BETA
This is a BETA version of the specification
This version is an engineering draft of the full specification. It includes majority of features but must not be treated as being complete. It is meant to obtain feedback from ARM partners and internally within ARM. It should be treated as a work in progress and is subject to change on the basis of this feedback. This document is confidential until released.
6.1 Overview
6.2 Device access management
6.3 Ownership and access attributes
   6.3.1 Ownership and access rules
   6.3.2 Ownership and access states
6.4 Memory management transactions
   6.4.1 Component roles
   6.4.2 Component configurations
   6.4.3 Transaction life-cycle
6.5 Donate memory transaction
   6.5.1 Donate memory state machine
   6.5.2 Donate memory transaction lifecycle
6.6 Lend memory transaction
   6.6.1 Lend memory transaction state machine
   6.6.2 Lend memory transaction lifecycle
6.7 Share memory transaction
   6.7.1 Share memory transaction state machine
   6.7.2 Share memory transaction lifecycle
6.8 Relinquish memory transaction
   6.8.1 Relinquish memory access state machine
   6.8.2 Relinquish memory transaction lifecycle
6.9 Memory region description
6.10 Transmission of transaction data

7 Interface overview

8 Status reporting interfaces
  8.1 SPCI_ERROR
  8.2 SPCI_SUCCESS
  8.3 SPCI_INTERRUPT

9 Setup and discovery interfaces
  9.1 SPCI_VERSION
  9.1.1 Overview
  9.2 SPCI_FEATURES
  9.3 SPCI_RX_RELEASE
  9.4 SPCI_RXTX_MAP
  9.5 SPCI_RXTX_UNMAP
  9.6 SPCI_PARTITION_INFO_GET
  9.7 SPCI_ID_GET
  9.8 SPCI_MSG_POLL

10 CPU cycle management interfaces
  10.1 SPCI_MSG_WAIT
    10.1.1 Component responsibilities for SPCI_MSG_WAIT
  10.2 SPCI_YIELD
    10.2.1 Component responsibilities for SPCI_MSG_YIELD
  10.3 SPCI_RUN
    10.3.1 Component responsibilities for SPCI_RUN

11 Messaging interfaces
  11.1 SPCI_MSG_SEND
    11.1.1 Target availability notification
    11.1.2 Component responsibilities for SPCI_MSG_SEND
    11.1.3 Mechanism for scheduler notification
  11.2 SPCI_MSG_SEND_DIRECT_REQ
Secure Partition Client Interface Specification

11.2.1 Component responsibilities for SPCI_MSG_SEND DIRECT_REQ 108
11.3 SPCI_MSG_SEND DIRECT RESP 110
11.3.1 Component responsibilities for SPCI_MSG_SEND DIRECT RESP 111

12 Memory management interfaces 113
  12.1 SPCI_MEM_DONATE 113
     12.1.1 Component responsibilities for SPCI_MEM_DONATE 114
  12.2 SPCI_MEM_LEND 117
     12.2.1 Component responsibilities for SPCI_MEM_LEND 118
  12.3 SPCI_MEM_SHARE 121
     12.3.1 Component responsibilities for SPCI_MEM_SHARE 122
  12.4 SPCI_MEM_RETRIEVE_REQ 125
     12.4.1 Component responsibilities for SPCI_MEM_RETRIEVE_REQ 128
  12.5 SPCI_MEM_RETRIEVE_RESP 131
     12.5.1 Component responsibilities for SPCI_MEM_RETRIEVE_RESP 133
  12.6 SPCI_MEM_RELINQUISH 134
     12.6.1 Component responsibilities for SPCI_MEM_RELINQUISH 135
  12.7 SPCI_MEM_RECLAIM 136
     12.7.1 Component responsibilities for SPCI_MEM_RECLAIM 136

13 Appendix 139
  13.1 S-EL0 partitions 139
     13.1.1 UEFI PI Standalone Management Mode partitions 139
## Release information

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 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 | • 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 above changes |
| 2018/Dec/21| Alpha 2 | • Changed content significantly based upon partner feedback since Alpha 1  
            • There is a clear separation between message passing and scheduling  
            • Introduced use of RX/TX buffers to enable message passing |
Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

**THIS 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.** 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, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

**TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.**

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at [http://www.arm.com/company/policies/trademarks](http://www.arm.com/company/policies/trademarks).

Copyright © 2019 Arm Limited (or its affiliates). All rights reserved.


110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349
1 About this document

1.1 Terms and abbreviations

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABI</td>
<td>Application Binary Interface</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>FF</td>
<td>Firmware Framework</td>
</tr>
<tr>
<td>GIC</td>
<td>Generic Interrupt Controller</td>
</tr>
<tr>
<td>HVC</td>
<td>Hypervisor Call</td>
</tr>
<tr>
<td>MBP</td>
<td>Must be preserved</td>
</tr>
<tr>
<td>MBZ</td>
<td>Must be zero</td>
</tr>
<tr>
<td>MM</td>
<td>Management Mode</td>
</tr>
<tr>
<td>MMIO</td>
<td>Memory Mapped Input Output</td>
</tr>
<tr>
<td>MP</td>
<td>Multi-processing</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PE</td>
<td>Processing Element</td>
</tr>
<tr>
<td>PPI</td>
<td>Private Peripheral Interrupt</td>
</tr>
<tr>
<td>PSA</td>
<td>Platform Security Architecture</td>
</tr>
<tr>
<td>SGI</td>
<td>Software Generated Interrupt</td>
</tr>
<tr>
<td>SMC</td>
<td>Secure Monitor Call</td>
</tr>
<tr>
<td>SMCCC</td>
<td>SMC Calling Convention</td>
</tr>
<tr>
<td>SMMU</td>
<td>System Memory Management Unit</td>
</tr>
<tr>
<td>SP</td>
<td>Secure Partition</td>
</tr>
<tr>
<td>SPCI</td>
<td>Secure Partition Client Interface</td>
</tr>
<tr>
<td>SPI</td>
<td>Shared Peripheral Interrupt</td>
</tr>
<tr>
<td>SPM</td>
<td>Secure Partition Manager</td>
</tr>
<tr>
<td>SPRT</td>
<td>Secure Partition Run Time</td>
</tr>
<tr>
<td>STMM</td>
<td>Standalone Management Mode</td>
</tr>
<tr>
<td>SVC</td>
<td>Supervisor Call</td>
</tr>
<tr>
<td>TEE</td>
<td>Trusted Execution Environment</td>
</tr>
<tr>
<td>UUID</td>
<td>Unique Universal Identifier</td>
</tr>
<tr>
<td>VCPU</td>
<td>Virtual CPU</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
</tbody>
</table>

1.2 References

This section lists publications by Arm and by third parties.

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

[1] ARM® System Memory Management Unit Architecture specification versions 3.0, 3.1 and 3.2. See https://static.docs.arm.com/ihi0070/ca/IHI_0070_C_a_System_Memory_Management_Unit_Arm_Architecture_Specification.pdf


1.3 Feedback

Arm welcomes feedback on its documentation.

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

- The title (Secure Partition Client Interface Specification).
- The document ID and version (DEN0077A 1.0).
- The page numbers to which your comments apply.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.
2 Introduction

The Armv8.4 architecture introduces the virtualization extension in the secure state. The Arm SMMU v3.2 architecture [1] adds support for stage 2 translations for secure streams to complement the secure EL2 translation regime in an Armv8.4 PE. These architectural features enable isolation of mutually mistrusting software components in the secure state from each other.

The term isolation is synonymous with the principle of least privilege: A software component must be able to access only regions in the physical address space and 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 e.g. 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
3. Normal world software can be isolated from software in S-EL1 to mitigate privilege escalation attacks

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

1. Leverages the virtualization extension to isolate software images provided by an ecosystem of vendors from each other
2. Describes interfaces that standardizes communication between the various software images. This includes communication between images in the Secure world and Normal world.

This software architecture is the firmware framework of the Platform Security Architecture in A-profile processors [2]. This framework also goes beyond the above stated goals to ensure that the interfaces can be used to standardize communication:

1. In the absence of the virtualization extension in the secure state. This provides a migration path for existing software images in the secure state to a system that implements the virtualization extension.
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]). Hence, a Hypervisor could use the firmware framework interfaces to enable communication 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].

2.1 Overview

The building blocks of the firmware framework described in this specification are listed below. Figure 1 illustrates an implementation of this framework and its components.

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

A SP typically encompasses the S-EL1 and S-EL0 exception levels. The firmware framework supports SPs that run only in S-EL0 as well. A S-EL0 SP could be managed by software in S-EL1 or EL3. This is
an implementation defined choice.

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

- In the Normal world, an endpoint could be a VM or the OS Kernel. These endpoints are called *NS-Endpoints* in scenarios where it is not necessary to distinguish between them.
- In the Secure world, an endpoint could be a SP running in S-EL0 or S-EL1 with or without virtualization extension enabled. These endpoints are called *S-Endpoints* in scenarios where it is not necessary to distinguish between them.

2. **A partition manifest** for each partition in the system. It describes the system resources a partition needs, services that a partition implements and other attributes of the partition that govern its run time behavior.

3. **A partition manager** (PM) that manages the partitions. In the Secure world it is called the *Secure Partition Manager* (SPM). In the Normal world it is a Hypervisor.

The partition manager is responsible for:

1. Initializing a partition as per the requirements stated in its manifest. The partition is isolated from other software components by the partition manager.
2. Ensuring the isolation properties are maintained during run time by policing address space accesses made by the partition.
3. Enabling communication between partitions through interfaces described in this document.

The SPM and Hypervisor are collectively referred to as the *Partition managers* in scenarios where they have the same responsibilities and it is not necessary to distinguish between them.

In an implementation of this framework, it is possible that a Hypervisor uses the framework only for communication between the Normal world and Secure world. It fulfills the responsibilities listed in 1 and 2 through implementation defined mechanisms.

4. **Application binary interfaces** that partitions can invoke at their exception level boundaries for the following purposes.

1. Discovery of a partition’s presence, its properties and services it implements.
2. Message passing amongst partitions and partition managers.
3. Memory management between partitions.

The interfaces described in this specification can be used with the following configurations of the virtualization extension in the system.

1. Virtualization is disabled in the Non-secure state and not present or disabled in the Secure state. An OS partition communicates with a single SP.
2. Virtualization is enabled in the Non-secure state and not present or disabled in the Secure state. One or more VMs communicate with each other and with a single SP.
3. Virtualization is disabled in the Non-secure state and enabled in the Secure state in Armv8.4 architecture. An OS partition communicates with one or more SPs. The SPs communicate with each other.
4. Virtualization is enabled in both security states in Armv8.4 architecture. One or more VMs communicate with each other, one or more SPs and vice versa.

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.
2.2 Document organization

The rest of this document is organized as follows.

1. Section 3 describes some fundamental concepts that are essential for understanding the firmware framework architecture.
2. Section 4 specifies the information contained in a partition manifest and how it is used to initialize a partition by a partition manager.
3. Section 5 describes the mechanisms that partitions can use for message passing.
4. Section 6 describes the mechanisms that partitions can use for managing memory amongst themselves.
5. Section 7 specifies the interfaces defined by the framework and used by framework components for communication.
6. Section 13 describes how an example use case of S-EL0 SPs can be implemented using the firmware framework.

Figure 1: Firmware framework with Secure EL2
3 Concepts

3.1 SPCI instances

A SPCI instance is a valid combination of two SPCI components at an exception level boundary. 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. Table 3 lists the SPCI instances that can exist between SPCI components. These instances are used to describe the interfaces in Section 7. An interface is accessed through a conduit described in Section 3.2. The responsibilities of the caller and callee in each interface depend upon the SPCI instance at which it is invoked.

Table 3: SPCI instances

<table>
<thead>
<tr>
<th>SPCI Instance number</th>
<th>Name</th>
<th>SPCI component at lower EL boundary</th>
<th>SPCI component at higher EL</th>
<th>Virtualization Extension support in applicable security state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Secure virtual SPCI instance</td>
<td>S-EL1/S-EL2SP</td>
<td>SP M</td>
<td>Enabled</td>
</tr>
<tr>
<td>2.</td>
<td>Secure virtual SPCI instance</td>
<td>Supervisor/S-EL2SP</td>
<td>SP M</td>
<td>Enabled</td>
</tr>
<tr>
<td>3.</td>
<td>Secure virtual SPCI instance</td>
<td>User/SupervisorSP</td>
<td>SP M</td>
<td>Not applicable</td>
</tr>
<tr>
<td>4.</td>
<td>Secure virtual SPCI instance</td>
<td>User/S-EL1SP</td>
<td>SP M</td>
<td>Not applicable</td>
</tr>
<tr>
<td>5.</td>
<td>Secure virtual SPCI instance</td>
<td>S-EL0/S-EL1SP</td>
<td>SP M</td>
<td>Not applicable</td>
</tr>
<tr>
<td>6.</td>
<td>Secure virtual SPCI instance</td>
<td>S-EL0/EL3SP</td>
<td>SP M</td>
<td>Disabled</td>
</tr>
<tr>
<td>7.</td>
<td>Secure physical SPCI instance</td>
<td>S-EL1/EL3 and/or (see Section 3.8) SPM</td>
<td>SP M</td>
<td>Disabled</td>
</tr>
<tr>
<td>8.</td>
<td>Secure physical SPCI instance</td>
<td>Supervisor/ESP and/or (see Section 3.8) SPM</td>
<td>SP M</td>
<td>Disabled</td>
</tr>
<tr>
<td>9.</td>
<td>Secure physical SPCI instance</td>
<td>Supervisor/Manager and/or (see Section 3.8) SPM</td>
<td>SP M</td>
<td>Disabled</td>
</tr>
<tr>
<td>10.</td>
<td>Secure physical SPCI instance</td>
<td>S-EL2/EL3 and/or (see Section 3.8) SPM</td>
<td>SP M</td>
<td>Enabled</td>
</tr>
<tr>
<td>11.</td>
<td>Non-secure virtual SPCI instance</td>
<td>Supervisor/EL2M</td>
<td>Hypervisor</td>
<td>Enabled</td>
</tr>
<tr>
<td>12.</td>
<td>Non-secure virtual SPCI instance</td>
<td>Supervisor/Hypervisor</td>
<td>Hypervisor</td>
<td>Enabled</td>
</tr>
</tbody>
</table>
### Provisional

**Open Topic**

The interfaces described in Section 7 have been specified with a view to support a subset of the SPCI instance combinations in Table 3. A non-exhaustive list of individual instances is provided below.

