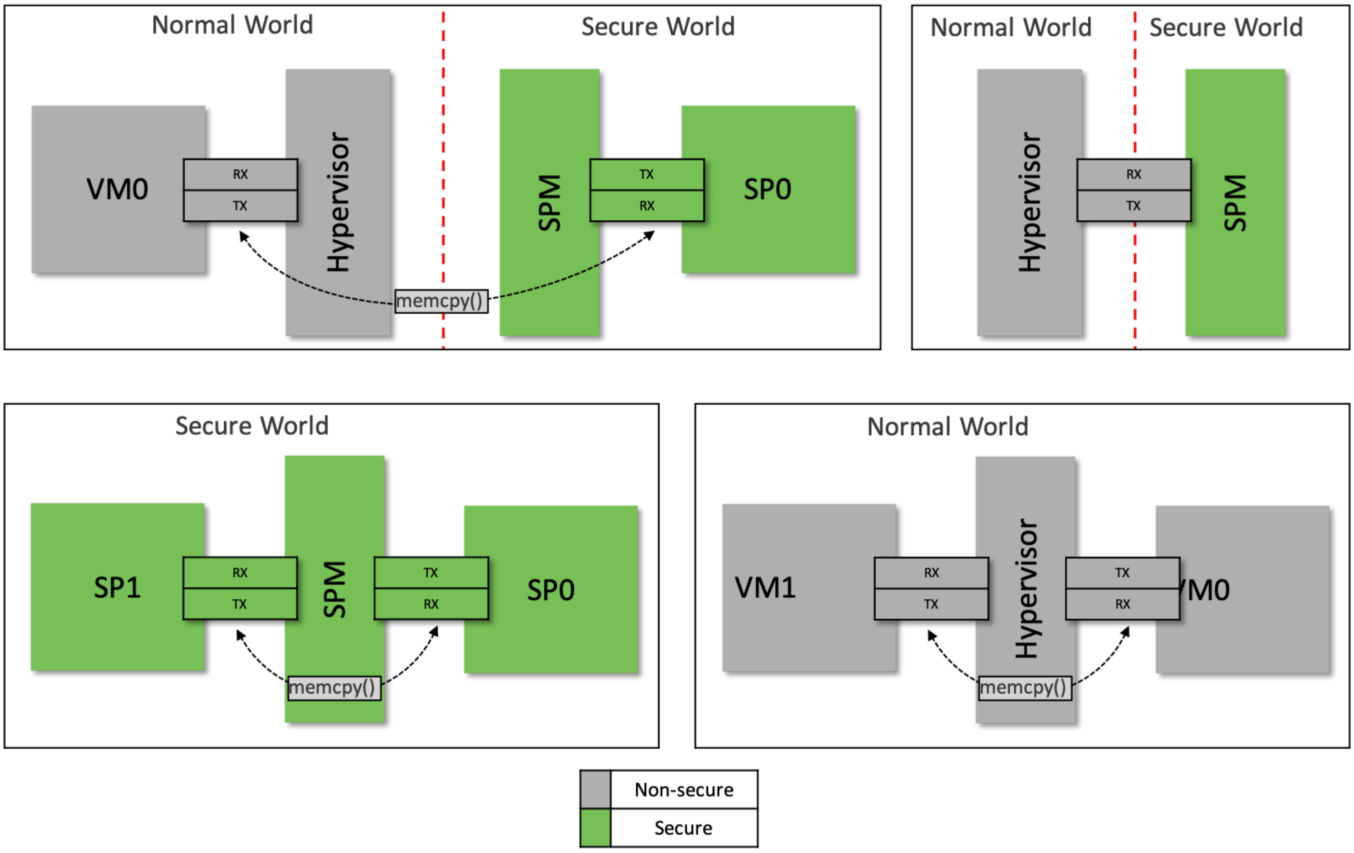

Figure 7.3: Configurations of RX/TX buffer pair between FF-A components

Mechanisms for message transmission through RX/TX buffers are described in 7.2.2.1 *Buffer-based message transmission*.

Mechanisms for discovery and setup of a RX/TX buffer pair are described in 7.2.2.2 *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 7.2.2.3 *Buffer attributes*.

7.2.2.1 Buffer-based message transmission

7.2.2.1.1 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 7.2.
2. The FFA_MSG_SEND2 ABI is used to 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 which must first acquire the RX buffer of the receiving VM from the SPMC (See 7.2.2.4.3 *Management of buffer ownership between Hypervisor and SPMC*).
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.

   In a configuration without the Hypervisor, ID *0* is assigned to the OS Kernel. It is used by a SP to send a partition message to the OS Kernel. The SPMC does not have a mechanism to detect the presence or absence of a Hypervisor. It is possible for an SP to use ID *0* to send a partition message to the Hypervisor. The SPMC copies the message from the TX buffer of the SP to the RX buffer shared between the Hypervisor and SPMC. The Hypervisor should cater for this scenario through an IMPLEMENTATION DEFINED mechanism.

Table 7.2: Encoding of a partition message header

Field	Byte length	Byte offset	Description
Flags	4	–	• Bits[31:0]: Reserved (SBZ).
Reserved	4	4	• Reserved (SBZ).
Message payload 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 payload size	4	16	• Length of message payload in bytes in the RX buffer.
Reserved	4	20	• Reserved (SBZ).
UUID	16	24	• Bytes[0..15] of UUID (see 6.2.3 *Partition UUID usage*).

7.2.2.1.2 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.
10. The producer of framework messages in the RX buffer of a consumer is responsible for preventing information from a higher Exception level leaking to a lower Exception level. The producer must ensure it treats all unused fields as Reserved (MBZ).

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

7.2.2.2 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 13.6 *FFA_RXTX_MAP*) to map it in the partition manager's translation regime. In an invocation of this ABI via the AArch32 calling convention (See [5]), the buffers allocated in the address space of the endpoint must have a 32-bit address. This is because the address of the buffer is encoded in a 32-bit register when this convention is used.

   In the latter case,

   1. The partition manager manages a stage of address translation in the translation regime of the endpoint as described in 4.1 *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 5.2.1 *Partition manifest*.
   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 13.7 *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 7.4 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.

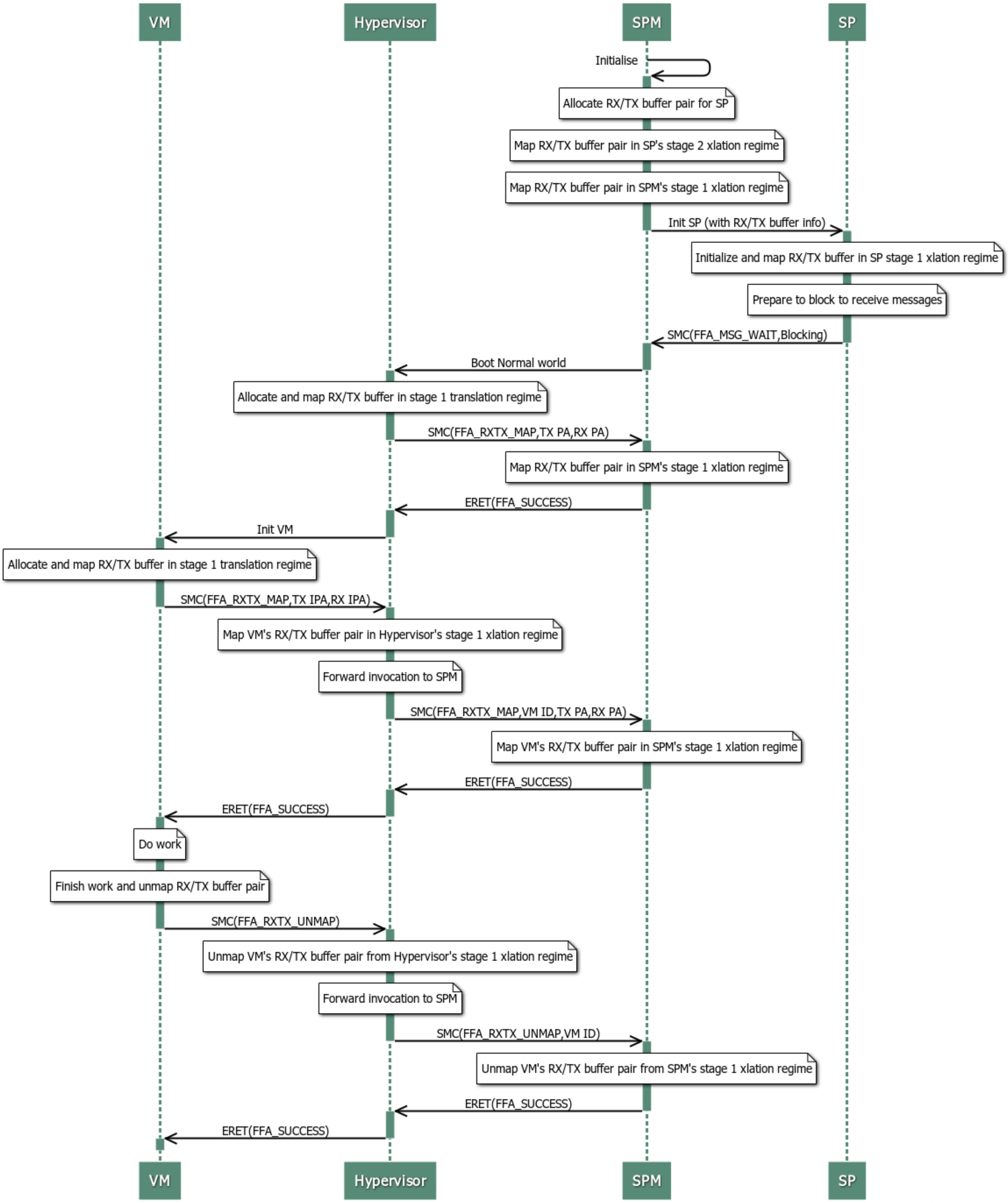

Figure 7.4: RX/TX Buffer setup

7.2.2.3 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 4.6 *Memory granularity and alignment*).
3. An endpoint discovers the minimum size, maximum 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 13.3 *FFA_FEATURES*). The maximum size is an optional field and a value of 0 means that the partition manager does not enforce a maximum size.
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 7.3 describes the minimum permission requirements of RX/TX buffer.

Table 7.3: 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.

7.2.2.3.1 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 7.4. 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 [6]).

- As a Producer, this must be done before the Consumer reads the buffer (see 7.2.2.4 *Buffer synchronization*).
- As a Consumer, this must be done before reading the buffer populated by the Producer.

Table 7.4: 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

7.2.2.3.2 Data privacy requirements

The producer of a buffer must ensure that there is no leakage of private information by clearing the unpopulated contents of the buffer. Within the data structures that populate the buffer, the following must be done:

- The producer at the lower EL must treat all unused fields in the data structure as Reserved (SBZ).
- The producer at the higher EL must treat all unused fields in the structure as Reserved (MBZ).

The rules differ based upon the Exception level of the producer because of the following reasons:

- The consumer at the higher EL does not flag an error if the unused fields are not set to zero. Hence, it is strongly recommended but not mandatory for the producer to zero the unused fields.
- The consumer at the lower EL is not trusted and the producer must set the unused fields to zero.

7.2.2.4 Buffer synchronization

The RX and TX buffers are written to by a Producer and read by a Consumer as described in Table 7.5. 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 7.5: 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

7.2.2.4.1 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.

7.2.2.4.2 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 10.8.1 *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 if the *Retain RX Buffer Ownership flag* is set to *b'0* (see Table 14.3).
   2. FFA_RX_RELEASE.

7.2.2.4.3 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 13.4 *FFA_RX_ACQUIRE*.

1. A VM's RX buffer is owned by the SPMC after it is mapped into its translation regime (see 7.2.2.2 *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 7.2.2.4.2 *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 7.5). 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.

7.3 Indirect messaging usage

7.3.1 Discovery and setup

An endpoint that can receive partition messages through Indirect messaging must specify this property in its manifest (see 5.2.1 *Partition manifest*). The scheduler that runs this endpoint can discover its presence and the number of execution contexts it implements through the following mechanisms.

1. Via an FF-A partition discovery mechanism. See 6.2 *Partition discovery*.
2. An IMPLEMENTATION DEFINED mechanism for example, Device tree.

Any endpoint can send a Indirect partition message to another endpoint by using the FFA_MSG_SEND2 ABI. In version 1.0 of the Framework, only VMs are allowed to send and receive messages through Indirect messaging. Also see 18.4 *Legacy Indirect messaging usage*.

7.3.2 Message delivery

The Framework defines the FFA_MSG_SEND2 interface to transmit a partition 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. 15.1 *FFA_MSG_SEND2* describes the FFA_MSG_SEND2 ABI. 7.2.2.1.1 *Transmission of partition messages* describes how an Indirect message is transmitted from the Sender to the Receiver through this interface.

18.6.2 *Example indirect messaging flows* illustrates example end to end flows of sending an indirect message between different combinations of endpoints.

7.3.3 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 10 *Notifications* & 10.8.1 *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 8 *Partition runtime models*.

7.4 Direct messaging usage

A Sender uses Direct messaging 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 and subsequently resumed one or more times. In the latter case, the Framework is responsible for resuming function execution.

The following ABIs are used to implement Direct messaging between a Sender and Receiver endpoint. Also see 8.3 *Runtime model for Direct request ABIs*.

- *FFA_MSG_SEND_DIRECT_REQ* & *FFA_MSG_SEND_DIRECT_REQ2*. These interfaces are used by a Sender to send a request message payload to a Receiver, allocate CPU cycles to the Receiver and wait for a response to arrive. Collectively, they are referred to as *Direct request ABIs*. Also see:
  - 15.2 *FFA_MSG_SEND_DIRECT_REQ*.
  - 15.4 *FFA_MSG_SEND_DIRECT_REQ2*.
- *FFA_MSG_SEND_DIRECT_RESP* & *FFA_MSG_SEND_DIRECT_RESP2*. These interfaces are used by a Receiver to send a response message payload to a Sender, return CPU cycles to the Sender and wait for a new message to arrive. Collectively, they are referred to as *Direct response ABIs*. Also see:
  - 15.3 *FFA_MSG_SEND_DIRECT_RESP*.
  - 15.5 *FFA_MSG_SEND_DIRECT_RESP2*.
- *FFA_INTERRUPT*. This interface is used by the Relayer to inform the Sender that Direct message processing in the Receiver was preempted (also see 9.3 *Physical interrupt actions*).
- *FFA_RUN*. This interface is used by the Sender to resume a preempted Receiver.

7.4.1 Discovery and setup

An endpoint could be capable of sending and/or receiving Direct messages. 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 a physical endpoint or a logical endpoint that is not co-resident with its partition manager in the same exception level (see Table 5.1 in 5.2.1 *Partition manifest*).
- In an IMPLEMENTATION DEFINED manner for a logical endpoint that is co-resident with its partition manager in the same exception level.

If an endpoint is able to send Direct requests, presence of support for each Direct request ABI must be specified in the manifest so that properties of this partition w.r.t Direct messaging can be populated as specified in Table 6.1 in an invocation of an FF-A discovery ABI.

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 fulfill this requirement, the Receiver must make one of the following implementation choices.

- The Receiver is implemented as a **UP** endpoint. This enables the SPMC 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 SPMC or Hypervisor to guarantee availability of an endpoint execution context for Direct messages on the same PE as the Sender.

This implementation choice is applicable to,

- A physical endpoint.
- A logical endpoint that is not co-resident with its partition manager in the same exception level.

It must be specified in the manifest of the endpoint (see Table 5.1 in 5.2.1 *Partition manifest*).

A partition manager or a logical endpoint that is co-resident with its partition manager in the same exception level can be recipients of Direct messages as well. The Framework assumes that they are implemented as per the constraints applicable to an MP endpoint i.e.

- 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.

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 via an FF-A discovery mechanism (see 6.2 *Partition discovery*). An endpoint could use the same mechanism to determine properties of other endpoints as well.

7.4.2 Message delivery and Receiver execution

The Framework uses the Direct messaging ABIs to transmit,

1. Direct partition messages between a pair of Sender and Receiver endpoints.
2. Direct framework messages between,
    1. The SPMD and SPMC.
    2. The SPMC and an SP
    3. The Hypervisor and an SP

    See the following sections for more details.

    - 18.2.4 *Power Management messages*.
    - 18.3 *VM availability signaling*.
    - 13.2.3.2 *Version discovery between Normal world and SPMC*.

The Relayers are responsible for ensuring that,

1. A request or response message has a valid message header.
2. A request or response message is sent only by an endpoint that is allowed to send that messages type.
3. An NS-Endpoint cannot send a response message to an S-Endpoint.
4. An S-Endpoint cannot send a request message to an NS-Endpoint

A request message is delivered by a Direct messaging ABI as follows.

1. The partition manager of the Receiver endpoint delivers the message to an execution context of the Receiver endpoint if,
    1. It supports receipt of Direct request messages.
    2. The execution context is in a *waiting* state.
    3. The execution context can be run on the physical PE where the request message was received by the partition manager.

    The message is delivered by the partition manager as described below.

    1. A Hypervisor delivers the message to the Receiver VM by invoking the ABI with the ERET conduit at the Non-secure virtual FF-A instance.
    2. An SPMC delivers the message to the Receiver SP as follows,
        1. An EL3 SPMC delivers the message by invoking the ABI with the ERET conduit at the,

        1. Secure physical FF-A instance for a logical Receiver SP in S-EL1.
        2. Secure virtual FF-A instance for a physical Receiver SP in S-EL0.

      2. A S-EL2 or S-EL1 SPMC delivers the message by invoking the ABI with the ERET conduit at the Secure virtual FF-A instance for a physical Receiver SP in S-EL0 or S-EL1.

      3. An SPMC delivers the message to a co-resident logical SP through an IMPLEMENTATION DEFINED mechanism.

      4. The SPMD (in a configuration without the EL3 SPMC) delivers the message to a co-resident EL3 LSP through an IMPLEMENTATION DEFINED mechanism.

2. If the partition managers for the Sender and Receiver endpoints are different, the message is forwarded by the former to the latter as described below:

   1. If the Sender of the message is a VM and the Receiver is an S-Endpoint, the message is forwarded to the SPMD by invoking the ABI at the Non-secure physical FF-A instance with the SMC conduit.

   2. The SPMD forwards a message to the S-EL2 or S-EL1 SPMC if it is targeted to a S-Endpoint managed by them. This is done by invoking the ABI at the Secure physical FF-A instance with the ERET conduit.

   3. The SPMD forwards a message targeted to a S-Endpoint managed by the EL3 SPMC through an IMPLEMENTATION DEFINED mechanism.

   4. If the Sender of the message is an S-Endpoint managed by the S-EL2 or S-EL1 SPMC and the receiver is an EL3 LSP, the SPMC forwards the message to the SPMD by invoking the ABI at the Secure physical FF-A instance with the SMC conduit.

The Receiver of the original request message is the Sender of the response message. The Sender of the original request message is the Receiver of the response message. The response message is delivered via a direct messaging ABI as follows.

1. The partition manager of the Receiver endpoint delivers the message to an execution context of the Receiver endpoint if,

   1. The execution context is the one that sent the request message.
   2. The execution context is in a *blocked* state.
   3. The execution context can be run on the physical PE where the response message was received by the partition manager.

   The responsibilities of each partition manager are listed below:

   1. A Hypervisor delivers the message to the Receiver VM by invoking the ABI with the ERET conduit at the Non-secure virtual FF-A instance.

   2. The SPM delivers the message to the Receiver SP as follows,

      1. An EL3 SPMC delivers the message by invoking the ABI with the ERET conduit at the,

         1. Secure physical FF-A instance for a logical Receiver SP in S-EL1.
         2. Secure virtual FF-A instance for a physical Receiver SP in S-EL0.

      2. A S-EL2 or S-EL1 SPMC delivers the message by invoking the ABI with the ERET conduit at the Secure virtual FF-A instance for a physical Receiver SP.

      3. The SPMC delivers the message to a co-resident LSP through an IMPLEMENTATION DEFINED mechanism.

      4. The SPMD (in a configuration without the EL3 SPMC) delivers the message to a co-resident EL3 LSP through an IMPLEMENTATION DEFINED mechanism.

2. If the partition managers for the Sender and Receiver endpoints are different, the message is forwarded by the former to the latter as described below:

1. If the Sender of the message is an S-Endpoint managed by the S-EL2 or S-EL1 SPMC and the Receiver is an NS-Endpoint or an EL3 LSP, the SPMC forwards the message to the SPMD by invoking the ABI at the Secure physical FF-A instance with the SMC conduit.
2. If the Sender of the message is an S-Endpoint managed by the EL3 SPMC and the Receiver is an NS-Endpoint, the message is forwarded to the SPMD through an IMPLEMENTATION DEFINED mechanism.
3. The SPMD forwards a message to an NS-Endpoint by invoking the ABI at the Secure physical FF-A instance with the ERET conduit.

Figure 7.5 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.

Hypcalls are used to exchange Direct framework messages between an endpoint and a partition manager.

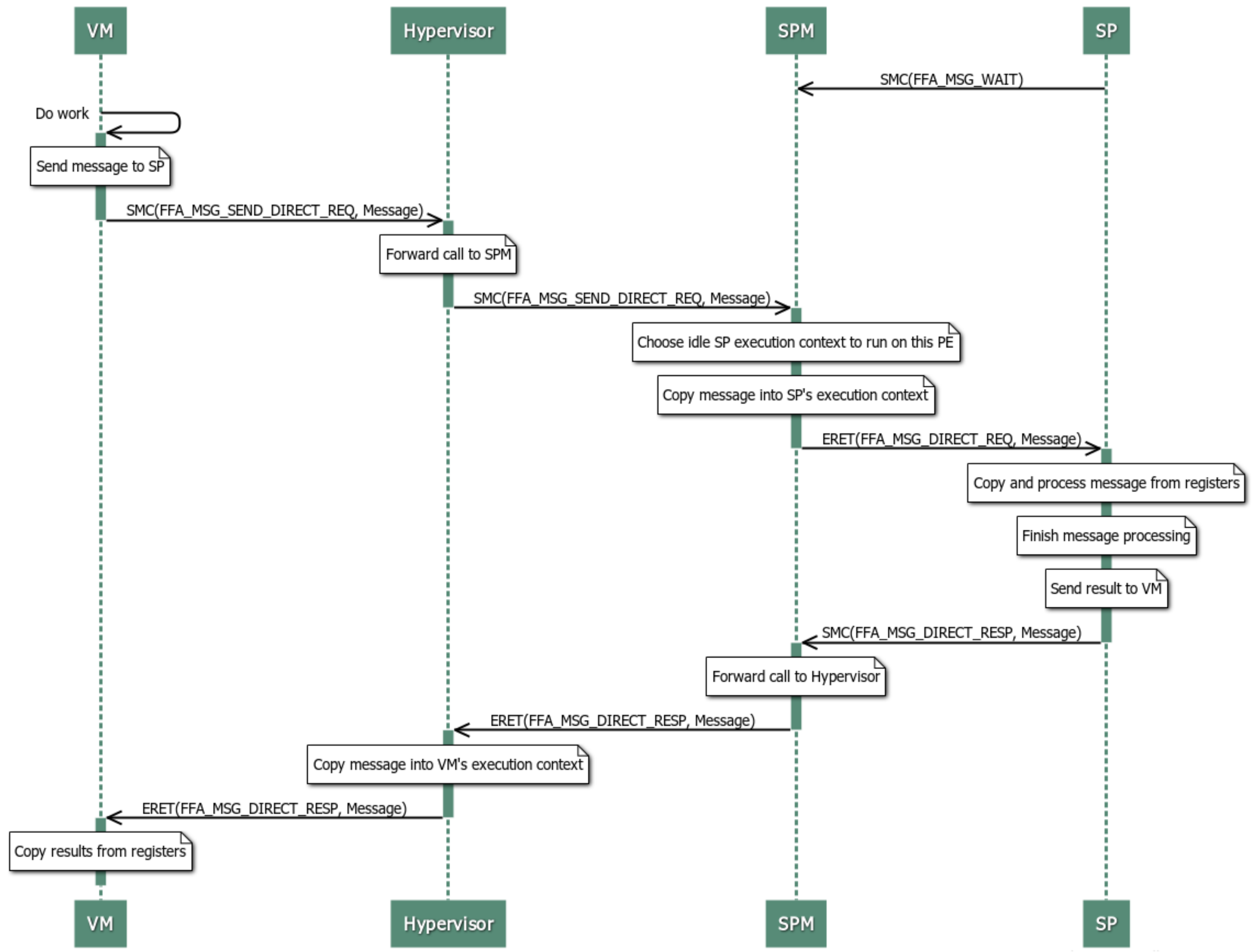

Figure 7.5: Example Direct message exchange between a VM and SP

Figure 7.6 illustrates an example flow in which an EL3 LSP sends a Direct request message to an SP managed by the S-EL2 or S-EL1 SPMC 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.

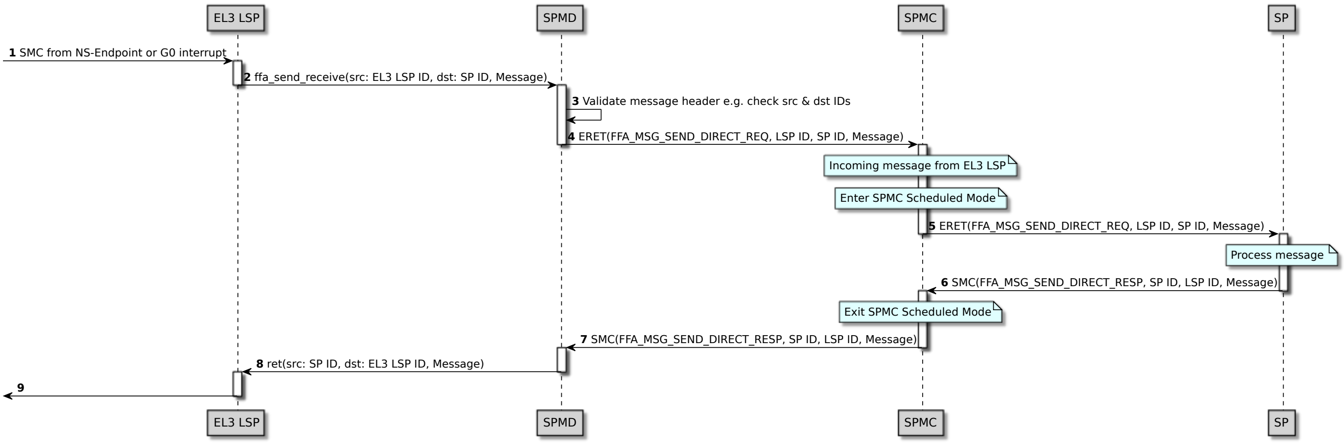

Figure 7.6: Example EL3 LSP Messaging Sequence

7.5 Compliance requirements

This section describes the compliance requirements that must be met by an FF-A message passing implementation. These requirements specify ABIs and conduits that must be implemented at a relevant FF-A instance to correctly support a message passing mechanism.

7.5.1 Compliance requirements for Direct messaging

Compliance requirements for Direct messaging depend upon the location of the message Sender and Receiver relative to each other. These are described below.

1. Sender and Receiver are at adjacent exception levels.
    1. Sender is at the higher exception level.
        1. Sender implements the FFA_MSG_SEND_DIRECT_REQ and/or FFA_MSG_SEND_DIRECT_REQ2 ABI with the ERET conduit to send a request message to the Receiver.
        2. Receiver implements the FFA_MSG_SEND_DIRECT_RESP and/or FFA_MSG_SEND_DIRECT_RESP2 ABI with the SMC, HVC or SVC conduit to send a response message to the Sender. The choice of conduit depends upon the FF-A instance where the message exchange takes place. Also see 4.4 *Conduits*.
    2. Sender is at the lower exception level.
        1. Sender implements the FFA_MSG_SEND_DIRECT_REQ and/or FFA_MSG_SEND_DIRECT_REQ2 ABI with the SMC, HVC or SVC conduit to send a request message to the Receiver. The choice of conduit depends upon the FF-A instance where the message exchange takes place. Also see 4.4 *Conduits*.
        2. Receiver implements the FFA_MSG_SEND_DIRECT_RESP and/or FFA_MSG_SEND_DIRECT_RESP2 ABI with the ERET conduit to send a response message to the Sender.
2. Sender and Receiver are not at adjacent exception levels.
    1. Sender implements the FFA_MSG_SEND_DIRECT_REQ and/or FFA_MSG_SEND_DIRECT_REQ2 ABI with the SMC, HVC or SVC conduit to send a request message to the Receiver. The choice of conduit depends upon the FF-A instance where the message exchange takes place. Also see 4.4 *Conduits*.
    2. Sender implements the FFA_MSG_SEND_DIRECT_RESP and/or FFA_MSG_SEND_DIRECT_RESP2 ABI with the ERET conduit to receive a response message from the Receiver.
    3. Receiver implements the FFA_MSG_SEND_DIRECT_REQ and/or FFA_MSG_SEND_DIRECT_REQ2 ABI with the ERET conduit to receive a request message from the Sender.
    4. Receiver implements the FFA_MSG_SEND_DIRECT_RESP and/or FFA_MSG_SEND_DIRECT_RESP2 ABI with the SMC, HVC, or SVC conduit to send a response message to the Sender. The choice of conduit depends upon the FF-A instance where the message exchange takes place. Also see 4.4 *Conduits*.
    5. Relayer (Hypervisor) at the Non-secure physical instance implements the FFA_MSG_SEND_DIRECT_REQ and/or FFA_MSG_SEND_DIRECT_REQ2 ABI with the SMC conduit to forward a request message from a VM to an SP.
    6. Relayer (Hypervisor) at the Non-secure physical instance implements the FFA_MSG_SEND_DIRECT_RESP and/or FFA_MSG_SEND_DIRECT_RESP2 ABI with the ERET conduit to forward a response message from an SP to a VM.
    7. Relayer (S-EL2 or S-EL1 SPMC) at the Secure physical instance implements the FFA_MSG_SEND_DIRECT_REQ and/or FFA_MSG_SEND_DIRECT_REQ2 ABI with the ERET conduit to forward a request message from a VM to an SP.

8. Relayer (S-EL2 or S-EL1 SPMC) at the Secure physical instance implements the FFA_MSG_SEND_DIRECT_RESP and/or FFA_MSG_SEND_DIRECT_RESP2 ABI with the SMC conduit to forward a response message from an SP to a VM.

7.5.2 Compliance requirements for Indirect messaging

Compliance requirements for Indirect messaging depend upon the role of a participating FF-A component in message transmission. These are described below.

1. Sender endpoint implements the FFA_MSG_SEND2 ABI with the SMC, HVC or SVC conduit to send a message populated in its TX buffer to the Receiver. The choice of conduit depends upon the FF-A instance where the ABI is invoked. Also see 4.4 *Conduits*.
2. The Scheduler implements the following FF-A ABIs and features to schedule a Receiver endpoint to process a message in its RX buffer.
    1. Support for the *Schedule receiver interrupt* (see 10.5 *Notification signaling*) and the FFA_NOTIFICATION_INFO_GET ABI with the SMC conduit at the Non-secure physical FF-A instance and the SMC or HVC conduits at the Non-secure virtual FF-A instance. Also see 10.7 *Compliance requirements*.
    2. If the Receiver endpoint is scheduled with the partition runtime model for FFA_MSG_SEND_DIRECT_REQ, support for Direct messaging is implemented as a Sender when the Receiver is not at the adjacent exception level as specified in 7.5.1 *Compliance requirements for Direct messaging*.
    3. If the Receiver endpoint is scheduled with the partition runtime model for FFA_RUN, support for this runtime model is implemented as specified below,
        1. FFA_RUN ABI with the SMC conduit at the Non-secure physical or the SMC or HVC conduit at Non-secure virtual FF-A instance.
        2. FFA_MSG_WAIT ABI with the ERET conduit at the Non-secure physical or virtual FF-A instance.
        3. FFA_MSG_YIELD ABI with the ERET conduit at the Non-secure physical or virtual FF-A instance.
3. Receiver endpoint implements support for,
    1. The RX buffer full notification (see 10.8.1 *RX buffer full notification*). Also see 10.7 *Compliance requirements*.
    2. The FFA_MSG_WAIT ABI with the SMC, HVC or SVC conduit, if it is scheduled with the partition runtime model for FFA_RUN (see 8.2 *Runtime model for FFA_RUN*). The choice of conduit depends upon the FF-A instance where the ABI is invoked. Also see 4.4 *Conduits*.
    3. Direct messaging as a Receiver when the Sender is not at the adjacent exception level as specified in 7.5.1 *Compliance requirements for Direct messaging*. This is applicable if it is scheduled with the partition runtime model for FFA_MSG_SEND_DIRECT_REQ (see 8.3 *Runtime model for Direct request ABIs*).
4. Relayers implement support for the FFA_RX_ACQUIRE ABI (see 13.4 *FFA_RX_ACQUIRE*) if both a VM and an SP send Indirect messages to another VM. This is described below.
    1. The Hypervisor implements this ABI with the SMC conduit at the Non-secure physical instance.
    2. A S-EL2 or S-EL1 SPMC implements this ABI with the ERET conduit at the Secure physical instance.

Chapter 8
Partition runtime models

8.1 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 4.10 *Run-time states*.
- The state transitions are described in 4.11 *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 8.2 *Runtime model for FFA_RUN*.
2. The Direct request interfaces are 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 8.3 *Runtime model for Direct request ABIs*.
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 8.4 *Runtime model for Secure interrupt handling*. Also see 9.2.1 *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 8.5 *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 Direct request 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*. Also see 9.2.3 *CPU cycle allocation modes*.

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.

8.2 Runtime model for FFA_RUN

Figure 8.1 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 Direct request ABIs 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 an endpoint execution context that is not in a call chain that leads to the currently running endpoint execution context. The execution context enters the *blocked* state. Also see *9.2.3 CPU cycle allocation modes*.
3. It cannot use the Direct response ABIs 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 available.

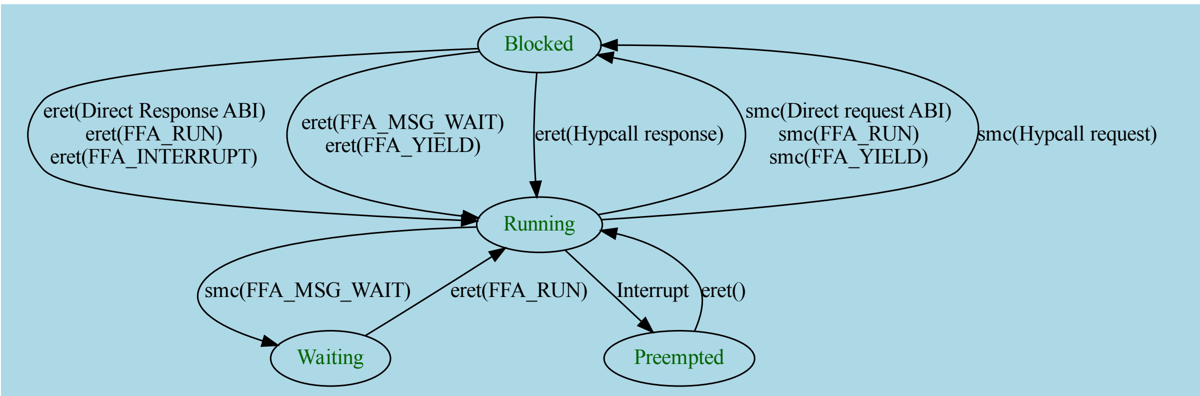

Figure 8.1: State machine for runtime model with FFA_RUN

8.3 Runtime model for Direct request ABIs

Figure 8.2 illustrates the state machine specified by the runtime model presented to an endpoint execution context that is allocated CPU cycles through the Direct request interfaces. Rules that govern this runtime model are listed below. [1]

1. It can use the Direct request interfaces 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 an endpoint execution context that is not in a call chain that leads to the currently running endpoint execution context. The execution context enters the *blocked* state. Also see 9.2.3 *CPU cycle allocation modes*.

3. It can use the smc(FFA_YIELD) transition to relinquish control back to the component that allocated CPU cycles to it and enter the *blocked* state. For example, to wait until an internal lock is available.

4. It uses the Direct response interfaces to return a response and relinquish control to the component that allocated CPU cycles to it and enter the *waiting* state. For example, to signal completion of message processing.

   It cannot use the Direct response interfaces transition to relinquish control back to any other component apart from the one that allocated CPU cycles to it and enter the *waiting* state.

5. It cannot use the smc(FFA_MSG_WAIT) transition to relinquish control back to any component and enter the *waiting* state.

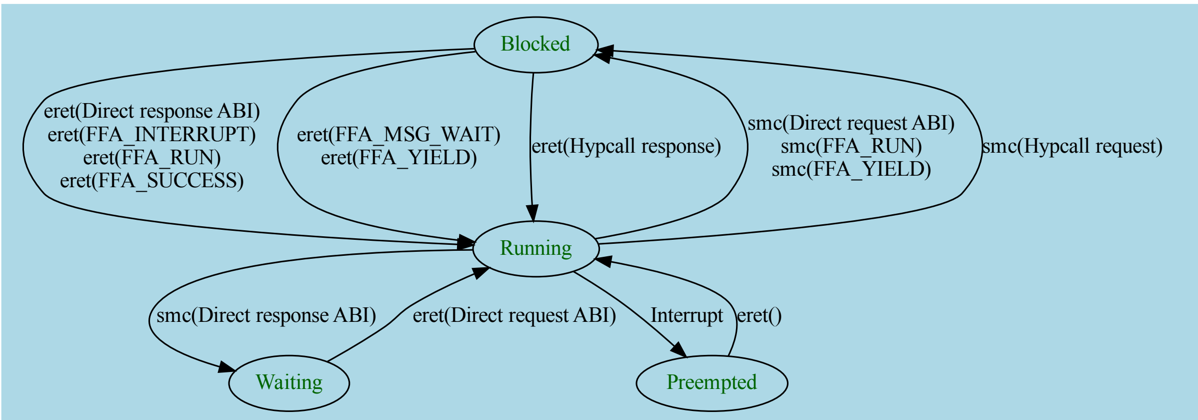

Figure 8.2: State machine for runtime model with Direct request ABIs

[1] Additional restrictions apply for power management framework messages. For more information see 18.2.4 *Power Management messages*.

8.4 Runtime model for Secure interrupt handling

Figure 8.3 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 9.2.1 *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_YIELD) transition to relinquish control back to its partition manager and enter the *blocked* state.
4. It can use the Direct request ABIs to send a message and allocate CPU cycles to any SP. The execution context enters the *blocked* state.
5. It cannot use the Direct response ABIs 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 a Direct request ABI. 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 Direct request ABIs. 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 9 *Interrupt management*.

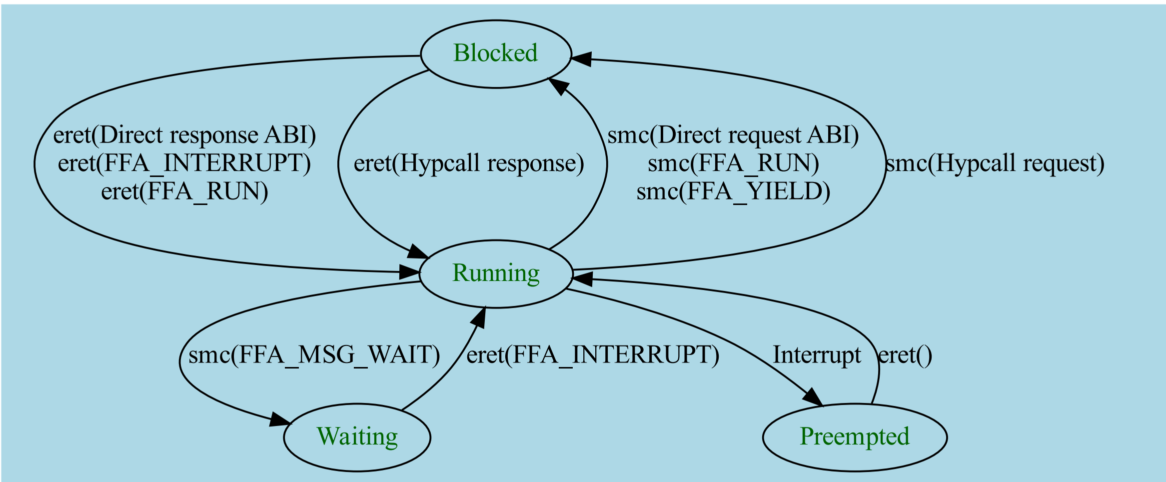

Figure 8.3: State machine for runtime model with Secure interrupt handling

8.5 Runtime model for SP initialization

Figure 8.4 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 Direct request ABIs 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 Direct response ABIs 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.