- SPCI Instance 5.
- SPCI Instance 6.
- SPCI Instance 7.
- SPCI Instance 10.
- SPCI Instance 13.
- SPCI Instance 14.
- SPCI Instance 15.

The correctness of interfaces at the instances listed above has undergone limited validation.

The interfaces are expected to work correctly at other SPCI instances but this has not been validated at the time of releasing this specification.

The specification of the semantics and syntax of these interfaces will become more robust as feedback from partners is received.

### 3.2 Conduits

The framework defines interfaces to enable communication between various SPCI components (see Section 7). Each interface is accessible through one or more conduits described below.

The SMC conduit as described in [5] should be used to invoke an interface by a SPCI 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 SPCI communication framework enables message exchange between any two SPCI components that
might be at the same or a different EL relative to each other. A request could be sent from a lower EL to a
higher EL and vice versa. Similarly, results of the request or an error status could be sent from a lower EL to
a higher EL and vice versa.

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 a SMC, HVC or SVC call is mirrored in an ERET call e.g. w0 contains a
function identifier parameter in the ERET call. This ensures that messages can be passed at any SPCI
instance irrespective of their direction of travel. An invocation through the SMC, HVC, SVC conduits is
completed through the ERET conduit. An invocation through the ERET conduit is completed through the
SMC, HVC, 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 SPCI components at an EL boundary at any SPCI instance.

The taxonomy of information transmitted through a conduit at an SPCI instance is described below. It could be:

1. An interface invocation described in Section 7
2. Results from the successful completion of the invoked interface
3. Error code from an unsuccessful completion of the invoked interface

Based upon the above taxonomy, an interface invocation through one conduit at an SPCI instance can
complete through another conduit in one of the following ways.

- An error code. The SPCI_ERROR function is used to return the error code (see Section 8.1).
- Results of the request. The SPCI_SUCCESS function is used to return the results (see Section 8.2).
- An invocation of another interface described in Section 7.

### 3.3 Execution State

The Armv8-A architecture defines two Execution states, AArch32 and AArch64 as described in [6]. The
execution states that are applicable to each SPCI component are listed below.

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

### 3.4 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 any component's translation regime is called Normal
  memory.
- Memory that is accessed with the NS bit cleared in the component translation regime is called Secure
  memory.
3.5 Memory granularity and alignment

The firmware framework specifies support to map a memory region in the translation regimes of the two SPCI components at a SPCI instance (see Section 5.2.2 & Section 6). 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 SPCI instance if a VM and the Hypervisor use translation granule sizes of 4K and 64K respectively, then 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.

Each endpoint must specify its translation granule size in its resource description as described in Section 4.1.1. The Hypervisor and SPM must use an implementation defined mechanism to specify their translation granule sizes to VMs and SPs respectively. This could be done through a platform discovery mechanism like ACPI or Device tree.

3.6 Partition identification and discovery

Partitions are identified by a 16-bit id and a UUID (Unique Universal Identifier) (see [7]). This helps partitions discover the presence of other partitions and their properties. The mechanism used to assign an id to a partition is implementation defined. It could be specified in the manifest of the partition or allocated at boot by the partition manager responsible for managing the partition.

An partition must use one of the following mechanisms to discover its id:

- A platform discovery mechanism like ACPI or Device tree
- SPCI_ID_GET interface (also see Section 9.7).

All SPCI components can discover the identities and properties of other partitions through the SPCI_PARTITION_INFO_GET interface. Once discovered, the ids must be used in the messaging interfaces to identify the target of a message.

The id value 0 is reserved for the Hypervisor as described in [5]. The id value assigned to the SPM is IMPLEMENTATION DEFINED.

---

**Provisional**

**Open Topic**

This version of the framework assumes that VMs and SPs are statically provisioned in the system such that they are loaded and initialized during system boot. Support for instantiation of partitions dynamically at run time is under discussion and partner feedback is welcome.

3.7 Execution context

Each endpoint has one or more execution contexts depending upon its implementation. An execution context comprises of general purpose, system and any memory mapped register state that must be maintained by a partition manager. A partition manager is responsible for allocating, initializing and running the execution context of an endpoint on a physical or virtual PE in the system. An execution context is identified by using
a 16-bit ID. This ID is referred to as the vCPU or execution context ID. Figure 2 illustrates an example configuration of endpoints and their execution contexts.

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 Section 3.10). Figure 3 illustrates this relationship.

An endpoint must be one of the following types:

- Uni-processor capable with a single execution context. It must run only on a single PE in the system at any point of time. This type of endpoint is called a UP endpoint.
- Multi-processor capable with multiple execution contexts. It could run concurrently on separate PEs in the system. These endpoints are called MP endpoints.
An endpoint's execution context 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 the endpoint’s execution. The endpoint must not make any assumptions about the PE it runs on.

This version of the framework mandates 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 an endpoint’s resource description (see Section 4.1.1).
- S-EL0 partitions must be UP and migrate capable.

For the purpose of power management (also see Section 4.2.1), an execution context of an endpoint in EL1 or S-EL1 is identified by the value programmed in the:

- MPIDR system register on the virtual or physical PE it executes on in the AArch32 execution state
- MPIDR_EL1 system register on the virtual or physical PE it executes on in the AArch64 execution state

On Armv8.3 and earlier systems, in an implementation of a MP Trusted OS, the number of execution contexts is equal to the number of PEs in the system. Furthermore, each execution context is pinned to a PE.

The number of execution contexts an endpoint implements can differ from the number of PEs in the system. This must be specified in the resource description of the endpoint (see Section 3.9). For example, a VM in the Normal world must use the resource description to inform the Hypervisor how many vCPUs it implements. The Hypervisor must maintain an execution context for each vCPU.

### 3.8 Split SPM configuration

As described in Section 2.1, the SPM is responsible for isolating SPCI components from each other and enabling communication between them. In all secure physical SPCI instances, both ELs have independent access to the physical address space. It is not possible to isolate components running on either side of the EL boundary from each other.

In all secure physical SPCI instances except SPCI instance number 10 in Table 3, the SPM could be implemented entirely in the higher EL and manage a single SP at the lower EL. These configurations are listed in Table 4. In these configurations:

- The SP must use the SPCI messaging mechanisms described in Section 5.
- In the absence of physical isolation, the SP and SPM could emulate:
  - SP setup and isolation at boot time as described in Section 4.1.1.
  - SPCI memory management transactions described in Section 6.4 at run time.
- The SP must use an implementation defined mechanism to isolate the physical address spaces of S-EL0 or User mode applications from each other.
- The SP must use an implementation defined mechanism to communicate with its S-EL0 or User mode applications.

<table>
<thead>
<tr>
<th>SPM/SP config. number</th>
<th>SPM EL</th>
<th>SP EL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>EL3</td>
<td>S-EL1</td>
</tr>
<tr>
<td>2.</td>
<td>EL3</td>
<td>Supervisor</td>
</tr>
</tbody>
</table>
In all secure physical SPCI instances, the SPM could also manage SPs at the secure virtual SPCI instance. This corresponds to the following SPCI instance numbers in Table 3.

- SPCI instance 1
- SPCI instance 2
- SPCI instance 3
- SPCI instance 4
- SPCI instance 5

In the absence of physical address space isolation, SPM responsibilities at the secure physical SPCI instance could be split amongst two SPM instances that run on either side of the EL boundary as described below.

- Both SPM instances participate in enabling generalized communication amongst themselves and SPs at the secure virtual SPCI instance through SPCI messaging mechanisms described in Section 5.
- The SPM instance at the lower EL manages isolation amongst one or more SPs running at the secure virtual SPCI instance. This implies that it is responsible for:
  - SP isolation and setup during boot time by using the SP's partition manifest (see Section 4.1.1).
  - SP isolation during run time by handling the memory management transactions specified in Section 6.4.
- The SPM instance at the higher EL allows the SPM instance in the lower EL to setup and isolate SPs at boot time.
- The SPM instance at the higher EL forwards memory management transactions between the Normal world and the SPM instance at the lower EL in both directions (also see Section 6.4.1).

The SPM implementations in either EL could be a part of the same or different software images. This choice is implementation defined. In all such scenarios, the two SPM instances must use the interfaces defined by the framework for communication.

Such a SPM implementation is called a *split SPM*. Valid split SPM configurations are listed in Table 5.

### Table 5: Split SPM configurations

<table>
<thead>
<tr>
<th>Split SPM config. number</th>
<th>Higher EL</th>
<th>Lower EL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>EL3</td>
<td>S-EL2</td>
</tr>
<tr>
<td>2.</td>
<td>EL3</td>
<td>S-EL1</td>
</tr>
<tr>
<td>3.</td>
<td>EL3</td>
<td>Supervisor</td>
</tr>
<tr>
<td>4.</td>
<td>Monitor</td>
<td>Supervisor</td>
</tr>
</tbody>
</table>

The SP in configurations listed in Table 4 could choose to manage SPs at the following secure virtual SPCI instances:
• SPCI instance 3
• SPCI instance 4
• SPCI instance 5

In this case, the SP must:

• Implement a split SPM in the EL it executes in.
• Only support direct messaging as a MP-capable partition as described in Section 5.4.

The mechanism used by the SP to function as a partition from the Normal world’s perspective and as a split SPM from the perspective of all SPCI components is implementation defined. The SP could choose to use SPCI interfaces for memory management and communication amongst SPs and applications in S-EL0/User mode.

3.9 System resource management

Components in the SP communication 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.

Additionally, 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 amongst components.

Rules associated with ownership and access of memory regions are described in Section 6.

Rules associated with ownership and access of CPU cycles are described in Section 3.10.

For a device, ownership and access attributes are associated with its MMIO region. If a device is upstream of an SMMU, then its access to the physical address space is managed using the rules associated with management of memory regions (also see Section 6.2).

A device could be a peripheral device or an autonomous coprocessor e.g. a DSP. It is assumed that the MMIO region contains an interface that the partition can use to communicate with the device. A partition can request access and/or ownership of a device through its manifest (see Table 8) in one of the following ways.

• A partition requests ownership and exclusive access to the MMIO region of a device during system boot (see Section 4). 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 system boot. The corresponding partition manager is the owner of the MMIO region and grants access to all the partitions.

This version of the framework does not permit ownership of or access to a device to be transferred to another partition during run time.
3.10 Primary scheduler

The primary scheduler is a module which is responsible for ensuring that all SPCI components are allocated CPU cycles for message processing on any PE in the system.

In terms of ownership and access rules, it is the owner of CPU cycles and lends them to threads of execution it manages. It does this in response to a scheduling decision.

In a scheduling decision, the scheduler chooses a thread of execution to run on a particular PE in the system as per the scheduling policy. The thread of execution is chosen from a set of runnable threads. Any thread that is not runnable is either blocked or sleeping until some event occurs.

The first type of thread of execution considered in the framework are Application threads. These can run inside any SPCI component e.g. an application that provides streaming media services inside the OS Kernel.

The second type of thread of execution considered in the framework are Execution context threads. Each thread corresponds to an execution context of an endpoint in the system e.g. For every VM managed by a Hypervisor, it implements an execution context thread for each vCPU of the VM.

The primary scheduler can manage one or both types of threads. An endpoint that is allocated CPU cycles by the primary scheduler could implement a secondary scheduler to manage the allocated cycles amongst its application threads.

Figure 4 illustrates an example of a primary scheduler that manages both types of threads of execution and a VM that manages application threads inside itself.

The framework assumes that execution context threads are managed only by the primary scheduler. See Section 5.3 for details.

![Figure 4: Example primary scheduler configuration](image)

The primary scheduler could be a part of:

- The OS kernel running in EL1 if virtualization extensions are not used in the Normal world
- The Host OS running in EL2 in the case of a Type 2 Hypervisor when the virtualization host extension is used.
• 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])
• A separate VM running in EL1 that has been delegated the responsibility of scheduling by the Hypervisor.
• The Hypervisor running in EL2

The following assumptions have been made regarding the role of the primary scheduler.

• There is a single primary scheduler across all PEs in the system
• An instance of the primary scheduler executes on each PE in the system.

Endpoint services could be accessed before the primary scheduler is initialized. At boot time, one or more software components are responsible for booting the primary scheduler. The framework assumes that each component executes a boot stage on the primary CPU. Ownership of cycles is relayed from one component to the next across these boot stages. Each component lends cycles to an endpoint if it access the endpoint’s services.

3.11 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 SPCI functions.

During run time, an endpoint’s execution context can be in one of the following states from the perspective of the primary scheduler, SPM and Hypervisor. These states are used to describe messaging mechanisms in Section 5.

• *Idle*. The execution context is waiting for a message.
• *Busy*. The execution context is processing a message.
• *Preempted*. The execution context was preempted by an interrupt while processing a message.
4 Partition setup

The firmware framework describes responsibilities of the Hypervisor and SPM for partition setup during boot time. *Boot time* refers to the stage during system boot before an endpoint is able to provide its services to other SPCI components.

The list of actions that a partition manager must perform to setup a partition is specified below.

1. Isolate the partition from other SPCI components based upon the contents of its partition manifest. This is described in Section 4.1.
2. Initialize the partition on the boot CPU. This is described in Section 4.2.
3. Initialize the partition on non-boot CPUs. This is described in Section 4.2.1.

4.1 Partition isolation

The SPM and Hypervisor are responsible for isolating all SPCI components from each other. This includes:

1. Isolation of an endpoint from another endpoint
2. Isolation of the Hypervisor from endpoints
3. Isolation of the SPM from the Hypervisor and endpoints
4. Isolation of SPs from the Hypervisor

Isolation is achieved by applying the principle of least privilege where an SPCI component is permitted to access only that part of the system’s physical address space and resources that are required for it to function correctly. This constraint is also applied to any devices that can be programmed by the SPCI component.

Isolation is enforced by utilizing features implemented by the CPU and system architecture. For example:

1. The Hypervisor uses the virtualization extension implemented by a CPU to restrict visibility of the physical address space and interrupts from a VM.
2. The Hypervisor uses stage 2 translations implemented by a System Memory Management Unit (SMMU) to restrict visibility of the physical address space from a device upstream of the SMMU.
3. SPCI components in the secure world are isolated from components in the Normal world by using the Arm TrustZone security extensions implemented by the CPU and devices in the system.