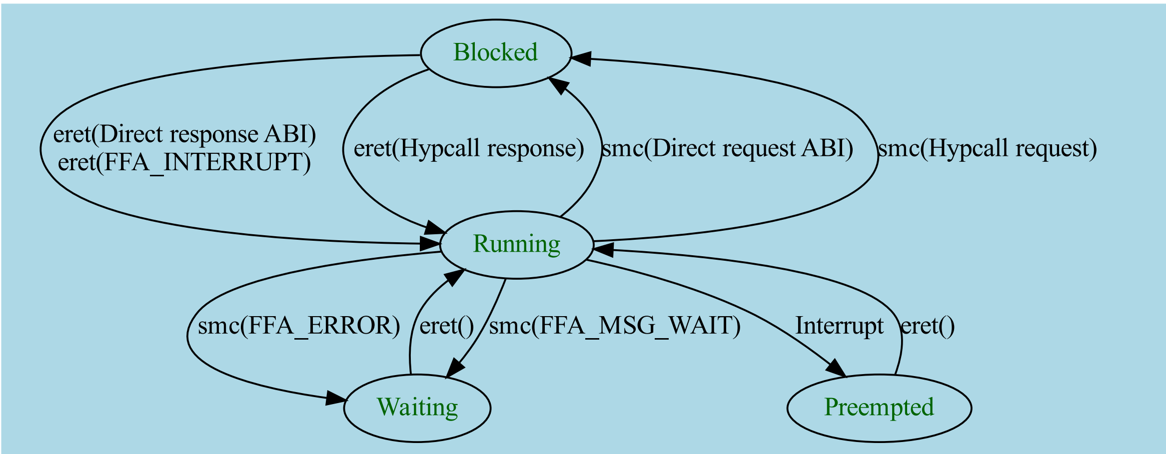

Figure 8.4: State machine for runtime model for initializing an SP execution context

Chapter 9
Interrupt management

9.1 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.

The scope of the guidance provided in this chapter applies to management of physical interrupts in relation to FF-A components in the Secure world as described below.

- Management of a Non-secure physical interrupt that triggers in the Secure world by the SPMC.
- Management of a Secure physical interrupt that triggers in the Normal world by the SPMC.
- Management of a Secure physical interrupt that triggers in the Secure world by the SPMC.
- Management of a Secure virtual interrupt targeted to a physical SP in the Secure world by the SPMC.
- Management of CPU cycles allocated by an NS-Endpoint or the SPMC.

Management of interrupts in the Normal world is IMPLEMENTATION DEFINED. The guidance in this chapter makes the following assumptions about the system configuration.

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 export a *para-virtualized* or *emulated* GIC to an SP.
3. The SPMC signals virtual interrupts to a S-EL0 SP through the ERET conduit. This is because the Arm

A-profile architecture does not support signaling of interrupts to the S-EL0 exception level through the virtual IRQ and FIQ lines.

4. 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 the SPMC in S-EL2.
5. 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.
6. Secure interrupts are configured as G1S or G0 interrupts if the GIC architecture version is 3.0 or later.
7. Non-secure interrupts are configured as G1NS interrupts if the GIC architecture version is 3.0 or later.
8. Secure interrupts are configured as G0 interrupts if the GIC architecture version is 2.0.
9. Non-secure interrupts are configured as G1 interrupts if the GIC architecture version is 2.0
10. 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.
11. Secure interrupts are routed to EL3 when execution is in the Non-secure state by programming SCR_EL3.FIQ=1.
12. All interrupts are routed to the SPMC when execution is in the Secure state. For example, with a S-EL2 SPMC, this is done by programming,
    - SCR_EL3.FIQ=0 and SCR_EL3.IRQ=0.
    - HCR_EL2.IMO=1 and HCR_EL2.FMO=1.

    On some implementations it is possible that some G0 interrupts must be handled by EL3 firmware even though they are routed to the SPMC. The Framework defines the *FFA_EL3_INTR_HANDLE* ABI to enable the SPMC to delegate handling of such an interrupt to EL3 firmware. Also see 17.1 *FFA_EL3_INTR_HANDLE*.

The guidance in this chapter based upon the above assumptions is aimed at fulfilling the following (non-exhaustive) list of requirements w.r.t interrupt management in the Secure world.

1. In the absence of GIC virtualization in the Secure world, Secure physical interrupts are delivered directly to a logical S-EL1 SP. In the absence of physical address space isolation, the physical GIC is accessible from the S-EL1 exception level. It can be configured by the logical S-EL1 SP. The SP relies on EL3 firmware to ensure that physical interrupt routing controls are programmed as described above. Together, these mechanisms guarantee that Secure physical interrupts are delivered to the SP.

   The SP does not depend on a primary or secondary scheduler in the Normal world to receive its interrupts and perform *top-half* interrupt handling. This is guaranteed by a combination of hardware and software configuration. The SP could still depend upon a primary or secondary scheduler in the Normal world for CPU cycles to perform *bottom-half* interrupt handling. This is an IMPLEMENTATION DEFINED aspect of the SP.

   In the presence of GIC virtualization in the Secure world, the physical GIC is shared among multiple physical S-EL1 SPs. Each S-EL1 SP sees a virtual GIC and handles virtual interrupts. As mentioned above, how the SPMC exposes a virtual GIC to each SP is IMPLEMENTATION DEFINED and beyond the scope of this specification. The guidance in this chapter enables the SPMC to preserve the interrupt delivery guarantee to S-EL1 SPs as described above.

   As per the interrupt routing controls described above, a Secure physical interrupt is delivered to the SPMC in S-EL2. The SPMC is responsible for signaling the corresponding Secure virtual interrupt to the target SP execution context. The SPMC ensures that this is done without a dependence on the primary or secondary scheduler in the Normal world for CPU cycles unless this is explicitly requested by the SP. This is discussed in 9.3.2 *Actions for a Secure interrupt*.
2. In the absence of GIC virtualization in the Secure world, Non-secure physical interrupts that trigger in the Secure world result in a *world-switch* to the Normal world. This enables a *co-operative* scheduling model

between the two worlds where the Secure world software strives to minimize delivery latency of Non-secure physical interrupts while maintaining its security and availability guarantees.

A commonly deployed mechanism to enable this model is where Non-secure physical interrupts are delivered directly to a logical S-EL1 SP. The SP manages its internal state and requests EL3 firmware to perform the *world switch*. The SP does not alter the state of the Non-secure interrupt in the GIC so that it can be handled as usual in the Normal world.

A less commonly deployed mechanism is to configure interrupt routing controls in the Secure world such that Non-secure physical interrupts are routed to EL3. A logical S-EL1 SP is interrupted when such an interrupt triggers. EL3 firmware arranges the *world-switch* after saving the SP's state. The SP is resumed when the Normal world runs it subsequently.

In the presence of GIC virtualization in the Secure world, a Non-secure physical interrupt is delivered to the SPMC in S-EL2. This is closer to the less commonly deployed mechanism described above. There is no equivalent for the more commonly deployed mechanism. The guidance in this chapter enables the SPMC to provide the equivalent of both mechanisms. This is discussed in 9.3.1 *Actions for a Non-secure interrupt*.

3. S-EL0 SPs are assigned devices which is akin to *user space* drivers in an OS. The device interrupts are handled by the driver in the S-EL0 SPs. The interrupts are configured as Secure physical interrupts in the physical GIC. The IRQ and FIQ lines cannot be used to signal interrupts to the S-EL0 exception level. The guidance in this chapter enables the SPMC to manage interrupts targeted to a S-EL0 SP.

9.2 Concepts

9.2.1 Secure interrupt signaling mechanisms

A virtual interrupt is signaled to a target SP execution context by the SPMC. *Signaling* means that the SPMC,

1. Uses a mechanism to indicate to the SP execution context that it has a pending virtual interrupt.
2. Runs the SP execution context so that it can handle the virtual interrupt.

The mechanism used by the SPMC to signal a virtual interrupt to the target execution context depends upon the type of SP and the run-time state from which the execution context will transition to the *running* state. The mechanisms used by the SPMC to signal an interrupt 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 SPMC *queues* the interrupt if it cannot be signaled. *Queuing* is an IMPLEMENTATION DEFINED mechanism used by the SPMC to maintain internal state that indicates that a virtual interrupt must be signaled to the target SP execution context subsequently.

For each runtime state that the target execution context of a S-EL0 or S-EL1 SP can be in, Table 9.1 and Table 9.2 respectively describe whether the SPMC can signal or must queue the Secure virtual interrupt. If it is possible to signal the interrupt, it describes the mechanism used by the SPMC to do so. If it is not possible to signal the interrupt, it describes the next runtime state when the interrupt can be signaled to the target execution context.

It is possible that a Secure virtual interrupt is queued even if the target execution context of an SP is in a runtime state where an interrupt can be signaled to it. The decision to signal or queue the interrupt is taken by the SPMC. This scenario is described in the following sections.

- 9.3.2.2 *Secure interrupt triggers in Secure state*.
- 9.3.2.2.1 *Signaling an Other S-Int in blocked state*.

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

Table 9.1: Secure interrupt signaling and queuing for a S-EL0 SP

No.	SP state	Conduit	Interface and parameters	Description
1.	Waiting	ERET	FFA_INTERRUPT, Interrupt ID	• The SPMC can signal an interrupt to the target execution context. • SPMC resumes execution of the SP through the ERET instruction.
2.	Blocked	NA	NA	• The SPMC cannot signal an interrupt to the target execution context. • The SPMC queues the interrupt and signals it when the SP execution context next enters the *waiting* state.
3.	Preempted	NA	NA	• The SPMC cannot signal an interrupt to the target execution context. • The SPMC queues the interrupt and signals it when the SP execution context next enters the *waiting* state.
4.	Running	NA	NA	• The SPMC cannot signal an interrupt to the target execution context. • The SPMC queues the interrupt and signals it when the SP execution context next enters the *waiting* state.

Table 9.2: Secure interrupt signaling and queuing for a S-EL1 SP

No.	SP state	Conduit	Interface and parameters	Description
1.	Waiting	ERET, vIRQ	FFA_INTERRUPT, Interrupt ID	• The SPMC can signal an interrupt to the target execution context. • 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.
2.	Blocked	ERET, vIRQ	FFA_INTERRUPT	• The SPMC can signal an interrupt to the target execution context. The ID of the interrupt is not specified. • 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.

No.	SP state	Conduit	Interface and parameters	Description
3.	Preempted	vIRQ	NA	• The SPMC can signal an interrupt to the target execution context. The ID of the interrupt is not specified. The FFA_INTERRUPT interface is not used. • 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.
4.	Running	ERET, vIRQ	NA	• The SPMC cannot signal an interrupt to the target execution context. • The SPMC queues the interrupt and signals it when the SP execution context next enters the *waiting*, *preempted* or *blocked* states as described above.

9.2.2 Physical interrupt types

From the perspective of an SP execution context, a physical interrupt is of one of the types listed in Table 9.3.

Table 9.3: Physical interrupt types

Acronym	Interrupt description
NS-Int	• A Non-secure physical interrupt. It requires a switch to the Normal world to be handled.
Other S-Int	• A Secure physical interrupt targeted to, – An execution context of another SP on the PE where the Secure physical interrupt is taken. – An execution context of the same SP that is different from the execution context currently running on the PE where the Secure physical interrupt is taken. ∗ For example, the physical interrupt and the corresponding virtual interrupt are SPIs. The virtual SPI is targeted to a different execution context of the same SP.
Self S-Int	• A Secure physical interrupt targeted to the SP execution context that is currently running.

9.2.3 CPU cycle allocation modes

CPU cycles are allocated to an SP execution context on a PE by either the Normal world or the SPMC so that it enters the *running* state.

1. An SP execution context runs in the *SPMC scheduled* mode if cycles are allocated by the SPMC.
2. An SP execution context runs in the *Normal world scheduled* mode if cycles are allocated by the Normal world.
3. An SP execution context in the *waiting* state enters the *running* state in the *SPMC scheduled* mode if any one of the following conditions is true:

1. The SPMC signals a virtual Secure interrupt to it. The SP execution context enters the runtime model for Secure interrupt handling. Also see 9.2.1 *Secure interrupt signaling mechanisms*.
2. Another SP execution context in the *SPMC scheduled* mode allocates cycles to it through an invocation of a Direct request ABI. The SP execution context enters the runtime model for this ABI (see 8.3 *Runtime model for Direct request ABIs*).
3. The SP execution context belongs to an EL3 LSP.

4. An SP execution context enters the *Normal world scheduled* mode when it is in the *waiting* state and it enters the *running* state when any one of the following conditions is true:
   1. A direct request ABI is used by any one of the following FF-A components to allocate CPU cycles:
      1. An NS-Endpoint execution context.
      2. An SP execution context that is in the *Normal world scheduled* mode.
   2. The FFA_RUN ABI is used by an NS-Endpoint to allocate CPU cycles.

   The SP execution context enters the runtime model corresponding to these ABIs. Also see:

   - 8.3 *Runtime model for Direct request ABIs*.
   - 8.2 *Runtime model for FFA_RUN*.

   The SPMC must return the DENIED error code in the case of an invalid state transition of an SP.
5. An SP execution context exits its CPU cycle allocation mode when it next enters the *waiting* state as described below.
   1. It was running in the runtime model for Direct request ABIs and invokes a Direct Response ABI.
   2. It was running in the runtime model for FFA_RUN and invokes the FFA_MSG_WAIT ABI.
   3. It was running in the runtime model for Secure interrupt handling and invokes the FFA_MSG_WAIT ABI.

9.2.4 SP call chains

An SP execution context in the *running* state in either CPU cycle allocation mode could run another SP execution context by invoking a Direct request ABI. This process could repeat any number of times. All SPs in this sequence of invocations are a part of a *call chain* (also see Chapter 8 *Partition runtime models*). The Framework defines two types of call chains.

1. Call chains in which all SP execution contexts run in the *SPMC scheduled* mode.
2. Call chains in which all SP execution contexts run in the *Normal world scheduled* mode.

Figure 9.1 illustrates an example of the two call chain types.

1. An NS-Endpoint issues a Direct request to SP0 to start the *Normal world scheduled* call chain in the Secure state. The NS-Endpoint enters the *blocked* state. SP0 enters the *running* state.
2. SP0 extends the call chain by issuing a Direct request to SP1. SP0 enters the *blocked* state. SP1 enters the *running* state.
3. SP1 gets preempted by a Secure physical interrupt. SP1 enters the *preempted* state.
4. The SPMC signals the corresponding Secure virtual interrupt to the target execution context of SP2. This starts the first call chain that runs in the *SPMC scheduled* mode. SP2 enters the *running* state.
5. SP2 extends the call chain by issuing a Direct request to SP3. SP2 enters the *blocked* state. SP3 enters the *running* state.
6. SP3 gets preempted by a Secure physical interrupt. SP3 enters the *preempted* state.

7. The SPMC signals the corresponding Secure virtual interrupt to the target execution context of SP4. This starts the second call chain that runs in the *SPMC scheduled* mode. SP4 enters the *running* state.

8. SP4 extends the call chain by issuing a Direct request to SP5. SP4 enters the *blocked* state. SP5 enters the *running* state.

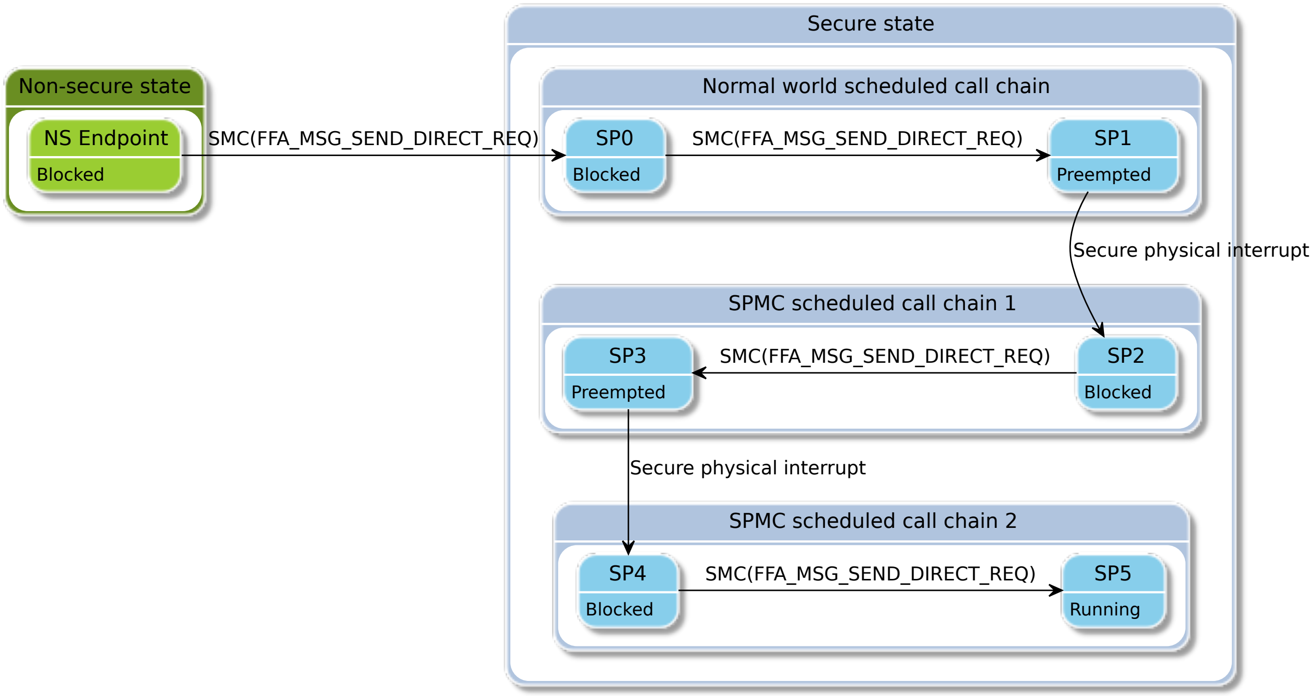

Figure 9.1: Example call chains

The following rules and guidelines govern the behavior of call chains on any PE in the Secure state.

1. A call chain cannot span PEs i.e. it can exist only on a single PE.

2. A call chain starts *winding* when the first SP execution context enters the corresponding CPU cycle allocation mode on a given PE.

3. A call chain that runs in the *SPMC scheduled* mode cannot be preempted by an *NS-Int*. This implies that an *NS-Int* is always queued when a SP runs in this mode (also see 9.3.1 *Actions for a Non-secure interrupt*).

4. A call chain in either mode starts *unwinding* on a given PE when the last SP execution context in the call chain relinquishes control to its caller by invoking a Direct response ABI.

   Additionally, a call chain in the *Normal world scheduled* mode starts *unwinding* on a given PE when the SPMC prepares to relinquish control to the Normal world in response to an *NS-Int*. Each SP execution context in the call chain either enters the *waiting* or *preempted* state depending upon the action that was specified in response to an *NS-Int* in the SP manifest. Also see 9.3.1 *Actions for a Non-secure interrupt*.

5. A call chain in the *SPMC scheduled* mode is *unwound* when each SP execution context in it enters the *waiting* state. It also exits its CPU cycle allocation mode and runtime model when it enters the *waiting* state.

6. A call chain in the *Normal world scheduled* mode is *unwound* when each SP execution context in it,

   1. Either enters the *waiting* state. In this case, it also exits its CPU cycle allocation mode and runtime model.

2. Or enters the *preempted* state. In this case, it continues to remain in its CPU cycle allocation mode and runtime model. This implies that this SP execution context enters the *running* state subsequently when it is resumed by an NS-Endpoint or a call chain that runs in the same mode.

7. A call chain is *unwound* in an order that is reverse of how it was *wound* or created.
8. On a given PE, any call chain that is created after entry into the Secure state must be unwound prior to the next exit from the Secure state.
9. When execution on a PE is in the Secure state, only a single call chain that runs in the *Normal world scheduled* mode can exist.
10. When execution on a PE is in the Secure state, any number of call chains that run in the *SPMC scheduled* mode can exist.
11. SP execution contexts in different CPU cycle allocation modes cannot be a part of the same call chain.
12. Presence of more than one call chain on a PE implies that each call chain apart from the currently active call chain was preempted by a Secure physical interrupt.
13. A call chain that runs in the *SPMC scheduled* mode cannot be preempted by the call chain that runs in the *Normal world scheduled* mode (if this call chain already exists on the same PE).

    This means that the call chain in the *SPMC scheduled* mode could be interrupted by a Secure physical interrupt. The corresponding Secure virtual interrupt could target an SP execution context in the call chain that runs in the *Normal world scheduled* mode. The SPMC queues this interrupt instead of signaling it. The interrupt is signaled after all call chains in the *SPMC scheduled* mode are unwound and the target execution context is subsequently resumed on this PE.

    This rule also implies that a call chain cannot run in the *Normal world scheduled* mode on a PE when there are unwound call chains present that run in the *SPMC scheduled* mode on the same PE. This scenario is possible only if one of the *SPMC scheduled* call chains was preempted by the *Normal world scheduled* call chain which is not possible as per the constraint described above.

    Figure 9.2 illustrates this constraint. SP3 is running in a call chain in the *SPMC scheduled* mode. This call chain was started the call chain comprising of SP0 and SP1 and running in the *Normal world scheduled* mode was preempted by *Secure physical interrupt 0*. *Secure physical interrupt 1* targeted to SP0 could pend while SP3 is running. The SPMC ensures that it remains pending until the call chain in the *SPMC scheduled* mode unwinds.

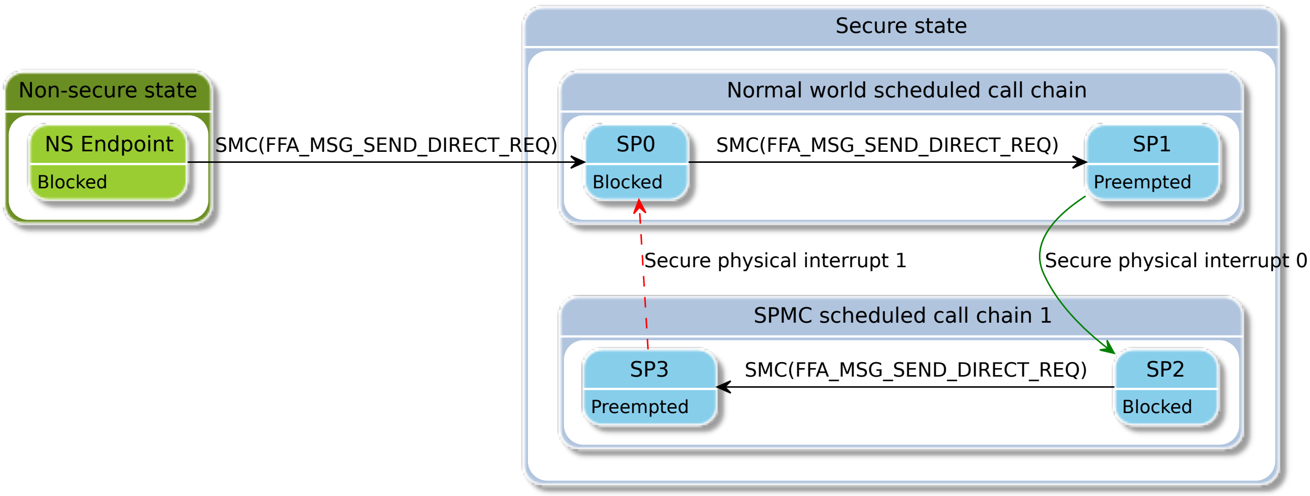

Figure 9.2: Example of queuing Other S-Ints in SPMC scheduled mode

The concept of a call chain is central to how the SPMC manages physical interrupts in the Secure world. The constraints around unwinding call chains prior to exit from the Secure state, preventing them from crossing PEs and tracking who allocated CPU cycles to them, enables requests that were posted on a given PE to be completed on the same PE. This prevents the SPMC from implementing complex scheduling policies to ensure that SP execution contexts in various runtime states with pending work are able to make progress across PEs upon being interrupted by Secure and Non-secure physical interrupts.

9.3 Physical interrupt actions

The Framework defines the actions that can be specified in response to each type of physical interrupt. An action is taken by the SPMC and depends upon the following factors,

1. The type of physical interrupt (see 9.2.2 *Physical interrupt types*).
2. The type of SP which is the target of the interrupt.
3. The runtime state of the target SP execution context.
4. IMPLEMENTATION DEFINED policy in the SPMC.

The actions are described as follows.

1. 9.3.1 *Actions for a Non-secure interrupt* describes actions in response to an *NS-Int*.
2. 9.3.2 *Actions for a Secure interrupt* describes actions in response to an *Self S-Int* or an *Other S-Int*.

9.3.1 Actions for a Non-secure interrupt

An SP specifies one of the following actions in its partition manifest (see 5.2.1 *Partition manifest*) in response to an *NS-Int* that triggers on a PE where an execution context of the SP was running.

9.3.1.1 Non-secure interrupt is signaled

The SPMC hands control to the Normal world on the PE where the interrupt triggers. The interrupt is handled in the Normal world through an IMPLEMENTATION DEFINED mechanism.

This action can be specified by both S-EL1 and S-EL0 SPs. This action is only applicable to SP execution contexts in a call chain in the *Normal world scheduled* mode. The interrupt is queued in the *SPMC scheduled* mode (see 9.3.1.3 *Non-secure interrupt is queued*).

Each applicable SP execution context in the call chain that runs in the *Normal world scheduled* mode (see 9.2.4 *SP call chains*) on that PE undergoes the following steps.

1. Enters the *preempted* state.
2. The state of the execution context is saved by the SPMC.
3. The SPMC informs the execution context that ran the preempted SP execution context that it was preempted. The FFA_INTERRUPT interface (also see 12.4 *FFA_INTERRUPT*) is used by the SPMC.

   This execution context subsequently uses the FFA_RUN interface to resume the preempted SP execution context.

Figure 9.3 illustrates an example flow where a Client in an 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.

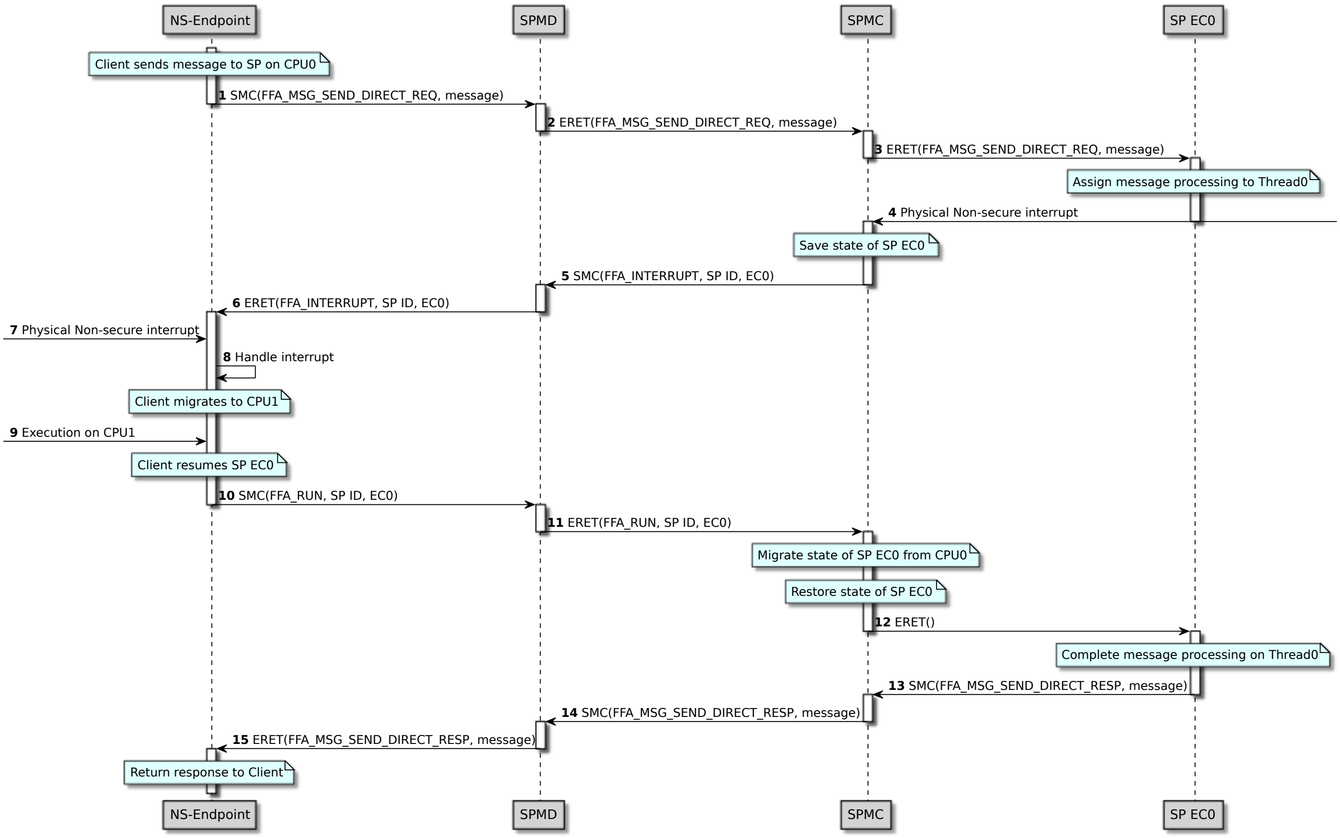

Figure 9.3: Example SP preemption flow

9.3.1.2 Non-secure interrupt is signaled after a managed exit (ME)

The SPMC hands control to the Normal world on the PE where the interrupt triggers. The interrupt is handled in the Normal world through an IMPLEMENTATION DEFINED mechanism.

This action can be specified only by S-EL1 SPs. This action is only applicable to SP execution contexts in a call chain in the *Normal world scheduled* mode. The interrupt is queued in the *SPMC scheduled* mode (see 9.3.1.3 *Non-secure interrupt is queued*).

Each applicable SP execution context on that PE enters the *waiting* state as described in 9.3.1.2.1 *Managed exit* prior to exit to the Non-secure state.

9.3.1.2.1 Managed exit

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. Within an SP execution context, the managed exit mechanism may place the running application thread in a preempted state. The execution context is able to enter the *waiting* runtime state upon completing the managed exit. This enables it to accept subsequent requests for work. Hence, other application threads running in the SP execution context are able to do work while one or more application threads have been preempted.

It is also possible that the preempted application thread is migrated to another SP execution context that runs on a different physical PE. This enables the thread to make progress. This is in contrast to the scenario where in the absence of a managed exit, the SP execution context gets preempted. In this case, the application thread gets preempted too and is unable to make progress until the SP execution context is resumed subsequently.

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 needs 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 18.2.4 *Power Management messages*.

Figure 9.4 illustrates a managed exit flow using this reason as an example where a Client 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.

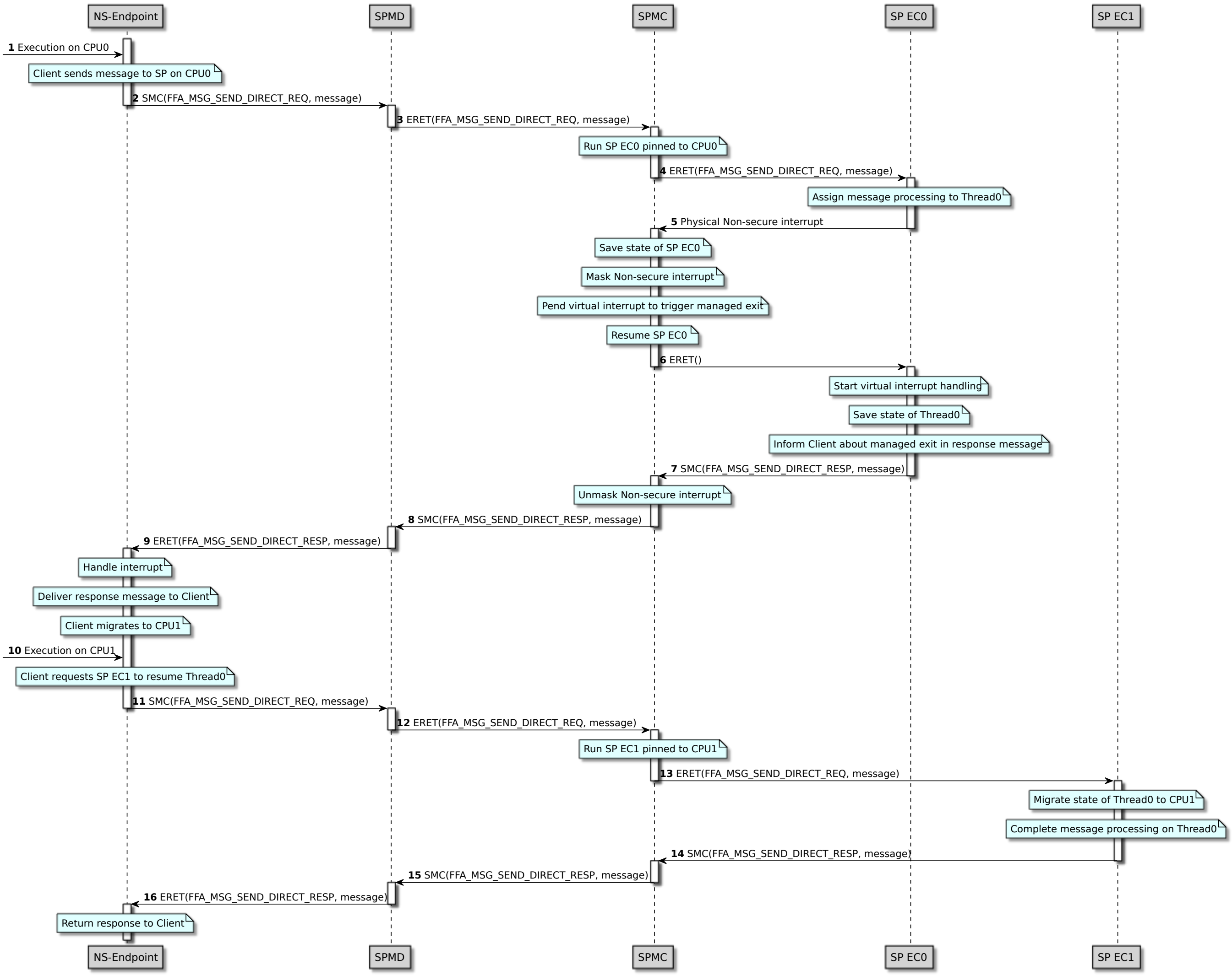

Figure 9.4: Example managed exit flow

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 5.1 in 5.2.1 *Partition manifest*) if it runs in S-EL1 in either execution state.
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* or *blocked* 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 a Direct response ABI to complete a managed exit if it was allocated cycles through a Direct request ABI (see 8.3 *Runtime model for Direct request ABIs*).
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 8.2 *Runtime model for FFA_RUN*).
9. A SP that has been asked to perform a managed exit could relinquish control and enter the *waiting* state 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.

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 9.5.

    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 13.3 *FFA_FEATURES*) and Table 13.13.

    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 9.6.

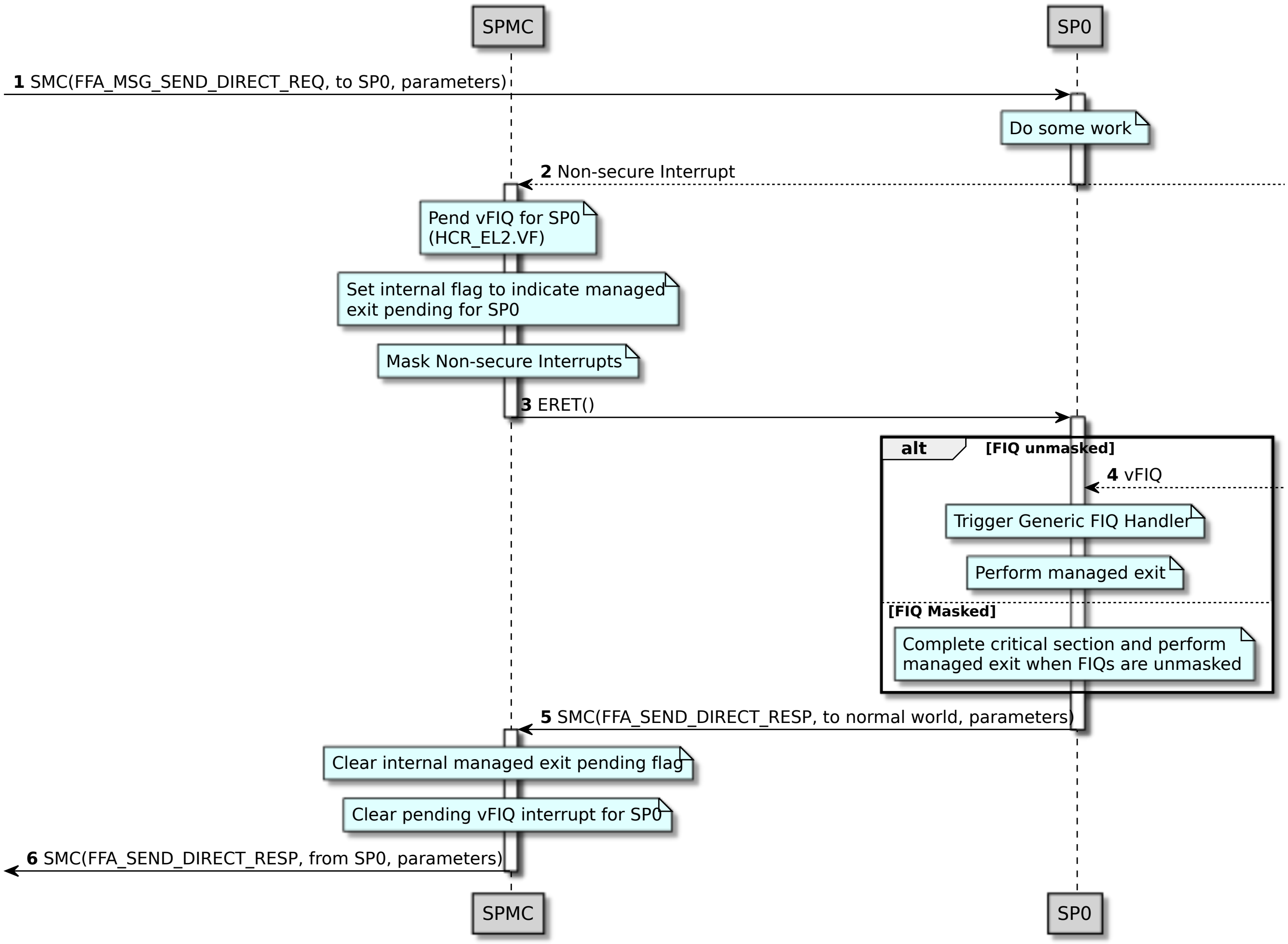

Figure 9.5: Managed exit signaling through a vFIQ

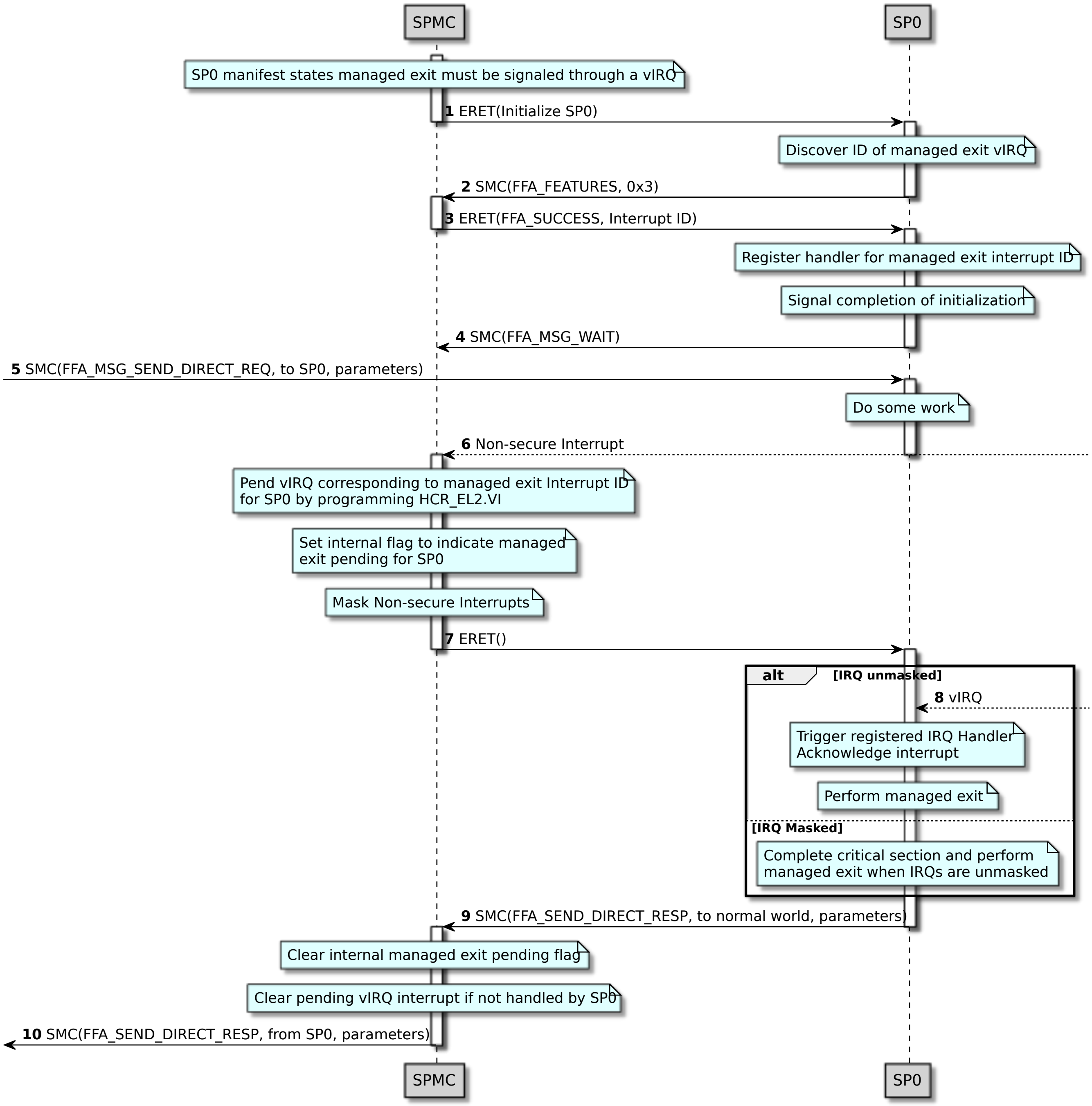

Figure 9.6: Managed exit signaling through a vIRQ

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 9.7, Figure 9.8, Figure 9.9 and Figure 9.10 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.

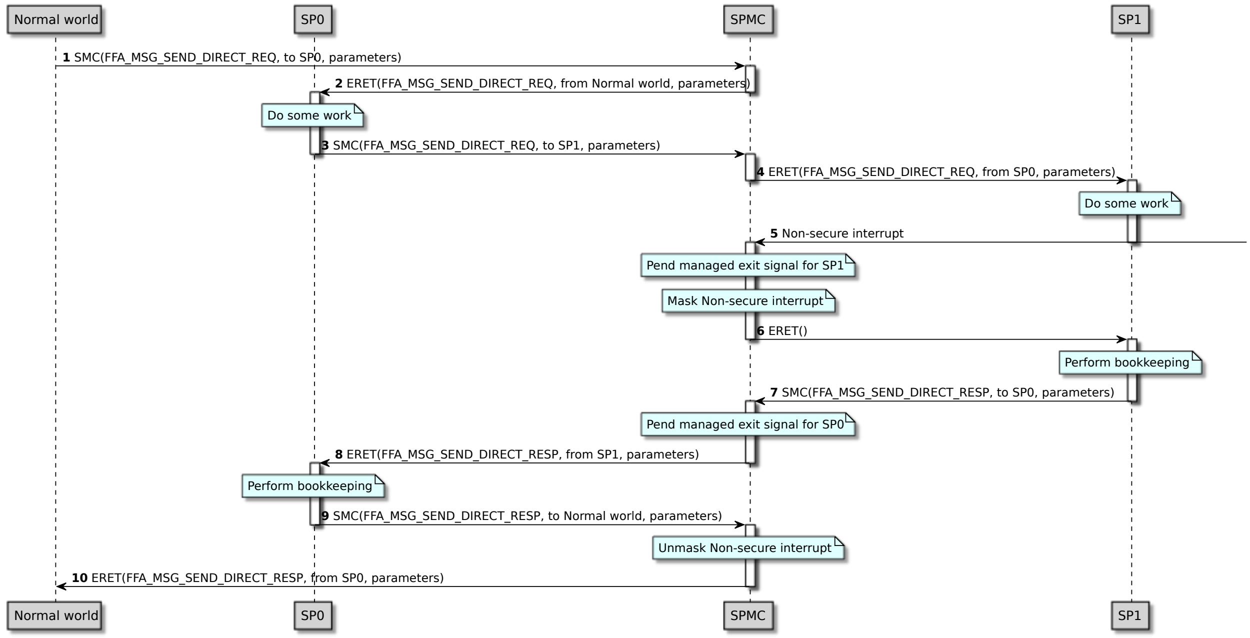

Figure 9.7: Managed exit is supported by SP0 and SP1

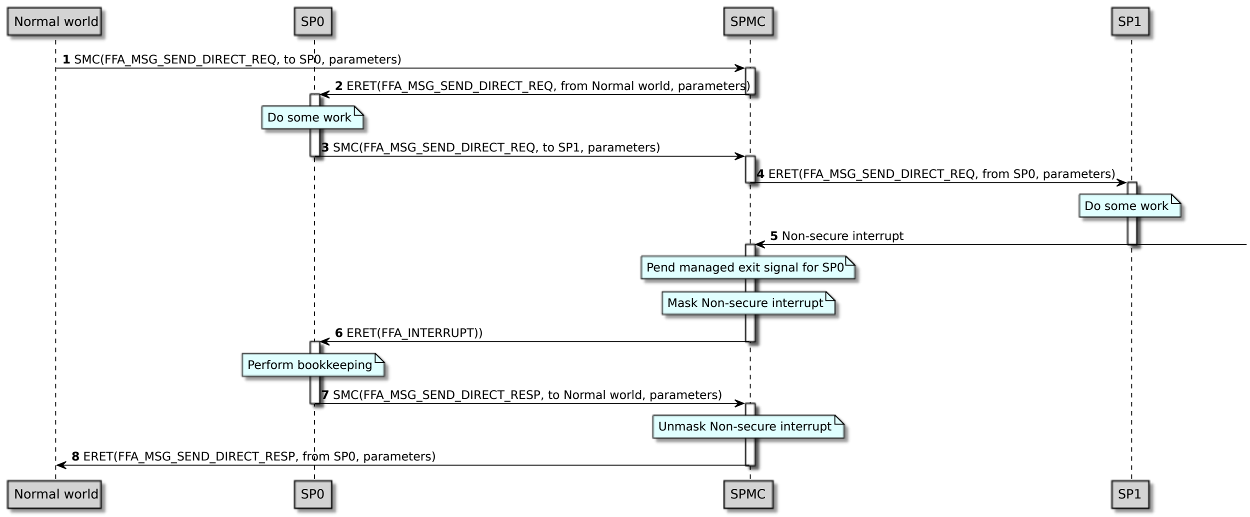

Figure 9.8: Managed exit is supported by SP0 but not by SP1

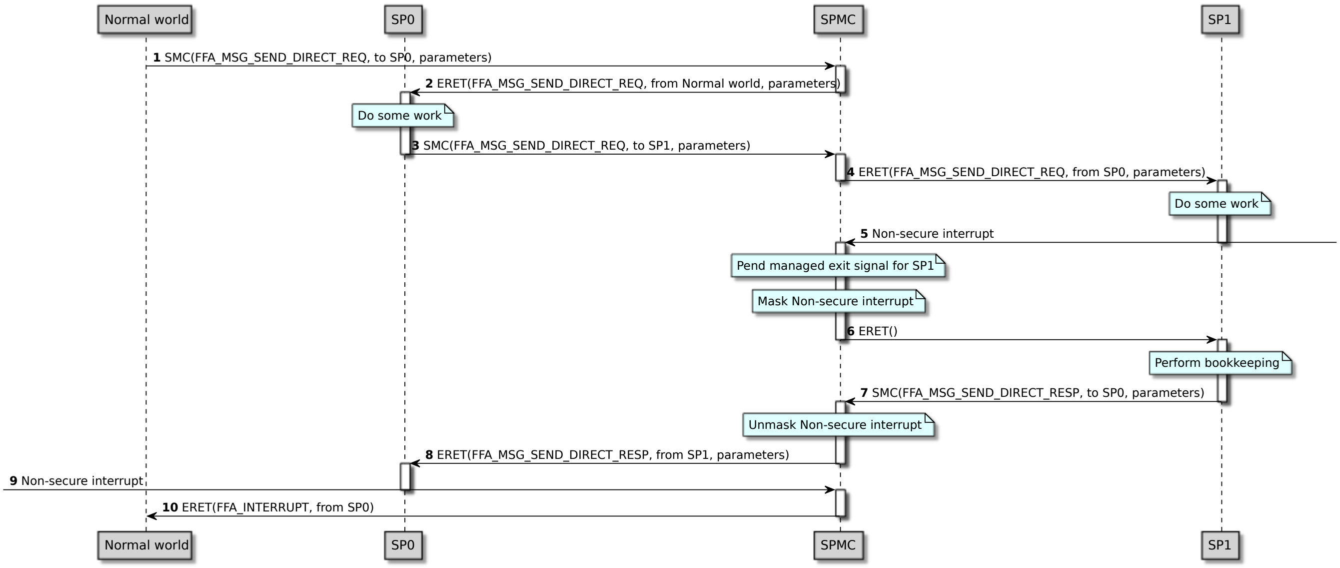

Figure 9.9: Managed exit is supported by SP1 but not SP0

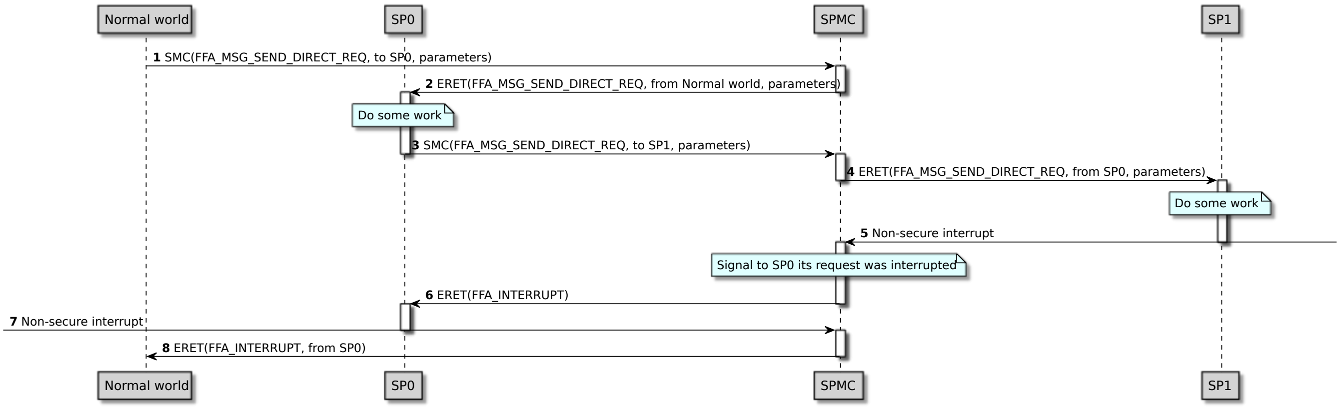

Figure 9.10: Managed exit is not supported by SP1 and SP0

9.3.1.3 Non-secure interrupt is queued

The SPMC uses an IMPLEMENTATION DEFINED mechanism to queue the interrupt such that it is never signaled to any execution context of this SP that is in the *running* state. For e.g. the SPMC could mask Non-secure interrupts in the GIC. This action can be specified by both S-EL1 and S-EL0 SPs.

Figure 9.11 illustrates an example flow where two SPs (SP0 and SP1) specify the action to queue *NS-Ints*. Normal world requests SP0 to do some work on its behalf. SP0 requests SP1 to do some work on its behalf. The SPMC ensures that *NS-Ints* are masked for the duration execution is in either SP0 or SP1. A pending *NS-Int* is handled when execution returns to the Normal world.

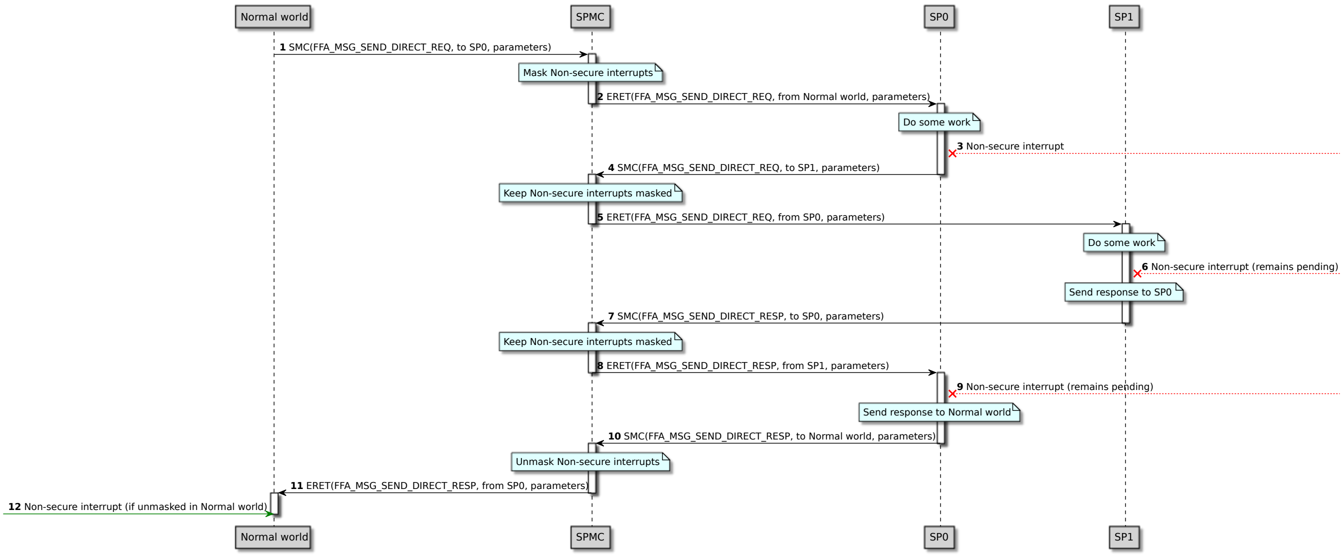

Figure 9.11: Non-secure interrupt is queued by SP0 and SP1

9.3.1.4 Precedence rules for NS-Int actions

The actions in response to an *NS-Int* are governed by the following precedence rules. < should be read as *is less permissive than*.

- *NS-Int* is queued < *NS-Int* is signaled after a ME < *NS-Int* is signaled.

An SP execution context in a call chain (see 9.2.4 *SP call chains*) could specify a less permissive action than subsequent SP execution contexts in the same call chain. The less permissive action takes precedence over the more permissive actions specified by the subsequent execution contexts. This is applicable to a call chain in either CPU cycle allocation mode.

The rationale behind this constraint is that a less permissive action effectively *decreases the priority* of an *NS-Int*. Taking a more permissive action when there are SP execution contexts in a call chain that rely on a less permissive action violates their priority model w.r.t *NS-Ints*. This constraint enables the next SP execution context to effectively *inherit the priority* of *NS-Ints* w.r.t the previous SP execution context in a call chain.

Figure 9.12 illustrates an example of this constraint as described below.

1. SP0 specifies the *NS-Int* is queued action for while running.
2. SP1 specifies the *NS-Int* is signaled action while running[1].
3. SP0 runs SP1 by invoking FFA_MSG_SEND_DIRECT_REQ.
4. The SPMC ensures that the action specified by SP0 takes precedence over the action specified by SP1.
5. An *NS-Int* that triggers while SP1 is running remains pending.
6. The *NS-Int* is handled when SPMC forwards the response from SP0 to the Normal world.

A variant of the above example is where SP0 is preempted by a Secure interrupt targeted to an SP1 execution context (*Other S-Int*). The new call chain in the *SPMC scheduled* mode is started on the same PE if the SP1 execution context is run to handle the corresponding Secure virtual interrupt. The SPMC applies the same mitigation described above to avoid violating the action specified by SP0 and leaving the SP0 call chain *unwound* prior to exit to the Non-secure state.

[1] SP1 could specify the *NS-Int* is signaled after a managed exit action. The action specified by SP0 would still take precedence.

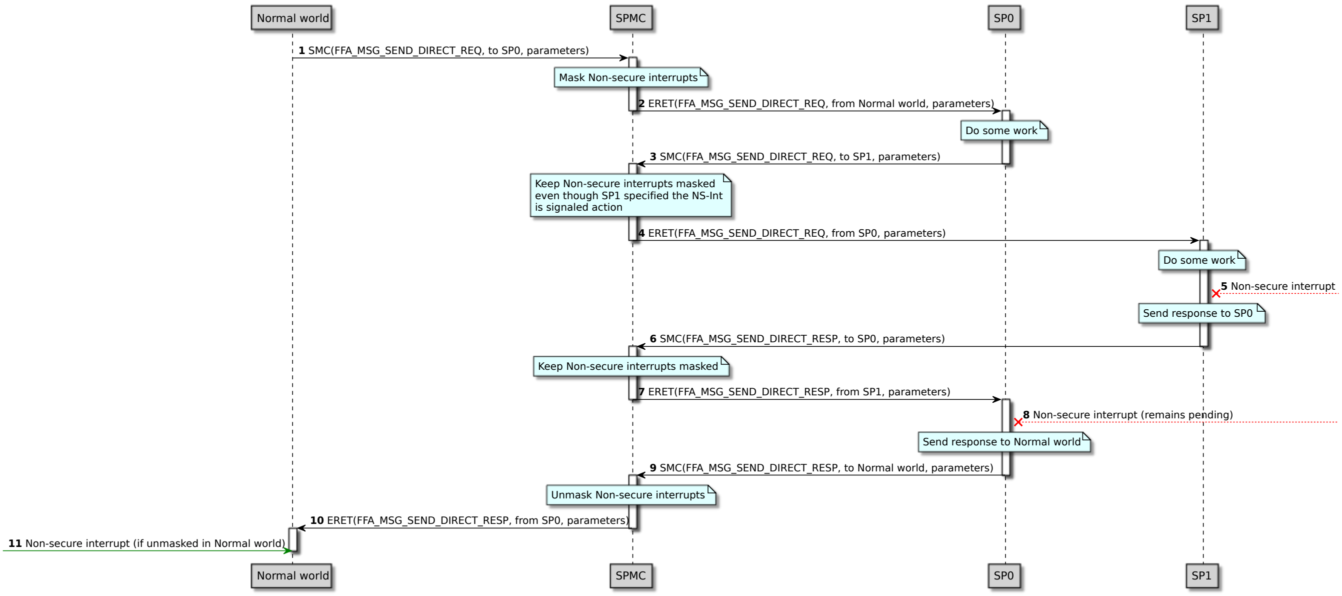

Figure 9.12: SP0's action takes precedence over SP1's action

9.3.2 Actions for a Secure interrupt

A Secure physical interrupt could trigger in the Normal world or the Secure world. The SPMC ensures that the corresponding Secure virtual interrupt is signaled to the target SP execution context without a dependence on the primary or secondary scheduler in the Normal world for CPU cycles.

9.3.2.1 Secure interrupt triggers in Non-secure state

A Secure physical interrupt preempts the Normal world if it triggers when execution is in the Non-secure state. The action taken by the SPMC in response to this interrupt depends upon the runtime state of the target SP execution context as listed below.

1. The execution context is in the *waiting* state. The SPMC signals the corresponding Secure virtual interrupt to the execution context as described in Table 9.1 and Table 9.2. This starts a new call chain that runs in the *SPMC scheduled* mode.

2. The execution context is in the *running* state on a different PE. The SPMC queues the corresponding Secure virtual interrupt and signals it to the target execution context as described in Table 9.1 and Table 9.2.

3. The execution context is in the *preempted* state on the same or a different PE. The SPMC queues the corresponding Secure virtual interrupt.

   In case of an S-EL1 SP, the interrupt is signaled when the execution context next enters the *running* state as described below.

   1. The execution context was preempted by an *NS-Int* in the *Normal world scheduled* mode. In this case, the queued Secure virtual interrupt is signaled when the Normal world resumes the call chain that the execution context is a part of subsequently.

   2. The execution context was preempted by an *Other S-Int* in either CPU cycle allocation mode on a PE different from where the Secure physical interrupt triggered. In this case, the queued Secure virtual interrupt is signaled when the SPMC resumes the execution context subsequently. This happens when the call chain that the SP execution context is a part of is resumed by the SPMC. This scenario is applicable to an un-pinned SP execution context.

      The execution context could not have been preempted by an *Other S-Int* on the PE where the Secure physical interrupt triggered. This is because the SPMC must unwind all call chains prior to exit to the

Non-secure state on a given PE. During unwinding, an SP execution context can enter the *preempted* state only in response to an *NS-Int*. This would happen if it does not request a managed exit instead. This scenario is applicable to a pinned SP execution context.

In case of an S-EL0 SP, the interrupt can be signaled only when the target SP execution context enters the *waiting* state (also see Table 9.1). Prior to entering this state, it enters the *running* state as described above. The interrupt is signaled only after it subsequently enters the *waiting* state.

4. The execution context is in the *blocked* state. This implies that the execution context is a part of a call chain on a different PE. This is because the SPMC must unwind all call chains prior to exit to the Non-secure state on a given PE. Hence, after unwinding, no execution context can be in the *blocked* state on the PE where the Secure physical interrupt has triggered. In this scenario, the Secure virtual interrupt is queued by the SPMC.

   As mentioned in Table 9.1, the interrupt cannot be signaled to a target S-EL0 SP execution context. It is signaled when the execution context next enters the *waiting* state via the *running* state.

   If the target execution context is of a S-EL1 SP, it is signaled when the execution context next enters the *running* state on the same PE where is entered the *blocked* state.

   These transitions happen when the call chain that the *blocked* SP execution context is a part of is unwound.

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. Also see 14.4 *FFA_NORMAL_WORLD_RESUME*.

9.3.2.2 Secure interrupt triggers in Secure state

A Secure physical interrupt preempts the running SP execution context if it triggers when execution is in the Secure state. The interrupted execution context enters the *preempted* state. The action chosen by the SPMC in response to this interrupt depends upon its type as described below (see 9.2.2 *Physical interrupt types*).

1. The interrupt is a *Self S-Int* and the target execution context belongs to a S-EL0 SP. The virtual interrupt is queued as it can be signaled only when the execution context enters the *waiting* state (also see Table 9.1).
2. The interrupt is a *Self S-Int* and the target execution context belongs to a S-EL1 SP. The virtual interrupt is signaled as specified in Table 9.2. The target execution context re-enters the *running* state.
3. The interrupt is an *Other S-Int*. The action taken by the SPMC depends upon its IMPLEMENTATION DEFINED policy. The SPMC could either signal or queue the corresponding Secure virtual interrupt. This decision depends upon the runtime state of the target SP execution context as listed below.
   1. The execution context is in the *waiting* state. The SPMC can signal the corresponding Secure virtual interrupt to the execution context as described in Table 9.1 and Table 9.2. This starts a new call chain that runs in the *SPMC scheduled* mode.

      The SPMC uses an IMPLEMENTATION DEFINED policy to decide whether the interrupt is signaled or not. The SPMC ensures that the virtual interrupt is signaled to the target SP execution context before the next exit to the Non-secure state on this PE.
   2. The execution context is in the *running* state on a different PE. The SPMC queues the corresponding Secure virtual interrupt and signals it to the target execution context as described in Table 9.1 and Table 9.2. The SPMC ensures that the virtual interrupt is signaled to the target SP execution context before the next exit to the Non-secure state on this PE.
   3. The execution context is in the *blocked* state. If the target execution context belongs to a S-EL0 SP, the interrupt is queued as it can be signaled only when the execution context enters the *waiting* state (also see Table 9.1). This happens when the corresponding call chain is unwound prior to exit from the Secure state.

      The action taken by the SPMC in case of a S-EL1 SP is described in 9.3.2.2.1 *Signaling an Other S-Int in blocked state*.
   4. The execution context is in the *preempted* state. In this case, the SPMC queues the Secure virtual interrupt. It is signaled when the execution context next enters the *running* state as described below.

1. The execution context was preempted by an *NS-Int* in the *Normal world scheduled* mode. In this case, the queued Secure virtual interrupt is signaled when the Normal world resumes the call chain that the SP execution context is a part of subsequently.
2. The execution context was preempted by an *Other S-Int* in either CPU cycle allocation mode. In this case, the queued Secure virtual interrupt is signaled when the SPMC subsequently resumes the call chain that the SP execution context is a part of.

9.3.2.2.1 Signaling an Other S-Int in blocked state

The action taken by the SPMC when an *Other S-Int* can be signaled to a target S-EL1 SP execution context in the *blocked* state depends upon the following factors.

1. The interrupted SP execution context (presently in *preempted* state) is a part of a call chain that was running in the *SPMC scheduled* mode or the *Normal world scheduled* mode.
2. The target SP execution context (presently in *blocked* state) is a part of a call chain that was running in the *SPMC scheduled* mode or the *Normal world scheduled* mode.
3. The interrupted and target execution contexts are a part of the same or different call chains. In the latter case, the call chains could reside on different PEs.

The scenarios that arise due to a combination of these factors are described in Table 9.4.

Table 9.4: Scenarios for signaling an Other S-Int in blocked state

No.	CPU cycle allocation mode of preempted execution context	CPU cycle allocation mode of target execution context	Part of the same call chain	Valid configuration
1	Normal world	Normal world	Yes	Yes
2	Normal world	Normal world	No	Yes
3	Normal world	SPMC	Yes	No[2]
4	Normal world	SPMC	No	Yes
5	SPMC	Normal world	Yes	No[3]
6	SPMC	Normal world	No	Yes
7	SPMC	SPMC	Yes	Yes
8	SPMC	SPMC	No	Yes

Each valid scenario in Table 9.4 is described below.

1. Scenario 1.
    1. The virtual interrupt is targeted to an SP execution context that ran earlier in the same call chain before entering the *blocked* state.

       An example of the scenario is illustrated in Figure 9.13. In an SP call chain comprising of SP0, SP1 and SP2, a *Other S-Int* targeted to SP0 occurs as step 3 when SP2 is running after step 2. SP0 is in the *blocked* state as it ran earlier in the same call chain.

[2] SP execution contexts in different CPU cycle allocation modes cannot be a part of the same call chain. Also see 9.2.4 *SP call chains*.
[3] SP execution contexts in different CPU cycle allocation modes cannot be a part of the same call chain. Also see 9.2.4 *SP call chains*.

Figure 9.13: Example of scenario 1

1. The SPMC can signal the virtual interrupt to the target SP execution context as described in Table 9.2. This *rewinds* the call chain.

   There could be intermediate SP execution contexts in the *blocked* state in the call chain between the preempted and the target execution context. For example, in Figure 9.13, SP1 is an intermediate execution context.

   1. The SPMC could leave all intermediate execution contexts in the *blocked* state and resume the target execution context for handling the interrupt. This is illustrated in Figure 9.14.

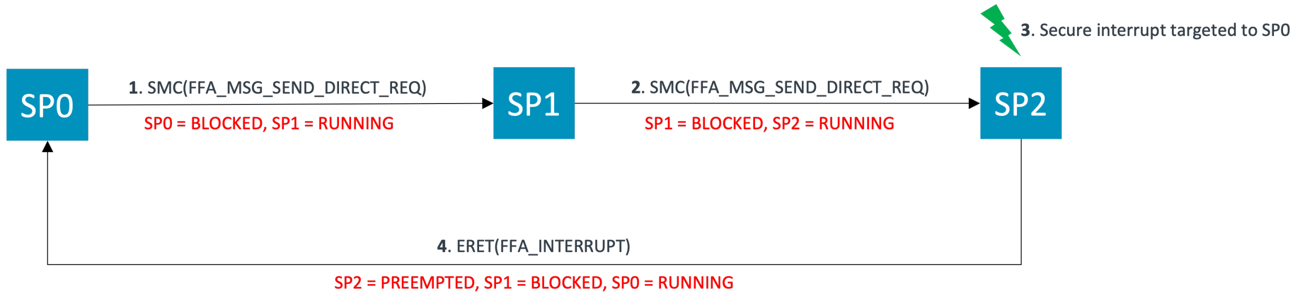

Figure 9.14: Intermediate execution context is left in blocked state in scenario 1

   1. The SPMC could place all intermediate execution contexts in the *preempted* state and resume the target execution context for handling the interrupt. This is illustrated in Figure 9.15.

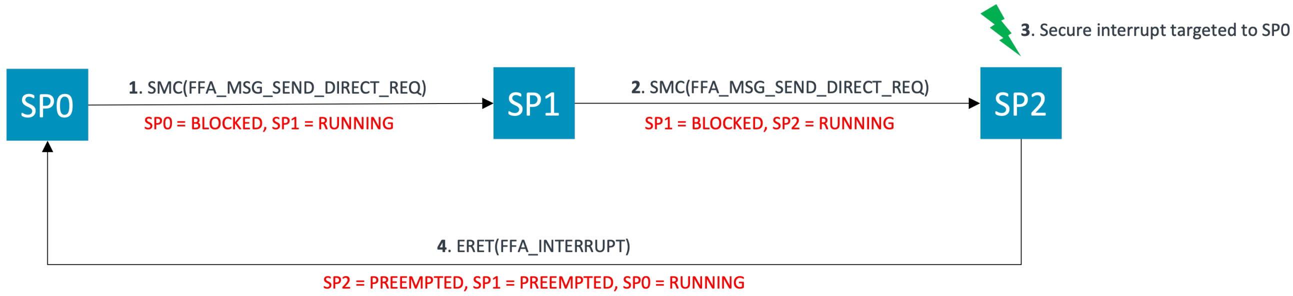

Figure 9.15: Intermediate execution context is left in preempted state in scenario 1

   The choice of mechanism used by the SPMC is IMPLEMENTATION DEFINED.

2. After the target SP execution context has handled the interrupt, it uses the FFA_RUN ABI to resume the request due to which it had entered the *blocked* state earlier.

   1. If the SPMC left all intermediate execution contexts in the *blocked* state as illustrated in Figure 9.14, then it bypasses these execution contexts and resumes the SP execution context that was originally preempted.

   2. If the SPMC left all intermediate execution contexts in the *preempted* state as illustrated in Figure 9.15, then it places these execution contexts in the *blocked* state and resumes the SP execution context that was originally preempted. Effectively, the call chain is *recreated*.

2. The virtual interrupt is targeted to an *intermediate* SP execution context in the *blocked* state as illustrated in Figure 9.16 i.e. an interrupt targeted to SP1 occurs while SP0 is handling the earlier interrupt.

   In this case, the SPMC queues the interrupt for the target execution context. It is signaled after the preempted execution context finishes interrupt handling. In the example illustrated in Figure 9.16, SP0 is resumed so that it can finish handling the original interrupt. SP1 is resumed subsequently.

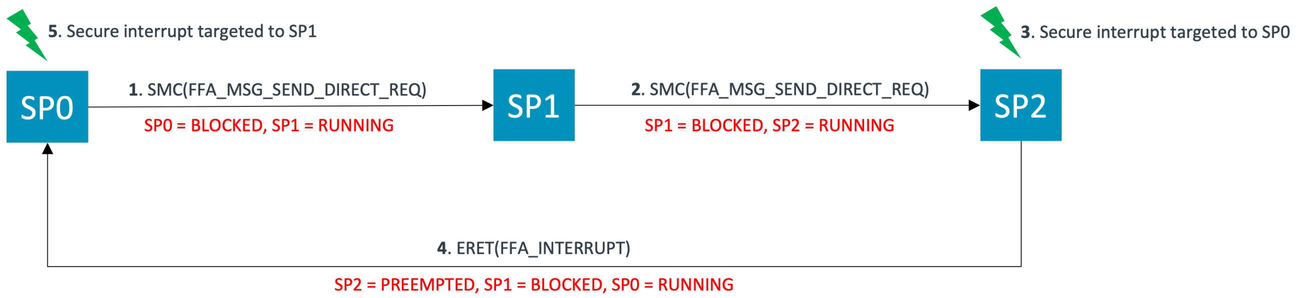

Figure 9.16: Interrupt is targeted to an intermediate execution context in scenario 1

2. Scenario 2.
   1. The virtual interrupt is targeted to an SP execution context that ran earlier in a different call chain. Since that call chain is running in the *Normal world scheduled* mode, it is active on a different PE.
   2. The SPMC queues the virtual interrupt and signals it to the target execution context when it next enters the *running* state on that PE. This happens when the call chain is unwound on that PE.
3. Scenario 4.
   1. The virtual interrupt is targeted to an SP execution context that ran earlier in a call chain on a different PE. This is because a call chain cannot run in the *Normal world scheduled* mode when there are unwound call chains that run in the *SPMC scheduled* mode on the same PE (also see 9.2.4 *SP call chains*).
   2. The SPMC queues the virtual interrupt and signals it to the target execution context when it next enters the *running* state on that PE. This happens when the call chain is unwound on that PE.
4. Scenario 6.
   1. The virtual interrupt is targeted to an SP execution context that ran earlier in a call chain on a different PE. This is because a call chain that runs in the *SPMC scheduled* mode cannot be preempted by a call chain that runs in the *Normal world scheduled* mode on the same PE (also see 9.2.4 *SP call chains*).
   2. The SPMC queues the virtual interrupt and signals it to the target execution context when it next enters the *running* state on that PE. This happens when the call chain is unwound on that PE.
5. Scenario 7.
   1. This scenario is the same as Scenario 1 apart from the difference that the call chain runs in the *SPMC scheduled* mode.
   2. This scenario is handled in the same way as Scenario 1.
6. Scenario 8.
   1. The virtual interrupt is targeted to an SP execution context that ran earlier in a different call chain. Since that call chain is running in the *SPMC scheduled* mode, it could be active on the same or a different PE.
   2. The SPMC queues the virtual interrupt and signals it to the target execution context when it next enters the *running* state on that PE. This happens when the call chain is unwound on that PE.

9.3.2.3 Exiting a CPU cycle allocation mode with pending virtual interrupts

An execution context of an S-EL1 SP running under the S-EL2 SPMC could exit its CPU cycle allocation mode with pending virtual interrupts.

To enable the SP execution context to handle the pending interrupts, one approach could be to request additional CPU cycles from the SP's scheduler. The mechanism used to inform the scheduler is IMPLEMENTATION DEFINED. An example mechanism is where the SP uses an FF-A notification before exiting its CPU cycle allocation mode to inform its scheduler (see 10.5 *Notification signaling*).

An alternative approach is where the SPMC arranges re-entry into the SP execution context so that it can handle its pending virtual interrupts. In this mechanism, after detecting that the SP execution context has pending virtual interrupts, the SPMC uses the FFA_INTERRUPT ABI with the ERET conduit to enter the SP execution context in the SPMC scheduled mode. The number of times the SPMC repeats this action is IMPLEMENTATION DEFINED. Figure 9.17 illustrates this mechanism.

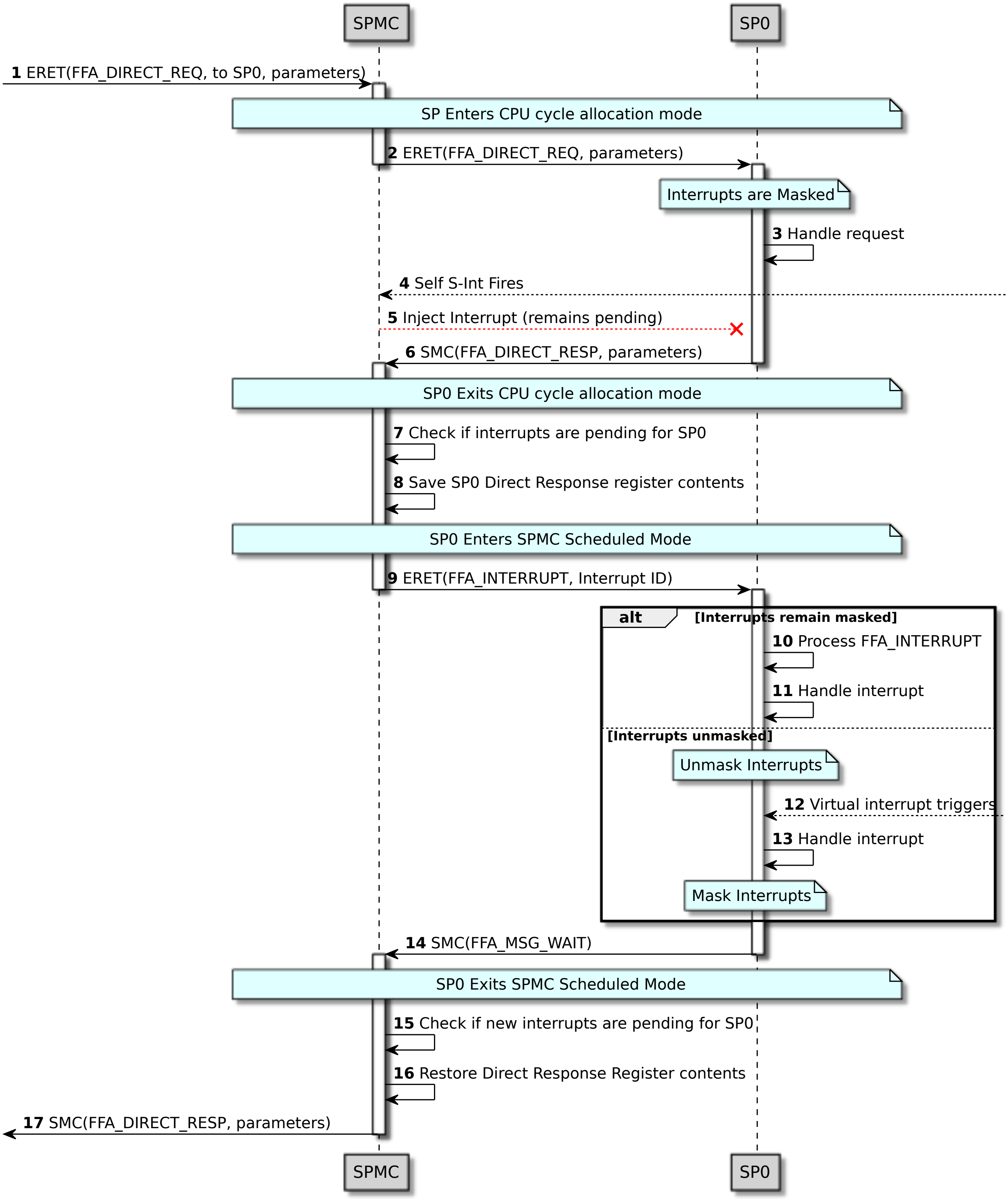

Figure 9.17: Example Self S-Int delivery for SEL1 SP with interrupts masked

9.4 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 scheduling actions as described below.