An endpoint describes the subset of the system’s physical address space and resources it needs access to through its manifest file. The SPM and Hypervisor use the manifest file of SPs and VMs respectively to validate the resource requests and allocate resources to the endpoint if the validation succeeds. See Section 4.1.1 for a description of the partition manifest.
4.1.1 Partition manifest

Each partition specifies the following information in its manifest file.

- Partition properties as described in Table 6
- Memory regions as described in Table 7
- Devices as described in Table 8
- Partition boot protocol as described in Table 9

The following aspects of the partition manifest are implementation defined.

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

<table>
<thead>
<tr>
<th>Information fields</th>
<th>Mandatory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPCI version</td>
<td>Yes</td>
<td>• Version of PSA-FF-A expected by the partition at the SPCI instance it will execute</td>
</tr>
</tbody>
</table>
| UUID                   | Yes       | • UUID of service implemented by this partition
|                        |           | • UUID can be shared by multiple instances of partitions that offer the same service
|                        |           | • For example,
|                        |           |  - If there are multiple instances of a Trusted OS, then the UUID can be shared by all instances
|                        |           |  - The TEE driver in the HLOS can use the UUID with the SPCI_PARTITION_INFO_GET interface to determine the:
|                        |           |   • Number of Trusted OSs
|                        |           |   • The partition ID of each instance of the Trusted OS
| Partition ID           | No        | • Pre-allocated partition ID
| Auxiliary ID           | No        | • Pre-allocated 16-bit ID that could be used in memory management transactions to allow a partition manager to handle the transaction in an implementation defined manner. |
| Stream endpoint IDs    | No        | • List of stream endpoint IDs this partition is acting as a proxy for. Also see Table 8 & Section 6.2.   |
| Name                   | No        | • Name of the partition e.g. 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 a MP capable partition irrespective of the number of PEs
<table>
<thead>
<tr>
<th>Information fields</th>
<th>Mandatory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run-Time EL</td>
<td>Yes</td>
<td>• EL1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• SEL0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• SEL1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Supervisor mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• User mode</td>
</tr>
<tr>
<td>Execution state</td>
<td>Yes</td>
<td>• AArch64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• AArch32</td>
</tr>
<tr>
<td>Load address</td>
<td>No</td>
<td>• Absence of this field indicates that the partition is position independent and can be loaded at any address chosen at boot time</td>
</tr>
<tr>
<td>Entry point address</td>
<td>No</td>
<td>• Absence of this field indicates that the entry point is at offset 0x0 from the base of the partition's binary image</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If present, this field specifies the offset of the entry point from the base of the partition's binary image</td>
</tr>
<tr>
<td>Translation Granule</td>
<td>Yes</td>
<td>• 4KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 16KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 64KB</td>
</tr>
<tr>
<td>Boot order</td>
<td>No</td>
<td>• A unique number amongst all partitions that specifies if this partition must be booted before others</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• For example, a partition could provide a service that other partitions need in order to initialize themselves. The manifest of this partition can use this field to ensure it is booted before others.</td>
</tr>
<tr>
<td>RX/TX information</td>
<td>No</td>
<td>• Reference to memory region entries in this resource description that describe the RX/TX buffers expected by the partition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The memory region entries must specify the base addresses of both buffers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The size and attributes fields must fulfill the requirements specified in Section 5.2.2.3</td>
</tr>
<tr>
<td>Messaging method</td>
<td>Yes</td>
<td>• This field specifies which messaging methods are supported by the partition. This could be one or both of direct and indirect messaging. These methods are described in Section 5. The following information must be provided in the manifest:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Indirect messaging is supported. This always includes support for both sending and receiving indirect messages.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Partition can receive direct requests.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– The number of execution contexts must be either equal to the number of PEs in the system or 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– The partition must support sending back direct responses.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– If the virtualization extension is enabled in the secure state then it must be specified whether the partition will use managed exits (see Section 5.5.4).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Partition can send direct requests.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– The partition must support receipt of direct responses.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Partition manager must not boot a partition if the messaging method it requires is not supported.</td>
</tr>
</tbody>
</table>
### Information fields

<table>
<thead>
<tr>
<th>Mandatory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Scheduler implemented</td>
<td>No • Presence of this field indicates that the partition implements the primary scheduler. • Run-time EL must be EL1 if this field is specified</td>
</tr>
<tr>
<td>Run time model</td>
<td>No • If the run-time EL is S-EL0 or User mode then this field specifies the run time model that the SPM must enforce for this SP. – <em>Run to completion</em>. SP's execution must not be preempted. An execution context of this SP must only transition between the <em>idle</em> and <em>busy</em> states described in Section 3.11. – <em>Preemptible</em>. SP's execution can be preempted. An execution context of this SP can transition between all states described in Section 3.11. This is the default run time model for a S-EL0/User mode SP if this field is not specified in the partition manifest.</td>
</tr>
</tbody>
</table>

### Table 7: Memory regions

<table>
<thead>
<tr>
<th>Mandatory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base address</td>
<td>No • Absence of this field indicates that a memory region of specified size and attributes must be mapped into the partition's translation regime. The PM must describe the memory region to the partition through an IMPLEMENTATION DEFINED mechanism • If present, this field could specify a PA, VA (for S-EL0) partitions or IPA (for S-EL1 and EL1 partitions). This information must be specified using an IMPLEMENTATION DEFINED mechanism – If a PA is specified then the memory region must be identity mapped with the same IPA or VA as the PA – If a VA or IPA is specified then the memory could be identity or non-identity mapped • If present, the address must be aligned to the Translation granule size</td>
</tr>
<tr>
<td>Size</td>
<td>Yes • Count of pages of memory region as a multiple of the translation granule size of the partition</td>
</tr>
<tr>
<td>Attributes</td>
<td>Yes • Memory attributes • Instruction and data access permissions • Shareability attributes • Cacheability attributes • Security attributes – Non-secure for a NS-Endpoint – Non-secure or secure for a S-Endpoint</td>
</tr>
<tr>
<td>Name</td>
<td>No • Name of the memory region e.g. for debugging purposes</td>
</tr>
</tbody>
</table>
## Table 8: Device regions

<table>
<thead>
<tr>
<th>Information fields</th>
<th>Mandatory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical base address</td>
<td>Yes</td>
<td>• PA of base of a physically contiguous device MMIO region</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• MMIO region must be mapped with the following attributes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Device-nGnRnE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Read-write or Read-only, Not executable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Security attributes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Non-secure for a NS-Endpoint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Non-secure or secure for a S-Endpoint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If the MMIO region is not physically contiguous, then an entry for each physically contiguous constituent region must be specified. The entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td>must specify the PA and size of the constituent region.</td>
</tr>
<tr>
<td>Size</td>
<td>Yes</td>
<td>• Count of pages allocated to the MMIO region as a multiple of the translation granule size of the partition.</td>
</tr>
<tr>
<td>Interrupts</td>
<td>No</td>
<td>• List of physical interrupt IDs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Attributes of each interrupt ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Interrupt type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- SPI</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- PPI</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- SGI</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Interrupt configuration</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Edge triggered</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Level triggered</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Interrupt security state</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Secure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Non-secure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Interrupt priority value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Target execution context/vCPU for each SPI</td>
</tr>
<tr>
<td>SMMU ID</td>
<td>No</td>
<td>• If present, then on a system with multiple SMMUs, this field must help the partition manager determine which SMMU instance is this device</td>
</tr>
<tr>
<td></td>
<td></td>
<td>upstream of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Absence of this field implies that the device is not upstream of a SMMU</td>
</tr>
<tr>
<td>Stream IDs</td>
<td>No</td>
<td>• List of stream IDs assigned to this device</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Absence of stream ID list indicates that the device is not upstream of a SMMU</td>
</tr>
<tr>
<td>Stage 2 translation</td>
<td>No</td>
<td>• For each stream ID that this device can generate, this field specifies how the stage 2 translations corresponding to each stream ID must be监督管理。The following properties must be specified. Also see</td>
</tr>
<tr>
<td>properties</td>
<td></td>
<td>section 6.2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- The stream endpoint ID must be specified for each stream ID the device can generate.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- If the stream endpoint is managed by a proxy endpoint then its partition ID must be specified as well. Absence of this field implies</td>
</tr>
<tr>
<td></td>
<td></td>
<td>that a proxy device is used instead.</td>
</tr>
</tbody>
</table>
4.2 Partition initialization

The SPM and Hypervisor are responsible for passing control to a partition for initialization during system boot. They must:

- Allocate memory for an execution context for the partition on the boot CPU.
- Program the appropriate system registers to ensure the following.
  - Caches and Stage 1 translation regime are turned off for S-EL1 and EL1 partitions
  - Caches and Stage 1 translation regime are turned on for S-EL0 partitions
- Use the following information in the partition manifest to program an entry into the partition through this execution context using the `ERET` instruction.
  - Run-time exception level
  - Execution state
  - Entry point address

The SPM and Hypervisor can also pass an array of `name-value-size` pairs to a partition upon the first `ERET` to it. This information is passed in a C programming language structure described in Figure 4.2.

```c
typedef struct name_value_pair {
    unsigned int name[4];
    uintptr_t value;
    unsigned int size;
} name_value_pair;

typedef struct spci_init_info {
    unsigned int magic; /* SPCI */
    unsigned int count; /* Count of name value size pairs */
    name_value_pair nvp[]; /* Array of name value size pairs */
} spci_init_info;
```

The SPM and Hypervisor must fulfill the following requirements for the memory region where the `spci_init_info` structure is populated.

- Size of memory region must be a multiple of the translation granule size used by the partition
- Address of memory region must be aligned to the translation granule size used by the partition
- The memory region must be mapped in the translation regime of the partition that is managed by the Hypervisor or SPM
  - This is Stage 1 for a S-EL0 partition
  - This is Stage 2 for a S-EL1 or EL1 partition
- The memory region must be mapped with the same memory attributes as the RX/TX buffers as described in Section 5.2.2.3 in the partition's translation regime managed by the Hypervisor or SPM
- Boot information must be populated at offset 0 in the memory region
- The address of boot information must be passed in the general purpose register specified in the partition manifest (see Table 9).

The partition must specify the information it expects to be populated in the `spci_init_info` structure in its manifest through an implementation defined mechanism.
Table 9: Boot protocol information

<table>
<thead>
<tr>
<th>Information fields</th>
<th>Mandatory</th>
<th>Description</th>
</tr>
</thead>
</table>
| SPCI boot protocol usage | No        | • Presence of this field indicates that the partition expects the `spci_init_info` structure to be passed in a general purpose register (see Figure 4.2).  
  • The register in which the `spci_init_info` structure must be passed must be specified. Register must be between w0/x0-w3/x3. The width of the register is derived from its execution state specified in the partition manifest. |

### 4.2.1 Partition initialization on non-boot CPUs

Each execution context of a partition is assigned a unique value by:

- Either the system integrator at build time or
- The Hypervisor and SPM for a VM and SP respectively at boot time

These values must be conveyed to the partition through an implementation defined mechanism. Each value must be the same as what would be read by the execution context from the `MPIDR` or `MPIDR_EL1 system registers on the virtual or physical PE it is running on (also see Section 3.7). A partition is responsible for initializing its execution contexts on non-boot virtual or physical PEs by using these values and interfaces specified in the PSCI specification [8] as described below.

- It must use the `PSCI_CPU_ON` interface at the virtual SPCI instance to initialize an execution context.  
  - The `target_cpu` parameter must contain the unique value assigned to the execution context as described above.  
  - The `entry_point_address` parameter must contain the address at which the SPM or Hypervisor must program an entry into the partition through the specified execution context.  
  - The SPM or Hypervisor must use the same properties from the partition manifest that were used to initialize its execution context on the boot CPU (see Section 4.2).  
  - The SPM or Hypervisor must not pass the `spci_init_info` structure during this initialization phase.

### 4.3 Partition shutdown

A physical PE could be turned off through the `PSCI_CPU_OFF` interface at the non-secure physical SPCI instance. A partition’s execution context executing on this physical PE must cope with this event in one or more of the following ways.

- If the partition is a SP and the execution context is not pinned to the PE, then the SPM must migrate it to another physical PE.
- If the partition is a VM and the execution context is not pinned to the PE, then the Hypervisor must migrate it to another physical PE.
- If the execution is pinned to the PE or if the PE is the last to be turned off, then the execution context must use the `PSCI_CPU_OFF` interface at the virtual SPCI instance to indicate that it is no longer in use and will not be impacted if the PE is physically powered off.

The `PSCI_CPU_OFF` interface must be invoked by the execution context at the virtual SPCI instance before it is invoked at the non-secure physical SPCI instance.
If the execution context does not do this then the invocation of this interface at the non-secure physical SPCI instance must be terminated with the **DENIED** error code. It is implementation defined how the OS kernel or Hypervisor handles this error code.
5 Message passing

5.1 Overview

The SPCI communication framework defines a set of ABIs that enable SPCI components to exchange messages with each other. A message exchange comprises of the following phases.

1. Transmission of the message payload from the sender to the receiver. The mechanisms specified by the framework to do this are described in Section 5.2.

2. Allocation of CPU cycles to the receiver to process the message on a PE in the system. Cycles could be allocated by the sender or the primary scheduler. These methods are described in Section 5.1.1 & Section 5.1.2. The partition manager must make at least one method available to an endpoint it manages.

3. Message processing using the allocated cycles. The role of the framework during message processing is described in Section 5.5.

SPCI components (also see Section 3.1) participate in a message exchange by playing one or more of the following roles.

- **Sender**.
- **Receiver**.
- **Relayer**.
  - Validates and forwards a message to the receiver.
  - Provide access to CPU cycles to the receiver to process the message.
- **Primary scheduler**. See Section 3.10.

Table 10 specifies the roles that each component can play in a message exchange. The responsibilities of a component in a role depend upon the type and phase of a message exchange. This is described in detail in subsequent sections. In all messaging scenarios, in the absence of a Hypervisor, the SPM assumes its responsibilities.

<table>
<thead>
<tr>
<th>Config. No.</th>
<th>SPCI Component</th>
<th>Sender</th>
<th>Receiver</th>
<th>Relayer</th>
<th>Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>NS-Endpoint</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>2.</td>
<td>S-Endpoint</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>3.</td>
<td>Hypervisor</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>4.</td>
<td>SPM</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

5.1.1 Indirect messaging

In this method, the message sender requests the primary scheduler to allocate CPU cycles to the receiver for processing the message. The receiver is scheduled by the primary scheduler. The sender could make progress concurrently with the receiver either on the same or a different PE. The sender either polls or is notified when a response from the receiver is available.
The term **Indirect messaging** is used to describe this method for CPU cycle allocation in conjunction with interfaces to transmit the message payload. A detailed description of this method's usage is provided in Section 5.3. Figure 5 illustrates this method. It assumes that both the sender and receiver run on the same PE.