- The *Run to completion* model is recommended for S-EL0 SPs that only handle Secure interrupts. Hence, these SPs never run in the *Normal world scheduled* mode. The SP specifies the queued action for *NS-Ints*. *Self S-Ints* are always queued in the running state. The SP relies on an IMPLEMENTATION DEFINED mechanism provided by the SPMC to specify which *Other S-Ints* can preempt its execution e.g. through an interrupt priority scheme.
- The *Preemptible* model is recommended for S-EL0 SPs that only process messages. Hence, these SPs never run in the *SPMC scheduled* mode. The SP specifies the signaled action for *NS-Ints*. *Self S-Ints* are not used. The SP relies on an IMPLEMENTATION DEFINED mechanism provided by the SPMC to specify that *Other S-Ints* are signalable.

Chapter 10
Notifications

10.1 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 4.9 *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 10.1 illustrates the notification mechanism and its participants.

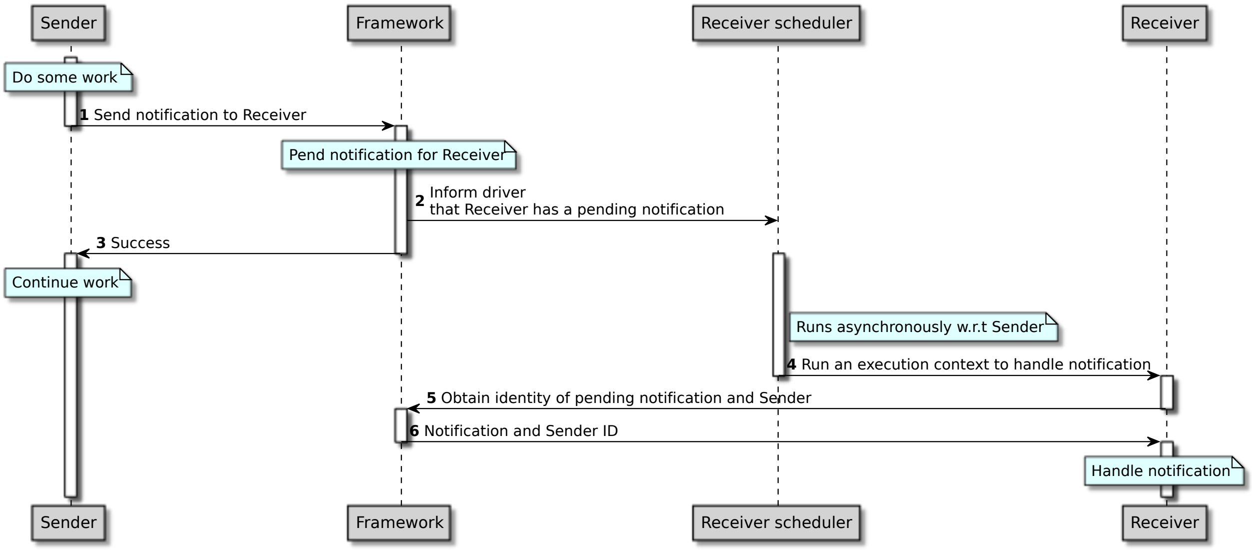

Figure 10.1: Example notification flow

Support for notifications in the Framework for a configuration that includes both the Hypervisor and SPM 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 10.8 *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 10.3 *Notification bitmap setup*.
8. The Framework provides an interface to the Sender to specify the notification to signal to the Receiver. Also see 16.5 *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 specify which endpoint can signal a particular notification. The Receiver notification is bound to the Sender endpoint. Also see 10.4.2 *Notification binding*.

13. The Framework provides an interface to the Receiver to determine the identity of the notification. Also see 16.6 *FFA_NOTIFICATION_GET*.

14. The Framework provides no guarantees when a notification will be handled by the Receiver.

15. 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 10.7 *Compliance requirements*.

Guidance on support for notifications in a Framework configuration without the Hypervisor is specified in 10.9 *Notification support without a Hypervisor*.

10.1.1 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 7.1.2 *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.

10.2 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[1] and SPMC on the notification bitmap of each type of endpoint are described in Table 10.1.
3. Permissions of VMs and SPs on the notification bitmap of each type of endpoint are described in Table 10.2.

Table 10.1: 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 10.2: 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

[1] RW permissions for the Hypervisor does not imply that it can access the memory allocated by the SPMC for the VM or HYP framework notifications bitmap. This implies that the Hypervisor can use the FFA_NOTIFICATION_SET ABI to Direct the SPMC to pend notifications in these bitmaps.

10.3 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 10.4 *Notification configuration* and 10.5 *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 16.1 *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 16.2 *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 10.2 illustrates how the Hypervisor and SPMC create notification bitmaps on behalf of a VM and SP respectively.

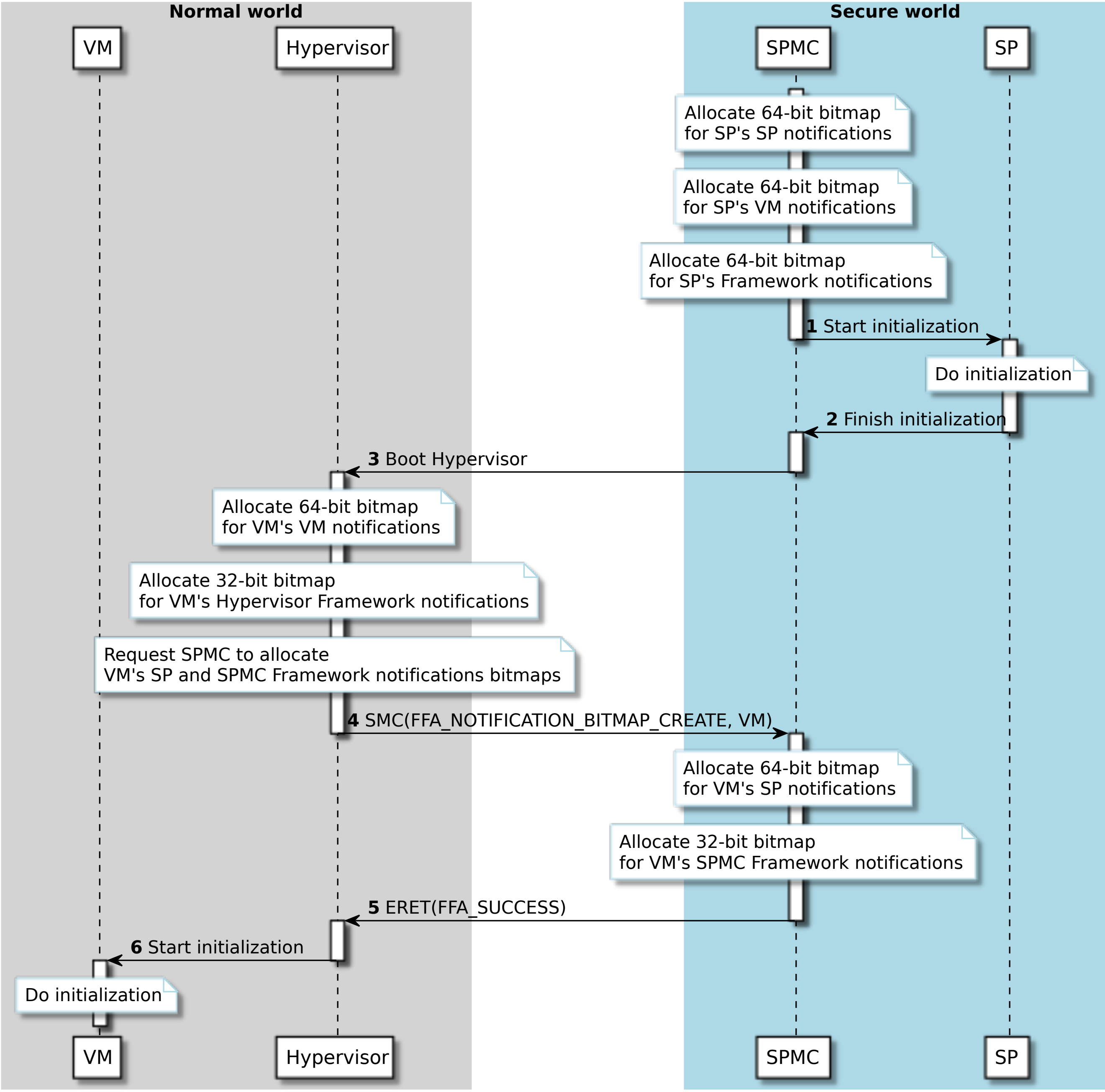

Figure 10.2: Notification bitmap creation for a VM and SP

10.4 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 10.5 *Notification signaling*.

1. The Receiver and its scheduler configure support for handling interrupts used by the Framework for notification signaling. See 10.4.1 *Notification interrupt setup*.
2. The Receiver binds a non-framework notification to an endpoint that is allowed to signal it. See 10.4.2 *Notification binding*.

10.4.1 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. An 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 13.3 *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 13.13.
4. A Receiver obtains the description of the *Notification pending interrupt* by invoking the FFA_FEATURES interface (see 13.3 *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 13.13.

Figure 10.3 illustrates an example setup of the *Schedule Receiver 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.

  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.
- The FF-A driver discovers the *Schedule Receiver 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.

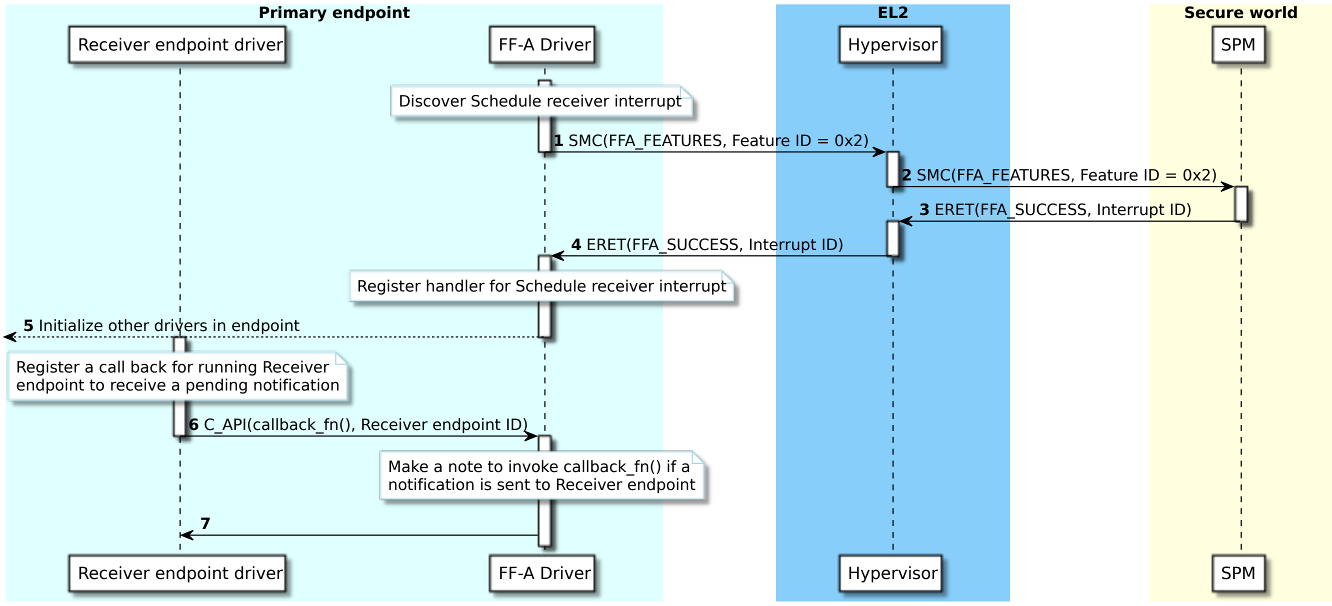

Figure 10.3: Schedule receiver interrupt setup in primary endpoint

Figure 10.4 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.

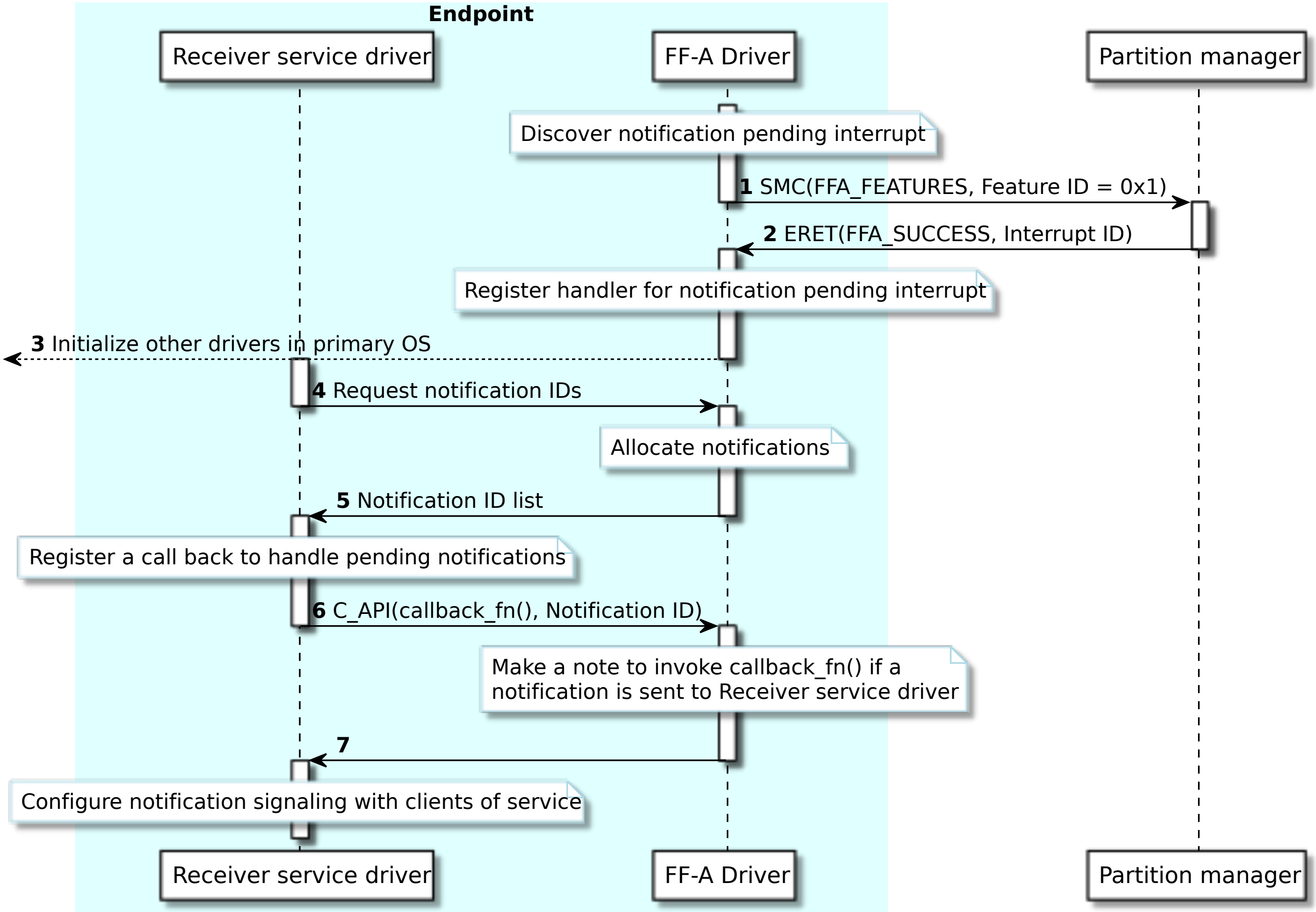

Figure 10.4: Notification pending interrupt setup in a Receiver endpoint

Receipt of the Schedule Receiver Interrupt and/or Notification Pending Interrupt by an FF-A component depends upon the following conditions:

1. Endpoint is a recipient of one or more notifications.
2. Endpoint is responsible for scheduling another endpoint that is a recipient of notifications.

Table 10.3 describes which notification interrupts an FF-A component may receive.

Table 10.3: Valid Notification Interrupt configurations

Receiver Component	Sender Component	Responsible for scheduling another recipient of notifications	Recipient of Notifications	Receives Schedule Receiver interrupt	Receives Notification Pending interrupt
VM	Hypervisor	Yes	Yes	Yes	Yes
VM	Hypervisor	No	Yes	No	Yes
VM	Hypervisor	Yes	No	Yes	No
VM	Hypervisor	No	No	No	No

Receiver Component	Sender Component	Responsible for scheduling another recipient of notifications	Recipient of Notifications	Receives Schedule Receiver interrupt	Receives Notification Pending interrupt
Physical SP (S-EL1)	SPMC (S-EL2)	No	Yes	No	Yes
Physical SP (S-EL1)	SPMC (S-EL2)	No	No	No	No
Hypervisor or OS Kernel	SPMC (S-EL1, S-EL2, EL3)	Yes	Yes	Yes	No

10.4.1.1 Interrupt properties

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

1. The type of interrupt should be inferred from the interrupt ID specified in Table 13.13. 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. The interrupt is edge-triggered.
3. The Security state of the 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 10.4.1 *Notification interrupt setup*.

The rules that govern the properties of the *Notification pending interrupt* are the same as the rules for the *Schedule Receiver interrupt* except for the following.

1. The type of the *Notification pending interrupt* is either a PPI or SGI.
2. The Security state of the *Notification pending interrupt* is the same as the Security state of the endpoint it is targeted to.

10.4.2 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 16.3 *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 16.4 *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*.

    A Receiver can have one or more *per-vCPU* and *global* notifications pending at any point of time. Additionally, the same *per-vCPU* notification could pend for multiple vCPUs of the same Receiver at the same time. Also see 10.5 *Notification signaling*.
8. The type of notification is specified by the Receiver endpoint when the notification is bound to the Sender endpoint.
9. An unbound notification is neither global nor per-vCPU i.e., it does not have a type associated with it.

Figure 10.5 illustrates an example flow of how a VM can bind a global notification to a SP

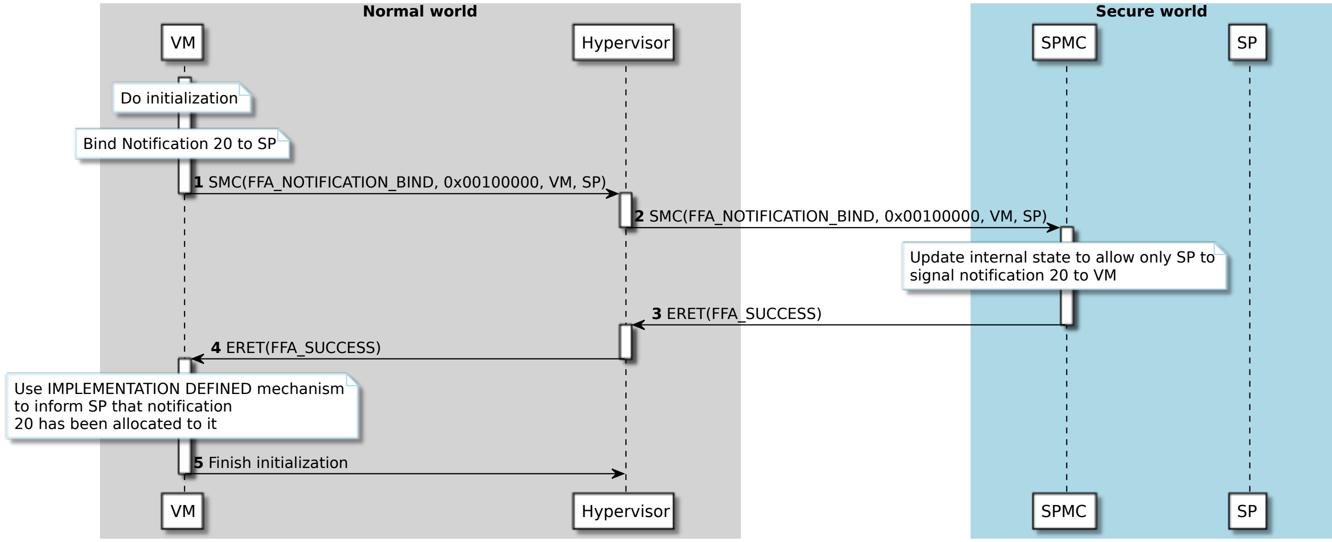

Figure 10.5: Binding a global notification from VM to SP

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 10.6 illustrates an example flow of how,

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

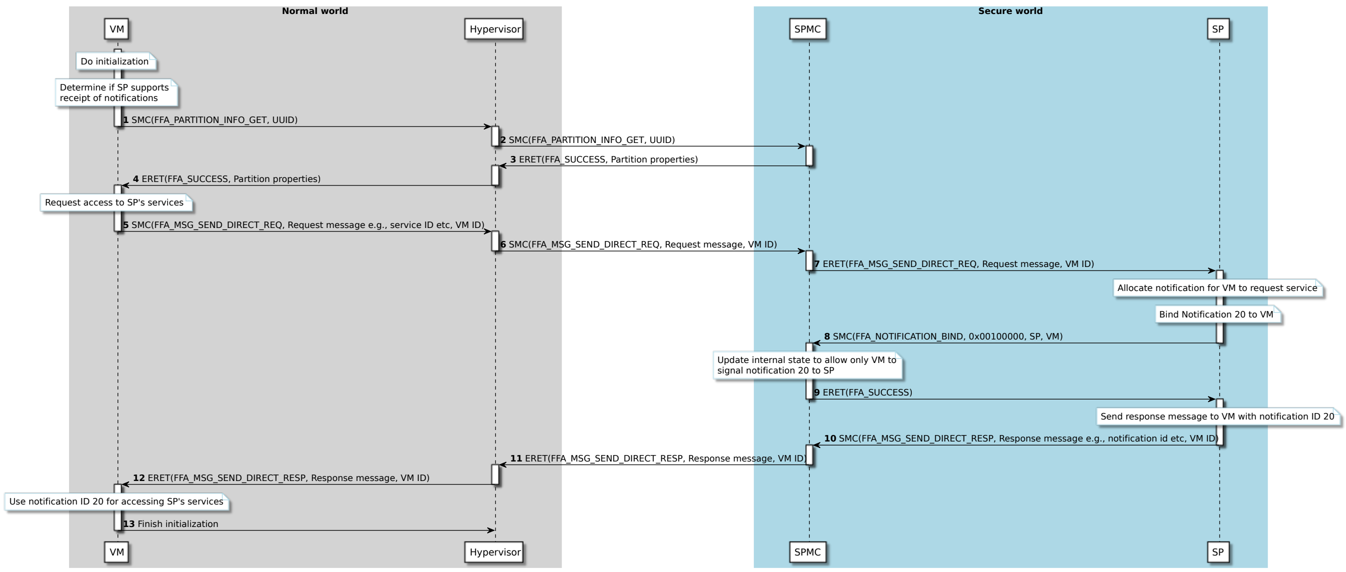

Figure 10.6: Notification binding between a VM and SP

10.5 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 obtains CPU cycles e.g. it 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 16.5 *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 is determined by the partition manager of the Receiver through an IMPLEMENTATION DEFINED mechanism.

      This mechanism is applicable to only partitions that run in EL1 or S-EL1.
   2. The Receiver's scheduler uses a Direct request 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 16.6 *FFA_NOTIFICATION_GET*).

   For example, a S-EL1 SP could invoke this interface while handling the *Notification pending interrupt*.
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 10.4.1 *Notification interrupt setup*) is used by the Partition manager to inform the Receiver's scheduler that the Receiver has one or more pending notifications. Assertion of this interrupt to signal a pending notification is the responsibility of the partition manager that writes to the notifications bitmap of the Receiver.

   The partition manager uses an IMPLEMENTATION DEFINED policy to determine when the *Schedule Receiver* interrupt must be asserted in response an invocation of the FFA_NOTIFICATION_SET interface. The interrupt could be asserted before or after an invocation of this interface completes.

   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 16.7 *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.

10.5.1 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 7.4.1 *Discovery and setup*.

5. The endpoints bind the following notifications as described in 10.4.2 *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 10.4.1 *Notification interrupt setup*.

18.6.1 *Example notification flows* illustrates some additional example end to end flows of signalling a notification between different combinations of endpoints and system configurations.

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

Figure 10.7 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.

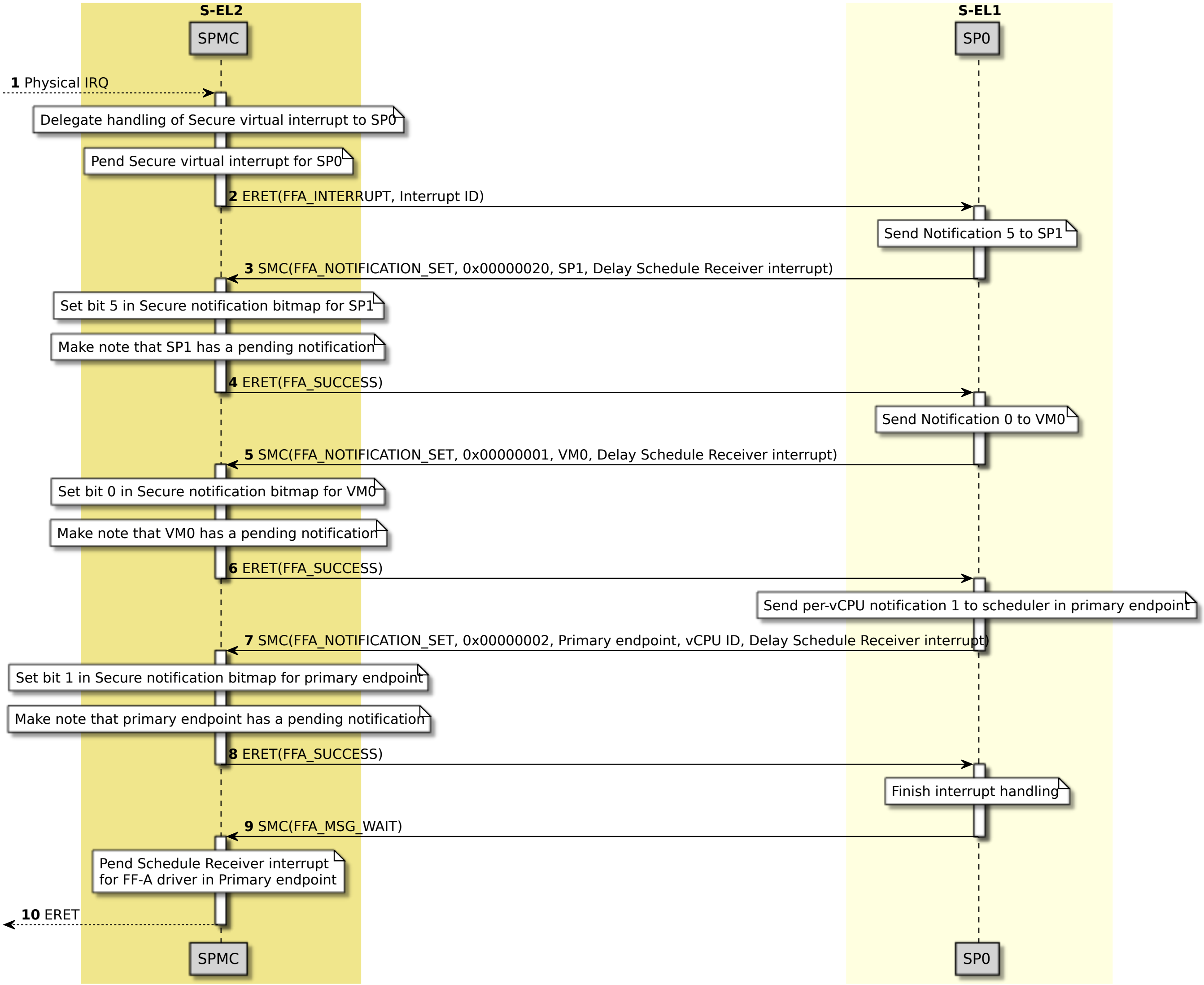

Figure 10.7: Signaling from SP0 to SP1, VM0 and its scheduler

10.5.1.2 Primary endpoint handles Schedule Receiver interrupt

Figure 10.8 illustrates an example flow where the FF-A driver in the primary endpoint handles the *Schedule Receiver interrupt*.

1. The Primary endpoint receives the Schedule Receiver interrupt.
2. The endpoint calls the FFA_NOTIFICATION_INFO_GET ABI to retrieve a list of endpoints that have pending notifications.
3. The endpoint prepares to iterate over each of the returned Endpoint IDs to invoke the corresponding schedule receiver callback.

Figure 10.9 illustrates an example flow where the SP1 driver in the primary endpoint schedules an SP1 execution context in response to the *Schedule Receiver interrupt*.

1. The Schedule Receiver callback that was previously registered for SP1 is invoked.
2. The SP1 Receiver Endpoint Driver allocates CPU cycles to SP1.
3. SP1 transitions to the Running state and receives the Notification Pending Interrupt.
4. SP1 invokes the FFA_NOTIFICATION_GET ABI to retrieve its notification bitmaps.
5. SP1 Handles its pending notifications.

Figure 10.10 illustrates an example flow where the VM0 driver in the primary endpoint schedules a VM0 execution context in response to the *Schedule Receiver interrupt*.

1. The Schedule Receiver callback that was previously registered for VM0 is invoked.
2. The VM0 Receiver Endpoint Driver allocates CPU cycles to VM0.
3. VM0 transitions to the Running state and receives the Notification Pending Interrupt.
4. VM0 invokes the FFA_NOTIFICATION_GET ABI to retrieve its notification bitmaps.
5. VM0 Handles its pending notifications.

Figure 10.11 illustrates an example flow where the SP0 driver in the primary endpoint handles the notification pended by SP0.

1. The primary endpoint Scheduler Receiver interrupt handler sees a pending notification for itself.
2. The endpoint invoked the FFA_NOTIFICATION_GET ABI to retrieve its notification bitmaps.
3. The endpoint iterates over each of its pending notifications to invoke the corresponding notification pending callback.
4. The Receiver Service Driver handles the signalled notification from SP0.

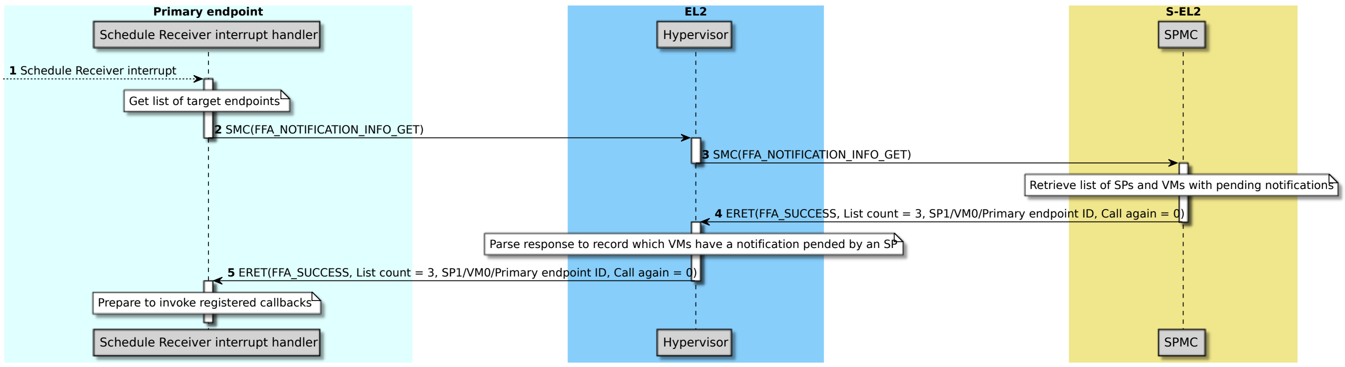

Figure 10.8: Schedule Receiver interrupt handling in primary endpoint

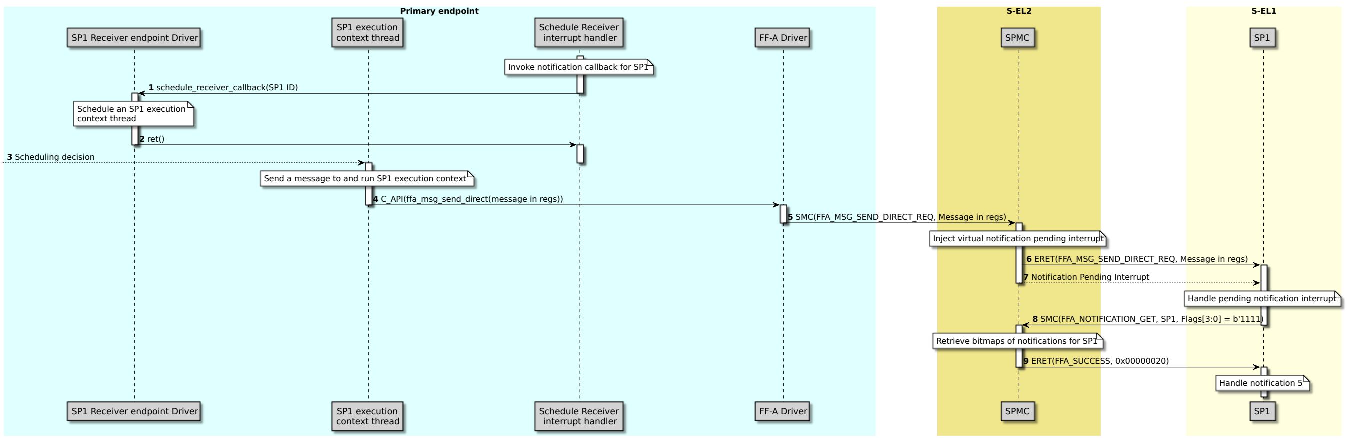

Figure 10.9: SP1 Receiver Endpoint driver in primary endpoint schedules SP1

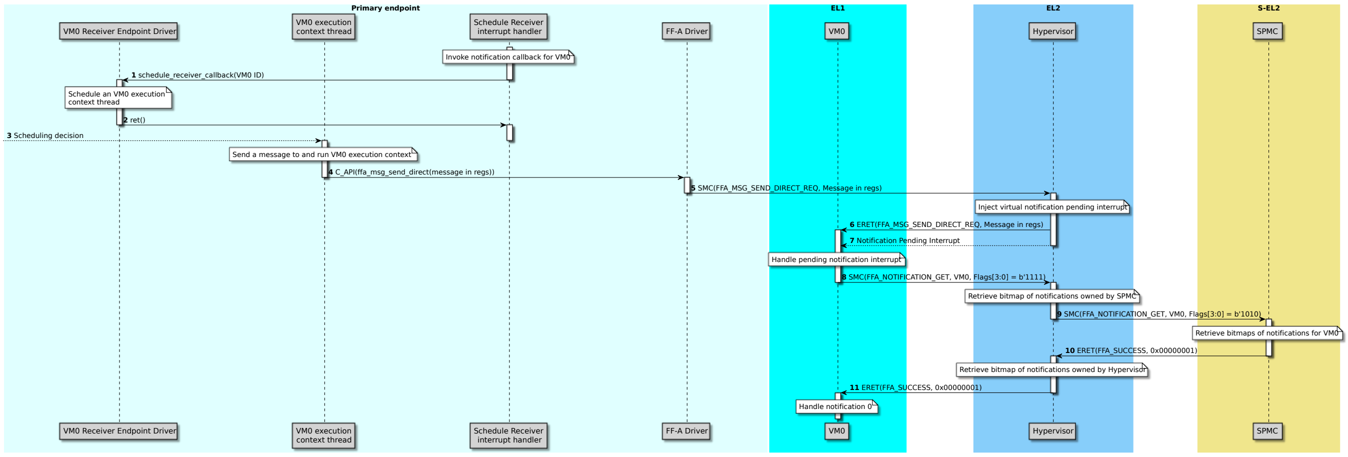

Figure 10.10: VM0 Receiver Endpoint driver in primary endpoint schedules VM0

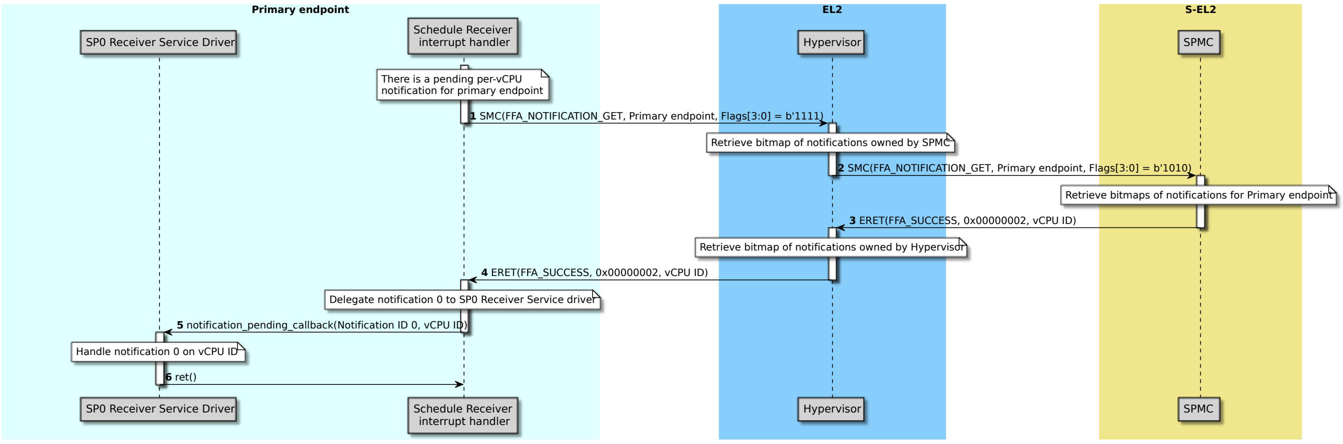

Figure 10.11: SP0 Receiver Service driver in primary endpoint receives a notification

10.5.1.3 Endpoint handles Notification Pending interrupt

Figure 10.12 illustrates an example flow where an Endpoint handles its pending notifications in response to the *Notification Pending interrupt*.

1. The endpoint Receives the Notification Pending interrupt.
2. The endpoint invokes the FFA_NOTIFICATION_GET ABI to retrieve its notification bitmaps.
3. The endpoint iterates over each of its pending notifications to invoke the corresponding notification pending callback.

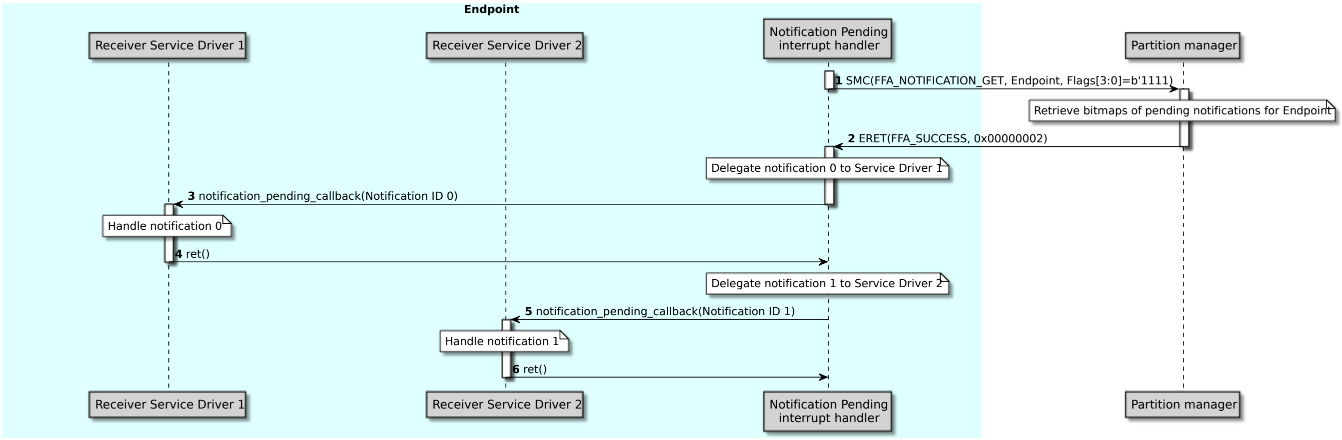

Figure 10.12: Endpoint receives Notification Pending interrupt

10.6 Notification state machine

I Figure 10.13 describes the state diagram of a notification.