In this method, the primary scheduler manages the receiver's execution context threads (also see Section 3.10). It chooses an execution context of the receiver and allocates cycles to it for message processing.

The framework assumes that this method is only used for passing messages between SPCI endpoints. This covers the following scenarios.

- Between VMs in the Normal world.
- Between a VM or OS Kernel in the Normal world and a SP in the Secure world.
- Between a SP in the Secure world and a VM or OS Kernel in the Normal world.
- Between SPs in the Secure world.

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

- **SPCI_MSG_SEND.** This interface is used by a sender to send a message payload to a receiver (also see Section 11.1).
- **SPCI_RUN.** This interface is used by the primary scheduler to allocate CPU cycles and hand control to a receiver for message processing (also see Section 10.3 and Section 5.5).
- **SPCI_MSG_WAIT.** This interface is used by a receiver to indicate that it is ready to receive a new message (also see Section 10.1).
- **SPCI_MSG_POLL.** This interface is used by a receiver to poll for a new message (also see Section 9.8).
5.1.2 Direct messaging

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

The term Direct messaging is used to describe this method for CPU cycle allocation in conjunction with interfaces to transmit the message payload. A detailed description of this method is provided in Section 5.4. Figure 6 illustrates this method.

This method is used for messaging between an endpoint and the Hypervisor or SPM. It is also used for messaging between endpoints before the primary scheduler is initialized and when the primary scheduler cannot be involved in the message exchange e.g. it is not possible to communicate with the scheduler, the latency associated with scheduler communication cannot be tolerated by the use case or the receiver must be run on the same PE as the sender.

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

- Between SPCI endpoints under the constraints described in Section 5.4.
- From an endpoint to the Hypervisor or SPM.
- From the Hypervisor or SPM to an endpoint.
- Between the Hypervisor and SPM.

The following ABIs participate are used to implement direct messaging between endpoints.
• **SPCI_MSG_SEND_DIRECT_REQ.** This interface is used by a sender to send a request message payload to a receiver, lend CPU cycles to the receiver and wait for a response to arrive. Also see Section 11.2.

• **SPCI_MSG_SEND_DIRECT_RESP.** This interface is used by a sender to send a response message payload to a receiver, return CPU cycles to the receiver and wait for a new message to arrive. Also see Section 11.3.

• **SPCI_INTERRUPT.** This interface is used by the relayer to inform the sender that direct message processing in the receiver was preempted.

• **SPCI_RUN.** This interface is used by the sender to resume a preempted receiver.

The following is a non-exhaustive list of ABIs used to implement direct messaging between endpoints and Hypervisor or SPM and between the Hypervisor and SPM.

- SPCI_VERSION
- SPCI_RXTX_MAP
- SPCI_RXTX_UNMAP
- SPCI_MEM_DONATE
- SPCI_MEM_SHARE
- SPCI_MEM_LEND
- SPCI_MEM_RETRIEVE_REQ
- SPCI_MEM_RETRIEVE_RESP
- SPCI_MEM_RELINQUISH
- SPCI_MEM_RECLAIM

![Figure 6: Direct messaging](image)

### 5.2 Message transmission

#### 5.2.1 Overview

Message payloads are exchanged between two SPCI components through general purpose registers or a single pair of shared memory regions to transmit and receive messages called *RX/TX buffers* (see also
Section 5.2.2).

- Direct messaging can use both these mechanisms in conjunction with the ABIs described in Section 5.1.2.
- Indirect messaging must use only the RX/TX message buffers in conjunction with the ABIs described in Section 5.1.1.

Each message has a header and a payload. The header describes the:

- Source and target of the message
- Size and type of message payload.

The version of the message header and payloads is the same as the version of the firmware framework as returned by `SPCI_VERSION` (see Section 9.1).

The header is encoded in the parameter registers of the ABI used for message transmission. This information is used to validate and route the message and decide if the message payload must be interpreted by a relayer.

There are two type of message payloads.

1. Payloads that are defined by the communication framework e.g. 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 relayers as well as the sender and receiver. They are used when:
   - Relayer participation is required to validate or modify message contents prior to 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 and vice versa.
   - Between the Hypervisor and SPM and vice versa.

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

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

The properties of framework and partition messages influence direct and indirect messaging as follows.

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

Table 11 lists examples of ABIs available for transmitting messages of both types in registers or RX/TX buffers using both types of messaging methods.
Table 11: Combinations of messaging and message transmission mechanisms

<table>
<thead>
<tr>
<th>Messaging method</th>
<th>Message type</th>
<th>Message payload location</th>
<th>Message transmission interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>Partition</td>
<td>Register</td>
<td>SPCI_MSG_SEND_DIRECT_REQ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>•</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SPCI_MSG_SEND_DIRECT_RESP</td>
</tr>
<tr>
<td>Direct</td>
<td>Partition</td>
<td>RX/TX</td>
<td>Invalid usage</td>
</tr>
<tr>
<td>Direct</td>
<td>Framework</td>
<td>Register</td>
<td>Any interface to send or receive information from the Hypervisor or SPM e.g.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• SPCI_VERSION</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• SPCI_RX_RELEASE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• SPCI_YIELD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• SPCI_RXTX_MAP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• SPCI_RXTX_UNMAP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• SPCI_RUN</td>
</tr>
<tr>
<td>Direct</td>
<td>Framework</td>
<td>RX/TX</td>
<td>Any interface to send or receive information from the Hypervisor or SPM e.g.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• SPCI_MEM_DONATE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• SPCI_MEM_SHARE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• SPCI_MEM_LEND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• SPCI_MEM_RELINQUISH</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• SPCI_MEM_RE林IVER_REQ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• SPCI_MEM_RElinque_RESP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• SPCI_MEM_RECLAIM</td>
</tr>
<tr>
<td>Indirect</td>
<td>Partition</td>
<td>Register</td>
<td>Invalid usage</td>
</tr>
<tr>
<td>Indirect</td>
<td>Partition</td>
<td>RX/TX</td>
<td>SPCI_MSG_SEND</td>
</tr>
<tr>
<td>Indirect</td>
<td>Framework</td>
<td>Register</td>
<td>Invalid usage</td>
</tr>
<tr>
<td>Indirect</td>
<td>Framework</td>
<td>RX/TX</td>
<td>Invalid usage</td>
</tr>
</tbody>
</table>

5.2.2 RX/TX buffers

A RX/TX buffer pair is shared between two SPCI components at a SPCI instance.

- The SPCI component at the lower EL is the consumer of the RX buffer and producer of the TX buffer.
- The SPCI component at the higher EL is the producer of the RX buffer and the consumer of the TX buffer.
- A SPCI component is permitted to share only a single RX/TX buffer pair with another SPCI component at a SPCI instance.
In *split SPM* configurations listed in Section 3.8, it is implementation defined whether the two SPM instances share an RX/TX buffer pair.

The endianess 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. Also see Section 5.2.2.3.
- Each VM shares its buffer pair with the Hypervisor and SPM.
- The OS kernel shares its buffer pair with the SPM.
- The Hypervisor shares its buffer pair with the SPM.

In the Secure world,

- Each SP has a secure buffer pair. Also see Section 5.2.2.3.
- Each SP shares its secure buffer pair with the SPM.

These message buffer configurations are illustrated in Figure 7.

**Figure 7: Configurations of RX/TX buffer pair between SPCI components**

Mechanisms for message transmission through RX/TX buffers are described in Section 5.2.2.1.

Mechanisms for discovery and setup of a RX/TX buffer pair are described in Section 5.2.2.2.

Requirements for correctly mapping a RX/TX buffer pair in the translation regimes of both SPCI components at any SPCI instance are described in Section 5.2.2.3.

5.2.2.1 *Buffer based message transmission*
A message is transmitted between endpoints by copying it from the TX buffer of the sender to the RX buffer of the receiver as described below.

- The message copy is done by the Hypervisor if the sender and receiver are VMs.
- The message copy is done by the SPM if the sender and receiver are SPs.
- The message copy is done by the SPM if the sender is a NS-Endpoint and the receiver is a SP or vice versa.

A message is transmitted between the Hypervisor as a sender and SPM as a receiver and vice versa through the RX/TX buffer pair shared between them. A TX to RX copy is not required as both components can access the single buffer pair.

A message is transmitted between a VM as a sender and Hypervisor or SPM as receiver through the non-secure RX/TX buffer pairs they share. The following message copies could be required in this case.

- A copy from the VM's TX buffer to the RX buffer shared between the Hypervisor and SPM
- A copy from the TX buffer shared between the Hypervisor and SPM to the VM's RX buffer.

If the message payload is modified by the Hypervisor before being forwarded to the SPM, the above message copies are required since the VM can see a modified message payload in its TX buffer.

If the message payload is forwarded unmodified to the SPM from the Hypervisor and vice versa, the above message copies are not required since the SPM can access the VM's buffer pair.

A message is transmitted between a SP as a sender and SPM as receiver through the secure RX/TX buffer they share. A TX to RX copy is not required as the SPM can access the SP's buffer pair.

A message is transmitted between a SP as a sender and Hypervisor as receiver by copying it from the SP's TX buffer to the TX buffer shared between the Hypervisor and SPM since the Hypervisor cannot access the SP's buffer pair.

5.2.2.2 Buffer discovery and setup

This version of the framework enables discovery and setup of RX/TX buffer pairs between SPCI components as described below.

1. The Hypervisor or SPM could allocate the buffer pair on behalf of a VM or SP respectively and convey this information to the endpoint through an implementation defined mechanism.

   The endpoint must specify this requirement in its resource description by specifying the base addresses (as IPAs or VAs) of where it expects its RX/TX buffer pair to be mapped by the SPM or Hypervisor as applicable (see also Section 4.1.1).

   The SPM is not permitted to allocate a buffer pair on behalf of the Hypervisor or a VM.

2. An endpoint could allocate the buffer pair and use the SPCI_RXTX_MAP interface to map it with the Hypervisor or SPM as applicable.

3. An endpoint could use the SPCI_RXTX_UNMAP interface to unmap a buffer pair from the Hypervisor or SPM as applicable.

4. The Hypervisor must allocate the buffer pair it shares with the SPM and use the SPCI_RXTX_MAP interface to map it with the SPM.

5. The Hypervisor could use the SPCI_RXTX_UNMAP interface to unmap the buffer pair from the SPM.

6. The Hypervisor must forward an invocation of the SPCI_RXTX_MAP interface from a VM to map its RX/TX buffer with the SPM.

7. The Hypervisor must forward an invocation of the SPCI_RXTX_UNMAP interface from a VM to unmap its RX/TX buffer from the SPM.

   - See Section 9.4 for a description of the SPCI_RXTX_MAP interface.
   - See Section 9.5 for a description of the SPCI_RXTX_UNMAP interface.
Figure 8 illustrates an example RX/TX buffer setup with a single VM and SP using the interfaces summarized above where the:

- SPM allocates the buffer pair on behalf of the SP
- Hypervisor registers its buffer pair with the SPM
- VM registers its buffer pair with the Hypervisor and SPM
- VM unregisters its buffer pair with the Hypervisor and SPM
Figure 8: RX/TX Buffer setup
5.2.2.3 Buffer attributes

The size of the RX and TX buffers in a pair must be the same and a multiple of the larger translation granule size used by the SPCI components at a SPCI instance.

The alignment of the RX and TX buffers in a pair must be equal to the larger translation granule size used by the SPCI components at a SPCI instance (see also Section 3.5).

The framework assumes that the memory attributes of all RX and TX buffers in the system are configured as described below.

- If a stage of address translation is enabled in a translation regime from where the buffer is accessed, it must be mapped with the following memory region attributes in that stage.
  - Normal memory
  - Write-Back Cacheable
  - Non-transient Read-Allocate
  - Non-transient Write-Allocate
  - Inner Shareable
  - Buffers shared between a SP and SPM must have the NS bit = 0 in their translation table descriptors.
  - Buffers shared between a VM, Hypervisor and SPM must have the NS bit = 1 in their translation table descriptors.
  - Table 12 describes the minimum permission requirements of RX/TX buffer.

<table>
<thead>
<tr>
<th>Buffer Type</th>
<th>Producer</th>
<th>Consumer</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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 write access to modify message payload if required. |

- A buffer pair could be accessed with different memory region attributes from the producer and consumer’s translation regimes if address translation is disabled in one of them.

To avoid memory coherency issues in this scenario, the SPCI component that has address translation disabled must perform cache maintenance on the buffer in scenarios listed in Table 13. 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 Section 5.2.2.4).
  - As a consumer, this must be done before reading the buffer populated by the producer.
Table 13: RX/TX buffer cache maintenance requirements

<table>
<thead>
<tr>
<th>Config No.</th>
<th>Address translation in Producer</th>
<th>Address translation in Consumer</th>
<th>Cache maintenance required</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Disabled</td>
<td>Disabled</td>
<td>No</td>
</tr>
<tr>
<td>2.</td>
<td>Disabled</td>
<td>Enabled</td>
<td>Yes</td>
</tr>
<tr>
<td>3.</td>
<td>Enabled</td>
<td>Disabled</td>
<td>Yes</td>
</tr>
<tr>
<td>4.</td>
<td>Enabled</td>
<td>Enabled</td>
<td>No</td>
</tr>
</tbody>
</table>

5.2.2.4 Buffer synchronization

The RX and TX buffers are written to by a producer and read by a consumer as described in Table 14. Concurrent accesses to these buffers from both entities on either side of an SPCI instance must be synchronized to preserve the integrity of their contents.

Table 14: RX/TX producers and consumers

<table>
<thead>
<tr>
<th>Buffer Type</th>
<th>Producers</th>
<th>Consumers</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM RX</td>
<td>Hypervisor, SPM</td>
<td>VM</td>
</tr>
<tr>
<td>VM TX</td>
<td>VM</td>
<td>Hypervisor, SPM</td>
</tr>
<tr>
<td>OS Kernel RX</td>
<td>SPM</td>
<td>OS Kernel</td>
</tr>
<tr>
<td>OS Kernel TX</td>
<td>OS Kernel</td>
<td>SPM</td>
</tr>
<tr>
<td>SP RX</td>
<td>SPM</td>
<td>SP</td>
</tr>
<tr>
<td>SP TX</td>
<td>SP</td>
<td>SPM</td>
</tr>
<tr>
<td>Hypervisor RX</td>
<td>SPM</td>
<td>Hypervisor</td>
</tr>
<tr>
<td>Hypervisor TX</td>
<td>Hypervisor</td>
<td>SPM</td>
</tr>
</tbody>
</table>

The framework defines states, access and ownership rules that must be followed by the producer and consumer of each buffer.