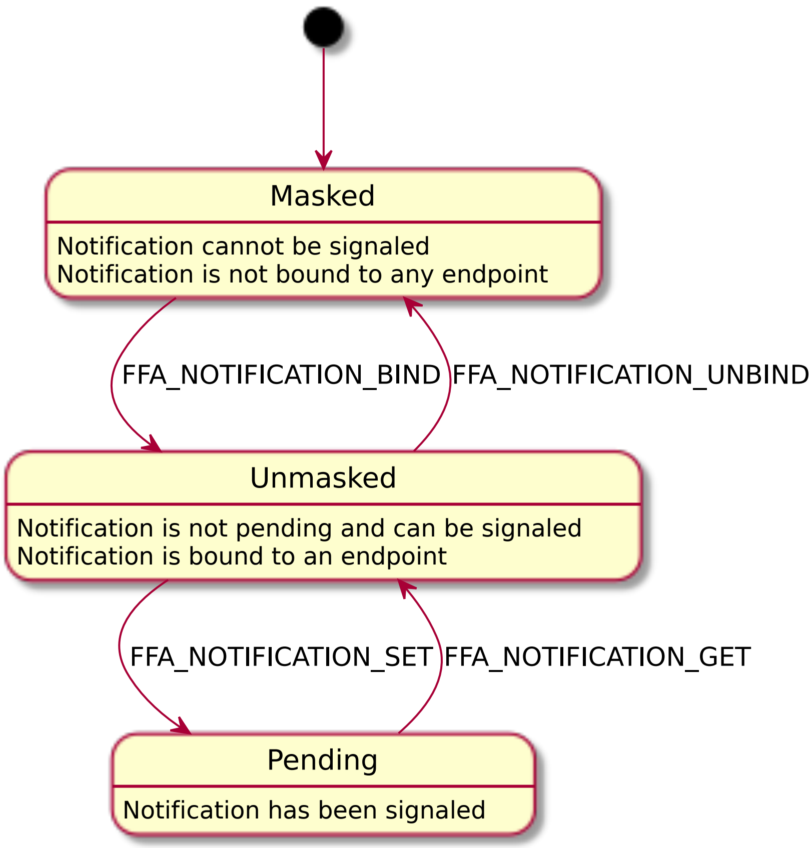

Figure 10.13: Notification state transition diagram

10.7 Compliance requirements

The following rules govern discovery of support for notifications.

1. Support for receipt of notifications is optional. If an endpoint implements this support, it specifies this in its manifest (see Chapter 5 *Setup*).
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.

   It is possible that the Hypervisor does not implement support for notifications while the SPMC and one or more SPs do. Notifications will not be delivered in this configuration since there is no recipient of the *Schedule Receiver interrupt* in the Normal world. The system integrator must ensure that notifications are supported by the Hypervisor before enabling use of this feature by the SPMC or SPs.
3. An FF-A component in the Normal world uses an FF-A discovery interface to determine if another endpoint supports receipt of notifications (see 6.2 *Partition discovery*).
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 by an endpoint, with Feature ID *0x1* or any Notification ABI, apart from FFA_NOTIFICATION_INFO_GET and FFA_NOTIFICATION_SET, 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.

The compliance requirements for an implementation of FF-A notifications at an FF-A instance are expressed as the set of ABIs and features that must be implemented at that instance. These requirements are listed in Table 10.4.

Table 10.4: Compliance requirements for FF-A notifications

Caller role	Instance	Mandatory Interface	Conduit
Sender	• Secure or NS virtual • NS physical[a] • Secure physical[b] [a] Interfaces are mandatory at this instance in the absence of an Hypervisor. [b] Interfaces are mandatory at this instance between the SPMC and a logical S-EL1 SP.	• FFA_NOTIFICATION_SET	SMC, HVC, SVC
Scheduler	• NS virtual • NS physical	• FFA_NOTIFICATION_INFO_GET	SMC, HVC
		• Handler for *Schedule receiver interrupt*.	Physical IRQ
Receiver	• Secure or NS virtual • Secure physical • Non-secure physical	• FFA_NOTIFICATION_BIND • FFA_NOTIFICATION_UNBIND • FFA_NOTIFICATION_GET	SMC, HVC, SVC

Caller role	Instance	Mandatory Interface	Conduit
		• Handler for *Notification pending interrupt*.	Virtual IRQ
Hypervisor or OS kernel [2]	• NS physical	• FFA_NOTIFICATION_BITMAP_CREATE • FFA_NOTIFICATION_BITMAP_DESTROY • FFA_NOTIFICATION_BIND • FFA_NOTIFICATION_UNBIND • FFA_NOTIFICATION_SET • FFA_NOTIFICATION_GET • FFA_NOTIFICATION_INFO_GET	SMC
S-EL2 or S-EL1 SPMC[3]	• Secure physical	• FFA_NOTIFICATION_BITMAP_CREATE • FFA_NOTIFICATION_BITMAP_DESTROY • FFA_NOTIFICATION_BIND • FFA_NOTIFICATION_UNBIND • FFA_NOTIFICATION_SET • FFA_NOTIFICATION_GET • FFA_NOTIFICATION_INFO_GET	ERET

[2] See 10.9 *Notification support without a Hypervisor*.
[3] Interface invocations from the Hypervisor are forwarded by the SPMD through the ERET conduit to the SPMC in S-EL2 or S-EL1.

10.8 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 10.8.1 *RX buffer full notification*.

10.8.1 RX buffer full notification

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

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

Also see 7.3 *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 if the Sender is a SP. It is pended by the Hypervisor if 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 10.5 *Notification signaling* and copies out the message from its RX buffer.

10.9 Notification support without a Hypervisor

Support for notifications on an Arm A-profile system without a Hypervisor is described below,

1. Only *SP notifications* and *Framework notifications* from the SPMC can be signaled to the OS Kernel. The SPMC has read-write and an SP has write-only permission on the notification bitmaps of the OS Kernel.
2. Both *SP and VM notifications* and *Framework notifications from the SPMC and Hypervisor* can be signaled to an SP from the OS Kernel.

   This is because it is not possible for the Secure world to reliably determine the presence or absence of the Hypervisor in the Normal world.
3. The OS Kernel has the same permissions on the *VM and Hypervisor's framework notification* bitmaps of an SP as the Hypervisor.
4. The bits corresponding to *VM notifications* in the notifications bitmap of the OS kernel are read-as-zero and write-ignore.
5. The bits corresponding to notifications from the Hypervisor in the framework notifications bitmap of the OS kernel are read-as-zero and write-ignore.
6. The SPMC acts as the partition manager of the OS Kernel for the purposes of,
   1. Signaling a notification to a SP.
   2. Retrieving pending notifications for the OS Kernel.
7. The OS Kernel uses *ID 0* (see Chapter 6 *Identification and Discovery*) as its endpoint ID as applicable in the notification ABIs listed in Chapter 16 *Notification interfaces*.
8. The OS Kernel uses the FFA_FEATURES ABI with the function ID of the FFA_NOTIFICATION_BITMAP_CREATE ABI to determine the absence of a Hypervisor.

   If a Hypervisor is not present, the OS Kernel is responsible for requesting the SPMC to allocate its notification bitmaps.
9. The OS Kernel uses the FFA_NOTIFICATION_BITMAP_CREATE interface to request the SPMC to allocate the SP and SPMC framework notification bitmaps during its initialization (see 16.1 *FFA_NOTIFICATION_BITMAP_CREATE*).

   Figure 10.14 illustrates notification bitmap creation for the OS Kernel.
10. The OS Kernel uses the FFA_NOTIFICATION_BITMAP_DESTROY interface to inform the SPMC prior to reset or shutdown (see 16.2 *FFA_NOTIFICATION_BITMAP_DESTROY*). The SPMC frees memory for the OS Kernel's SP and SPMC framework notification bitmaps.
11. Discovery and setup of the *Schedule Receiver interrupt* is done by the OS Kernel as the primary endpoint.
12. The *Notification pending interrupt* is a virtual interrupt and not used for signaling to the OS Kernel that it has a pending notification.
13. The OS Kernel is the Receiver endpoint for the purposes of binding and unbinding notifications.
14. The OS Kernel uses the FFA_NOTIFICATION_SET interface to signal a notification to a SP. The notification is signaled through the VM notifications bitmap of the SP.

    The SPMC pends the *Schedule Receiver interrupt* to inform the OS Kernel that one or more SPs have pending notifications and must be run.
15. An SP uses the FFA_NOTIFICATION_SET interface to signal a notification to the OS Kernel. The notification is signaled through the SP notifications bitmap of the OS Kernel.

    The SPMC pends the *Schedule Receiver interrupt* to inform the OS Kernel that it has pending notifications that must be handled when it is run next.

16. The OS Kernel uses the FFA_NOTIFICATION_INFO_GET interface to,
    1. Retrieve the list of SPs that have pending notifications and must be run.
    2. Determine if it has pending notifications.
17. The SP's scheduler in the OS Kernel uses a Direct request interface to run and inform the SP through a partition message that it has a pending notification.
18. The OS Kernel uses the FFA_NOTIFICATION_GET interface to retrieve its pending notifications and handle them.

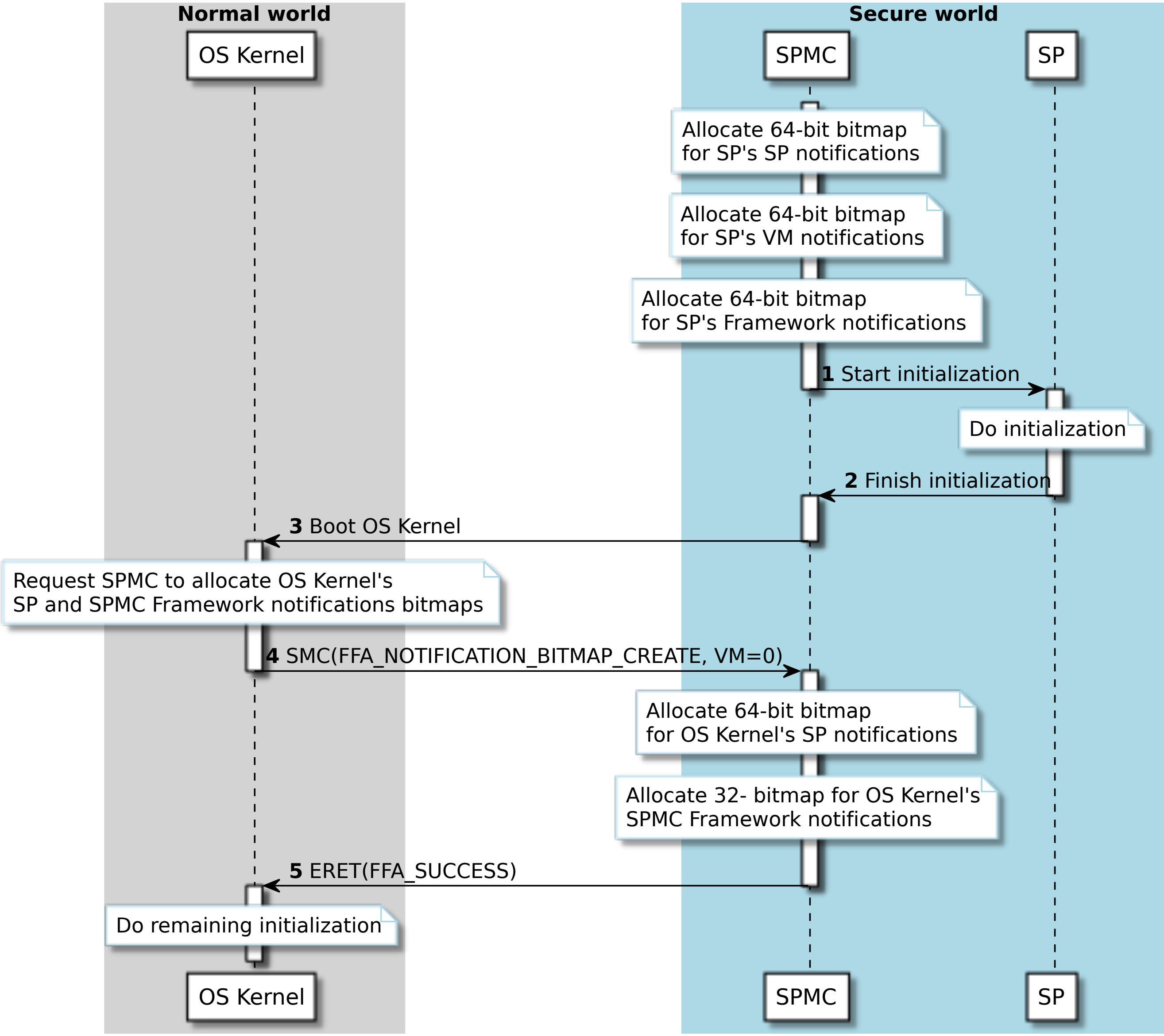

Figure 10.14: Notification bitmap creation for an OS Kernel and SP

10.10 Notification support for a Hypervisor

The guidance in this chapter assumes that endpoints are Senders and Receivers of notifications. It is also possible for the SPMC to signal Framework notifications to a Hypervisor. This mechanism is enabled by the fact that the endpoint ID 0 is assigned to the Hypervisor (see 6.3 *Partition manager identification*).

The Hypervisor can invoke the FFA_NOTIFICATION_BITMAP_CREATE interface with a VM ID of 0 to request the SPMC to allocate the SP and SPMC framework notification bitmaps. A Framework notification pended by the SPMC in the corresponding bitmap for VM ID 0 will be received by the Hypervisor.

It is IMPLEMENTATION DEFINED whether the Hypervisor allows an SP or VM to pend notifications via the endpoint notification bitmaps. For example, the Host operating system at EL1 in a Type-2 Hypervisor implements a subset of the Hypervisor functionality and fulfils the role of an endpoint. It can use the endpoint notification bitmaps to receive IMPLEMENTATION DEFINED notifications from any SP or VM just like any other endpoint.

Chapter 11
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 12 *Status reporting interfaces*.
- Interfaces for partition setup and discovery using Framework messages are described in Chapter 13 *Setup and discovery interfaces*.
- Interfaces to manage CPU cycles allocated to an endpoint are described in Chapter 14 *CPU cycle management interfaces*.
- Interfaces to implement exchange of Direct and Indirect Partition messages between endpoints are described in Chapter 15 *Messaging interfaces*.
- Additional interfaces for interfaces pertaining to power management are described in Chapter 18 *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 4.4 *Conduits*.
2. Each interface relies on the SMC calling convention v1.2 described in [5]. The divergences from the calling convention are described in 11.1 *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 [5].
    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 FF-A ABI could support both the SMC32 and SMC64 conventions e.g. FFA_RXTX_MAP, FFA_NOTIFICATION_INFO_GET. A callee that runs in the AArch64 execution state and implements such an ABI must implement both SMC32 and SMC64 conventions of the ABI.

6. 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 13.3 *FFA_FEATURES*) to discover if an FF-A ABI is implemented by the FF-A component at the higher EL.

11.1 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. The interpretation of *bit[31]* of the *Function ID* parameter by the Framework depends upon the FF-A ABI. For example, *hypcalls* generally behave as *fast calls*. FF-A ABIs that allocate CPU cycles to a partition generally behave as yielding calls.

11.2 Reserved parameter convention

The SMCCC refers to the documentation of each SMC or HVC call to determine if parameter registers in that call are used or preserved. Unused parameter registers in FF-A ABIs are reserved for future use by the Framework.

In an invocation of an ABI via the SMC, HVC or SVC conduit, the callee treats the unused parameter registers as Reserved (SBZ). The caller is expected to write zeroes to these registers. The callee ignores the values in these registers.

The ERET conduit is used in the following scenarios:

- To complete the invocation of a hypcall. The parameter registers contain return results. E.g. FFA_ID_GET is invoked via the SMC conduit at the Non-secure physical FF-A instance. The invocation completes via the ERET conduit.
- To invoke a new ABI that is independent of the ABI that was previously invoked via the SMC, HVC or SVC conduits. The parameter registers contain input arguments of the new ABI. E.g. An SP invokes FFA_MSG_WAIT to enter the waiting state. The SPMC invokes FFA_MSG_SEND_DIRECT_REQ with the ERET conduit to transition the SP to the running state.

In both scenarios, the caller of the ERET instruction treats all unused parameter registers as Reserved (MBZ). This has the following implications:

- Information from a higher Exception level never leaks to a lower Exception level in ABI invocations.
- Values specified by a lower Exception level in the last invocation of the SMC, HVC or SVC conduits are not preserved.

Chapter 12
Status reporting interfaces

12.1 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 13 *Setup and discovery interfaces*.
- Interfaces to implement memory management transactions in the FF-A memory management protocol [1].
- Interfaces to manage CPU cycles in Chapter 14 *CPU cycle management interfaces*.
- Interfaces to implement messaging between endpoints in Chapter 15 *Messaging interfaces*.

[1] The *FFA_VERSION* interface (see 13.2 *FFA_VERSION*) is used for discovering the presence of a Framework implementation. It does not use the status reporting interfaces.

12.2 FFA_ERROR

Description

- Returns error code in response to a previous invocation of an FF-A function.
- Table 12.2 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 12.3.
- Syntax of this function is described in Table 12.4.
- Figure 12.1 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.

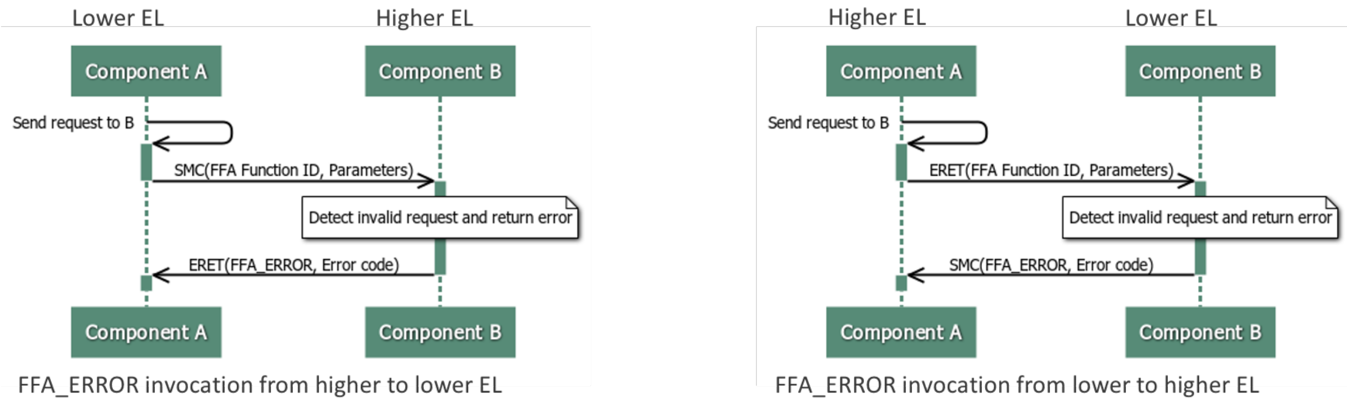

Figure 12.1: Example usage of FFA_ERROR

Table 12.2: Error status codes

Status code	Description
-1	NOT_SUPPORTED
-2	INVALID_PARAMETERS
-3	NO_MEMORY
-4	BUSY
-5	INTERRUPTED
-6	DENIED
-7	RETRY
-8	ABORTED

Status code	Description
-9	NO_DATA
-10	NOT_READY

Table 12.3: 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 12.4: 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-x17	• Reserved (SBZ).

12.3 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 12.6.
- Syntax of this function is described in Table 12.7.
- Figure 12.2 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.

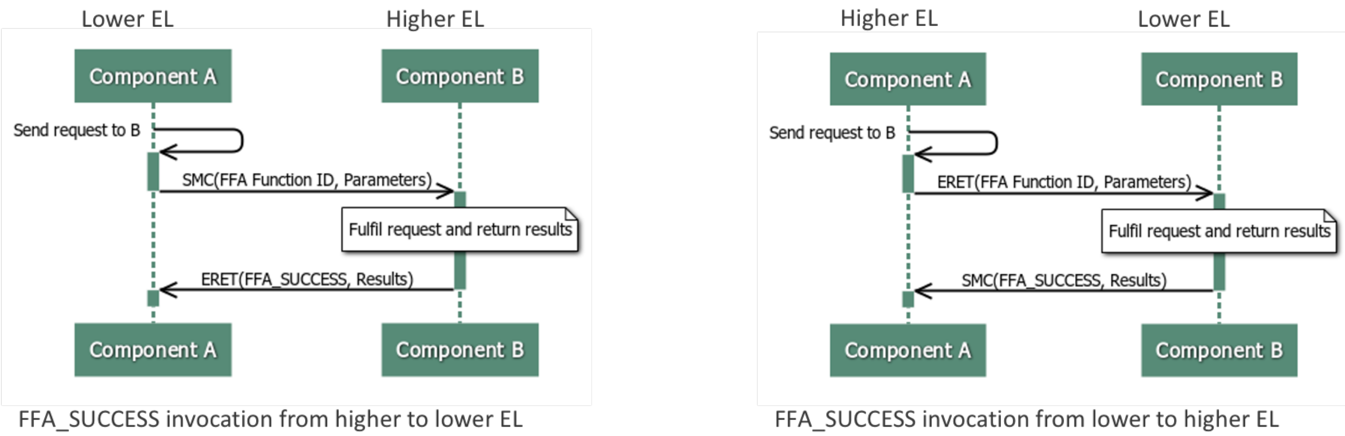

Figure 12.2: Example usage of FFA_SUCCESS

Table 12.6: 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 12.7: FFA_SUCCESS function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000061. • 0xC4000061. – This function ID, also denoted as FFA_SUCCESS64, is used only if any result register encodes a 64-bit parameter.
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-x17	• FF-A function specific return results. See function definition for result encoding. Reserved (SBZ) if not explicitly specified.

12.4 FFA_INTERRUPT

Description
• Returns control from the caller to the callee in response to an interrupt. See 12.4.1 *Usage* for details. • Valid FF-A instances and conduits are listed in Table 12.9. • Syntax of this function is described in Table 12.10.

Table 12.9: 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, ERET

Table 12.10: FFA_INTERRUPT function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000062.
uint32 Endpoint/vCPU IDs	w1	• Endpoint and vCPU IDs of the caller in a valid usage scenario described in 12.4.1 *Usage*. MBZ otherwise. – Bits[31:16]: Endpoint ID. – Bits[15:0]: vCPU ID.
uint32 Interrupt ID	w2	• Interrupt ID in a valid usage scenario described in 12.4.1 *Usage*. MBZ otherwise.
Other Parameter registers	w3-w7 x3-x17	• Reserved (SBZ).

12.4.1 Usage

1. FFA_INTERRUPT is used with the SMC conduit at the Secure physical FF-A instance only by the S-EL1 or S-EL2 SPMC to inform the SPMD that a Non-secure interrupt has preempted execution of an SP and the NS-Endpoint on this physical PE must be resumed so that the interrupt can be handled.
    1. The SPMC returns the ID of the SP and its execution context that was doing work on behalf of the NS-Endpoint. The NS-Endpoint is expected to resume execution of the SP execution context once the interrupt is handled.
    2. The interrupt ID field is invalid and MBZ. It must be ignored by the SPMD.
    3. An example of this usage is illustrated in Figure 9.3.
2. FFA_INTERRUPT is used with the ERET conduit by the Hypervisor, SPMC or the SPMD to inform a callee in the *blocked* runtime state that its request was preempted by an interrupt.
    1. The callee had entered the *blocked* runtime state after requesting a partition to do work on its behalf e.g through an invocation of a Direct request interface.

2. The partition manager returns the ID of the partition and its vCPU or execution context that was doing work on behalf of the callee. The callee is expected to resume execution of the partition vCPU or execution context once the interrupt is handled. This is applicable only if the callee runs in a privileged exception level i.e. not in S-EL0. Also see 9.2.1 *Secure interrupt signaling mechanisms*.
3. The interrupt ID field is invalid and MBZ. It must be ignored by the callee.
4. Valid caller and callee combinations at specific FF-A instances are listed below.
    1. SPMD and an NS-Endpoint at the Non-secure physical FF-A instance.
    2. Hypervisor and a VM at the Non-secure virtual FF-A instance.
    3. SPMC and an SP at the Secure virtual FF-A instance.
5. An example of this usage is illustrated in Figure 9.3.

3. FFA_INTERRUPT is used with the ERET conduit by the Hypervisor, SPMC or the SPMD to delegate interrupt handling to a callee in the *waiting* runtime state.
    1. The callee had entered the *waiting* runtime state after finishing work requested by another partition e.g through an invocation of FFA_MSG_SEND_DIRECT_RESP.
    2. The partition manager returns the ID of the pending interrupt to the callee. This is applicable only if the callee is a partition that runs in a privileged exception level i.e. not in S-EL0. It is also not applicable if the callee is an SPMC and the caller is the SPMD. This is because the SPMD is not expected to determine the identity of the pending interrupt by querying the GIC. Also see 9.2.1 *Secure interrupt signaling mechanisms*.
    3. The endpoint and vCPU ID fields are invalid and MBZ. They must be ignored by the callee.
    4. Valid caller and callee combinations at specific FF-A instances are listed below.
        1. SPMD and the S-EL1 or S-EL2 SPMC at the Secure physical FF-A instance.
        2. EL3 SPMC and a logical S-EL1 SP at the Secure physical FF-A instance.
        3. Hypervisor and a VM at the Non-secure virtual FF-A instance.
        4. SPMC and an SP at the Secure virtual FF-A instance.

Chapter 13
Setup and discovery interfaces

13.1 Compliance requirements

Table 13.1 lists the discovery and setup interfaces that must be implemented at a given FF-A instance with a specific conduit.

1. Combinations of interface and conduit that are not listed in the table but listed in the corresponding interface description are optional.
2. Combinations of interface and conduit that are neither listed in Table 13.1 nor in the corresponding interface description are not supported.

Table 13.1: Mandatory discovery and setup interfaces

Interface	Conduit	Mandatory FF-A Instance	Notes
FFA_VERSION	SMC, HVC, SVC	• All	
FFA_FEATURES	SMC, HVC, SVC	• All	
FFA_FEATURES	ERET	• Secure physical instance	• Mandatory at this instance between the SPMD and a S-EL2 or S-EL1 SPMC.
FFA_RX_RELEASE	SMC, HVC, SVC	• All	
FFA_RX_RELEASE	ERET	• Secure physical instance	• Mandatory at this instance between the SPMD and a S-EL2 or S-EL1 SPMC.
FFA_RXTX_MAP	SMC, HVC, SVC	• All except Secure physical instance	• Optional at this instance between the SPMD and a S-EL2 or S-EL1 SPMC.
FFA_RXTX_MAP	ERET	• Secure physical instance	• Mandatory at this instance between the SPMD and a S-EL2 or S-EL1 SPMC.
FFA_RXTX_UNMAP	SMC, HVC, SVC	• All except Secure physical instance	• Optional at this instance between the SPMD and a S-EL2 or S-EL1 SPMC.
FFA_PARTITION_INFO_GET	SMC, HVC, SVC	• All except Secure physical instance	• Optional at this instance between the SPMD and a S-EL2 or S-EL1 SPMC.
FFA_PARTITION_INFO_GET	ERET	• Secure physical instance	• Mandatory at this instance between the SPMD and a S-EL2 or S-EL1 SPMC.

Interface	Conduit	Mandatory FF-A Instance	Notes
FFA_PARTITION_INFO_GET_REGS	SMC	• Secure physical instance	• Mandatory at this instance between the SPMD and a S-EL2 or S-EL1 SPMC only if EL3 LSPs are implemented and not using an IMPLEMENTATION DEFINED discovery | mechanism.
FFA_PARTITION_INFO_GET_REGS	ERET	• Secure physical instance	• Mandatory at this instance between the SPMD and a S-EL2 or S-EL1 SPMC only if EL3 LSPs are implemented and not using an IMPLEMENTATION DEFINED discovery | mechanism.
FFA_ID_GET	SMC, HVC, SVC	• All	
FFA_SPM_ID_GET	SMC, HVC, SVC	• All	

13.2 FFA_VERSION

Description
• Returns version of the Firmware Framework implementation at an FF-A instance as described in 13.2.1 *Overview*. • Valid FF-A instances and conduits are listed in Table 13.3. • Syntax of this function is described in Table 13.4. • Encoding of a version number in return parameters is described in Table 13.5. • Encoding of error codes in return parameters is described in Table 13.6.

Table 13.3: 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 13.4: 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-x17	• Reserved (SBZ).

Table 13.5: Encoding of a version number

Parameter	Register	Value
int32 Output version number	w0	• Supported version number (see 13.2.2 *Usage*). – Bit[31]: Must be 0. – Bits[30:16]: Major Version number. – Bits[15:0]: Minor Version number.
Other Result registers	w1-w7 x1-x17	• Reserved (MBZ).

Table 13.6: Encoding of error codes

Parameter	Register	Value
int32 Error code	w0	• NOT_SUPPORTED: A Firmware Framework implementation of the requested version does not exist at this FF-A instance.

13.2.1 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 $x \mathrel{!=} a$, 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 $x == a$.

13.2.2 Usage

This function enables the caller to determine if the callee implements the Firmware Framework and the version number of the implementation. Conversely, it enables the callee to determine the version number of the Framework that the caller implements. This negotiation of version numbers enables both the caller and callee at an FF-A instance to determine if they are compatible. If they are compatible, it enables them to determine which Framework functionalities can be used. Hence, negotiation of the version must happen before an invocation of any other FF-A ABI. The responsibilities of the caller and callee in an invocation of this ABI are listed below:

- 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.

  It is strongly recommended that the callee returns the version number instead of the NOT_SUPPORTED error code. Also see,

  * 18.5 *Changes to FF-A v1.0 data structures for forward compatibility*.
  * 18.5.3 *Compatibility requirements for FF-A v1.0 data structures*.

- 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 2.

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.

Once the caller invokes any FF-A ABI other than *FFA_VERSION*, the version negotiation phase is complete.

Once an FF-A version has been negotiated between a caller and a callee, the version may not be changed for the lifetime of the calling component. The callee must treat the negotiated version as the only supported version for any subsequent interactions with the caller.

13.2.3 SPM usage

13.2.3.1 Version discovery between SPMD and SPMC

In SPM configurations where the SPMD and SPMC reside in separate Exception levels (see Table 4.1), 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 e.g. through the SPMC manifest (see 5.2.2 *SPMC manifest*).
- 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 version that fulfills the below requirements in response to an invocation of FFA_VERSION function at the Secure physical FF-A instance.
    * The highest major version supported by both the SPMC and SPMD.
    * The highest minor version for this major version supported by both the SPMC and SPMD.

13.2.3.2 Version discovery between Normal world and SPMC

The Hypervisor or OS Kernel invoke the FFA_VERSION function at the Non-secure physical FF-A instance to specify the version of the Framework to the SPMD. In this case,

1. If the highest version of the Framework implemented by the SPMC is major version 1 and minor version 0, the SPMD must return this version to the Normal world.
2. If the SPMC implements this version of the Framework, the SPMD must forward the FFA_VERSION function invocation to the SPMC through,

1. The message described in Table 13.7 if the SPMC is implemented in S-EL2 or S-EL1.
2. An IMPLEMENTATION DEFINED mechanism if the SPMC is implemented in EL3.

It is possible that the SPMC implements multiple versions of the Framework such that each version is less than the common version negotiated between the SPMC and the SPMD. The SPMC needs to know the version presented by the Hypervisor or OS Kernel to determine which version it should use to maintain compatibility.

The forwarded message from the SPMD enables the SPMC to make this choice and either return a compatible version or return the NOT_SUPPORTED error code to the Normal world as per the callee specific guidelines described in 13.2.2 *Usage*. The SPMC must return the response to the forwarded message through,

1. The message described in Table 13.8 if the SPMC is implemented in S-EL2 or S-EL1.
2. An IMPLEMENTATION DEFINED mechanism if the SPMC is implemented in EL3.

The SPMD must forward the return value in *w0* in the response message to the Hypervisor or OS as the return value of the original FFA_VERSION call. Figure 13.1 illustrates how the SPMD forwards,

1. An FFA_VERSION call from the Normal world to an SPMC in S-EL2 or EL3.
2. The response from the SPMC back to the Normal world.

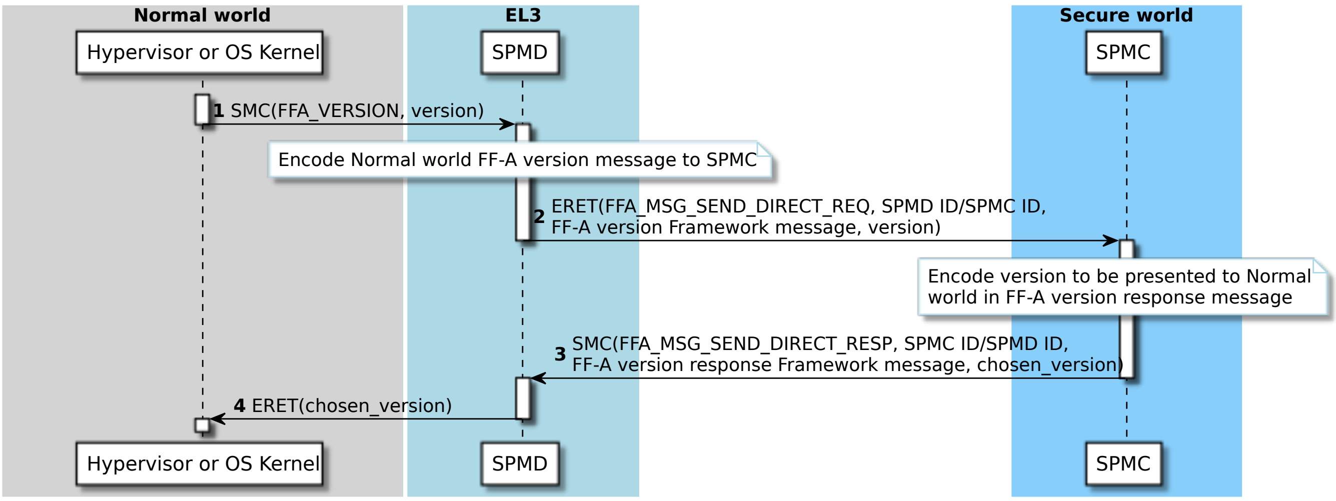

Figure 13.1: Forwarding of FFA_VERSION call from SPMD to SPMC at lower EL

Table 13.7: Normal world FF-A version message

Register	Parameter
w0	0x8400006F: FFA_MSG_SEND_DIRECT_REQ Function ID
w1	• Sender and Receiver endpoint IDs. – Bit[31:16]: ∗ SPMD ID. – Bit[15:0]: ∗ SPMC ID.
w2	• Message flags. – Bit[31] = b'1: Framework message. – Bit[30:8]: Reserved (SBZ). – Bit[7:0] = b'00001000: Message for forwarding FFA_VERSION call from Normal world to the SPMC.

Register	Parameter
w3	• FFA version from Normal world.
w4-w7	Reserved (SBZ).

Table 13.8: Response to Normal world FF-A version message

Register	Parameter
w0	0x84000070: FFA_MSG_SEND_DIRECT_RESP Function ID
w1	• Sender and Receiver endpoint IDs. – Bit[31:16]: ∗ SPMC ID. – Bit[15:0]: ∗ SPMD ID.
w2	• Message flags. – Bit[31] = b'1: Framework message. – Bit[30:8]: Reserved (SBZ). – Bit[7:0] = b'00001001: Response message to forwarded FFA_VERSION call from the Normal world.
w3	• Encoding of *w0* in Table 13.5 if a version is returned by the SPMC. • NOT_SUPPORTED if a version is not returned by the SPMC.
w4-w7	Reserved (SBZ).

13.3 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 13.10.
- Syntax of this function is described in Table 13.11.
- 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 13.12.
- 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 13.10: 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 13.11: FFA_FEATURES function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000064.
uint32 FF-A function ID or Feature ID	w1	• Bit[31]: *w1* contains an FF-A Function ID or Feature ID. – b'1: *w1* must be interpreted as the Function ID of the FF-A interface whose implementation is being queried. Effectively, bit[31] of the SMCCC Function ID i.e. the Fastcall bit is used to distinguish between an FF-A feature and function ID. ∗ If an interface defines both SMC32 and SMC64 FIDs, then either FID could be used. (Also see common rules that govern definition and behavior of FF-A ABIs in Chapter 11 *Interface overview*). – b'0: *w1* must be interpreted as the ID of a feature supported by the Framework at this FF-A instance. IDs of supported features are listed in Table 13.13. ∗ Bit[30:8]: Reserved (MBZ). ∗ Bit[7:0]: Feature ID.

Parameter	Register	Value
uint32 Input properties	w2	• A list of properties expected by the caller corresponding to the Function ID specified in *w1*. • This parameter is Reserved (MBZ) if a Feature ID is specified in *w1*.
Other Parameter registers	w3-w7 x3-x17	• Reserved (SBZ).

Table 13.12: 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 13.14. – Feature IDs and encodings of their properties are listed in Table 13.13. • MBZ if no optional features are implemented or no implementation details are exported by the queried interface.
Other Result registers	w4-w7 x4-x17	• Reserved (MBZ).

Table 13.13: 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 13.14: Encoding of interface properties parameters

FF-A Function ID (w1)	Input properties (w2)	Return parameters (w2-w3)
FFA_RXTX_MAP	Reserved (SBZ).	• *w2*: Buffer sizes. – Bit[1:0]: Minimum buffer size and alignment boundary (see 7.2.2.3 *Buffer attributes*). ∗ b'00: 4K. ∗ b'01: 64K. ∗ b'10: 16K. ∗ b'11: Reserved. – Bit[15:2]: Reserved (MBZ). – Bit[31:16]: Maximum buffer size in number of pages (see 7.2.2.3 *Buffer attributes*). ∗ MBZ if there is no size limit. • *w3/x3* : Reserved (MBZ).

13.4 FFA_RX_ACQUIRE

Description

- Acquire ownership of a RX buffer before writing a message to it (see 7.2.2.4.3 *Management of buffer ownership between Hypervisor and SPMC*).
- Valid FF-A instances and conduits are listed in Table 13.16.
- Syntax of this function is described in Table 13.17.
- Returns FFA_SUCCESS without any further parameters on successful completion.
- Encoding of error code in the FFA_ERROR function is described in Table 13.18.

Table 13.16: FFA_RX_ACQUIRE instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure Physical	SMC
2	Secure Physical	ERET

Table 13.17: FFA_RX_ACQUIRE function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000084.
uint32 VM ID	w1	• ID of VM ownership of whose RX buffer should be acquired. – Bit[31:16]: Reserved (SBZ). – Bit[15:0]: VM ID.
Other Parameter registers	w2-w7 x2-x17	• Reserved (SBZ).

Table 13.18: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• DENIED: Callee cannot relinquish ownership of the RX buffer. • INVALID_PARAMETERS: There is no buffer pair registered on behalf of the VM. • NOT_SUPPORTED: This function is not implemented at this FF-A instance.

13.5 FFA_RX_RELEASE

Description
• Relinquish ownership of a RX buffer after reading a message from it (see 7.2.2.4 *Buffer synchronization*). • Valid FF-A instances and conduits are listed in Table 13.20. • Syntax of this function is described in Table 13.21. • Returns FFA_SUCCESS without any further parameters on successful completion. • Encoding of error code in the FFA_ERROR function is described in Table 13.22.

Table 13.20: 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 13.21: FFA_RX_RELEASE function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000065.
uint32 VM ID	w1	• ID of VM ownership of whose RX buffer should be released. Only valid at the Non-secure physical FF-A instance. MBZ otherwise. – Bit[31:16]: Reserved (SBZ). – Bit[15:0]: VM ID.
Other Parameter registers	w2-w7 x2-x17	• Reserved (SBZ).

Table 13.22: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• DENIED: Caller did not have ownership of the RX buffer. • INVALID_PARAMETERS: There is no buffer pair registered by the Hypervisor on behalf of the VM. • NOT_SUPPORTED: This function is not implemented at this FF-A instance.

13.6 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 13.27.
  - 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 13.24.
- Syntax of this function is described in Table 13.25.
- Returns FFA_SUCCESS without any further parameters on successful completion.
- Encoding of error code in the FFA_ERROR function is described in Table 13.26.

Table 13.24: 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 13.25: 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 (SBZ).
		• Bit[5:0]: Number of contiguous 4K pages allocated for each buffer.
		  – 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.

Other Parameter registers	w4-w7 x4-x17	• Reserved (SBZ).

Table 13.26: FFA_ERROR encoding

Parameter	Register	Value

int32 Error code	w2	• INVALID_PARAMETERS:
		  – One or more fields in input parameters is incorrectly encoded.
		  – Invalid number of pages specified (see 13.3 *FFA_FEATURES*).
		• 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.
		  – A VM's buffer pair cannot be registered by the SPMC since no SP sends or receives Indirect messages.
		• NOT_SUPPORTED: This function is not implemented at this FF-A instance.

Table 13.27: Endpoint RX/TX descriptor[1]

Field	Byte length	Byte offset	Description
Endpoint ID	2	0	• ID of endpoint that allocated the RX/TX buffer.
Reserved	2	2	• Reserved (SBZ).
RX buffer memory region description offset	4	4	• 8-byte aligned offset from the base address of this descriptor to the *Composite memory region descriptor* that describes the RX buffer memory region. (See [1] for more information).
TX buffer memory region description offset	4	8	• 8-byte aligned offset from the base address of this descriptor to the *Composite memory region descriptor* that describes the TX buffer memory region. (See [1] for more information).

13.7 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 13.29.
- Syntax of this function is described in Table 13.30.
- Returns FFA_SUCCESS without any further parameters on successful completion.
- Encoding of error code in the FFA_ERROR function is described in Table 13.31.

Table 13.29: 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 13.30: FFA_RXTX_UNMAP function syntax

Parameter	Register	Value
uint32 Function ID	w0/x0	• 0x84000067.
uint32 ID	w1	• ID of VM that allocated the RX/TX buffer. Only valid at the Non-secure physical FF-A instance. MBZ otherwise. – Bit[31:16]: ID. – Bit[15:0]: Reserved (SBZ).
Other Parameter registers	w2-w7 x2-x17	• Reserved (SBZ).

Table 13.31: 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.

13.8 FFA_PARTITION_INFO_GET

Description

- Returns information about FF-A components implemented in the system as described in 13.8.1 *Overview*.
- Valid FF-A instances and conduits are listed in Table 13.33.
- Syntax of this function is described in Table 13.34.
- Encoding of result parameters in the FFA_SUCCESS function is described in Table 13.35.
- Encoding of error code in the FFA_ERROR function is described in Table 13.36.

Table 13.33: 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 13.34: 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 [5].
uint32 Flags	w5	• Bit[0]: Return information type flag. – b'1: Return the count of partitions deployed in the system corresponding to the specified UUID in *w2* as specified in Table 13.35. – b'0: Return partition information descriptors corresponding to the specified UUID in the RX buffer of the caller (see Table 6.1). • Bit[31:1]: Reserved (SBZ).
Other Parameter registers	w6-w7 x6-x17	• Reserved (SBZ).

Table 13.35: FFA_SUCCESS encoding

Parameter	Register	Value
uint32 Count	w2	• If *Bit[0] = b'0* in the *Flags* input parameter, this field contains the count of partition information descriptors corresponding to the specified *UUID* in *w1-w4*. The descriptors are populated in the RX buffer of the caller. • If *Bit[0] = b'1* in the *Flags* input parameter, this field contains the count of partitions deployed in the system corresponding to the specified *UUID* in *w1-w4*.
uint32 Size	w3	• If *Bit[0] = b'0* in the *Flags* input parameter, this field contains the size of each partition information descriptor populated in the RX buffer of the caller. • If *Bit[0] = b'1* in the *Flags* input parameter, this field MBZ.
Other Result registers	w4-w7 x4-x17	• Reserved (SBZ).

Table 13.36: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• BUSY: RX buffer of the caller is required to return partition information but is either not free or not mapped. • 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. • NOT_READY: Callee is not ready to handle this request.

13.8.1 Overview

FFA_PARTITION_INFO_GET is used by FF-A components for the following purposes.

- To discover the ID (see Chapter 6 *Identification and Discovery*) and other properties of partitions. This information is,
  - Requested by specifying a UUID as an input parameter as described in Table 13.34.
    - The *Return information type flag* is set to *b'0* to indicate that partition properties are being requested.
  - Encoded in a *partition information descriptor* as described in Table 6.1.
  - 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 13.35).
- To discover the count of partitions of a particular type. This information is,
  - Requested by specifying a UUID as an input parameter as described in Table 13.34.

    * The *Return information type flag* is set to *b'1* to indicate that a partition count is being requested.
  - Returned in *w2* (see Table 13.35).

13.8.2 Usage

The result of an invocation of this ABI depends upon the version of the Framework, specified UUID, Flags parameter and the FF-A instance where the ABI is invoked. This is described in 6.2.2 *Partition discovery ABI usage*.

The caller transfers ownership of the RX buffer back to the producer through a mechanism described in 7.2.2.4.2 *Transfer of buffer ownership*.

13.9 FFA_PARTITION_INFO_GET_REGS

Description

- Returns information about FF-A components implemented in the system as described in 13.9.1 *Overview*.
- Valid FF-A instances and conduits are listed in Table 13.38.
- Syntax of this function is described in Table 13.39.
- Encoding of result parameters in the FFA_SUCCESS function is described in Table 13.40.
- Encoding of error code in the FFA_ERROR function is described in Table 13.41.

Table 13.38: FFA_PARTITION_INFO_GET_REGS instances and conduits

Config	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 13.39: FFA_PARTITION_INFO_GET_REGS function syntax

Parameter	Register	Value
uint32 Function ID	w0	0xC400008B
uint64 UUID_Lo	x1	Bytes[7:0] of the UUID with byte 0 in the low-order bits.
uint64 UUID_Hi	x2	Bytes[15:8] of the UUID with byte 8 in the low-order bits.
uint32 Start index and tag	x3	• Bits[15:0]: Start index. – Index into an array of partition information maintained by the callee from where information must be returned. ∗ E.g. if there are 10 partitions corresponding to a UUID, this value could be any number in the range (0-9). • Bits[31:16]: Information tag for the queried UUID. – MBZ if Start Index = 0. – Information tag known to the caller if *Start Index > 0*. • Bits[63:32]: Reserved (SBZ).
Other Parameter registers	x4-x17	• Reserved (SBZ).

Table 13.40: FFA_SUCCESS encoding

Parameter	Register	Value
uint32 Function ID	w0	FFA_SUCCESS64
uint64 Information metadata	x2	• Bits[15:0]: Last index – Maximum index of array of partition information maintained by the callee relative to the Start index specified by the caller. ∗ E.g. if there are 10 partitions corresponding to a UUID, this value will be 9. – Total number of entries = *Last index + 1* • Bits[31:16]: Current Index. – Maximum index of array of partition information returned by the callee relative to the Start index specified by the caller. ∗ E.g. If there are 10 partitions corresponding to a UUID, each invocation of this ABI returns 5 array entries and the caller specifies 0 as the start index, this value would be 4. – Number of entries returned by the callee in this ABI invocation = *(Current index - Start index)* + *1* • Bits[47:32]: – Information tag for the queried UUID known to the callee. • Bits[63:48]: – Size in bytes of each partition information entry descriptor.

Parameter	Register	Value

uint64 Partition information	x3-x17	• Partition information descriptor.
		  – Size of each descriptor is 24 bytes as per the FF-A v1.1 spec.
		  – Each descriptor is encoded in 3 registers in little endian format as described below.
		• Registers *N=3,6,9,12,15*
		  – *xN*
		    ∗ Bits[15:0]: 16-bit ID of the partition, stream or auxiliary endpoint.
		    ∗ Bits[31:16]: Number of execution contexts implemented by this partition (also see 4.7 *Execution context*).
		    ∗ Bits[64:32]: Partition properties as encoded in Table 6.2
		  – *xN + 1*
		    ∗ MBZ if a non-Nil UUID is specified as input.
		    ∗ If the Nil UUID is specified as input.
		      · Bytes[0. . . 7] of the UUID with byte 0 in the low-order bits.
		  – *xN* + 2
		    ∗ MBZ if a non-Nil UUID is specified as input.
		    ∗ If the Nil UUID is specified as input.
		      · Bytes[8. . . 15] of the UUID with byte 0 in the low-order bits.

Table 13.41: Encoding of return codes

Parameter	Register	Value
uint32 Function ID	w0	FFA_ERROR
int32 Error code	w2	• INVALID_PARAMETERS: Unrecognized UUID or start index. • NOT_SUPPORTED: This function is not implemented at this FF-A instance. • DENIED: Callee is not in a state to handle this request. • RETRY: The provided tag is not valid. • NOT_READY: Callee is not ready to handle this request.

13.9.1 Overview

FFA_PARTITION_INFO_GET_REGS can be used by FF-A components for the following purposes.

- To discover the ID (see Chapter 6 *Identification and Discovery*) and other properties of partitions. This information is,
  - Requested by specifying a UUID as an input parameter as described in Table 13.39.
  - Returned in registers as encoded in a *partition information descriptor* as described in Table 13.40.

13.9.2 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 in 6.2.2 *Partition discovery ABI usage*.

To cater for the scenario where the full list of descriptors does not fit in a single invocation, the callee exports an abstraction to the caller in which the partition information corresponding to the queried UUID is organized in an array. If the array cannot be returned in a single response to the caller, the callee must encode an IMPLEMENTATION DEFINED tag as part of the response that is used to identify the version of the information being provided as part of the call. The mechanism is described as follows,

1. The caller starts retrieval of partition information by specifying the start index 0 of the array.
2. The callee returns the last index of the array to inform the caller about the number of entries in the array. Total number of entries = *last index + 1*.
3. The callee also returns the current index which identifies the last array entry that could fit in the returned partition information.
4. The number of entries returned in a single invocation = *(current index - start index) + 1*.
5. If all the partition information cannot fit into the available register space, *current index < last index*. The caller invokes the ABI again with *start index = current index + 1*.
6. All partition information entries in the array have been returned when *last index == current index*.
   - *last index >= start index*.
   - *current index >= start index*.
7. The ABI needs to be invoked only once if *start_index == 0 && last_index == current_index*.
8. The callee returns the tag of the partition information in response to an invocation of this ABI with *start_index = 0*.
9. The caller encodes this tag in every invocation of this ABI with *start_index > 0*.
10. The callee must return the *RETRY* error if *tag(callee) != tag(caller)*.

Figure 13.2 illustrates an example where an SP uses the above mechanism to discover the presence of 10 SPs by using the Nil UUID.

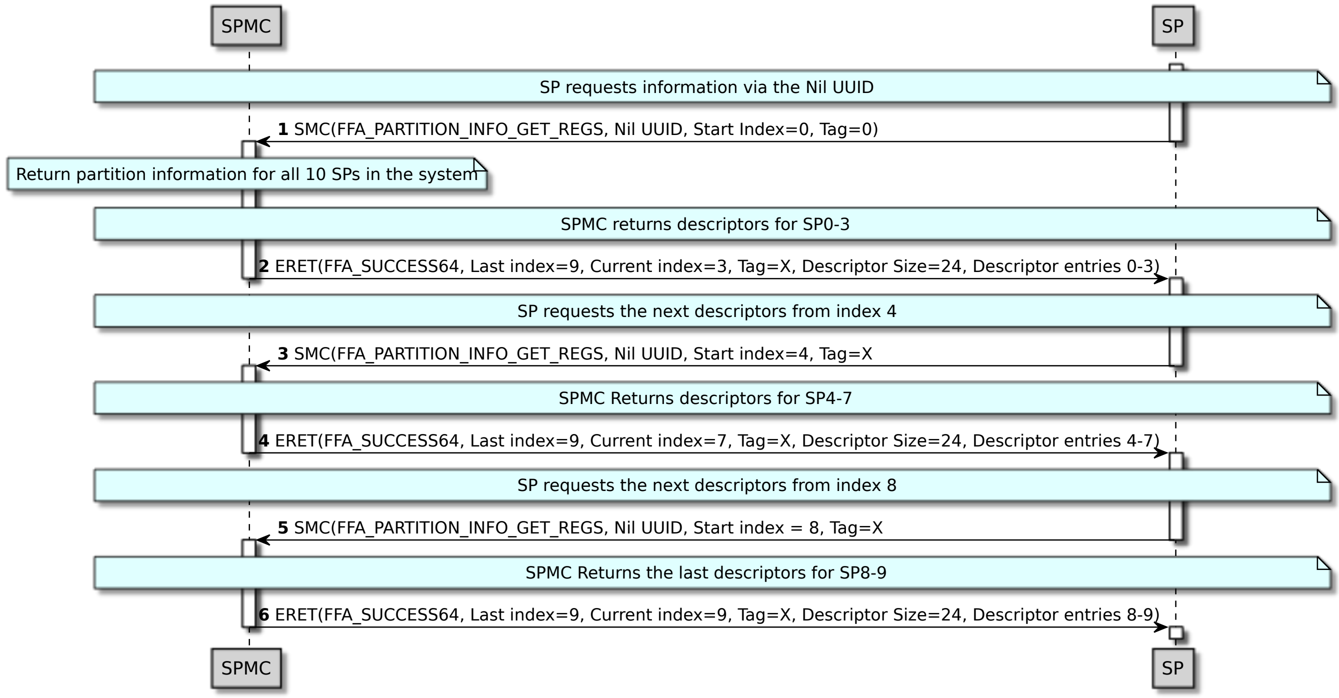

Figure 13.2: Example usage of FFA_PARTITION_INFO_GET_REGS by an SP.

13.10 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 Chapter 6 *Identification and Discovery*). • Valid FF-A instances and conduits are listed in Table 13.43. • Syntax of this function is described in Table 13.44. • Encoding of result parameters in the FFA_SUCCESS function is described in Table 13.45. • Encoding of error code in the FFA_ERROR function is described in Table 13.46.

Table 13.43: 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 13.44: FFA_ID_GET function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000069.
Other Parameter registers	w1-w7 x1-x17	• Reserved (SBZ).

Table 13.45: 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-x17	• Reserved (MBZ).

Table 13.46: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• NOT_SUPPORTED: This function is not implemented at this FF-A instance.

13.11 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 13.11.1 *Overview* for details.
- Valid FF-A instances and conduits are listed in Table 13.48.
- Syntax of this function is described in Table 13.49.
- Encoding of result parameters in the FFA_SUCCESS function is described in Table 13.50.
- Encoding of error code in the FFA_ERROR function is described in Table 13.51.

Table 13.48: 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 13.49: FFA_SPM_ID_GET function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000085.
Other Parameter registers	w1-w7 x1-x17	• Reserved (SBZ).

Table 13.50: FFA_SUCCESS encoding

Parameter	Register	Value
uint32 ID	w2	• ID of the SPMD or SPMC as described in 13.11.2 *Usage*. – Bit[31:16]: Reserved (MBZ). – Bit[15:0]: ID.
Other Result registers	w3-w7 x3-x17	• Reserved (MBZ).

Table 13.51: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• NOT_SUPPORTED: This function is not implemented at this FF-A instance.

13.11.1 Overview

v1.1 of the Framework mandates that the SPMC and SPMD components must be assigned unique IMPLEMENTATION DEFINED 16-bit IDs (see Chapter 6 *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 13.11.2 *Usage*.

The Framework assumes that no FF-A component apart from the SPMC needs to discover and use the SPMD ID.

13.11.2 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 4.1 *SPM architecture*), the SPMD could either return the SPMC ID or forward the ABI invocation to the SPMC through the ERET conduit at the Secure physical FF-A instance. This is an IMPLEMENTATION DEFINED choice.
- 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 if the SPMC is implemented at S-EL1 or S-EL2.
- An invocation of this ABI at the Secure physical FF-A instance returns the ID of the SPMC if the SPMC is implemented at EL3.

13.12 FFA_CONSOLE_LOG

Description

- Allow an entity to provide debug logging to the console.
- Valid FF-A instances and conduits are listed in Table 13.53.
- Syntax of this function is described in Table 13.54.
- Returns FFA_SUCCESS without any further parameters on successful completion.
  - In this case, the characters are logged to the console in finite time.
- Encoding of error codes in the FFA_ERROR function is described in Table 13.55.

Table 13.53: FFA_CONSOLE_LOG instances and conduits

Config No.	FF-A instance	Valid conduits
1	Secure physical	SMC
2	Secure virtual	SMC, HVC, SVC

Table 13.54: FFA_CONSOLE_LOG function syntax

Parameter	Register	Value
uint32 Function ID	w0/x0	• 0x8400008A. • 0xC400008A.
uint32 Character Count	w1/x1	• Count of characters *i* provided in *w2/x2-w7/x7* – Bit[31:8]: Reserved (SBZ). – Bit[7:0]: Count of characters. ∗ *1 <= i <= 24* if the SMC32 convention is used. ∗ *1 <= i <= 128* if the SMC64 convention is used.
uint32/uint64 Character lists	w2-w7 x2-x17	• Tightly packed list of characters – Character *i* = Bits[M:N] – M = *((8 x i) - 1)* – N = *(8 x (i - 1))*

Table 13.55: Encoding of return codes

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: Parameters are not correctly encoded. • NOT_SUPPORTED: This function is not implemented at this FF-A instance. • RETRY: Some or all characters could not be logged.

Parameter	Register	Value
uint32 Character Count	w3	• Number of characters that were successfully logged. Count starts from the first character. – Valid only with RETRY error code. MBZ otherwise.

Chapter 14
CPU cycle management interfaces

14.1 FFA_MSG_WAIT

Description

- An invocation of this ABI at a virtual or Secure physical FF-A instance with a valid conduit does the following,
  - Transitions the state of the calling execution context from *running* to *waiting* in the following runtime models.
    - 8.5 *Runtime model for SP initialization*.
    - 8.4 *Runtime model for Secure interrupt handling*.
    - 8.2 *Runtime model for FFA_RUN*.
  - Optionally relinquishes ownership of the caller's RX buffer (see 7.2.2.4 *Buffer synchronization*).
- 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 14.5.
- 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 8 *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 14.2.
- Syntax of this function is described in Table 14.3.
- Encoding of error codes in the FFA_ERROR function is described in Table 14.4.

Table 14.2: 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 14.3: FFA_MSG_WAIT function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x8400006B.
Reserved	w1/x1	• Reserved (SBZ).

Parameter	Register	Value
uint32 Flags	w2	• Flags. – Bit[0]: Retain RX Buffer Ownership flag. ∗ This flag specifies if the caller relinquishes Ownership of its RX buffer. ∗ It is ignored by a callee if the caller does not have ownership of its RX Buffer. ∗ Only valid at the virtual or Secure physical FF-A instance. It is ignored by a callee at all other instances. ∗ b'0: The caller relinquishes ownership of its RX buffer. ∗ b'1: The caller does not relinquish ownership of its RX buffer. – Bits[31:1]: Reserved (SBZ).
Other Parameter registers	w3-w7 x3-x17	• Reserved (SBZ).

Table 14.4: 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 14.5: 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.

Parameter	Register	Value
Other Parameter registers	w4-w7 x4-x17	• Reserved (MBZ).

14.2 FFA_YIELD

Description

- This ABI is invoked by an endpoint execution context to yield execution back to the FF-A component that scheduled it. E.g. SP0 yields execution back to VM0 instead of busy waiting for an IO operation to complete as illustrated in Figure 14.1.
  - The endpoint execution context transitions from the *running* to the *blocked* state.
  - The endpoint execution context invokes this ABI in the following runtime models,
    - 8.2 *Runtime model for FFA_RUN*.
    - 8.3 *Runtime model for Direct request ABIs*.
    - 8.4 *Runtime model for Secure interrupt handling*.
  - The endpoint execution context invokes this ABI with the combinations of FF-A instances and conduits listed in Table 14.7.
  - An invocation of this ABI by an endpoint execution context is completed through the following transitions. The endpoint execution context transitions from the *blocked* to the *running* state.
    - eret(FFA_RUN). This transition is applicable to all endpoints listed in Table 14.7.
    - eret(FFA_INTERRUPT). This transition is not applicable to S-EL0 endpoints (see Table 9.1).
  - The endpoint execution context is scheduled by FF-A components as described below,
    - An SP or VM is scheduled by an S-Endpoint, NS-Endpoint or Hypervisor via the FFA_RUN or Direct request ABIs.
    - An SP is scheduled by the SPMC in response to a Secure interrupt via the FFA_INTERRUPT ABI.
- This ABI is invoked by an SPMC in response to a corresponding invocation by an S-Endpoint execution context. This is done to yield execution back to the S-Endpoint (via the ERET conduit), Hypervisor or NS-Endpoint execution context (via the SMC conduit) that originally scheduled the calling S-Endpoint. The valid combinations of SPMCs, FF-A instances and conduits are listed in Table 14.8.
- This ABI is invoked by the SPMD in response to a corresponding invocation by an SPMC on behalf of an S-Endpoint execution context. This is done to yield execution back to the Hypervisor or NS-Endpoint execution context (via the ERET conduit) that originally scheduled the calling S-Endpoint. The valid combinations of FF-A instances and conduits are listed in Table 14.8.
- This ABI is invoked by a Hypervisor in response to a corresponding invocation by a VM execution context or the SPMD on behalf of an S-Endpoint execution context. This is done to yield execution back to the VM (via the ERET conduit) that originally scheduled the calling endpoint. The valid combinations of FF-A instances and conduits are listed in Table 14.8.
  - 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 is described in Table 14.9.
- Encoding of error codes in the FFA_ERROR function is described in Table 14.10.

Table 14.7: Valid combinations of endpoints, instances and conduits for invoking FFA_YIELD

Instance/Conduit	SMC	HVC	SVC
Secure virtual	S-EL1 SP	S-EL1 SP	S-EL0 SP
Secure physical	S-EL1 SP	NA	NA
Non-secure virtual	EL1 SP	EL1 SP	NA

Table 14.8: Valid combinations of partition managers, instances and conduits for invoking FFA_YIELD

Instance/Conduit	SMC	ERET
Secure virtual	NA	Any SPMC
Secure physical	S-EL1 SPMC, S-EL2 SPMC	NA
Non-secure virtual	NA	Hypervisor
Non-secure physical	NA	SPMD

Table 14.9: FFA_YIELD function syntax

Parameter	**Register**	**Value**
uint32 Function ID	w0	• 0x8400006C.
uint32 Endpoint/vCPU IDs	w1	• Endpoint and vCPU IDs of the caller endpoint. – Bit[31:16]: Endpoint ID. – Bit[15:0]: vCPU ID. • This parameter is used by the Hypervisor, SPMC and SPMD at the following combinations of FF-A instances and conduits. It is Reserved (MBZ) in all other scenarios. – By an SPMC at the, ∗ Secure physical FF-A instance with the SMC conduit. ∗ Secure virtual FF-A instance with the ERET conduit. – By the SPMD at the Non-secure physical FF-A instance with the ERET conduit. – By the Hypervisor at the Non-secure virtual FF-A instance with the ERET conduit.
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. • This parameter is used by the Hypervisor at the following combinations of FF-A instances and conduits. It is Reserved (MBZ) at in all other scenarios. – Non-secure virtual FF-A instance with the ERET conduit.

Parameter	Register	Value
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. • This parameter is used by the Hypervisor at the following combinations of FF-A instances and conduits. It is Reserved (MBZ) at in all other scenarios. – Non-secure virtual FF-A instance with the ERET conduit.
Other Parameter registers	w4-w7 x4-x17	• Reserved (SBZ).

Table 14.10: 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.

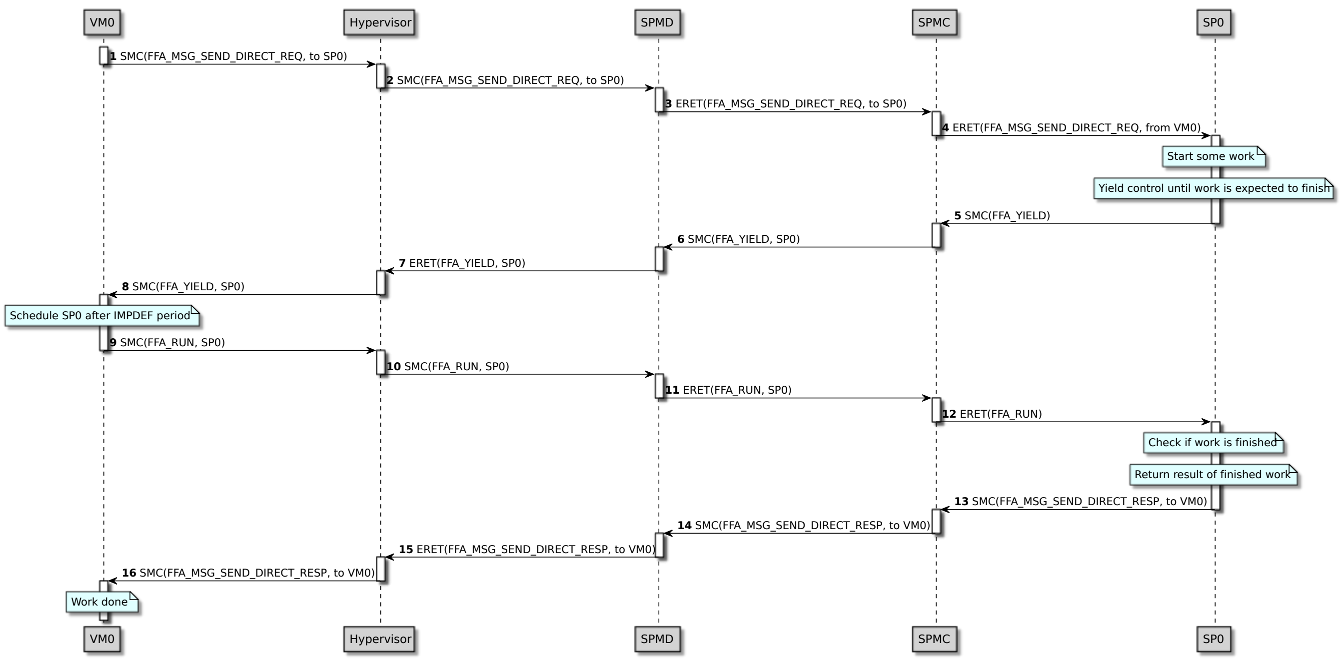

Figure 14.1: SP0 yields execution back to VM0

14.3 FFA_RUN

Description

- This ABI is used by a scheduler (see *4.9 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.
  - *8.2 Runtime model for FFA_RUN*.
  - *8.3 Runtime model for Direct request ABIs*.
- 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.
    - *8.2 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.
    - *8.2 Runtime model for FFA_RUN*.
    - *8.3 Runtime model for Direct request ABIs*.
- 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.
  - *8.2 Runtime model for FFA_RUN*.
  - *8.3 Runtime model for Direct request ABIs*.
- 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 14.12.
- Syntax of this function is described in Table 14.13.
- Encoding of error code in the FFA_ERROR function is described in Table 14.14.

Table 14.12: FFA_RUN 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, ERET

Config No.	FF-A instance	Valid conduits
4	Secure virtual	SMC, HVC, SVC, ERET

Table 14.13: 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-x17	• Reserved (SBZ).

Table 14.14: 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. – Caller is not allowed to invoke this ABI (see 9.2.3 *CPU cycle allocation modes*). • BUSY: vCPU is busy and caller must retry later. • ABORTED: vCPU or VM ran into an unexpected error and has aborted.

14.4 FFA_NORMAL_WORLD_RESUME

Description
• Request SPMD to resume execution of Normal world on current PE after the exception that originally preempted the Normal world has been handled. See 14.4.1 *Overview* for details. • Valid FF-A instances and conduits are listed in Table 14.16. • Syntax of this function is described in Table 14.17. • 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 14.18.

Table 14.16: FFA_NORMAL_WORLD_RESUME instances and conduits

Config No.	FF-A instance	Valid conduits
1	Secure physical	SMC

Table 14.17: FFA_NORMAL_WORLD_RESUME function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x8400007C.
Other Parameter registers	w1-w7 x1-x17	• Reserved (SBZ).

Table 14.18: 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.

14.4.1 Overview

Execution in Normal world could be preempted in response to an exception for example, a Secure physical interrupt. As per the Arm A-profile 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 4.1.1 *Secure EL2 SPM core component*) and S-EL1 (see 4.1.2 *S-EL1 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 15
Messaging interfaces

15.1 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 7.2.2.1.1 *Transmission of partition messages*.
  - Notify the Receiver's scheduler that the Receiver endpoint must be run to process the partition message as described in 10.8.1 *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 7.2.
- 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 code in the FFA_ERROR function is described in Table 15.4.

Table 15.2: 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 15.3: FFA_MSG_SEND2 function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000086.
uint32 Sender VM ID	w1	• Sender VM ID from whose TX buffer the message must be copied into the RX buffer of the target SP. – Bit[31:16]: Sender VM ID at the Non-secure physical instance. MBZ otherwise. – Bit[15:0]: Reserved (SBZ).
uint32 Flags	w2	• Message flags. – Must be ignored by callee when SVC conduit is used. – Bit[0]: Reserved (SBZ). – Bit[1]: Delay *Schedule Receiver* interrupt flag. Guidance in. 16.5.1 *Delay Schedule Receiver interrupt flag* applies to the FFA_MSG_SEND2 ABI. – Bit[31:2]: Reserved (SBZ).
Other Parameter registers	w3-w7 x3-x17	• Reserved (SBZ).

Table 15.4: 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. – Caller is not allowed to invoke this ABI. – 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.

15.2 FFA_MSG_SEND_DIRECT_REQ

Description

- Send a Partition or Framework message in parameter registers as a request to a Receiver endpoint, run the endpoint and block until a response is available. Also see 7.4 *Direct messaging usage*.
- Valid FF-A instances and conduits are listed in Table 15.6.
- Syntax of this function is described in Table 15.7.
- 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_YIELD* to indicate that the Receiver endpoint has transitioned to the *blocked* runtime state 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 15.8.

Table 15.6: FFA_MSG_SEND_DIRECT_REQ 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, ERET
4	Secure virtual	SMC, HVC, SVC, ERET

Table 15.7: FFA_MSG_SEND_DIRECT_REQ function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x8400006F. • 0xC400006F.
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	• 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 (SBZ). – Bit[7:0]: ∗ Reserved (MBZ) if bit[31] = b'0. ∗ Framework message type if bit[31] = b'1. · See Table 18.6 & Table 18.7 in 18.2.4 *Power Management messages*.
IMPLEMENTATION DEFINED values	w3-w7 x3-x7	• IMPLEMENTATION DEFINED values.
Other Parameter registers when using the SMC64, HVC64 or SVC64 convention.	x8-x17	• Reserved (SBZ).

Table 15.8: 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. – Receiver endpoint does not support receipt of Direct request messages. • NOT_SUPPORTED: This function is not implemented at this FF-A instance. • BUSY: Receiver endpoint is in a *running*, *blocked* or *preempted* state. • ABORTED: Receiver endpoint ran into an unexpected error and has aborted. • NOT_READY: Receiver endpoint is not ready to handle this request.

15.3 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 wait until a new message is available. Also see 7.4 *Direct messaging usage*.
- Valid FF-A instances and conduits are listed in Table 15.10.
- Syntax of this function is described in Table 15.11.
- Successful completion of this function is indicated in the same manner as that of the *FFA_MSG_WAIT* function (also see 14.1 *FFA_MSG_WAIT*).
- Encoding of error code in the FFA_ERROR function is described in Table 15.12.

Table 15.10: FFA_MSG_SEND_DIRECT_RESP instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure physical	ERET
2	Secure physical	SMC, ERET
3	Non-secure virtual	SMC, HVC, ERET
4	Secure virtual	SMC, HVC, SVC, ERET

Table 15.11: 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.
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 (SBZ). – Bit[7:0]: ∗ Reserved (MBZ) if Bit[31] = b'0. ∗ Framework message type if bit[31] = b'1. · See Table 18.8 in 18.2.4 *Power Management messages*.
IMPLEMENTATION DEFINED values	w3-w7 x3-x7	• IMPLEMENTATION DEFINED values.
Other Parameter registers when using the SMC64, HVC64 or SVC64 convention.	x8-x17	• Reserved (SBZ).

Table 15.12: 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.
		  – Caller is not allowed to invoke this ABI.
		  – Receiver endpoint does not support receipt of Direct response messages.
		• NOT_SUPPORTED: This function is not implemented at this FF-A instance.
		• ABORTED: Receiver endpoint ran into an unexpected error and has aborted.

15.4 FFA_MSG_SEND_DIRECT_REQ2

Description

- Send a Partition message in parameter registers as a request to a Receiver endpoint, run the endpoint and block until a response is available. The UUID parameter is used as described in 6.2.3 *Partition UUID usage*.
- Valid FF-A instances and conduits are listed in Table 15.14.
- Syntax of this function is described in Table 15.15.
- Successful completion of this function is indicated through an invocation of the following interfaces by the callee:
  - *FFA_MSG_SEND_DIRECT_RESP2* 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_YIELD* to indicate that the Receiver endpoint has transitioned to the *blocked* runtime state 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 15.16.

Table 15.14: FFA_MSG_SEND_DIRECT_REQ2 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, ERET
4	Secure virtual	SMC, HVC, SVC, ERET

Table 15.15: FFA_MSG_SEND_DIRECT_REQ2 function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0xC400008D.
uint32 Sender/Receiver IDs	w1	• Sender and Receiver endpoint IDs. – Bits[31:16]: Sender endpoint ID. – Bits[15:0]: Receiver endpoint ID.
uint64 UUID Lo	x2	• Bytes[0...7] of UUID with byte 0 in the low-order bits.
uint64 UUID Hi	x3	• Bytes[8...15] of UUID with byte 8 in the low-order bits.
Other Parameter registers	x4-x17	• IMPLEMENTATION DEFINED values.

Table 15.16: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: – Invalid endpoint ID or message flags. – Unrecognized UUID. • DENIED: – Callee is not in a state to handle this request. – Caller is not allowed to invoke this ABI. – Receiver endpoint does not support receipt of Direct request messages. • NOT_SUPPORTED: This function is not implemented at this FF-A instance. • BUSY: Receiver endpoint is in a *running*, *blocked* or *preempted* state. • ABORTED: Receiver endpoint ran into an unexpected error and has aborted. • NOT_READY: Receiver endpoint is not ready to handle this request.

15.5 FFA_MSG_SEND_DIRECT_RESP2

Description

- Send a Partition message in parameter registers as a response to a target endpoint, run the endpoint and wait until a new message is available. Also see 7.4 *Direct messaging usage*.
- Valid FF-A instances and conduits are listed in Table 15.18.
- Syntax of this function is described in Table 15.19.
- Successful completion of this function is indicated in the same manner as that of the *FFA_MSG_WAIT* function (also see 14.1 *FFA_MSG_WAIT*).
- Encoding of error code in the FFA_ERROR function is described in Table 15.20.

Table 15.18: FFA_MSG_SEND_DIRECT_RESP2 instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure physical	ERET
2	Secure physical	SMC, ERET
3	Non-secure virtual	SMC, HVC, ERET
4	Secure virtual	SMC, HVC, SVC, ERET

Table 15.19: FFA_MSG_SEND_DIRECT_RESP2 function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0xC400008E.
uint32 Source/Destination IDs	w1	• Source and destination endpoint IDs. – Bit[31:16]: Source endpoint ID. – Bit[15:0]: Destination endpoint ID.
uint64 Reserved	w2	• Reserved (SBZ).
uint64 Reserved	w3	• Reserved (SBZ).
Other Parameter registers	x4-x17	• IMPLEMENTATION DEFINED values.

Table 15.20: 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. – Caller does not support sending of Direct response messages. – Receiver endpoint does not support receipt of Direct response messages. • NOT_SUPPORTED: This function is not implemented at this FF-A instance. • ABORTED: Receiver endpoint ran into an unexpected error and has aborted.

Chapter 16
Notification interfaces

16.1 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 10.3 *Notification bitmap setup*.
- Valid FF-A instances and conduits are listed in Table 16.2.
- Syntax of this function is described in Table 16.3.
- Returns FFA_SUCCESS without any further parameters on successful completion.
- Encoding of error codes in the FFA_ERROR function is described in Table 16.4.

Table 16.2: FFA_NOTIFICATION_BITMAP_CREATE instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure physical	SMC
2	Secure physical	ERET

Table 16.3: 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 (MBZ). – Bit[15:0]: VM ID.
uint32 vCPU count	w2	• Number of vCPUs implemented by the VM.
Other Parameter registers	w3-w7 x3-x17	• Reserved (SBZ).

Table 16.4: 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.

16.2 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 10.3 *Notification bitmap setup*. • Valid FF-A instances and conduits are listed in Table 16.6. • Syntax of this function is described in Table 16.7. • Returns FFA_SUCCESS without any further parameters on successful completion. • Encoding of error codes in the FFA_ERROR function is described in Table 16.8.

Table 16.6: FFA_NOTIFICATION_BITMAP_DESTROY instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure physical	SMC
2	Secure physical	ERET

Table 16.7: 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 (SBZ). – Bit[15:0]: VM ID.
Other Parameter registers	w2-w7 x2-x17	• Reserved (SBZ).

Table 16.8: 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.

16.3 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 10.4.2 *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 16.10.
- Syntax of this function is described in Table 16.11.
- Returns FFA_SUCCESS without any further parameters on successful completion.
- Encoding of error codes in the FFA_ERROR function is described in Table 16.12.

Table 16.10: 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 16.11: 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[0]: Per-vCPU notification flag (see 10.4.2 *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 (SBZ).
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.

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 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-x17	• Reserved (SBZ).

Table 16.12: 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. – Caller is not allowed to invoke this ABI. • ABORTED: Sender partition ran into an unexpected error and has aborted.

16.4 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 10.4.2 *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 16.14. • Syntax of this function is described in Table 16.15. • Returns FFA_SUCCESS without any further parameters on successful completion. • Encoding of error codes in the FFA_ERROR function is described in Table 16.16.

Table 16.14: 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 16.15: 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 (SBZ).
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.
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.

Parameter	Register	Value
Other Parameter registers	w5-w7 x5-x17	• Reserved (SBZ).

Table 16.16: 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. – Caller is not allowed to invoke this ABI. • ABORTED: Sender partition ran into an unexpected error and has aborted.

16.5 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 Receiver endpoint. Also see 10.5 *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 16.18. • Syntax of this function is described in Table 16.19. • Returns FFA_SUCCESS without any further parameters on successful completion. • Encoding of error codes in the FFA_ERROR function is described in Table 16.20.

Table 16.18: 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 16.19: 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]: Per-vCPU notification flag (see 10.4.2 *Notification binding*). ∗ b'1: All notifications in the bitmap are per-vCPU notifications. · Each notification must be signaled to the vCPU specified in the *Receiver vCPU ID* field. ∗ b'0: All notifications in the bitmap are global notifications · The *Receiver vCPU ID* field MBZ. – Bit[1]: Delay *Schedule Receiver* interrupt flag. See 16.5.1 *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-x17	• Reserved (SBZ).

Table 16.20: Encoding of return codes

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: – Unrecognized partition ID or invalid flags. – *Per-vCPU notification flag = b'0* and *Receiver vCPU ID* != 0. – *Per-vCPU notification flag = b'0* and a per-vCPU notification is specified in the *Notification bitmap*. – *Per-vCPU notification flag = b'1* and a global notification is specified in the *Notification bitmap*. • 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.

16.5.1 Delay Schedule Receiver interrupt flag

The partition manager uses an IMPLEMENTATION DEFINED policy to determine when the *Schedule Receiver* interrupt must be asserted in response an invocation of the FFA_NOTIFICATION_SET interface. The interrupt could be asserted before or after an invocation of this interface completes.

The SPMC could choose to assert this interrupt before completion of an FFA_NOTIFICATION_SET interface invocation by an SP. The interrupt would either preempt or trigger a managed exit of the caller SP execution context immediately upon the completion of the interface invocation. This might be undesirable for the SP in some scenarios. The non-secure *Schedule Receiver* interrupt could trigger a switch to the Normal world when the SP is about to request the same switch itself. For example, an SP sends notifications while handling a Direct request from the Normal world. It could switch back to the Normal world through a Direct response immediately after sending the notifications thereby avoiding the need to pend the *Schedule Receiver* interrupt until switch takes place.