- Each buffer can be either empty or full (has a message in it) at any given time. This state must be tracked internally by the producer and consumer using an implementation defined mechanism.
- The producer of a buffer owns it when it is empty.
- The consumer of a buffer owns it when it is full.
- The producer must write to the buffer only when it is empty.
- The consumer must read from the buffer only when it is not empty.

After a producer has written to a buffer, it must transfer its ownership to the consumer for reading the message. The mechanism used for this depends upon the following factors.

1. Exception level of each entity. The producer could reside in a higher exception level than the consumer and vice-versa. This in turn governs the conduits available to signal transfer of ownership. These are described in the Table 15.

2. Type of Secure partition. Architectural constraints on the highest EL in which a SP runs determine which conduit can be used to signal the transfer of ownership e.g. a S-EL1 SP can use all conduits described in Table 15 while a S-EL0 SP cannot use the interrupt conduit.
Table 15: Conduits to signal transfer of buffer ownership

<table>
<thead>
<tr>
<th></th>
<th>Producer to Consumer</th>
<th>Consumer to Producer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer at lower EL</td>
<td>SMC, SVC, HVC</td>
<td>ERET</td>
</tr>
<tr>
<td>Producer at higher EL</td>
<td>Interrupt or ERET</td>
<td>SMC, SVC, HVC</td>
</tr>
</tbody>
</table>

The SPM and Hypervisor must describe the interrupts they will use as a producer to each SP and VM respectively during their initialization through an implementation defined mechanism e.g. a device tree.

Transfer of ownership of the TX buffer takes place as described below.

- Through an invocation of the `SPCI_MSG_SEND` interface from the producer to the consumer.
- Through an invocation of the `ERET` instruction from the consumer to complete a previous invocation of the `SPCI_MSG_SEND` interface from the producer.

Transfer of ownership of the RX buffer takes place as described below.

- Through an invocation of the `SPCI_MSG_SEND` function through the `ERET` conduit from the producer to the consumer.
- Through an interrupt pended by the producer to indicate to the consumer that the RX buffer is full.
- Through an invocation of the `SPCI_RX_RELEASE` or `SPCI_MSG_WAIT` interfaces from the consumer to the producer.

A buffer could be shared amongst multiple producers and consumers and multiple instances of the same producer and consumer (also see Table 14). 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 spin lock to protect each buffer from accesses made concurrently by its own instances.

Provisional

Open Topic

A message could be copied by the SPM from the TX buffer of a SP to the RX buffer of VM0 at the same time as a copy from the TX buffer of VM1 to the RX buffer of VM0 by the Hypervisor.

In this case, both the Hypervisor and SPM are producers for VM0's RX buffer. Their accesses to the buffer must be synchronized. In this version of the framework, this must be done through an implementation defined mechanism.

The need for support in the framework to implement this synchronization is currently under discussion and partner feedback is welcome.

5.2.2.5 Example buffer synchronization flows

This section illustrates examples of how the states and mechanisms to transfer ownership of a buffer can be used in an implementation.

Figure 9 illustrates interaction between a SP and the SPM for transferring ownership of the RX buffer through the `ERET` instruction and `SPCI_RX_RELEASE` interface.

Figure 10 illustrates interaction between a SP and the SPM for transferring ownership of the RX buffer through an interrupt and `SPCI_RX_RELEASE` interface.
Figure 11 illustrates interaction between a SP and the SPM for transferring ownership of the TX buffer through the \texttt{SPCI\_MSG\_SEND} interface and \texttt{ERET} instruction.

The following aspects of these interactions have been assumed.

- The same interactions can be applied to a RX/TX buffer pair shared between a VM and Hypervisor and Hypervisor and SPM as well.
- The SP and SPM are MP-capable. This means that it has multiple instances that can run concurrently on separate physical PEs.
Figure 9: Buffer synchronization with producer at higher EL
Figure 10: Interrupt based buffer synchronization with producer at higher EL
Figure 11: Buffer synchronization with producer at lower EL
5.3 Indirect messaging usage

There are three phases in an indirect message exchange

- Transmission of message from sender to receiver.
- Notification to the primary scheduler that the receiver is runnable.
- Allocation of CPU cycles by the primary scheduler to the receiver to process the delivered message.

A SPCI component could play one or more of the roles listed in Table 10. A list of configurations that result from valid combinations of these roles for indirect messaging is specified in Table 16.

In each configuration the primary scheduler could be resident either in a NS-Endpoint or the Hypervisor. This implies that the primary scheduler could be:

- Co-resident with the sender if the sender is in the Normal world.
- Co-resident with the receiver if the receiver is in the Normal world.
- Co-resident with the relayer i.e. the Hypervisor.
- Resident in a VM different from the sender and receiver.

<table>
<thead>
<tr>
<th>Config no.</th>
<th>Sender</th>
<th>Receiver</th>
<th>Relayer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>VM</td>
<td>VM</td>
<td>Hypervisor</td>
</tr>
<tr>
<td>2.</td>
<td>NS-Endpoint</td>
<td>SP</td>
<td>Hypervisor (if present) and SPM</td>
</tr>
<tr>
<td>3.</td>
<td>SP</td>
<td>SP</td>
<td>SPM</td>
</tr>
<tr>
<td>4.</td>
<td>SP</td>
<td>NS-Endpoint</td>
<td>SPM and Hypervisor (if present)</td>
</tr>
</tbody>
</table>

5.3.1 Discovery and setup

An endpoint that can receive messages through indirect messaging must specify this property in its resource description (see Section 4.1.1). The primary scheduler must be aware of the presence of this component and the number of execution contexts it implements. The Hypervisor and/or SPM could provide this information to the primary scheduler through an implementation defined mechanism e.g. Device tree. Alternatively, the SPCI component that implements the primary scheduler could use the SPCI_PARTITION_INFO_GET interface to obtain this information.

An attempt to send a direct message to an endpoint that only supports indirect messaging must be rejected as described below.

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

5.3.2 Message delivery and scheduler notification

The framework defines the SPCI_MSG_SEND interface to transmit a message from a sender to a receiver and notify the primary scheduler that the receiver is runnable.

Section 11.1 describes the SPCI_MSG_SEND interface. Section 11.1.2 describes how a message is transmitted using this interface and the responsibilities of the participating components in all the configurations.
listed in Table 16.

The sender must notify the primary scheduler to schedule the receiver to process the message in its RX buffer. The mechanism to do this depends upon the SPCI component where the primary scheduler is implemented relative to the sender and relayer. An applicable mechanism in Section 11.1.3 must be used. In all cases, after the scheduler has been notified, an invocation of the `SPCI_MSG_SEND` call must be completed to enable the sender to make progress.

Figure 12 illustrates an example flow in which a VM sends an indirect message to a SP. The primary scheduler is resident in another VM. The Hypervisor forwards the message transmission request to the SPM, notifies the scheduler and completes the `SPCI_MSG_SEND` call from the VM.

![Figure 12: Example VM to SP indirect message delivery and scheduler notification](image-url)
5.3.3 Scheduling the receiver

The primary scheduler must make the following choices after receiving the notification that the receiver is runnable.

- Choose a PE on which CPU cycles will be allocated to the receiver.
- Choose an execution context of the receiver to which the cycles will be allocated on the chosen PE.

The mechanism used by the primary scheduler to run the receiver depends upon the SPCI component where the scheduler is implemented relative to the receiver and relayer. A combination of applicable mechanisms listed below must be used.

1. The primary scheduler resides in the receiver VM. The scheduler must use an implementation defined mechanism to run the thread responsible for processing the message e.g. use an OS primitive to run an application thread.

2. The primary scheduler is co-resident with the Hypervisor. If the receiver is a VM, the Hypervisor is responsible for programming a return to the receiver in response to a decision by the primary scheduler to run the receiver. The method used to do this depends upon the state of the receiver (also see Section 3.11).

   - If the receiver’s execution context is in the *idle* state, the relayer must complete the invocation of the interface used by the receiver to enter this state e.g. `SPCI_MSG_WAIT`, `SPCI_MSG_SEND_DIRECT_RESP`.
   - If the receiver’s execution context is in the *preempted* state, the relayer must resume its execution.

3. In all other scenarios, the primary scheduler must use the `SPCI_RUN` interface to run the receiver. Section 10.3.1 describes the responsibilities of the participating components in an invocation of this interface.

Once the receiver starts processing the message after a scheduling decision, it could interact with the primary scheduler in ways described in Section 5.5.

Figure 13 builds upon the example flow in Figure 12 to illustrate how the primary scheduler runs the SP during a scheduling decision.
Figure 13: Complete indirect messaging flow from VM to SP

Figure 14 builds upon the example flow in Figure 13 to illustrate how a SP sends an indirect message back to the VM and notifies the primary scheduler. The scheduler runs the VM subsequently.
Figure 14: Complete indirect messaging flow from SP to VM
5.4 Direct messaging usage

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

The framework assumes that direct messaging is used by a sender as an equivalent of invoking a procedure or function in the receiver. The receiver executes the function and returns the results through a direct message. For framework messages, execution of the function in the Hypervisor or SPM runs to completion from the sender’s perspective. For partition messages, execution of the function in an endpoint could run to completion or be preempted by interrupts one or more times. In the latter case, the communication framework is responsible for resuming function execution.

A SPCI component could play one or more of the roles listed in Table 10 except for the role of the primary scheduler during a direct message exchange. Direct messaging is used for:

1. Exchanging framework messages with the Hypervisor and SPM in the configurations listed in Table 17. These messages can be exchanged in both RX/TX buffers and registers.
2. Exchanging partition messages between endpoints in the configurations listed in Table 18. These messages can be exchanged only in registers.

Table 17: Valid configurations for exchanging framework messages through direct messaging

<table>
<thead>
<tr>
<th>Config no.</th>
<th>Sender</th>
<th>Receiver</th>
<th>Relayer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>VM</td>
<td>Hypervisor</td>
<td>•</td>
</tr>
<tr>
<td>2.</td>
<td>NS-Endpoint</td>
<td>SPM</td>
<td>Hypervisor (if present)</td>
</tr>
<tr>
<td>3.</td>
<td>SP</td>
<td>SPM</td>
<td>•</td>
</tr>
<tr>
<td>4.</td>
<td>SP</td>
<td>Hypervisor</td>
<td>SPM</td>
</tr>
<tr>
<td>5.</td>
<td>Hypervisor</td>
<td>VM</td>
<td>•</td>
</tr>
<tr>
<td>6.</td>
<td>Hypervisor</td>
<td>SPM</td>
<td>•</td>
</tr>
<tr>
<td>7.</td>
<td>Hypervisor</td>
<td>SP</td>
<td>SPM</td>
</tr>
<tr>
<td>8.</td>
<td>SPM</td>
<td>NS-Endpoint</td>
<td>Hypervisor (if present)</td>
</tr>
<tr>
<td>9.</td>
<td>SPM</td>
<td>Hypervisor</td>
<td>•</td>
</tr>
<tr>
<td>10.</td>
<td>SPM</td>
<td>SP</td>
<td>•</td>
</tr>
</tbody>
</table>
Table 18: Valid configurations for exchanging partition messages through direct messaging

<table>
<thead>
<tr>
<th>Config no.</th>
<th>Sender</th>
<th>Receiver</th>
<th>Relayer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>VM</td>
<td>VM</td>
<td>Hypervisor</td>
</tr>
<tr>
<td>2</td>
<td>NS-Endpoint</td>
<td>SP</td>
<td>Hypervisor (if present) and SPM</td>
</tr>
<tr>
<td>3</td>
<td>SP</td>
<td>SP</td>
<td>SPM</td>
</tr>
<tr>
<td>4</td>
<td>SP</td>
<td>NS-Endpoint</td>
<td>SPM and Hypervisor (if present)</td>
</tr>
</tbody>
</table>

5.4.1 Discovery and setup

An endpoint that can receive messages through direct messaging must specify this property in its resource description (see Section 4.1.1). The Hypervisor and SPM must be aware about the presence of this endpoint and the number of execution contexts it implements. They must use the `SPCI_PARTITION_INFO_GET` interface to obtain this information from each other. An endpoint could use the same interface to obtain this information as well.

An attempt to send a indirect message to an endpoint that only supports direct messaging must be rejected as described below.

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

An endpoint that needs to send direct messages must specify this property in its resource description as well.

In a direct message exchange, it is possible that the primary scheduler cannot participate in choosing the execution context of the receiver and the PE it is run on. To minimize conflict with the primary scheduler's scheduling decisions and aid its availability on a PE, the receiver must make one of the following implementation choices.

- The receiver is implemented as a **UP** endpoint. This enables the SPM or Hypervisor to migrate the endpoint's execution context to the PE on which a direct messaging request is made.

- The receiver is implemented as a **MP** endpoint. In this case, the number of execution contexts that the endpoint implements must be equal to the number of PEs in the system. Each execution context must be pinned to a PE at system boot. This enables the SPM or Hypervisor to guarantee availability of an endpoint's execution context for direct messages on the same PE as the sender.

The receiver must specify these implementation choices to the Hypervisor or SPM in its resource description (see Section 4.1.1)

For framework messages that are exchanged with and between the Hypervisor and SPM, it is assumed that:

- Both components are implicitly configured as a **MP** endpoint as described above. This is true for an SPM implementation in any privileged exception level in the Secure world.
- Both components 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.
5.4.2 Message delivery and receiver execution

This version of the firmware framework defines a number of interfaces for passing framework messages with and between the Hypervisor and SPM. The description of these interfaces and the responsibilities of the participating components in all the configurations listed in Table 17 is provided in Section 7. A non-exhaustive list of these interfaces is provided below.

- SPCI_VERSION
- SPCI_RXTX_MAP
- SPCI_RXTX_UNMAP
- SPCI_MEMDonate
- SPCI_MEM_SHARE
- SPCI_MEM_LEND
- SPCI_MEM_RETRIEVE_REQ
- SPCI_MEM_RETRIEVE_RESP
- SPCI_MEM_RELINQUISH
- SPCI_MEM_RECLAIM

The rest of this section describes direct messaging between endpoints.

The framework defines the SPCI_MSG_SEND_DIRECT_REQ and SPCI_MSG_SEND_DIRECT_RESP interfaces (also see Section 11.2 & Section 11.3) to transmit a direct message from a sender endpoint to a receiver endpoint.

Section 11.2.1 describes how a message is transmitted using the SPCI_MSG_SEND_DIRECT_REQ interface and the responsibilities of the participating components in all the configurations listed in Table 18.

Section 11.3.1 describes how a message is transmitted using the SPCI_MSG_SEND_DIRECT_RESP interface and the responsibilities of the participating components in all the configurations listed in Table 18.

Figure 15 illustrates an example flow in which a VM sends a direct message to a SP through the SPCI_MSG_SEND_DIRECT_REQ interface. The SP processes the messages and returns the results using the SPCI_MSG_SEND_DIRECT_RESP interface.
Figure 15: Example direct message exchange between a VM and SP
5.5 Partition message processing

The run time model available to an endpoint for processing a partition message depends upon the method used for sending the message.

- Section 5.5.1 describes processing of an indirect message.
- Section 5.5.2 describes processing of a direct message.

In both cases, an endpoint can exchange framework messages with the Hypervisor and SPM. It is assumed that the processing of these messages runs to completion from the endpoint's perspective.

5.5.1 Indirect message processing

After an endpoint starts processing an indirect message, one or more of the following events could occur.

1. It relinquishes control back to the primary scheduler after completing message processing. This is done using the `SPCI_MSG_WAIT` interface and described in Section 10.1.

2. It sends an indirect message to another endpoint and requests the primary scheduler to schedule the message target. This is described in Section 5.3.

3. It gets preempted by an interrupt targeted to the primary scheduler or another SPCI component. This is described in Section 5.5.3.

4. It yields control back to the primary scheduler because it cannot continue execution due to an internal dependency. This is done using the `SPCI_YIELD` interface and described in Section 10.2.

5. It sends a direct partition message request to an endpoint and blocks until it receives a response. This is described in Section 5.4.

6. It sends a direct framework message to a partition manager and blocks until it receives a response.

5.5.2 Direct message processing

After a receiver starts processing a message, one or more of the following events could occur. The Hypervisor or SPM must treat the invocation of any other events as invalid.

1. It sends a direct partition message response to the endpoint that sent it the request. It blocks until it receives another direct or indirect message. This is described in Section 5.4.

2. It sends a direct partition message request to an endpoint and blocks until it receives a response. This is described in Section 5.4.

3. It gets preempted by an interrupt targeted to the primary scheduler or another SPCI component. This is described in Section 5.5.3.

5.5.3 Preemption during message processing

An endpoint could be interrupted during message processing by a physical interrupt targeted to any SPCI component. The endpoint must be resumed after the interrupt has been handled so that it can continue processing the message. The method to do this depends upon the location of the endpoint relative to the SPCI component that implements the interrupt handler. Responsibilities of participating components are listed below.

- An endpoint must enter the `preempted` state upon being interrupted
- A VM's execution context must be saved upon interruption and restored upon resumption by the Hypervisor
• A SP’s execution context must be saved upon interruption and restored upon resumption by the SPM
• The OS kernel’s execution context must be saved upon interruption and restored upon resumption by the SPM
• If the Hypervisor or SPM must pass control to another SPCI component for handling the physical interrupt, the SPCI_INTERRUPT interface must be used with the appropriate conduit (also see Section 8.3) if the component is:
  – The primary scheduler that resides in EL1
  – An endpoint waiting for direct message response
  – An endpoint in the idle state
• If the physical interrupt is handled in the SPCI component that implements the primary scheduler, then it must use the SPCI_RUN interface to resume the preempted endpoint.

5.5.4 Managed exit

A managed exit is a mechanism in which an endpoint that is processing a direct message is notified about the occurrence of an interrupt targeted to another SPCI component. This allows the endpoint to save the state of its application threads before relinquishing control to the SPM or the Hypervisor.

An endpoint must use the SPCI_MSG_SEND_DIRECT_RESP interface (see Section 11.3) to hand control to the SPM or Hypervisor and complete a managed exit.

The Hypervisor or SPM must save the endpoint’s execution context after it has performed a managed exit. This is in contrast to the endpoint’s execution context being saved and restored transparently by the SPM or Hypervisor as described in Section 5.5.3.

Use of this mechanism leaves the endpoint’s execution context in an idle state. This means that when it is next run on a PE, it can either start processing a new request or resume processing the preempted request.

The use of a managed exit by an endpoint is optional and only available during direct message processing (see Section 5.1.2). An endpoint must specify that it needs to use managed exits in its resource description (see Section 4.1.1).

A managed exit could be used for the following reasons.

1. It enables other application threads in an endpoint to make progress while one or more application threads have been preempted.
2. It ensures that the CPU cycles allocated to an endpoint’s execution context are used to process the request that the calling endpoint 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 on a PE. Endpoint execution contexts pinned to that PE might need to be notified about this event. An endpoint’s execution context could be in a state where it can only resume the request that was previously preempted. This could lead to an unacceptable delay before the endpoint processes the event.

4. It enables application threads to be migrated to a different PE and resumed under the execution context pinned on that PE as opposed to remaining in a preempted state on the original PE until subsequently resumed.

An endpoint could either depend upon the SPM or the Hypervisor for performing a managed exit or implement it on its own. This depends upon the architectural environment the endpoint is executing in as described below.

• An endpoint’s accesses to the GIC (see [9]) could be virtualized e.g. while running in a virtual machine under the control of the Hypervisor or SPM. In this case, the endpoint can only manage virtual interrupts. Physical interrupts are targeted to the exception level that the Hypervisor or SPM are running in. If a physical interrupt must be handled in another SPCI component, the SPM or the Hypervisor must notify
the preempted endpoint that a managed exit is required through an implementation defined mechanism.

- An endpoint could access the physical GIC e.g. a Trusted OS running in S-EL1 under the control of the SPM when the secure virtualization extension is either unavailable or disabled. The endpoint can manage physical interrupts.

An interrupt targeted to another SPCI component can be used by the endpoint to perform a managed exit without changing the state of the interrupt in the GIC. This flow is illustrated in Figure 16.

![Figure 16: Example managed exit flow](image-url)
6 Memory Management

6.1 Overview

The firmware framework describes mechanisms and interfaces that enable SPCI components to manage access and ownership of memory regions in the physical address space to fulfill use cases such as:

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

SPCI components can use a combination of framework and partition messages to manage memory regions in the following ways.

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

6.2 Device access management

The framework enables SPCI components to manage access to the physical address space from a device that is upstream of a SMMU using the memory management transactions described in Section 6.4.

- The manifest of an endpoint to which a device is assigned must also specify the ID of the SMMU and stream IDs the device can generate (also see Table 8).
- The Hypervisor manages the stage 2 translations corresponding to a non-secure stream ID.
- The SPM manages the stage 2 translations corresponding to a secure stream ID.
- 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 translations could map to one or more stream IDs.

A set of SMMU stage 2 translations maintained by a partition manager is called a Stream endpoint. Each stream endpoint is assigned a 16-bit ID called the Stream endpoint ID or SEPID. A SEPID is used in memory management transactions to identify the correct stage 2 translation context in the SMMU corresponding to a combination of the following:

- A unique SMMU instance
- One or more stream IDs
- A unique device corresponding to each stream ID

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

There is a 1:N (N >= 1) mapping between a SEPID and stream IDs assigned to different devices. The stage 2 translations corresponding to the SEPID are shared by one or more stream IDs.

At boot, the partition manifest of each endpoint a device is assigned to specifies the SEPID corresponding to each stream ID generated by the device. This enables the partition manager to discover the identities of the stream IDs that share a SEPID. The stage 2 translations corresponding to a SEPID are configured with no access to the physical address space. Transactions that use the stream IDs that map to this SEPID cannot access physical memory.
An endpoint uses memory management transactions to request changes to the stage 2 translation tables corresponding to the SEPID. A change could:

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

A SEPID could have a proxy PE endpoint that is responsible for authorizing changes to the corresponding stage 2 translations.

- The identity of the proxy PE endpoint must be specified in the manifest of the PE endpoint that a device that uses the SEPID is assigned to.
- The manifest of the proxy PE endpoint must list all the SEPDIs it is a proxy for.
- The proxy PE endpoint must discover the stream IDs that use a SEPID using an implementation defined mechanism.

The devices that generate the stream IDs that use the SEPID rely on the proxy endpoint to validate a memory management transaction with the SEPID as a receiver. The validation takes place before the partition manager changes the translation tables. The memory region is not mapped into the proxy PE endpoint’s translation regime managed by the partition manager.

Alternatively, changes to a SEPID’s stage 2 translations could be authorized by a proxy device that uses the SEPID through its stream IDs e.g. an autonomous device like a DSP. SPCI components must discover the identity of the proxy device through an implementation defined mechanism.

The devices that generate the stream IDs that use the SEPID rely on the proxy device to validate a memory management transaction with the SEPID as a receiver. The validation takes place after the partition manager has changed the translation tables. The partition manager must use an implementation defined mechanism to describe the transaction to the proxy device.

A set of stream IDs must specify one of the above mechanisms (proxy endpoint or device) to manage a SEPID they share. Each stream ID in the set must specify the same mechanism.

### 6.3 Ownership and access attributes

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

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

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

Access control must be enforced through an implementation defined mechanism and/or by encoding these permissions in,

- The stage 1 and stage 2 (if applicable) translation tables of the translation regime of each participating component on each PE.
- The stage 1 and stage 2 (if applicable) translation tables of the SMMUs of DMA capable devices assigned to each participating component. Also see Section 6.2.

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

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

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

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

An endpoint requests access to and/or ownership of a memory region through its partition manifest (also see Table 7).

### 6.3.1 Ownership and access rules

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

1. The size of a memory region to which ownership and access rules apply must be a multiple of the smallest translation granule size supported on the system. This can be discovered.
   - By reading the ID_AA64MMFR0_EL1 system register in the AArch64 execution state
   - Through an implementation defined platform discovery mechanism e.g. DT or ACPI tables
2. A normal memory region must be mapped with the non-secure security attribute in any component that is granted access to it.
3. Each memory region in the physical address space must have a single owner.
4. A SPCI component must have access to a memory region it owns unless it has granted exclusive access to the region to another SPCI component.
5. Only the owner of a memory region must be able to change the security state of the memory region through an implementation defined mechanism.
6. Only the owner of a memory region can grant access to it to one or more borrowers in the system.
7. Only the owner of a memory region can transfer its ownership to another endpoint in the system.
8. If a SP is terminated ‘cause of a fatal error condition, access to the SP’s memory regions and their ownership must be transferred to the SPM.
9. If a VM is terminated ‘cause of a fatal error condition, access to the VM’s memory regions and their ownership must be transferred to the Hypervisor.
10. If the Hypervisor or OS kernel are terminated ‘cause of a fatal error condition, access to the their memory regions and ownership must be transferred to the SPM.
11. The number of distinct components to whom an owner can grant access to a memory region is implementation defined.
12. The owner of a memory region must not be able to change its ownership or access attributes until all borrowers have relinquished access to it.

### 6.3.2 Ownership and access states

Table 19 describes the ownership states applicable to a SPCI component for a memory region.
Table 19: Ownership states

<table>
<thead>
<tr>
<th>No.</th>
<th>Ownership state</th>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Owner</td>
<td>Owner</td>
<td>Component owns the memory region.</td>
</tr>
<tr>
<td>2</td>
<td>Not Owner</td>
<td>!Owner</td>
<td>Component does not own the memory region.</td>
</tr>
</tbody>
</table>

Table 20 describes the access states applicable to a SPCI component for a memory region.

Table 20: Access states

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

Table 21 describes the valid combination of access and ownership states applicable to a SPCI component for a memory region.

Table 21: Valid combinations of ownership and access states

<table>
<thead>
<tr>
<th>No.</th>
<th>Ownership state</th>
<th>Access state</th>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Not owner</td>
<td>No access</td>
<td>!Owner-NA</td>
<td>Component has neither ownership nor access to the memory region.</td>
</tr>
<tr>
<td>2</td>
<td>Not owner</td>
<td>Exclusive access</td>
<td>!Owner-EA</td>
<td>Component has exclusive access without ownership of the memory region.</td>
</tr>
<tr>
<td>3</td>
<td>Not owner</td>
<td>Shared access</td>
<td>!Owner-SA</td>
<td>Component has shared access with one or more components without ownership of the memory region.</td>
</tr>
<tr>
<td>4</td>
<td>Owner</td>
<td>No access</td>
<td>Owner-NA</td>
<td>Component owns the memory region and has granted exclusive access to the memory region to another component.</td>
</tr>
<tr>
<td>5</td>
<td>Owner</td>
<td>Exclusive access</td>
<td>Owner-EA</td>
<td>Component owns the memory region and has exclusive access to it.</td>
</tr>
<tr>
<td>6</td>
<td>Owner</td>
<td>Shared access</td>
<td>Owner-SA</td>
<td>Component owns the memory region and shares access to it with one or more components.</td>
</tr>
</tbody>
</table>

For two SPCI components A and B and a memory region, valid combinations of states defined in Table 21 are described in Table 22. Other combinations of states are considered invalid.
<table>
<thead>
<tr>
<th>No.</th>
<th>Component A state</th>
<th>Component B state</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Owner-EA</td>
<td>!Owner-NA</td>
<td>Component A has exclusive access and ownership of a memory region that is inaccessible from component B.</td>
</tr>
<tr>
<td>2</td>
<td>Owner-NA</td>
<td>!Owner-NA</td>
<td>Component A has granted exclusive access to a memory region it owns to another component. It is inaccessible from component B.</td>
</tr>
<tr>
<td>3</td>
<td>Owner-NA</td>
<td>!Owner-EA</td>
<td>Component A has granted exclusive access to a memory region it owns to component B.</td>
</tr>
<tr>
<td>4</td>
<td>Owner-NA</td>
<td>!Owner-SA</td>
<td>Component A has relinquished access to a memory region it owns. Access to the memory region is shared between component B and at least one other component</td>
</tr>
<tr>
<td>5</td>
<td>Owner-SA</td>
<td>!Owner-NA</td>
<td>Component A shares access to a region of memory it owns with another component. Component B cannot access the memory region.</td>
</tr>
<tr>
<td>6</td>
<td>Owner-SA</td>
<td>!Owner-SA</td>
<td>Component A shares access to a region of memory it owns with component B and possibly other components.</td>
</tr>
</tbody>
</table>

To fulfill the use cases and enforce the rules listed earlier, SPCI components should track the state of a memory region. This could be done as described below:

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

### 6.4 Memory management transactions

The ownership and access attributes described in Section 6.3 apply to all SPCI components. The memory management transactions described in this section apply only to endpoints. The use of these transactions to transfer ownership and manage access to a memory region between an endpoint and a partition manager is currently beyond the scope of this version of the communication framework. Any partner feedback in this regard is welcome.