The *Delay Schedule Receiver interrupt* flag is a hint from the SP execution context to the SPMC it does not have to assert this interrupt upon completion of the FFA_NOTIFICATION_SET interface invocation. The use of this flag by the SP could help the SPMC optimize the policy it uses for asserting this interrupt. This flag is only used at the Secure virtual FF-A instance. It MBZ at all other FF-A instances.

The above guidance for this flag applies to the FFA_MSG_SEND2 ABI described in 15.1 *FFA_MSG_SEND2* as well.

16.6 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 10.5 *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 16.22.
- Syntax of this function is described in Table 16.23.
- Encoding of result parameters in the FFA_SUCCESS function is described in Table 16.24.
- Returns FFA_SUCCESS without any further parameters on successful completion.
- Encoding of error codes in the FFA_ERROR function is described in Table 16.25.

Table 16.22: 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 16.23: 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 vCPU ID. – Bit[15:0]: Receiver endpoint 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 SBZ 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 SBZ 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 (SBZ).
Other Parameter registers	w3-w7 x3-x17	• Reserved (SBZ).

Table 16.24: 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.
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 16.25: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• INVALID_PARAMETERS: Unrecognized partition ID or incorrectly encoded Flags parameter. • DENIED: Caller is not allowed to invoke this ABI. • NOT_SUPPORTED: This function is not implemented at this FF-A instance.

16.7 FFA_NOTIFICATION_INFO_GET

Description

- This ABI returns lists of endpoints that have pending notifications and must be run to handle their notifications. This is described in 16.7.1 *Usage*.
- Valid FF-A instances and conduits are listed in Table 16.27.
- Syntax of this function is described in Table 16.28.
- Encoding of result parameters in the FFA_SUCCESS function is described in Table 16.29.
- Encoding of error codes in the FFA_ERROR function is described in Table 16.30.

Table 16.27: 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

Table 16.28: FFA_NOTIFICATION_INFO_GET function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x84000083. • 0xC4000083.
Other Parameter registers	w1-w7 x1-x17	• Reserved (SBZ).

Table 16.29: FFA_SUCCESS encoding

Parameter	Register	Value
uint32/uint64 Pending notification flags	w2/x2	• See 16.7.1 *Usage*.
uint32/uint64 ID lists	w3-w7 x3-x17	• See 16.7.1 *Usage*.

Table 16.30: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• NO_DATA: There is no pending notification information available. • NOT_SUPPORTED: This function is not implemented at this FF-A instance.

16.7.1 Usage

This ABI is used by an NS-Endpoint or the Hypervisor to retrieve a list of endpoints that have pending notifications as described below. Also see 10.5 *Notification signaling*.

- This ABI is invoked by a VM at the Non-secure virtual FF-A instance with the SMC or HVC conduits to request the Hypervisor to return the list of SPs and VMs that have pending notifications. The Hypervisor returns the list of those endpoints whose schedulers are implemented in the calling VM.
- This ABI is invoked by the Hypervisor or the OS Kernel at a Non-secure physical FF-A instance with the SMC conduit to request the SPMC to return the list of SPs and VMs that have pending notifications. The ABI invocation is forwarded by the SPMD to the SPMC as described below,
  - Through the ERET conduit if they do not reside in the same exception level. Also see 4.1.1 *Secure EL2 SPM core component* and 4.1.2 *S-EL1 SPM core component*.
  - Through an IMPLEMENTATION DEFINED mechanism if they reside in the same exception level. Also see 4.1.3 *EL3 SPM core component*.

  The Hypervisor is responsible for signaling the *Notification pending interrupt* to any VM that has a pending notification. It uses the list of VMs returned by the SPMC to discover the VMs that have pending notifications signaled by SPs.

The lists of endpoints with pending notifications is returned in *w2/x2-w7/x7* registers as described below.

1. One or more lists of 16-bit IDs are returned in the *ID lists registers w3/x3-w7/x7*. This is subject to the following rules.
   1. An ID is of one of the following types.
      1. An endpoint ID.
      2. A vCPU ID.
   2. If an endpoint has only one or more pending global notifications, its ID is returned in a list of size 1.
   3. If an endpoint has one or more pending per-vCPU notifications, its ID is the first element in the list followed by the IDs of vCPUs that have pending notifications. The size of the list is > 1 in this case.
   4. Each list has a minimum size of 1 and a maximum size of 4. 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.
   5. The *ID lists* are tightly packed in the registers as follows.
      1. The first ID of the first list is encoded as follows,
         1. In *Bit[15:0]* in *w3* if the SMC32 convention is used.
         2. In *Bit[15:0]* in *x3* if the SMC64 convention is used.
      2. The bit position of the first ID of the next list is calculated by using the number of IDs in the previous list. Subsequent lists follow in the same or a higher numbered register.
   6. With the SMC32 calling convention, the *ID lists* registers can accommodate 10 IDs.
   7. With the SMC64 calling convention, the *ID lists* registers can accommodate 20 IDs.

2. 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* as described in Table 16.31.
3. All information about endpoints with pending notifications may not fit in one invocation of this ABI. The partition manager sets the *More pending notifications flag* in *w2/x2* in this case. This ABI is invoked until the flag is unset by the partition manager to retrieve all the information.

   Information about pending notifications is returned by the partition manager only once i.e. an *ID list* retrieved in one invocation of this interface cannot be retrieved again in a subsequent invocation.
4. 16.7.1.1 *Example usage* describes an example encoding of pending notification information as described above.

Table 16.31: Pending notifications flags encoding

Field	Description
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 of Receiver endpoints with pending notifications. It must invoke this interface again to retrieve the remaining lists.
Bit[6:1]	• Reserved (MBZ).
Bit[11:7]	• Count of lists returned in ID lists registers. – Bit[11] Reserved (MBZ) if the SMC32 convention is used.
Bit[M:N]	• Count of IDs in list *i* where, – Count = Bit[M:N] + 1. – M = ((2 x *i*) - 1) + *off*. – N = (2 x (*i* - 1)) + *off*. – *off* is the starting bit offset = 12. – 1 <= *i* <= 10 if the SMC32 convention is used. – 1 <= *i* <= 20 if the SMC64 convention is used. • Value of Bit[M:N] in unused lists is ignored.
Bit[63:52]	• Reserved (MBZ) if the SMC64 convention is used.

16.7.1.1 Example usage

Table 16.32 considers an example scenario where partitions listed in the first column have pending notifications of the type specified in the second column. If a per-vCPU notification is pending, the IDs of the vCPUs are listed in the third column.

Table 16.32: Example encoding of notification information

Partition ID	Notification type	vCPU IDs
0	• Per vCPU	0, 2, 3, 4, 6
2	• Global Notification	NA
3	• Per vCPU	1

This information is encoded by the partition manager of the partition that invokes the SMC32 variant of the FFA_NOTIFICATION_INFO_GET ABI as illustrated in Figure 16.1. The encoding in response to an invocation of the SMC64 variant of the FFA_NOTIFICATION_INFO_GET ABI is illustrated in Figure 16.2.

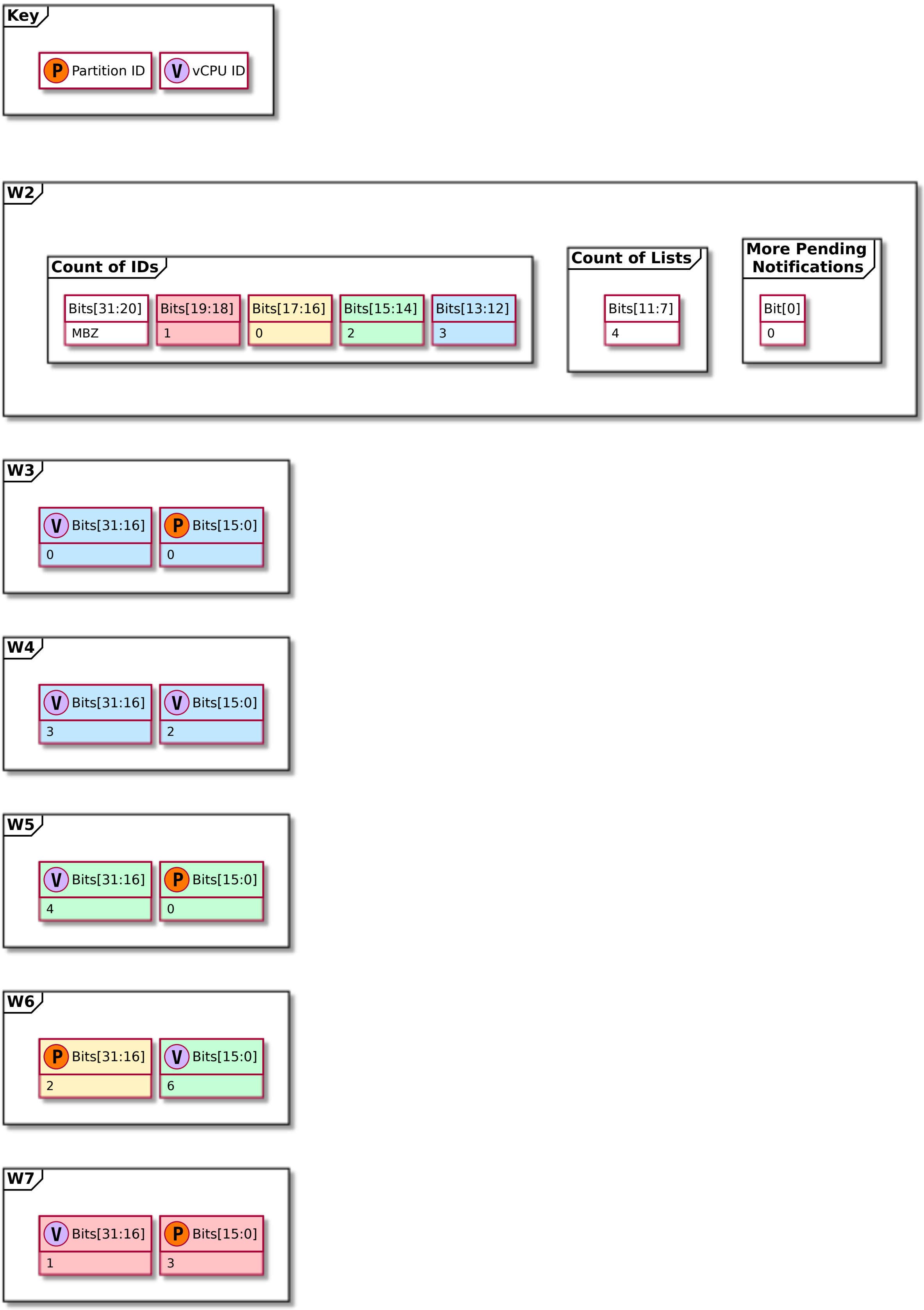

Figure 16.1: Encoding of NOTIFICATION_INFO_GET using 32bit ABI

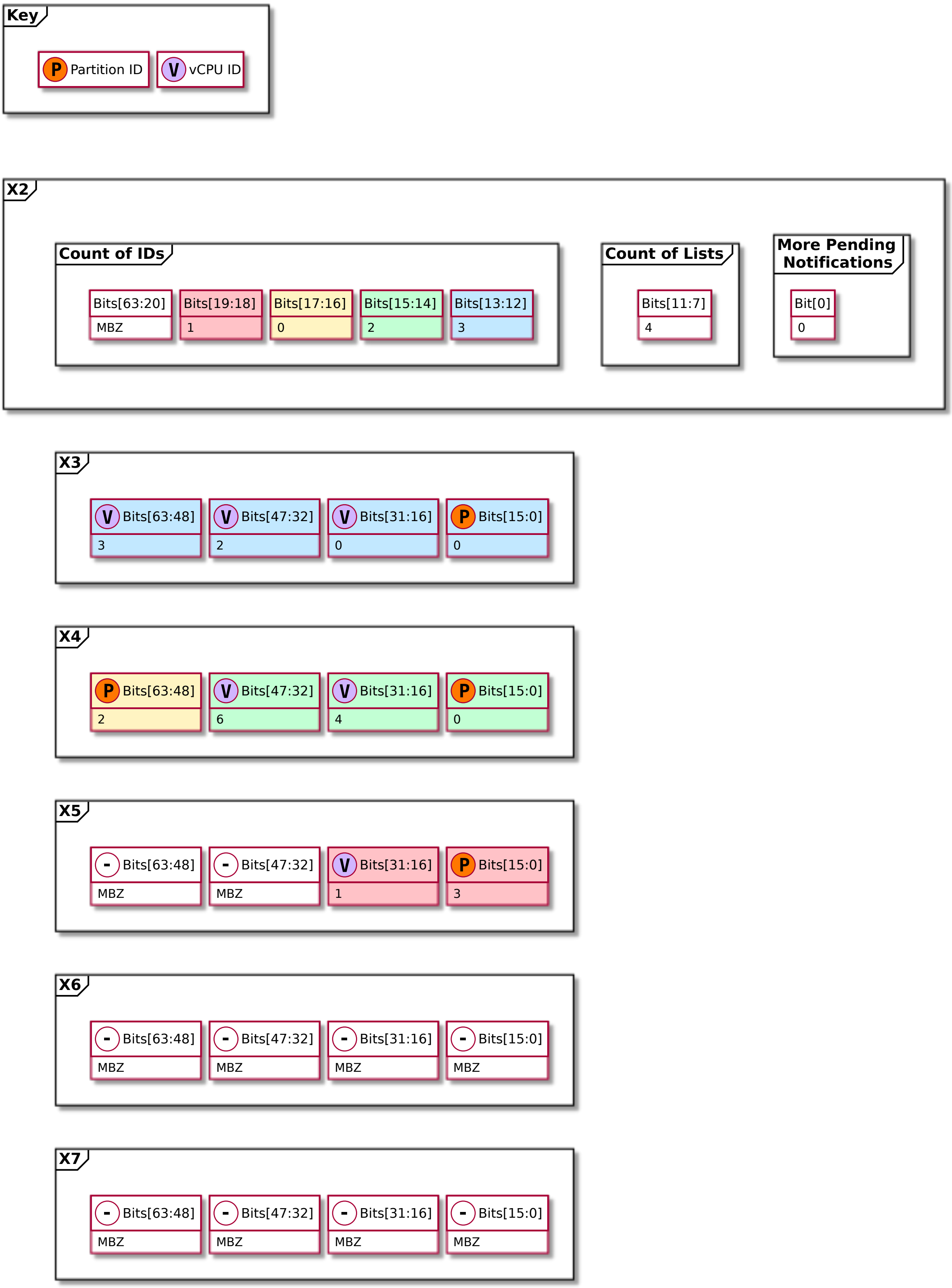

Figure 16.2: Encoding of NOTIFICATION_INFO_GET using 64bit ABI

Chapter 17
Interrupt management interfaces

17.1 FFA_EL3_INTR_HANDLE

Description
• Request EL3 firmware to handle a pending interrupt. • Valid FF-A instances and conduits are listed in Table 17.2. • Syntax of this function is described in Table 17.3. • Returns FFA_SUCCESS without any further parameters on successful completion. • Encoding of error code in the FFA_ERROR function is described in Table 17.4.

Table 17.2: FFA_EL3_INTR_HANDLE instances and conduits

Config No.	FF-A instance	Valid conduits
1	Secure physical FF-A	SMC

Table 17.3: FFA_EL3_INTR_HANDLE function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x8400008C.
Other Parameter registers	w1-w7 x1-x17	• Reserved (SBZ).

Table 17.4: FFA_ERROR encoding

Parameter	Register	Value
int32 Error code	w2	• NOT_SUPPORTED: This function is not implemented at this FF-A instance.

17.1.1 Overview

This ABI is used by a S-EL2 or S-EL1 SPMC to delegate handling of an EL3 interrupt to EL3 firmware. On a GICv3 system, when SCR_EL3.FIQ=0, the SPMC at S-EL2 or S-EL1 uses this ABI to request EL3 firmware to handle a pending Group 0 interrupt that cannot be handled at the same or lower Exception level than the SPMC.

The following rules govern the behaviour of this ABI.

- An invocation of this ABI returns the NOT_SUPPORTED error code in the following scenarios:
  - The ABI is invoked at an unsupported FF-A instance.
  - The ABI is invoked at S-EL1 or S-EL2 and SCR_EL3.FIQ=1.
- EL3 firmware does not perform a switch to another security state as a part of handling an invocation of this ABI.

  This rule ensures that an invocation of this ABI is handled entirely in EL3 firmware before returning to the calling Exception level in the Secure state. This helps preserve the FF-A programming model where an exit to another Security state is always explicitly requested by software in the Secure state in S-EL1 or S-EL2.

Figure 17.1 illustrates how a Group 0 interrupt can be handled by TF-A at EL3 in coordination with an SPMC at S-EL1 or S-EL2 via the use of this ABI when SCR_EL3.FIQ=0 in the Secure state.

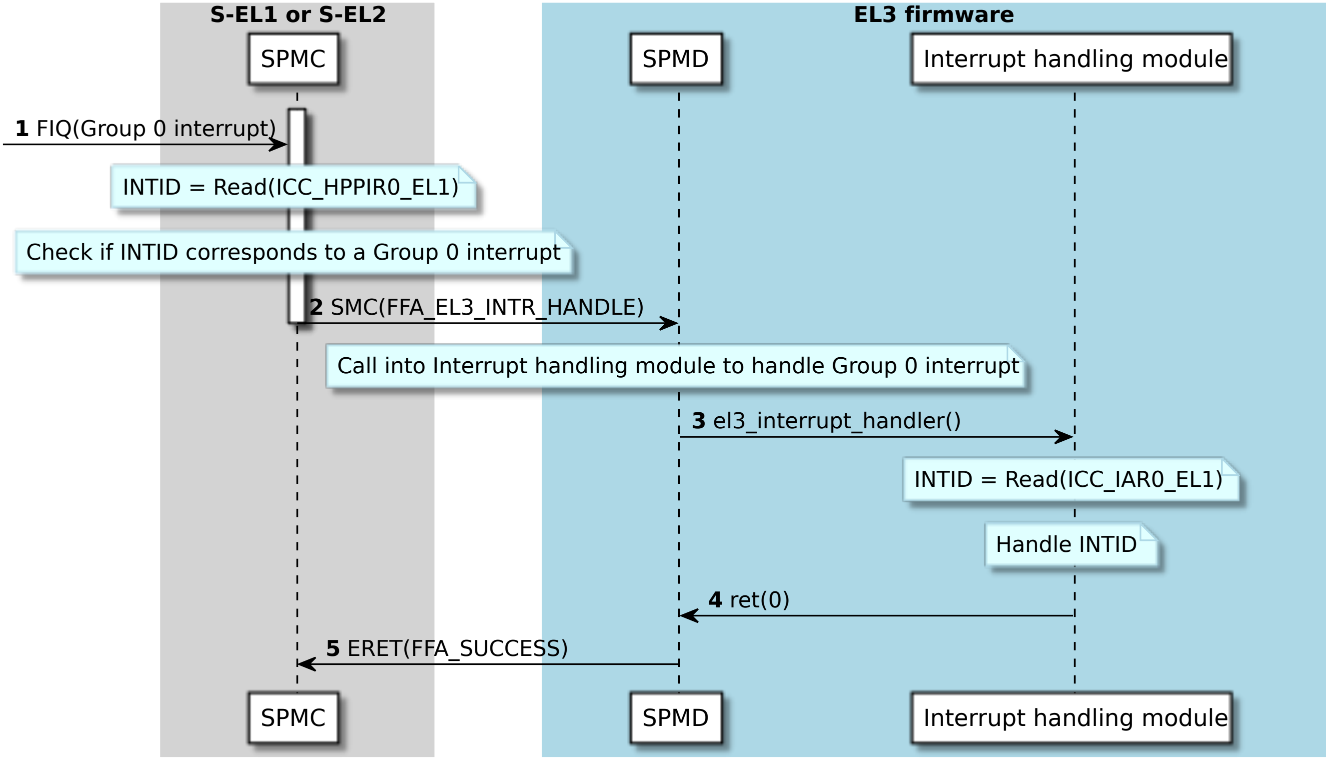

Figure 17.1: Example usage of FFA_EL3_INTR_HANDLE

Chapter 18
Appendix

18.1 S-EL0 partitions

S-EL0 partitions in either execution state are used to achieve isolation among Secure services on Arm A-profile systems where it is either not possible or not desirable to deploy a S-EL1 physical partition. They could host one or more device drivers to control hardware or security services that are accessed by the Normal world through the message passing interfaces described in this specification. An example use case of S-EL0 partitions is described in 18.1.1 *UEFI PI Standalone Management Mode partitions*.

18.1.1 UEFI PI Standalone Management Mode partitions

Standalone management mode (STMM) is described in [8] 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 Arm A-profile 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.

18.1.1.1 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 re-entrant 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 7.4 *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 [9] 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 model and CPU cycle allocation mode are described in Chapter 8 *Partition runtime models* and 9.4 *Support for legacy run-time 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 [10]). The *FFA_MEM_PERM_GET* and *FFA_MEM_PERM_SET* interfaces are described in the FF-A memory management protocol [1].
3. A canonical UUID to discover the presence of STMM SPs.

STMM SP UUID:	378daedc-f06b-4446-8314-40ab933c87a3

Some example flows to illustrate common aspects of interaction with a STMM SP based on the preceding concepts are as follows.

- Figure 18.1 describes how the MM communication driver can discover the presence of STMM SPs and their properties. It is assumed that:
  - All STMM SPs share a MM service UUID. The Framework allows a 1:N mapping between the UUID and partitions (also see 6.2.3 *Partition UUID usage*). Each STMM SP specifies this UUID, its run-time model, memory regions, devices etc. in its partition manifest.
  - The MM service UUID is used by MM communication driver to discover the partition IDs and properties of all the STMM SPs through a FF-A partition discovery mechanism.
  - The MM communication buffer for each STMM SP is allocated by the EFI MM communication driver.
- Figure 18.2 describes how the MM communication driver and a STMM SP can communicate using Direct Partition messages and the communication buffer shared between them.
- Figure 18.3 describes how the STMM SP can be invoked in response to an interrupt.

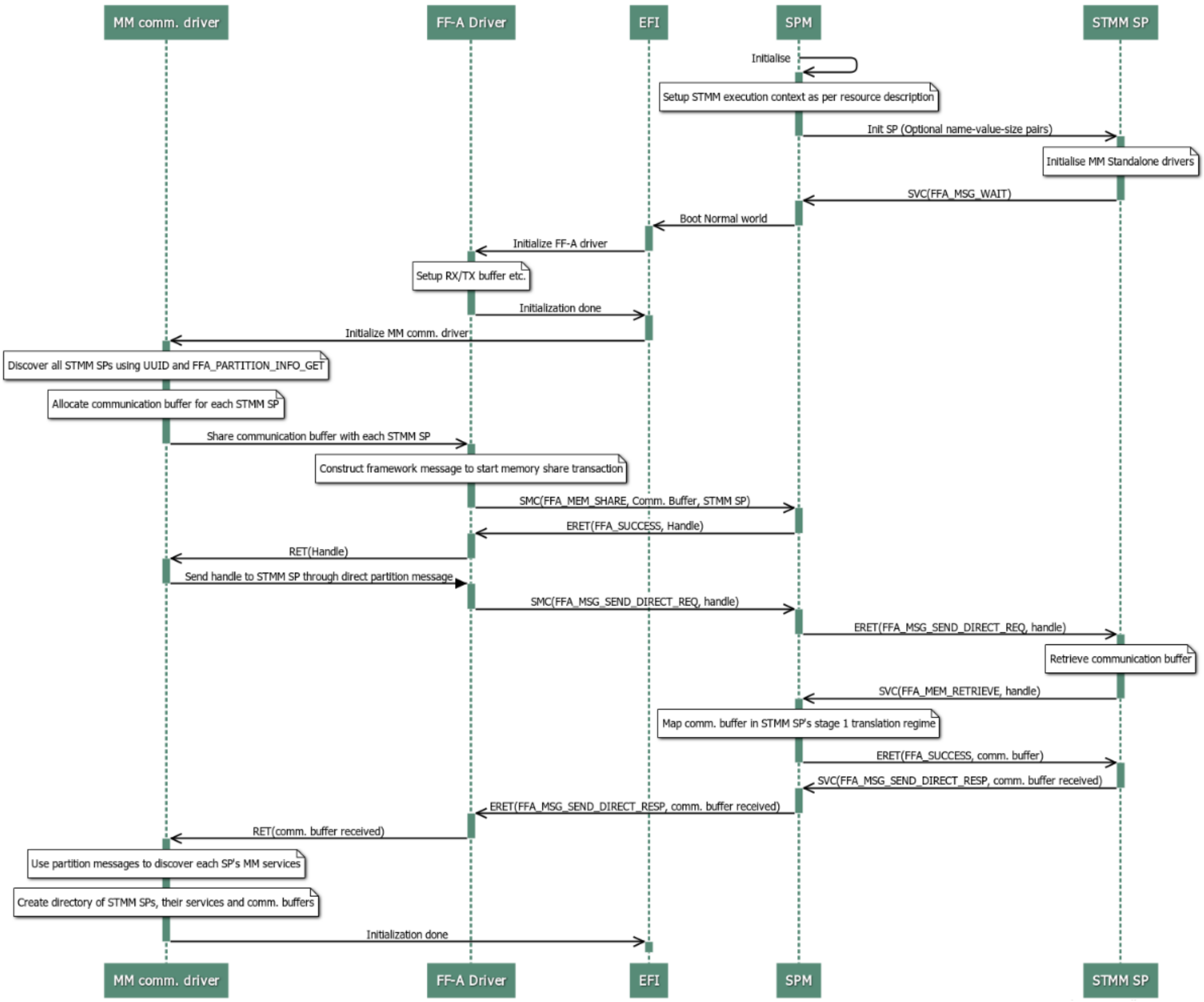

Figure 18.1: MM communication driver initialization

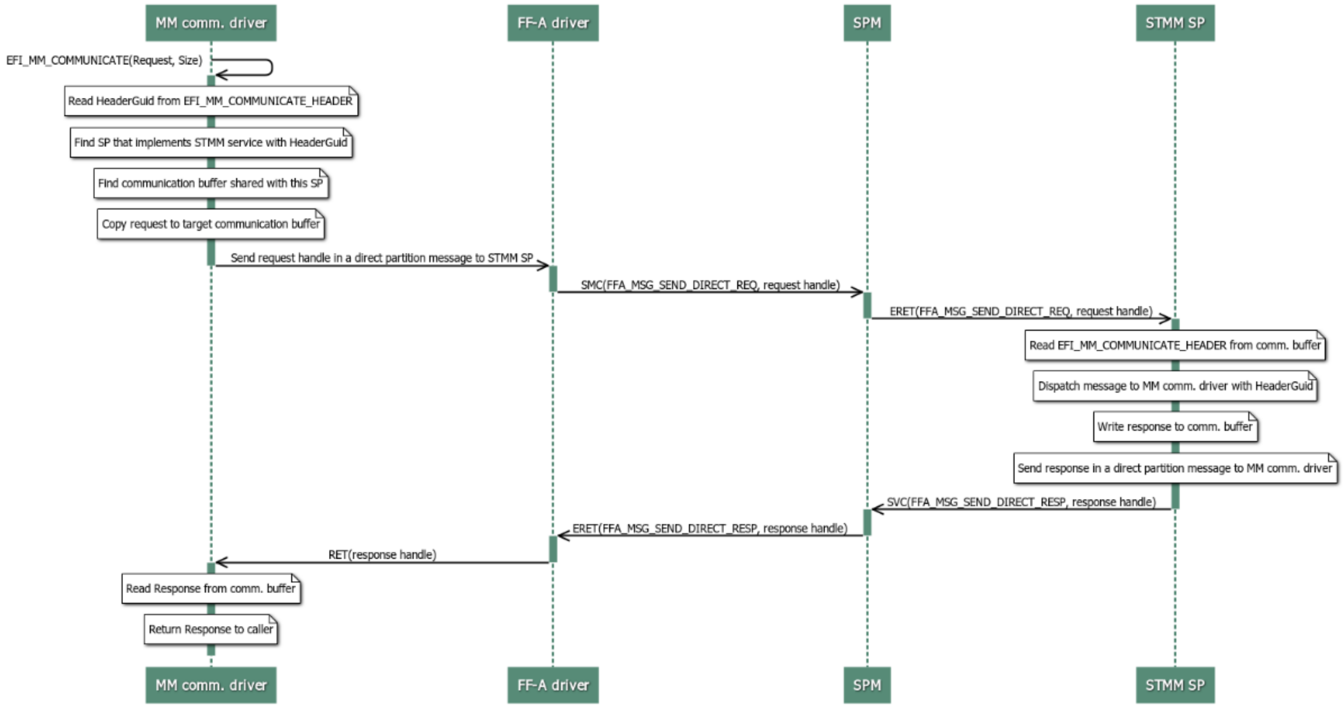

Figure 18.2: Message exchange between a STMM SP and MM communication driver

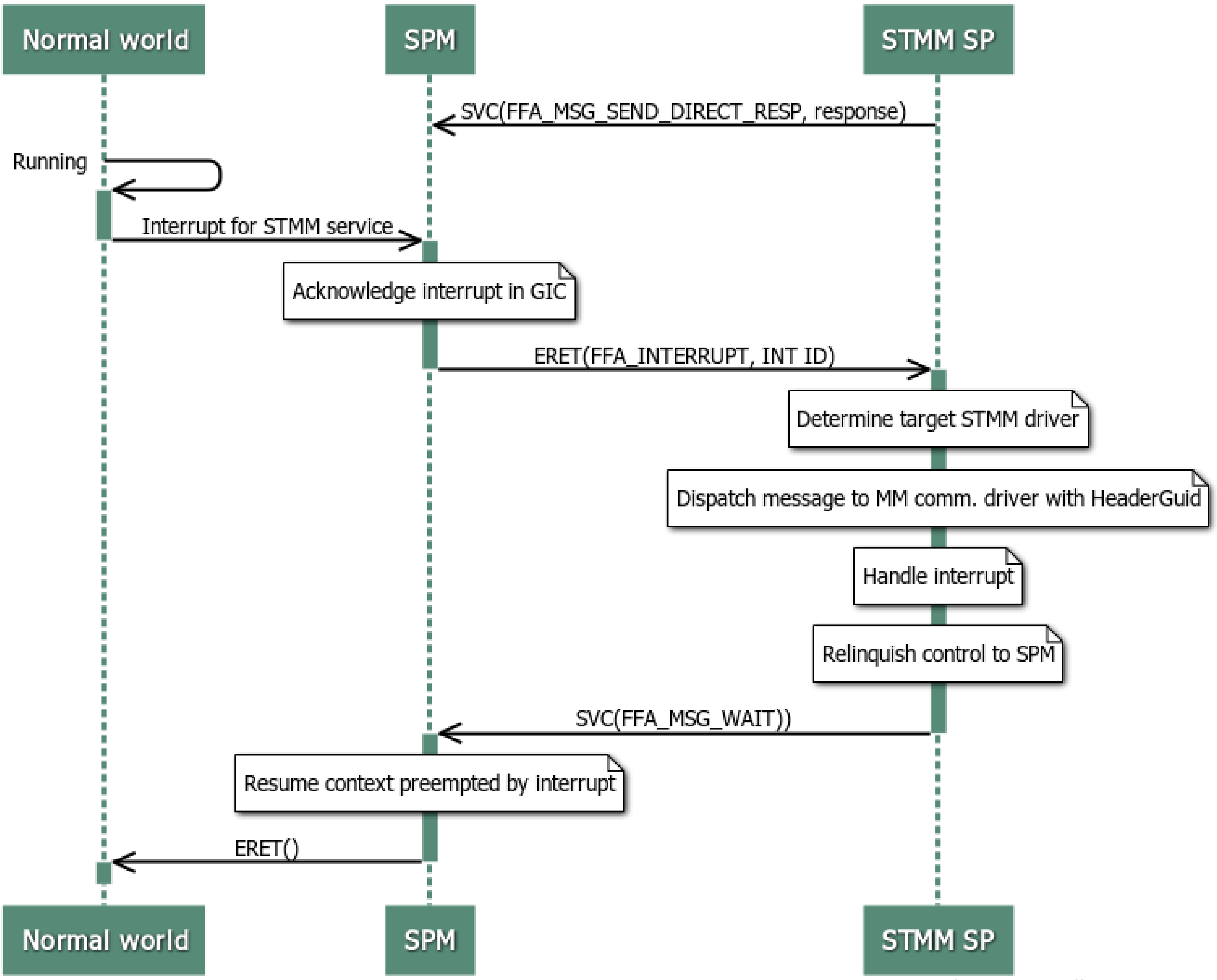

Figure 18.3: Invocation of a STMM SP in response to an interrupt

18.2 Power Management

18.2.1 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 [11] 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 [11]) 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 [11]). 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 5 *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 18.2.2 *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 18.2.3 *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 18.2.4 *Power Management messages*.

18.2.2 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 5 *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 5.1.
    - Table 5.4.

2. The FFA_SECONDARY_EP_REGISTER function (see 18.2.2.1 *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 8.5 *Runtime model for SP initialization* to initialize the SP execution context.

18.2.2.1 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 18.2.2.1.1 *Usage*.
- Valid FF-A instances and conduits are listed in Table 18.3.
- Syntax of this function is described in Table 18.4.
- Returns FFA_SUCCESS without any further parameters on successful completion.
- Encoding of error code in the FFA_ERROR function is described in Table 18.5.

Table 18.3: FFA_SECONDARY_EP_REGISTER instances and conduits

Config No.	FF-A instance	Valid conduits
1	Secure physical	SMC
2	Secure virtual	SMC, HVC

Table 18.4: 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 physical 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. – Address is a PA at the Secure physical FF-A instance with a EL3 SPMD and S-EL2 SPMC.
Other Parameter registers	w2-w7 x2-x17	• Reserved (SBZ).

Table 18.5: 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. • DENIED: – This function was invoked by a S-EL0 SP which only has a single execution context. – This function was invoked when the caller is not in the runtime model for SP initialization.

18.2.2.1.1 Usage

This function is invoked by a SP or the SPMC during the initialization of their execution context during a primary cold boot (see 18.2 *Power Management*). The callee returns DENIED if this function is invoked post-initialization on the primary PE or at any time on a secondary PE.

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 5.3 *Register state*.

18.2.3 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 fulfill 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 18.2.4 *Power Management messages*.

18.2.4 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 18.2 *Power Management* and 18.2.3 *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 physical SPs at S-EL0 about the power management operation that was invoked.
- The SPMC at EL3 uses these messages at the Secure physical FF-A instance, to inform one or more logical SPs 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 5.1 and Table 5.4.

- 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 7.4 *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 13.11 *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 as described in Table 18.6 and Table 18.7.

Table 18.6: Power management request message encoding for PSCI functions

Register	Parameter
w0	FFA_MSG_SEND_DIRECT_REQ Function ID (0x8400006F or 0xC400006F).
w1	• Sender and Receiver endpoint IDs. – Bit[31:16]: ∗ SPMD ID in a message to the SPMC. ∗ SPMC ID in a message to a SP. – Bit[15:0]: ∗ SPMC ID in a message from the SPMD. ∗ SP ID in a message from the SPMC.
w2	• Message flags. – Bit[31] = b'1: Framework message. – Bit[30:8] = 0: Reserved (SBZ). – Bit[7:0] = b'00000000: Message for a power management operation initiated by a PSCI function.
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 (SBZ).

Table 18.7: Power management request message encoding for a warm boot

Register	Parameter
w0	0x8400006F: FFA_MSG_SEND_DIRECT_REQ Function ID
w1	• Sender and Receiver endpoint IDs. – Bit[31:16]: ∗ SPMD ID in a message to the SPMC. ∗ SPMC ID in a message to a SP. – Bit[15:0]: ∗ SPMC ID in a message from the SPMD. ∗ SP ID in a message from the SPMC.
w2	• Message flags. – Bit[31] = b'1: Framework message. – Bit[30:8] = 0: Reserved (SBZ). – Bit[7:0] = b'00000001: Message for a warm boot.
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-x17	Reserved (SBZ).

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-w7* registers as described in Table 18.8.

Table 18.8: Power management response message encoding

Register	Parameter
w0	0x84000070: FFA_MSG_SEND_DIRECT_RESP Function ID
w1	• Sender and Receiver endpoint IDs. – Bit[31:16]: ∗ SPMC ID in a message to the SPMD. ∗ SP ID in a message to the SPMC. – Bit[15:0]: ∗ SPMD ID in a message from the SPMC. ∗ SPMC ID in a message from a SP.
w2	• Message flags. – Bit[31] = b'1: Framework message. – Bit[30:8] = 0: Reserved (SBZ). – Bit[7:0] = b'00000010: Response message to indicate return status of the last power management request message.
w3	Return error code SUCCESS or DENIED as defined in [11].
w4-w7	Reserved (SBZ).

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.
- The SPMC denies a request from an SP to switch to the Normal world during message processing.
- An SP is run in the SPMC scheduled mode during message processing (see 9.2.3 *CPU cycle allocation modes*).
- The runtime model for direct messaging is used during message processing (see 8.3 *Runtime model for Direct request ABIs*) with the following additional restrictions. The SPMC denies any such request.
  - An SP does not use the smc(FFA_RUN) transition to allocate CPU cycles to any other component.
  - An SP does not use the smc(FFA_YIELD) transition to relinquish control back to the SPMC.
  - An SP does not use the Direct request interfaces to send a message and allocate CPU cycles to any other component.
- The SPMD denies a request from the SPMC to switch to the Normal world during message processing.

Figure 18.4 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.

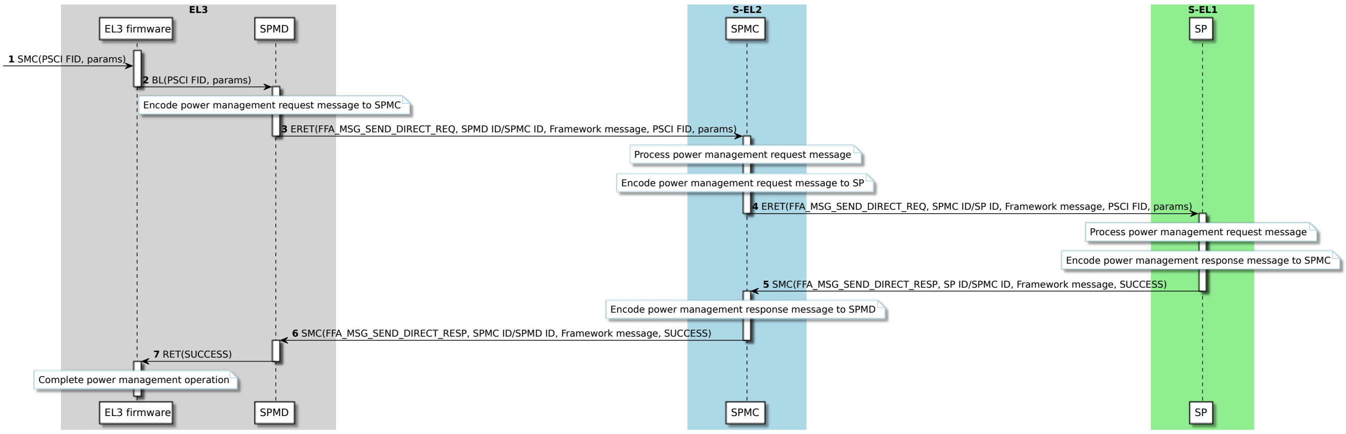

Figure 18.4: Example power management message usage

18.3 VM availability signaling

18.3.1 Overview

An SP could provide services to VMs in the Normal world. A VM could be created by the Hypervisor at runtime, access an SP's services and be destroyed by the Hypervisor when its work is complete. Alternatively, a VM could crash and its resources reclaimed by the Hypervisor. An SP could allocate resources when a VM is created and de-allocate them when the VM is destroyed. Alternatively, it could perform some IMPLEMENTATION DEFINED actions in response to one or both events. In either case, the SP needs to know when a VM is created or destroyed. To cater for this use case, the Framework specifies a mechanism that enables the Hypervisor to inform an SP when it creates or destroys a VM. This mechanism consists of,

1. Framework messages to signal and acknowledge VM creation and destruction. These messages are defined in 18.3.2 *VM availability messages*.
2. A discovery mechanism that enables an SP to subscribe to receipt of VM creation and destruction messages. This mechanism is described in 18.3.3 *Discovery and setup*

18.3.2 VM availability messages

The Framework defines the following messages to enable the Hypervisor inform an SP about VM availability.