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

- Transitions between states described in Section 6.3.2 happen in response to transactions described in Table 23. Each transaction involves exchange of one or more framework and partition messages.
Each transition is described as a transaction involving two endpoints (A and B) and a memory region. Endpoint A is the owner of the memory region.

**Table 23: Memory region transactions**

<table>
<thead>
<tr>
<th>No.</th>
<th>Transaction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Donate</td>
<td>Endpoint A transfers ownership of a memory region it owns to endpoint B. See Section 6.5.</td>
</tr>
<tr>
<td>2.</td>
<td>Lend</td>
<td>Endpoint A relinquishes access to a memory region and grants it to only endpoint B. Endpoint B gains exclusive access to the memory region. Endpoint A relinquishes access to a memory region and grants it to endpoint B and at least one other endpoint simultaneously. Endpoint B gains shared access to the memory region. See Section 6.6.</td>
</tr>
<tr>
<td>3.</td>
<td>Share</td>
<td>Endpoint A grants access to a memory region to endpoint B and optionally to other endpoints simultaneously. See Section 6.7.</td>
</tr>
<tr>
<td>3.</td>
<td>Relinquish</td>
<td>Endpoint B relinquishes access to a memory region granted to it by Endpoint A. Endpoint A reclaims exclusive access to the memory region. See Section 6.8.</td>
</tr>
</tbody>
</table>

### 6.4.1 Component roles

In this version of the framework, endpoints can fulfill the role of an owner, lender or borrower (see Section 6.3). The Hypervisor and SPM participate in memory management transactions to validate and transmit them from a sender to a receiver. They are also responsible for managing the translation regime of an endpoint and tracking the ownership and access attributes of a memory region. This collective role is termed as a Relayer. Table 24 specifies the roles each SPCI component can play in a memory management transaction.

In a split SPM configuration (see Section 3.8), the role of the SPM instance at the higher EL (see Table 5) as a relayer is to forward memory management transactions between the secure and non-secure physical SPCI instances.

- The SPM at the higher EL does not modify the out-bound transaction from the SPM at the lower EL.
- The SPM at the higher EL does not modify the inbound transaction to the SPM at the lower EL.
- The SPM at the higher EL does not map or unmap memory regions in any translation regime of the SPM at the lower EL.

It is possible that in a split SPM configuration, the SPM instance at the lower EL coexists with a SP at the same EL. The role of the SPM instance as the relayer and the SP as the owner, borrower or lender are still considered to be logically separate. The interface used by the two components to exchange memory management transactions is implementation defined. It is strongly recommended that this interface between the two components mirrors that between a SPM and a SP in separate exception levels as closely as possible. The SP and SPM must still appear as separate SPCI components to software in the Normal world and SPs at the secure virtual SPCI instance.
### 6.4.2 Component configurations

In all transactions, a list of valid combinations of roles played by various SPCI components is specified in Table 25. The specific roles of the sender and receiver depend upon the type of transaction. This is described below.

- In a transaction to donate ownership of a memory region, the sender is the current owner and the receiver is the new owner.
- In a transaction to lend or share access to a memory region, the sender is the lender and the receiver is the borrower.
- In a transaction to relinquish access to a memory region, the sender is the borrower and the receiver is the lender.

The role of the relayer is described in detail later in this section.

**Table 25: Valid component configurations for memory management transactions**

<table>
<thead>
<tr>
<th>Config no.</th>
<th>Sender</th>
<th>Receiver</th>
<th>Relayer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>VM</td>
<td>VM</td>
<td>Hypervisor</td>
</tr>
<tr>
<td>2.</td>
<td>NS-Endpoint</td>
<td>SP</td>
<td>Hypervisor (if present) and SPM</td>
</tr>
<tr>
<td>3.</td>
<td>SP</td>
<td>SP</td>
<td>SPM</td>
</tr>
<tr>
<td>4.</td>
<td>SP</td>
<td>NS-Endpoint</td>
<td>SPM and Hypervisor (if present)</td>
</tr>
</tbody>
</table>

### 6.4.3 Transaction life-cycle

Each transaction described in Table 23 takes place in three steps described below and illustrated in Figure 17.

1. The sender sends a framework message to the relayer to start a transaction involving one or more receivers
2. The sender sends a partition message requesting each receiver to complete the transaction.
3. Each receiver sends a framework message to the relayer to complete the transaction

If the sender is the owner of the memory region involved in a transaction, then it can also abort the transaction before the receiver completes it.

In a split SPM configuration (see Section 3.8), the framework message sent in step 1 could be directly delivered to the SP at the secure physical SPCI instance thus making steps 2 and 3 unnecessary. A successful return of the interface used in step 1 completes the transaction. The following aspects of memory management transaction in this scenario are implementation defined.

1. How the sender discovers the presence of a split SPM configuration
2. How the relayer delivers the framework message to the receiver
3. How the receiver interacts with the relayer to complete the transaction

![Figure 17: Memory management transaction lifecycle](image)

### 6.5 Donate memory transaction

This transaction is used to transfer the ownership of a memory region from the endpoint that owns it to another endpoint. The owner also specifies the memory attributes with which the memory region must be mapped in the receiver's translation regime.

#### 6.5.1 Donate memory state machine

Table 26 describes the state machine for transferring ownership to a memory region from the perspective of two components A & B.

A owns the memory region. It could have lent or shared the memory region with B or another component. Valid and invalid state transitions in response to a transaction where A attempts to donate the memory region to B have been listed.

In each valid transition,
• A loses both ownership and access to the memory region and enters the !Owner-NA state.
• B gains ownership and exclusive access to the memory region and enters the Owner-EA state.

<table>
<thead>
<tr>
<th>No.</th>
<th>Current Endpoint A state</th>
<th>Current Endpoint B state</th>
<th>Next Endpoint A state</th>
<th>Next Endpoint B state</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Owner-EA</td>
<td>!Owner-NA</td>
<td>!Owner-NA</td>
<td>Owner-EA</td>
<td>• Owner has exclusive access to the memory region and transfers ownership to endpoint B.</td>
</tr>
<tr>
<td>2</td>
<td>Owner-NA</td>
<td>!Owner-NA</td>
<td>Error</td>
<td>–</td>
<td>• Owner does not have exclusive access to the memory region. It cannot transfer its ownership.</td>
</tr>
<tr>
<td>3</td>
<td>Owner-NA</td>
<td>!Owner-EA</td>
<td>!Owner-NA</td>
<td>Owner-EA</td>
<td>• Owner does not have exclusive access to the memory region. Endpoint B has exclusive access to it. Owner transfers ownership to endpoint B.</td>
</tr>
<tr>
<td>4</td>
<td>Owner-NA</td>
<td>!Owner-SA</td>
<td>Error</td>
<td>–</td>
<td>• Owner has lent access to the memory region to endpoint B and possibly other endpoints. It cannot transfer its ownership.</td>
</tr>
<tr>
<td>5</td>
<td>Owner-SA</td>
<td>!Owner-NA</td>
<td>Error</td>
<td>–</td>
<td>• Owner has shared access to the memory region with one or more endpoints. It cannot transfer its ownership.</td>
</tr>
<tr>
<td>6</td>
<td>Owner-SA</td>
<td>!Owner-SA</td>
<td>Error</td>
<td>–</td>
<td>• Owner has shared access to the memory region with endpoint B and possibly other endpoints. It cannot transfer its ownership.</td>
</tr>
</tbody>
</table>

### 6.5.2 Donate memory transaction lifecycle

This transaction takes place in three steps described below (also see Section 6.4.3).

1. The owner uses the SPCI_MEM_DONATE interface to describe the memory region and convey the identity of the receiver to the relayer as specified in Table 32. This interface is described in Section 12.1.

2. The owner uses a partition message to request the receiver to receive the donated memory region. This message contains a description of the memory region relevant to the receiver.

3. The receiver uses the SPCI_MEM_RETRIEVE_REQ and SPCI_MEM_RETRIEVE_RESP interfaces to map the memory region in its translation regime and complete the transaction. These interfaces are described in Section 12.4 & Section 12.5 respectively.
6.6 Lend memory transaction

This transaction is used by an owner to relinquish its access to a memory region and grant access to it to one or more borrowers. If the region is lent to a single borrower, it is granted exclusive access to it. If the region is lent to more than borrower, they are granted shared access to it.

6.6.1 Lend memory transaction state machine

This section describes the state machine for the lend memory transaction for a memory region from the perspective of two components A & B.

A owns the memory region. It relinquishes its access to the memory region and grants shared or exclusive access to it to B. Valid and invalid state transitions in response to the following transactions have been specified.

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

6.6.2 Lend memory transaction lifecycle

This transaction takes place in three steps described below (also see Section 6.4.3).

1. The owner uses the SPCI_MEM_LEND interface to describe the memory region and convey the identities
of the borrowers to the relayer as specified in Table 32. This interface is described in Section 12.2.

2. The owner uses a partition message to request each borrower to receive the lent memory region. This message contains a description of the memory region relevant to the borrower.

3. Each borrower uses the `SPCI_MEM_RETRIEVE_REQ` and `SPCI_MEM_RETRIEVE_RESP` interfaces to map the memory region in its translation regime and complete the transaction. These interfaces are described in Section 12.4 & Section 12.5 respectively.

### 6.7 Share memory transaction

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

#### 6.7.1 Share memory transaction state machine

This section describes the state machine for the share memory transaction for a memory region from the perspective of two components `A` & `B`.

A owns the memory region. It grants access to the memory region to `B`. Valid and invalid state transitions in response to the following transactions have been specified.

**Table 28: Share memory transaction state machine**

<table>
<thead>
<tr>
<th>No.</th>
<th>Current Endpoint A state</th>
<th>Current Endpoint B state</th>
<th>Next Endpoint A state</th>
<th>Next Endpoint B state</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Owner-EA</td>
<td>!Owner-NA</td>
<td>Owner-SA</td>
<td>!Owner-SA</td>
<td>• Owner has exclusive access to the memory region and grants access to it to one or more borrowers including endpoint B.</td>
</tr>
<tr>
<td>2</td>
<td>Owner-NA</td>
<td>!Owner-NA</td>
<td>Error</td>
<td>–</td>
<td>• Owner has already lent the memory region to one or more endpoints. It cannot share it with endpoint B.</td>
</tr>
<tr>
<td>3</td>
<td>Owner-NA</td>
<td>!Owner-EA</td>
<td>Error</td>
<td>–</td>
<td>• Owner has already lent the memory region to endpoint B with exclusive access.</td>
</tr>
<tr>
<td>4</td>
<td>Owner-NA</td>
<td>!Owner-SA</td>
<td>Error</td>
<td>–</td>
<td>• Owner has already lent the memory region to endpoint B and other endpoints.</td>
</tr>
<tr>
<td>5</td>
<td>Owner-SA</td>
<td>!Owner-NA</td>
<td>Error</td>
<td>–</td>
<td>• Owner has already shared the memory region with one or more endpoints. It cannot share it with endpoint B.</td>
</tr>
<tr>
<td>6</td>
<td>Owner-SA</td>
<td>!Owner-SA</td>
<td>Error</td>
<td>–</td>
<td>• Owner has already shared the memory region with endpoint B and possibly other endpoints.</td>
</tr>
</tbody>
</table>
6.7.2 Share memory transaction lifecycle

This transaction takes place in three steps described below (also see Section 6.4.3).

1. The owner uses the SPCI_MEM_SHARE interface to describe the memory region and convey the identities of the borrowers to the relayer as specified in Table 32. This interface is described in Section 12.2.

2. The owner uses a partition message to request each borrower to receive the shared memory region. This message contains a description of the memory region relevant to the borrower.

3. Each borrower uses the SPCI_MEM_RETRIEVE_REQ and SPCI_MEM_RETRIEVE_RESP interfaces to map the memory region in its translation regime and complete the transaction. These interfaces are described in Section 12.4 & Section 12.5 respectively.

6.8 Relinquish memory transaction

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

6.8.1 Relinquish memory access state machine

Table 29 describes the state machine for the relinquish memory access transaction for a memory region from the perspective of two components A & B.

A owns the memory region. It could have granted shared or exclusive access to the memory region to B. Alternatively, B might not have access to the memory region. B attempts to relinquish access to this memory region. Valid and invalid state transitions in response to this transaction have been specified.

<table>
<thead>
<tr>
<th>No.</th>
<th>Current Endpoint A state</th>
<th>Current Endpoint B state</th>
<th>Next Endpoint A state</th>
<th>Next Endpoint B state</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Owner-EA</td>
<td>!Owner-NA</td>
<td>Error</td>
<td>–</td>
<td>• Endpoint B tries to relinquish access to a memory region that the owner has exclusive access to.</td>
</tr>
<tr>
<td>2</td>
<td>Owner-NA</td>
<td>!Owner-NA</td>
<td>Error</td>
<td>–</td>
<td>• Endpoint B tries to relinquish access to a memory region that the owner has granted shared or exclusive access to one or more other borrowers.</td>
</tr>
<tr>
<td>3</td>
<td>Owner-NA</td>
<td>!Owner-EA</td>
<td>Owner-EA</td>
<td>!Owner-NA</td>
<td>• Endpoint B relinquishes exclusive access to the memory region and transfers it back to the owner.</td>
</tr>
<tr>
<td>No.</td>
<td>Current Endpoint A state</td>
<td>Current Endpoint B state</td>
<td>Next Endpoint A state</td>
<td>Next Endpoint B state</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>--------------------------</td>
<td>--------------------------</td>
<td>----------------------</td>
<td>----------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>4</td>
<td>Owner-NA</td>
<td>!Owner-SA</td>
<td>Owner-EA</td>
<td>!Owner-NA</td>
<td>• Endpoint B relinquishes access to the memory region that it shares with other borrowers. Owner reclaims exclusive access once all borrowers have relinquished access.</td>
</tr>
<tr>
<td>5</td>
<td>Owner-SA</td>
<td>!Owner-NA</td>
<td>Error —</td>
<td>—</td>
<td>• Endpoint B tries to give up access to a memory region that the owner shares with one or more other borrowers.</td>
</tr>
<tr>
<td>6</td>
<td>Owner-SA</td>
<td>!Owner-SA</td>
<td>Owner-EA</td>
<td>!Owner-NA</td>
<td>• Endpoint B relinquishes access to the memory region that it shares with the owner and possibly other borrowers. Owner reclaims exclusive access once all borrowers have relinquished access.</td>
</tr>
</tbody>
</table>

### 6.8.2 Relinquish memory transaction lifecycle

This transaction takes place in three steps described below (also see Section 6.4.3). It is assumed that the memory region was originally lent or shared by the owner to the borrowers.