1. A pair of Framework messages to signal and acknowledge VM creation. These messages are defined in 18.3.2.4 *VM creation message*.
2. A pair of Framework messages to signal and acknowledge VM destruction. These messages are defined in 18.3.2.5 *VM destruction message*.

18.3.2.1 SPMC responsibilities

The SPMC is responsible for ensuring that these messages are,

1. Validated as per the responsibilities associated with all Direct messages listed in 7.4 *Direct messaging usage*.
2. Exchanged only between valid senders and recipients i.e. these messages are not exchanged,
   1. Between SPs.
   2. Between an SP and a VM.
   3. Between the Hypervisor and SPMC.
   4. Between the Hypervisor and an SP if the SP has not subscribed for receipt of VM creation and destruction messages (also see 18.3.3 *Discovery and setup*).

   The SPMC returns INVALID_PARAMETERS if an invalid message exchange is attempted.

18.3.2.2 Hypervisor responsibilities

The Hypervisor is responsible for ensuring that these messages are,

1. Sent to each SP that has subscribed to them.
2. Validated as per the responsibilities associated with all Direct messages listed in 7.4 *Direct messaging usage*.
3. Exchanged only between valid senders and recipients i.e. these messages are not exchanged,
   1. Between VMs.
   2. Between a VM and an SP.

   The Hypervisor returns INVALID_PARAMETERS if an invalid message exchange is attempted.

18.3.2.3 VM availability state machine

An SP maintains a state machine to track availability of each VM. State transitions are effected through the receipt of VM creation and destruction messages. The states are described below. The state machine is described in Table 18.9.

1. *VM available*.
   1. The SP is aware of the presence of the VM and ready to communicate with it. The SP has discarded any state associated with a previous instance of this VM.
2. *VM unavailable*.
   1. The SP is aware that the VM has been either destroyed or has not yet been created. In the former case, it might not have freed all resources associated with the VM.
3. *Error*.
   1. The Hypervisor has requested an invalid state transition or sent an invalid message to the SP. The SP has returned an error response to the Hypervisor.

Table 18.9: VM availability state transition diagram

State/Transition	VM creation message	VM destruction message
VM available	Error	VM unavailable
VM unavailable	VM available	Error
Error	Error	Error

18.3.2.4 VM creation message

Table 18.10: Message to signal VM creation

Register	**Parameter**
w0	0x8400006F: FFA_MSG_SEND_DIRECT_REQ Function ID
w1	• Sender and Receiver endpoint IDs. – Bit[31:16]: ∗ Hypervisor ID. – Bit[15:0]: ∗ SP ID.
w2	• Message flags. – Bit[31] = b'1: Framework message. – Bit[30:8] = 0: Reserved (SBZ). – Bit[7:0] = b'00000100: Message to signal creation of a VM .
w3/w4	• Globally unique Handle to identify a memory region that contains IMPLEMENTATION DEFINED information associated with the created VM. • The invalid memory region handle must be specified by the Hypervisor if this field is not used.
w5	• Bit[31:16]: Reserved (SBZ). • Bit[15:0]: ID of VM that has been created.
w6	• Reserved (SBZ).
w7	• Reserved (SBZ).

Table 18.11: Message to acknowledge VM creation

Register	Parameter
w0	0x84000070: FFA_MSG_SEND_DIRECT_RESP Function ID
w1	• Sender and Receiver endpoint IDs. – Bit[31:16]: ∗ SP ID. – Bit[15:0]: ∗ Hypervisor ID.
w2	• Message flags. – Bit[31] = b'1: Framework message. – Bit[30:8] = 0: Reserved (SBZ). – Bit[7:0] = b'00000101: Message to acknowledge creation of a VM.
w3	• SP return status code. – 0: SUCCESS. ∗ The SP acknowledges successful receipt of VM creation message by transitioning to the *VM available* state. – -2: INVALID_PARAMETERS. ∗ One or more parameters were incorrectly encoded. ∗ One or more parameters contain invalid values. ∗ The SP transitions to the *Error* state. – -5: INTERRUPTED. ∗ The SP was interrupted by a Non-secure interrupt. It performed a managed exit before handling the message. The Hypervisor should resend the message to resume SP execution. This enables the SP to finish handling the VM creation message. ∗ The SP remains in the *VM unavailable* state. – -6: DENIED. ∗ The SP cannot acknowledge successful receipt of VM creation message due to an IMPLEMENTATION DEFINED reason. ∗ The SP remains in its current state. – -7: RETRY. ∗ The SP is in an IMPLEMENTATION DEFINED state that prevents it from acknowledging the VM creation message. The Hypervisor should resend the VM creation message. ∗ The SP remains in the *VM unavailable* state.
w4	• Reserved (SBZ).
w5	• Reserved (SBZ).
w6	• Reserved (SBZ).
w7	• Reserved (SBZ).

18.3.2.5 VM destruction message

Table 18.12: Message to signal VM destruction

Register	Parameter
w0	0x8400006F: FFA_MSG_SEND_DIRECT_REQ Function ID

Register	Parameter
w1	• Sender and Receiver endpoint IDs. – Bit[31:16]: ∗ Hypervisor ID. – Bit[15:0]: ∗ SP ID.
w2	• Message flags. – Bit[31] = b'1: Framework message. – Bit[30:8] = 0: Reserved (SBZ). – Bit[7:0] = b'00000110: Message to signal destruction of a VM.
w3/w4	• Globally unique Handle to identify a memory region that contains IMPLEMENTATION DEFINED information associated with the destroyed VM. • The invalid memory region handle must be specified by the Hypervisor if this field is not used.
w5	• Bit[31:16]: Reserved (SBZ). • Bit[15:0]: ID of VM that has been destroyed.
w6	• Reserved (SBZ).
w7	• Reserved (SBZ).

Table 18.13: Message to acknowledge VM destruction

Register	Parameter
w0	0x84000070: FFA_MSG_SEND_DIRECT_RESP Function ID
w1	• Sender and Receiver endpoint IDs. – Bit[31:16]: ∗ SP ID. – Bit[15:0]: ∗ Hypervisor ID.
w2	• Message flags. – Bit[31] = b'1: Framework message. – Bit[30:8] = 0: Reserved (SBZ). – Bit[7:0] = b'00000111: Message to acknowledge destruction of a VM.

Register	Parameter
w3	• SP return status code. – 0: SUCCESS. ∗ The SP acknowledges successful receipt of VM destruction message by transitioning to the “VM unavailable” state. – -2: INVALID_PARAMETERS. ∗ One or more parameters were incorrectly encoded. ∗ One or more parameters contain invalid values. ∗ The SP transitions to the *Error* state. – -5: INTERRUPTED. ∗ The SP was interrupted by a Non-secure interrupt. It performed a managed exit before handling the message. The Hypervisor should resend the message to resume SP execution. This enables the SP to finish handling the VM destruction message. ∗ The SP remains in the *VM available* state. – -6: DENIED. ∗ The SP cannot acknowledge successful receipt of VM destruction message due to an IMPLEMENTATION DEFINED reason. ∗ The SP remains in its current state. – -7: RETRY. ∗ The SP is in an IMPLEMENTATION DEFINED state that prevents it from acknowledging the VM destruction message. The Hypervisor should resend the VM destruction message. ∗ The SP remains in the *VM available* state.
w4	• Reserved (SBZ).
w5	• Reserved (SBZ).
w6	• Reserved (SBZ).
w7	• Reserved (SBZ).

18.3.3 Discovery and setup

An SP informs the SPMC that it wants to receive VM creation and/or destruction messages through its manifest (see Table 5.1).

The Hypervisor discovers that an SP wants to receive VM creation and/or destruction messages by retrieving the SP properties through the FFA_PARTITION_INFO_GET ABI (see Table 6.1).

18.4 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 now 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 7.2. 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.

18.4.1 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 18.4.1.2 *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 18.15.
  - 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 18.16.
- Successful completion of this function call is indicated as follows.
  - *w0* contains *FFA_SUCCESS* function ID.
  - *w1/x1-w7/x7* are Reserved (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 18.17.
  - See 18.4.1.1 *Target availability notification* for behavior when BUSY is returned and caller must be notified about availability of TX buffer.

Table 18.15: FFA_MSG_SEND instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure virtual	SMC, HVC, ERET

Table 18.16: 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).
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.

Parameter	Register	Value
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 18.17: 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.

18.4.1.1 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 7.2.2.4 *Buffer synchronization*). The callee must complete the interface invocation with a *BUSY* error code in this case.

A VM running in EL1 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 7.2.2.4 *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 7.2.2.4 *Buffer synchronization* for example, through the **FFA_RX_RELEASE** interface.

18.4.1.2 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 7.1.1 *Indirect messaging*.

18.4.1.2.1 Sender VM responsibilities

1. Must acquire ownership of empty TX buffer (see 7.2.2.4 *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 18.4.1.2.2 *Hypervisor responsibilities* for Hypervisor responsibilities in this message transmission.

18.4.1.2.2 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 18.4.1.3 *Legacy mechanism for scheduler notification*.
12. Must return *SUCCESS* to *Sender* if message is successfully transmitted.
13. Must mark the RX buffer as available when the Receiver releases it.

18.4.1.2.3 Receiver VM responsibilities

1. Copy message from RX buffer.
2. Transfer ownership of the RX buffer by invoking the *FFA_RX_RELEASE* interface.

18.4.1.3 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.

18.4.2 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 18.19.
- Syntax of this function is described in Table 18.20.
- Successful completion of this function is indicated through the invocation of the FFA_MSG_SEND interface (see 18.4.1 *FFA_MSG_SEND*).
- Encoding of error code in the FFA_ERROR function is described in Table 18.21.

Table 18.19: FFA_MSG_POLL instances and conduits

Config No.	FF-A instance	Valid conduits
1	Non-secure virtual	SMC, HVC

Table 18.20: FFA_MSG_POLL function syntax

Parameter	Register	Value
uint32 Function ID	w0	• 0x8400006A.
Other Parameter registers	w1-w7 x1-x7	• Reserved (MBZ).

Table 18.21: 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.

18.5 Changes to FF-A v1.0 data structures for forward compatibility

Version 1.1 of the Framework specifies changes to make the following data structures defined in version 1.0 of the Framework forwards compatible.

1. Memory transaction descriptor (see [1] for more information).
2. Endpoint memory access descriptor (see [1] for more information).
3. Endpoint RX/TX descriptor in Table 13.27.
4. Partition information descriptor in Table 6.1.

These changes enable forward compatibility as described below.

1. A new field is always added at the end these data structures.
2. A producer of this data structure specifies its size corresponding to the FF-A version it implements to a consumer.
3. A consumer of this data structure uses this size to correctly read the version of the data structure implemented by the producer.
4. A consumer of this data structure uses the size corresponding to the Framework version it implements to consume only fields defined in its version. Additional fields in the producer's version of this data structure are safely ignored enabling forward compatibility.

18.5.1 Changes to Partition information descriptor

The *Partition information descriptor* (see Table 6.1) has undergone the following changes based upon partner feedback.

1. The *Partition UUID* field has been added at the 8-byte offset. This enables the caller of FFA_PARTITION_INFO_GET with the Nil UUID to determine the UUIDs of all the endpoints deployed in the system.

2. New flags corresponding to properties of a partition have been added in Bits[8:4] of the *Partition properties* field of the FF-A v1.0 descriptor (see Table 18.22).

To ensure that changes to this data structure in future versions of the Framework can be introduced in a forward compatible manner, the *Size* parameter has been added to the return parameters of FFA_PARTITION_INFO_GET as described in Table 13.35. This enables a consumer of Table 6.1 to determine the size of the version of this data structure used by the producer as described above.

Table 18.22: FF-A v1.0 Partition information descriptor

Field	Byte length	Byte offset	Description
Partition ID	2	0	• 16-bit ID of the partition.
Execution context count	2	2	• Number of execution contexts implemented by this partition (also see 4.7 *Execution context*).

Field	Byte length	Byte offset	Description
Partition properties	4	4	• Flags to determine partition properties. – 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 7.4 *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[31:3]: Reserved (MBZ).

18.5.2 Changes to Endpoint RX/TX descriptor

The changes for the *Endpoint RX/TX descriptor* (see Table 13.27) are listed below.

1. All fields in FF-A v1.0 descriptor (see Table 18.23) except for the *Endpoint ID* field at the 0-byte offset and the *Reserved* field at the 2-byte offset have been replaced by the following fields.
    1. *RX buffer memory region description offset* at the 4-byte offset.
    2. *TX buffer memory region description offset* at the 8-byte offset.

    These changes enable new fields to be added to this data structure in future in a forward compatible manner. They also enable reuse of the composite memory region descriptor (see [1] instead of duplicating the same functionality.

Table 18.23: FF-A v1.0 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.
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 the constituent memory region descriptor in [1] 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 the constituent memory region descriptor in [1] for how the address ranges are encoded.

18.5.3 Compatibility requirements for FF-A v1.0 data structures

A partition manager that implements major version 1 and a higher minor version (>= 1) of the Framework implements the v1.0 version of the data structures described in *18.5 Changes to FF-A v1.0 data structures for forward compatibility* to maintain backward compatibility with a client that implements version 1.0 of the Framework. Each client specifies its version of the Framework in an invocation of FFA_VERSION (see 13.2 *FFA_VERSION*). For each client, the partition manager ensures it only uses that version of the data structure that is implemented by the client.

A partition manager that implements major version 1 and a higher minor version (>=1) of the Framework and cannot maintain backwards compatibility returns the NOT_SUPPORTED error code in response to an FFA_VERSION invocation with v1.0 as the input version. Such a partition manager implementation is not compliant with the specification as it violates the requirement to maintain backward compatibility with a client that implements the same major version.

18.6 Example notification and in-direct messaging flows

Provisional

The flows described in this section are to facilitate the understanding of the notification and indirect messaging mechanisms. The format of these diagrams are not yet finalized and will be updated in a subsequent revision of the specification.

18.6.1 Example notification flows

These flow diagrams illustrate how a notification is signalled between the following combinations of endpoints with the Hypervisor acting as the primary scheduler.

- A VM sets up and sends a notification to another VM (see 18.6.1.1 *VM to VM notification*).
- An SP sets up and sends a notification to a VM (see 18.6.1.2 *SP to VM notification*).
- A VM sets up and sends a notification to an SP (see 18.6.1.3 *VM to SP notification*).

The bind procedure is not represented in the flows but the FFA_NOTIFICATION_BIND ABI must be called by the receiver of a notification to allow the sender to raise it (see 16.3 *FFA_NOTIFICATION_BIND* and 10.4.2 *Notification binding*.

The sequences are simplified and the details in each flow are provided earlier (see 10.5 *Notification signaling*).

18.6.1.1 VM to VM notification

Figure 18.5 illustrates an example sequence where VM1 sends a notification to VM0.

1. VM1 uses the FFA_NOTIFICATION_SET interface to send notification 10 to VM0.
2. Hypervisor injects the virtual notification pending interrupt into VM0.
3. VM0 handles the notification pending interrupt and uses the FFA_NOTIFICATION_GET interface to retrieve its pending notifications including notification 10.

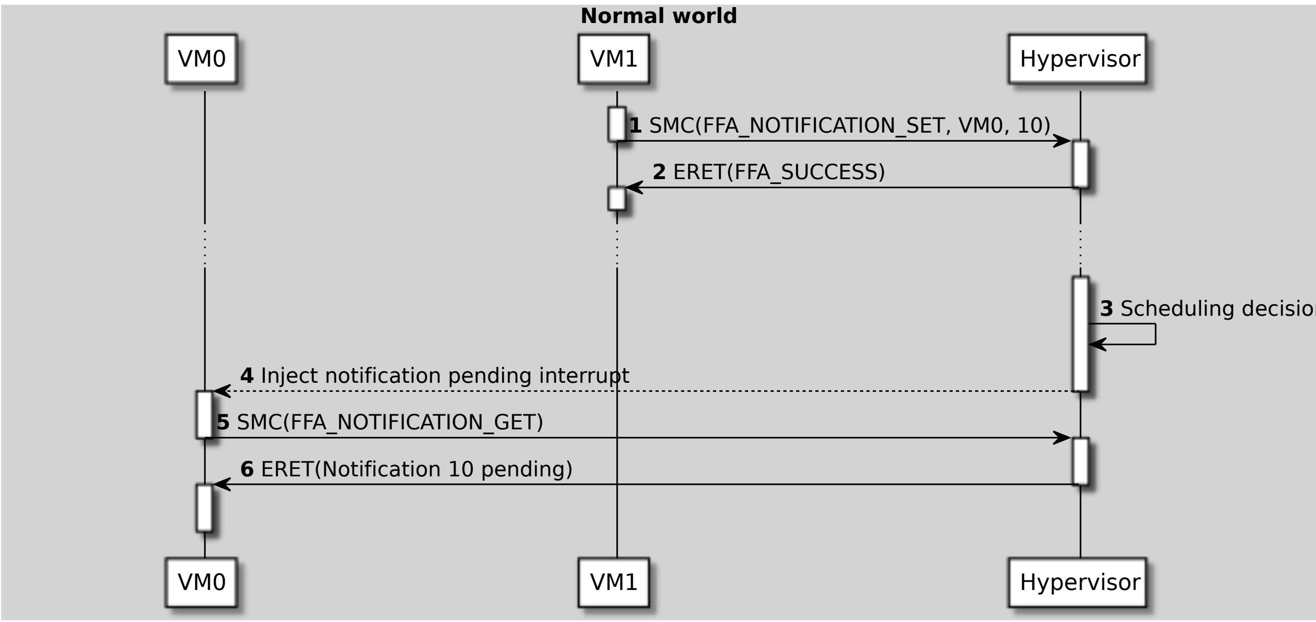

Figure 18.5: VM to VM notification flow

18.6.1.2 SP to VM notification

Figure 18.6 illustrates an example sequence where SP0 sends a notification to VM0 as follows.

1. SP0 uses the FFA_NOTIFICATION_SET interface to send notification 5 to VM0.
2. SPMC pends the schedule receiver interrupt for the Hypervisor.
3. Hypervisor uses the FFA_NOTIFICATION_INFO_GET to retrieve the list of endpoints with currently pending notifications.
4. Hypervisor injects the virtual notification pending interrupt into VM0.
5. VM0 handles the notification pending interrupt and uses the FFA_NOTIFICATION_GET interface to retrieve its pending notifications including notification 5.

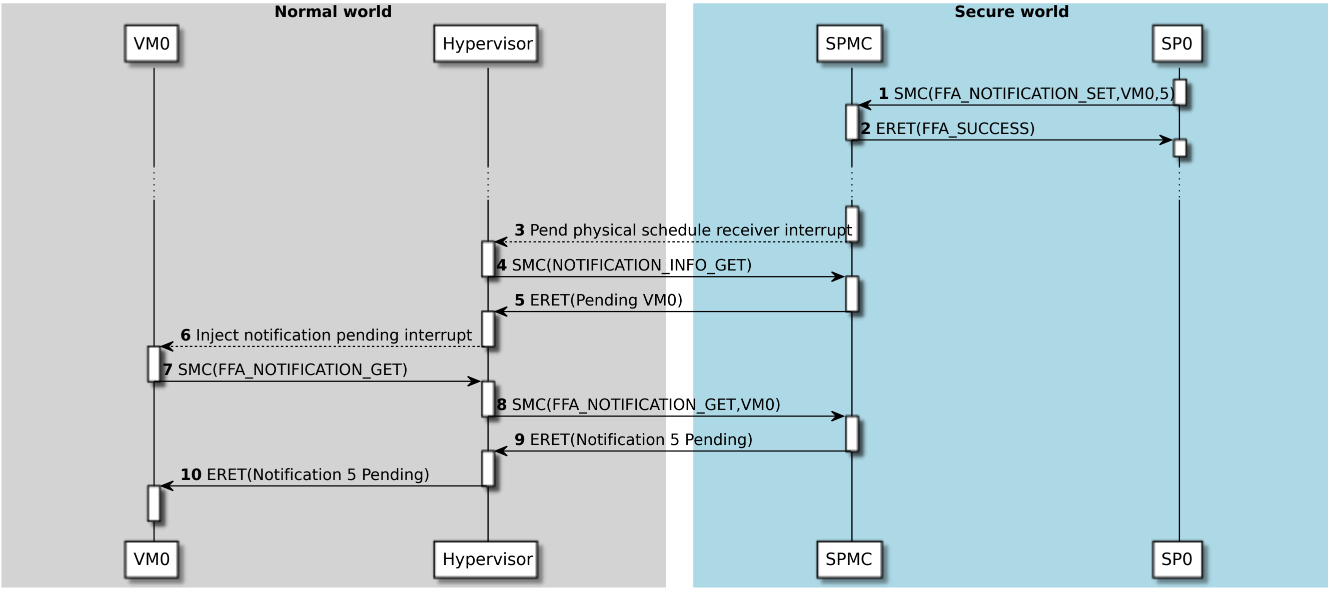

Figure 18.6: SP to VM notification flow

18.6.1.3 VM to SP notification

Figure 18.7 illustrates an example sequence where VM0 sends a notification to SP0 in S-EL1 where VM1 is responsible for scheduling SP0 i.e. it implements SP0's secondary scheduler.

1. VM0 uses the FFA_NOTIFICATION_SET interface to send notification 8 to SP0.
2. SPMC pends the physical schedule receiver interrupt for the Hypervisor to inform it of pending notifications.
3. Hypervisor uses the FFA_NOTIFICATION_INFO_GET interface to retrieve the list of endpoints with pending notifications from the SPMC including SP0.
4. Hypervisor injects the virtual schedule receiver interrupt into VM1.
5. VM1 uses the FFA_NOTIFICATION_INFO_GET interface to retrieve the list of endpoints with pending notifications including SP0.
6. VM1 uses the FFA_MSG_SEND_DIRECT_REQ interface with an IMPLEMENTATION DEFINED message to SP0 to provide it CPU cycles and inform it that it has a notification to handle.
7. SP0 receives the direct message and uses the FFA_NOTIFICATION_GET interface to retrieve the bitmap of pending VM notifications including notification 8.
8. SP0 does the IMPLEMENTATION DEFINED work related to notification 8 and uses the FFA_MSG_SEND_DIRECT_RESP interface with an IMPLEMENTATION DEFINED content to inform VM1 that it has handled its notifications.

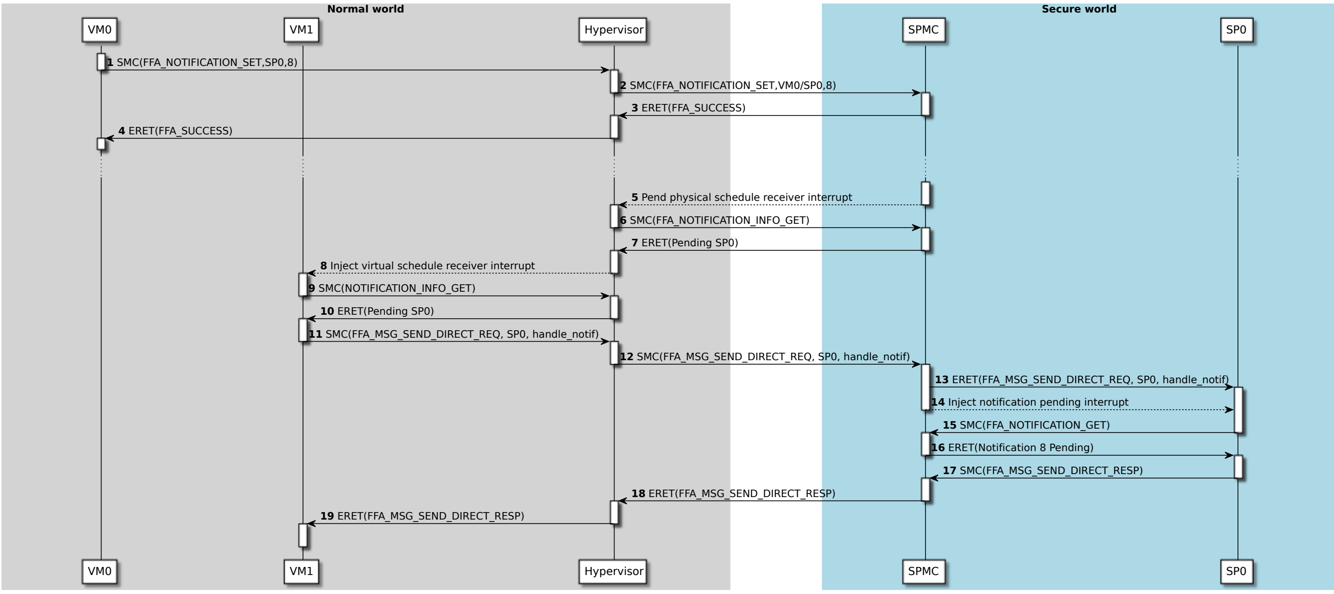

Figure 18.7: VM to SP notification flow

18.6.2 Example indirect messaging flows

These flow diagrams illustrate how an indirect message is transmitted between the following combinations of endpoints with the Hypervisor acting as the primary scheduler.

- A VM sending an indirect message to another VM (see 18.6.2.1 *VM to VM indirect message flow*).
- A VM sending an indirect message to an SP (see 18.6.2.2 *VM to SP indirect message flow*).
- An SP sending an indirect message to a VM (see 18.6.2.3 *SP to VM indirect message flow*).

The sequences are simplified and the details in each flow are provided in earlier sections (see 7.3 *Indirect messaging usage*).

18.6.2.1 VM to VM indirect message flow

Figure 18.8 illustrates an example sequence VM1 sends an indirect message to VM0 as follows.

1. VM1 populates a message in its TX buffer and uses the FFA_MSG_SEND2 interface to send an indirect message to VM0.
2. If a system supports sending indirect messages from an SP to a VM the Hypervisor uses the FFA_RX_ACQUIRE interface to obtain the ownership of the RX buffer of VM0 from the SPMC (see 7.2.2.4.3 *Management of buffer ownership between Hypervisor and SPMC*).
3. Hypervisor copies the message from the TX buffer of VM1 to the RX buffer of VM0.
4. Hypervisor injects a notification pending interrupt in VM0.
5. VM0 handles the notification pending interrupt and uses the FFA_NOTIFICATION_GET interface to retrieve its pending notifications.
6. Hypervisor returns the bitmap of notifications that are pending for VM0 including the RX buffer full notification.
7. VM0 copies the data from its RX buffer and releases ownership to the Hypervisor (see 7.2.2.4.2 *Transfer of buffer ownership*).
8. If the Hypervisor obtained ownership of the RX buffer of VM, it uses the FFA_RX_RELEASE interface to release ownership back to the SPMC.

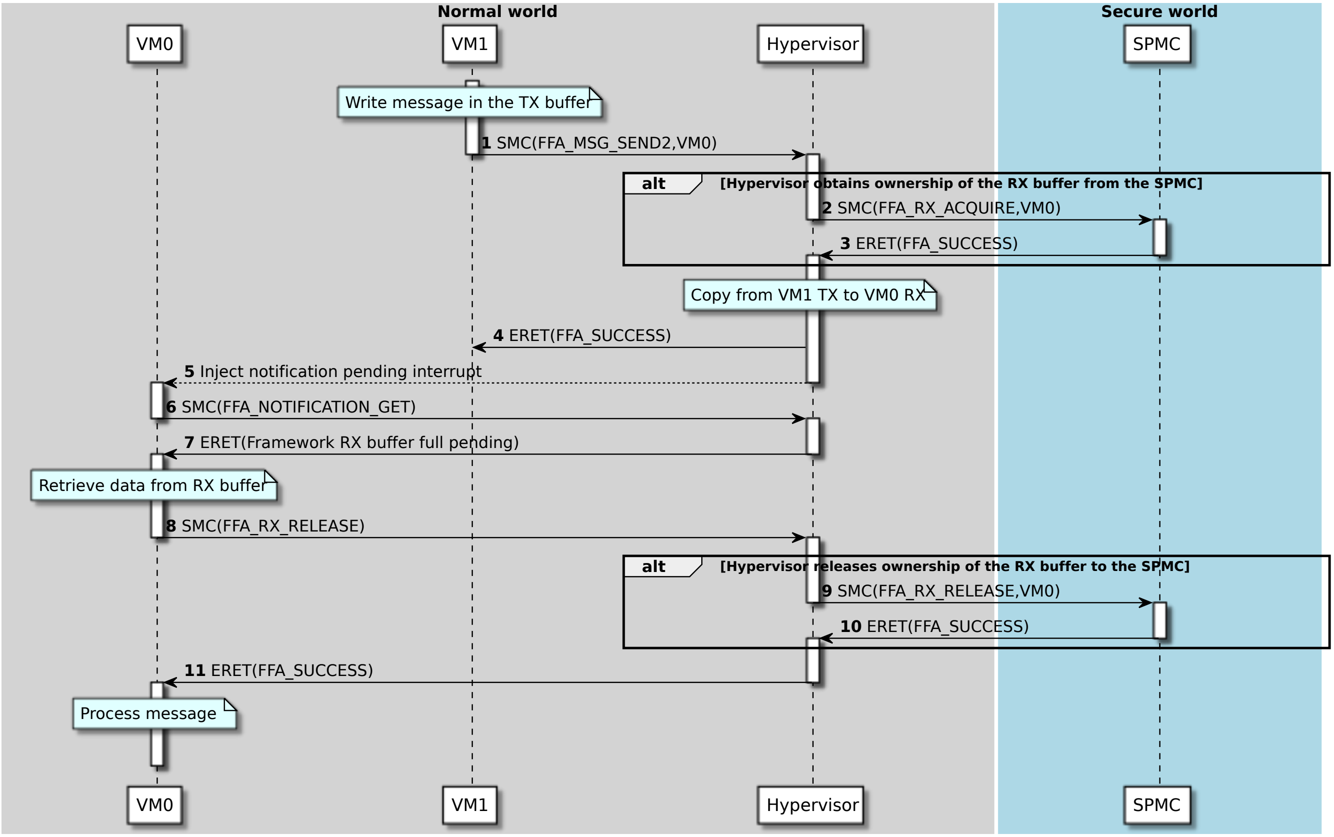

Figure 18.8: VM to VM indirect message flow

18.6.2.2 VM to SP indirect message flow

Figure 18.9 illustrates an example sequence where VM0 sends an indirect message to SP0 in S-EL1 where VM1 is responsible for scheduling SP0 i.e. it implements SP0's secondary scheduler.

1. VM0 writes a message in its TX buffer and uses the FFA_MSG_SEND2 interface to send an indirect message to SP0.
2. SPMC copies the message from the TX buffer of VM0 to the RX buffer of SP0 and returns success to the Hypervisor
3. SPMC pends the physical schedule receiver interrupt for the Hypervisor to inform it of pending notifications.
4. Hypervisor handles the schedule receiver interrupt and uses the FFA_NOTIFICATION_INFO_GET interface to retrieve the list of endpoints with pending notifications from the SPMC including SP0.
5. Hypervisor injects the virtual schedule receiver interrupt into VM1.
6. VM1 handles the scheduler receiver interrupt and uses the FFA_NOTIFICATION_INFO_GET interface to retrieve the list of endpoints with pending notifications including SP0.
7. VM1 uses the FFA_MSG_SEND_DIRECT_REQ interface with an IMPLEMENTATION DEFINED content to provide CPU cycles to SP0 and inform it that is has notifications to handle.
8. SP0 use the FFA_NOTIFICATION_GET interface to retrieve the list of pending notifications including the RX buffer full notification.
9. SP0 processes the data from its RX buffer and uses the FFA_RX_RELEASE interface to return ownership of its RX buffer to the SPMC.
10. SP0 uses the FFA_MSG_SEND_DIRECT_RESP interface to inform VM1 it has finished processing its notifications.

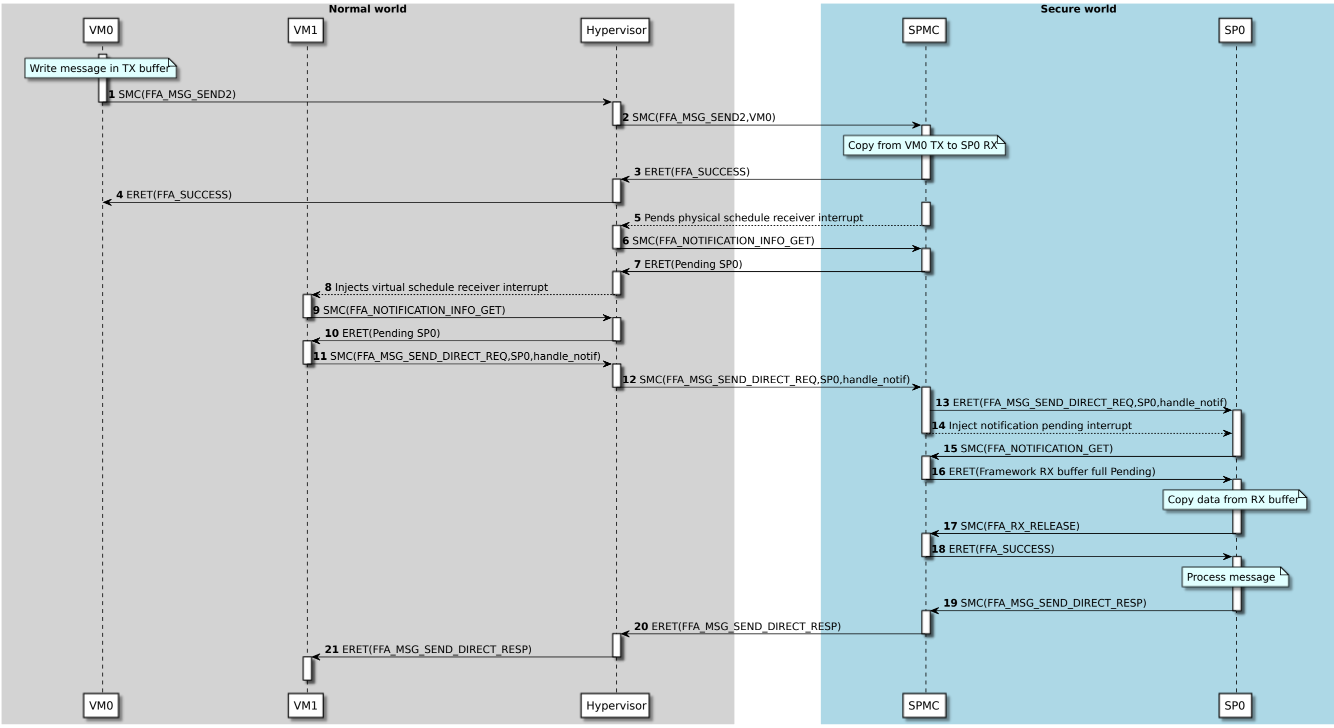

Figure 18.9: VM to SP indirect message flow

18.6.2.3 SP to VM indirect message flow

Figure 18.10 illustrates an example sequence where SP0 sends an indirect message to a VM0 as follows.

1. SP0 writes a message in its TX buffer and uses the FFA_MSG_SEND2 interface to send an indirect message to VM0.
2. SPMC copies the message from the RX buffer of SP0 to the TX buffer of VM0.
3. SPMC pends the physical schedule receiver interrupt for the Hypervisor.
4. Hypervisor handles the schedule receiver interrupt and uses the FFA_NOTIFICATION_INFO_GET interface to retrieve the list of endpoints with pending notifications from the SPMC including VM0.
5. Hypervisor injects the notification pending interrupt in VM0.
6. VM0 uses the FFA_NOTIFICATION_GET interface to retrieve the list of pending notifications.
7. Hypervisor uses the FFA_NOTIFICATION_GET interface to retrieve the list of pending notifications for VM0 from the SPMC including the RX buffer full notification.
8. Hypervisor returns the bitmaps of notifications that are pending for VM0 including the RX buffer full notification.
9. VM0 copies the data from its RX buffer and releases ownership to the Hypervisor (see 7.2.2.4.2 *Transfer of buffer ownership*).
10. Hypervisor uses the FFA_RX_RELEASE interface to return the ownership of the RX buffer of VM0 to the SPMC.

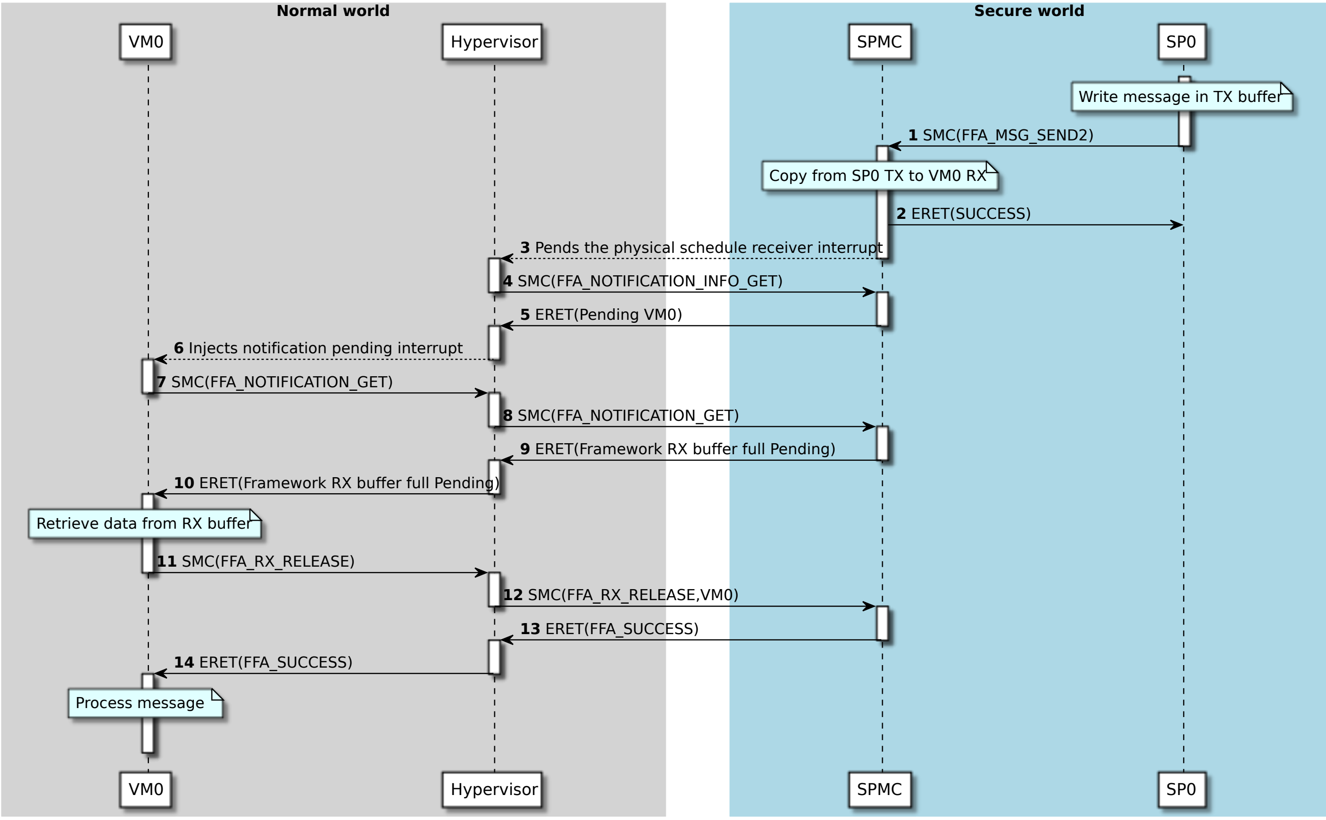

Figure 18.10: SP to VM indirect message flow

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

ME

Managed exit

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

SBZ

Should be zero

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