1. Each borrower uses the `SPCI_MEM_RELINQUISH` interface with the global handle allocated by the relayer to indicate that the memory region can be unmapped from its translation regime. This interface is described in Section 12.6.

2. Each borrower uses a partition message to inform the owner that it has relinquished access to the memory region. The memory region should be identified by its handle allocated by the relayer.

3. Once all borrowers have relinquished access to the memory region, the owner uses the `SPCI_MEM_RECLAIM` interface with the handle to reclaim exclusive access to the memory region. This interface is described in Section 12.7.

### 6.9 Memory region description

A memory region is described in a memory management transaction by specifying one or more of the following properties:

1. The list of 4K pages that constitute it. This list is specified by using one or more constituent memory region descriptors (see Table 30). Each descriptor specifies the base address and size of a virtually or physically contiguous memory region. The pages are addressed using VAs, IPAs or PAs depending upon the SPCI instance at which the transaction is taking place. This is described below.
   - VAs are used at the following secure virtual SPCI instances listed in Table 3.
     - SPCI instance 3.
     - SPCI instance 4.
– SPCI instance 5.
– SPCI instance 6.

• IPAs are used at the following secure and non-secure virtual SPCI instances listed in Table 3.
  – SPCI instance 1.
  – SPCI instance 2.
  – SPCI instance 11.
  – SPCI instance 12.
  – SPCI instance 13.

• PAs are used at all the secure and non-secure physical SPCI instances listed in Table 3. These instances are also listed below.
  – SPCI instance 7.
  – SPCI instance 8.
  – SPCI instance 9.
  – SPCI instance 10.
  – SPCI instance 14.
  – SPCI instance 15.
  – SPCI instance 16.

2. Identity of the endpoint, access permissions and memory attributes with which the memory region must be mapped in the translation regime of this endpoint managed by the relayer (see Table 31).

3. A globally unique *Handle* to identify the memory region allocated by the relayer.

### Table 30: Constituent memory region descriptor

<table>
<thead>
<tr>
<th>Field</th>
<th>Byte length</th>
<th>Byte offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>8</td>
<td>–</td>
<td>• Base VA, PA or IPA of constituent memory region aligned to the page size (4K) granularity.</td>
</tr>
<tr>
<td>Page count</td>
<td>4</td>
<td>8</td>
<td>• Number of 4K pages in constituent memory region.</td>
</tr>
</tbody>
</table>

### Table 31: Memory region attributes descriptor

<table>
<thead>
<tr>
<th>Field</th>
<th>Byte length</th>
<th>Byte offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID of receiver</td>
<td>2</td>
<td>–</td>
<td>• 16-bit ID of receiver endpoint</td>
</tr>
<tr>
<td>Field</td>
<td>Byte length</td>
<td>Byte offset</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------</td>
<td>-------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Memory attributes</td>
<td>2</td>
<td>2</td>
<td>• Memory attributes with which this memory region must be/is mapped in receiver’s translation tables.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- bits[15:7]: Reserved (MBZ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- bits[6:5]: Data and instruction access permissions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• b’00: Read-only, Not executable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• b’01: Read-only, Executable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• b’10: Read-write, Not executable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• b’11: Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- bits[4]: Memory type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• b’0: Device memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• b’1: Normal memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- bits[3:2]:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• b’00: Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• b’01: Non-cacheable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• b’10: Write-Through</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• b’11: Write-Back</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Device memory attributes if bit[4] = b’0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• b’00: Device-nGnRnE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• b’01: Device-nGnRE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• b’10: Device-nGRE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• b’11: Device-GRE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• b’00: Non-shareable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• b’01: Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• b’10: Outer Shareable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• b’11: Inner Shareable</td>
</tr>
</tbody>
</table>

Figure 18 illustrates example memory region descriptors of a virtually contiguous region VA_0 of size 16K. It is composed of two IPA regions IPA_0 and IPA_1 of size 8K each.

- IPA_0 is comprised of two PA regions PA_0 and PA_1. Each PA region is of size 4K.
- IPA_1 is comprised of two PA regions PA_2 and PA_3. Each PA region is of size 4K.
Figure 18: Memory region descriptor example

Constituent memory region descriptors:
- **Count = 2**
  - Address = IPA_0
  - Page Count = 2
  - Address = IPA_1
  - Page Count = 2

Constituent memory region descriptors:
- **Count = 4**
  - Address = PA_0
  - Page Count = 1
  - Address = PA_1
  - Page Count = 1
  - Address = PA_2
  - Page Count = 1
  - Address = PA_3
  - Page Count = 1
Table 32 specifies the data structure that must be used by the owner to initiate a transaction to donate, lend or share a memory region. It uses the constituent memory region and memory region attributes descriptors described in Table 30 and Table 31.

**Table 32: Descriptor to donate, lend or share a memory region**

<table>
<thead>
<tr>
<th>Field</th>
<th>Byte length</th>
<th>Byte offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>4</td>
<td>0</td>
<td>• Implementation defined value associated with the receiver and the memory region.</td>
</tr>
<tr>
<td>Flags</td>
<td>4</td>
<td>4</td>
<td>• Flags to govern behavior of the transaction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>– bit[0]: Clear memory region contents after unmapping it from the sender's translation regime and before mapping it in any receiver's translation regime.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>– bit[31:1]: Reserved (MBZ).</td>
</tr>
<tr>
<td>Total page count</td>
<td>4</td>
<td>8</td>
<td>• Size of the memory region described as the count of 4K pages.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Must be equal to the sum of page counts specified in each constituent memory descriptor.</td>
</tr>
<tr>
<td>Constituent memory region count</td>
<td>4</td>
<td>12</td>
<td>• Count of constituent memory regions.</td>
</tr>
<tr>
<td>Constituent memory region descriptor offset</td>
<td>4</td>
<td>16</td>
<td>• Offset from the base address of this descriptor to an array of constituent memory region descriptors. See Table 30.</td>
</tr>
<tr>
<td>Memory region attributes descriptor count</td>
<td>4</td>
<td>20</td>
<td>• Count of memory region attributes descriptors.</td>
</tr>
<tr>
<td>Memory region attributes descriptors</td>
<td>–</td>
<td>24</td>
<td>• Array of memory region attributes descriptors. See Table 31. Each entry in the array describes attributes with which the memory region must be mapped in the specified endpoint.</td>
</tr>
</tbody>
</table>

### 6.10 Transmission of transaction data

Interfaces that implement a memory management transaction use data structures to transmit information related to the transaction. For example,

- Table 32 specifies the data structure used to initiate a transaction to donate, lend or share a memory region.
- Table 124 specifies the data structure used to retrieve a memory region.

This version of the framework assumes that by default, these data structures are populated in the RX/TX buffers of an endpoint, Hypervisor or SPM when an applicable memory management interface is invoked e.g. `SPCI_MEM_DONATE`, `SPCI_MEM_LEND` etc.

- The TX buffer is used to transmit data structures from an endpoint to the Hypervisor or SPM
- The TX buffer is used to transmit data structures from the Hypervisor to the SPM
- The RX buffer is used to transmit data structures from the Hypervisor or SPM to an endpoint
- The RX buffer is used to transmit data structures from the SPM to the Hypervisor

It is possible that the size of a data structure is larger than the RX or TX buffer. The following mechanisms can be used by the implementation to address this scenario.
1. The caller of an interface could allocate a separate area of memory that is large enough to accommodate the data structure. The applicable memory management interfaces include support to let the caller specify this memory as described below.

   - The VA, IPA or PA of the memory region.
     - If a value of 0 is specified then it is assumed that the RX or TX buffer is used as specified above and the attributes of the memory region are ignored.
   - The remaining attributes of the memory region must be the same as those specified for a RX/TX buffer in Section 5.2.2.3.

The callee must map the memory region in its translation regime upon invocation of the interface and unmap it upon completion.

2. It is possible that the caller can only allocate a memory region in that is larger than the RX/TX buffer but still smaller than the size of the data structure. The applicable memory management interfaces include support to let the caller break the data structure into fragments such that each fragment fits into the memory region. The caller can then invoke the interface as many times as there are fragments to transmit the complete data structure to the callee. The callee must ensure that the memory area remains mapped in its translation regime until transmission is over.

3. It is possible that the callee does not implement support to dynamically map a region of memory in its translation regime apart from the RX/TX buffers using the SPCI_RXTX_MAP interface. The applicable management interfaces include support to let the caller break the data structure into fragments such that each fragment fits into the RX or TX buffer. The caller can then invoke the interface as many times as there are fragments to transmit the complete data structure to the callee.

To support mechanisms described in 2 and 3 above,

   - The first invocation of the interface must provide the following information. If values of 0 are specified then it is assumed that the data structure can fit in the specified memory region.
     - A handle unique to the caller to associate a fragment with the overall data structure.
     - Number of fragments or invocations of this interface after the first invocation that the callee must expect.
     - Total length of the data structure to be transmitted.
   - Subsequent invocations of the interface to transmit the remaining fragment must provide the following information.
     - The handle unique to the caller to associate a fragment with the overall data structure.
     - The fragment number.
   - For each invocation of the interface to transmit a fragment of the data structure, the callee must respond with an invocation of the:
     - SPCI_SUCCESS function to indicate successful transmission
     - SPCI_ERROR function to indicate a failed transmission. The RETRY error code must be used to request the caller to retry the transaction

The implementation at a SPCI instance must support at least one of the above mechanisms to transmit a data structure larger than the RX/TX buffer. The partition managers must describe the mechanisms they implement to an endpoint through an implementation defined mechanism.

The ABIs that incorporate support for these mechanisms are listed below.

   - SPCI_MEMDonate. This interface is described in Section 12.1.
   - SPCI_MEM_Lend. This interface is described in Section 12.2.
   - SPCI_MEM_Share. This interface is described in Section 12.3.
   - SPCI_MEM_Retrieve_REQ. This interface is described in Section 12.4.
   - SPCI_MEM_Retrieve_RESP. This interface is described in Section 12.5.
Figure 19 illustrates an example where the *SPCI_MEM_RETRIEVE_REQ* and *SPCI_MEM_RETRIEVE_RESP* interfaces are used to retrieve a memory region description through multiple invocations. The following assumptions have been made.

- The memory region is shared with only a single borrower.
- The RX/TX buffers of the borrower are used by these interfaces.
- The descriptor in Table 124 requires two invocations of the *SPCI_MEM_RETRIEVE_REQ* interface to be delivered to the relayer.
- The descriptor in Table 130 requires two invocations of the *SPCI_MEM_RETRIEVE_RESP* interface to be delivered to the borrower.
Figure 19: Example of fragment transmission while retrieving memory
The interfaces used by SPCI components for communication at an SPCI instance are described in the following sections.

- Interfaces for reporting status of execution of other interfaces are described in Section 8.
- Interfaces for partition setup and discovery using framework messages are described in Section 9.
- Interfaces to implement memory management transactions using framework messages are described in Section 12.
- Interfaces to manage CPU cycles allocated to an endpoint are described in Section 10.
- Interfaces to implement exchange of direct and indirect partition messages between endpoints are described in Section 11.

Each interface can be invoked using one more conduits described in Section 3.2. Each interface is based upon the AArch64 and AArch32 SMC calling convention described in [5]. Usage of only those architectural registers that are relevant to an interface is specified. The values of all other architectural registers must be ignored.

The following standard secure service call identifier ranges have been reserved for SPCI interfaces in the SMCCC [5].

1. 0x84000060-0x8400007F: SPCI 32-bit calls.
2. 0xC4000060-0xC400007F: SPCI 64-bit calls.

- If the caller is in the AArch32 execution state, it must use the function identifiers for 32-bit calls.
- If the callee is in the AArch64 execution state, it could use a function identifier for 32-bit or 64-bit calls.

**Provisional**

**Open Topic**

The standard secure service call identifier ranges have been provisionally allocated. The semantics of message passing interfaces do not completely align with the definition of Fast calls as described in [5]. For example, SPCI_MSG_WAIT could be interrupted inspite of being a Fast call. Its semantics are more aligned with Standard calls. The Standard call function identifier range is currently allocated to Trusted OSs. Hence, the use of that range for SPCI will require a significant update to [5]. This topic is under internal discussion and partner feedback is welcome.

**Provisional**

**Open Topic**

The names of interfaces described in the following sections could be changed to better reflect their purpose. This is being considered for the next version of this specification. Any partner feedback in this regard is welcome.
8 Status reporting interfaces

8.1 SPCI_ERROR

Description

- Returns error code in response to a previous invocation of a SPCI function
- Table 34 defines the values for status codes used with SPCI functions. All values are considered to be 32-bit signed integers
- Valid SPCI instances and conduits are listed in Table 35
- Syntax of this function is described in Table 36
- Figure 20 illustrates example usage of this function with the following assumptions
  - Component A makes an invalid request to Component B through an SPCI function described in this specification
  - Component B uses the SPCI_ERROR function to return the error code to Component A
  - The SPCI function used by component A can be invoked through the SMC and ERET conduits
  - Both components could be interacting at any SPCI instance supported by the SPCI function. The two possible scenarios have been considered.
    - Component A is at a lower EL than component B at the SPCI instance
    - Component A is at a higher EL than component B at the SPCI instance

![Figure 20: Example usage of SPCI_ERROR](image)

Table 34: Error status codes

<table>
<thead>
<tr>
<th>Status code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>NOT_SUPPORTED</td>
</tr>
<tr>
<td>-2</td>
<td>INVALID_PARAMETERS</td>
</tr>
<tr>
<td>-3</td>
<td>NO_MEMORY</td>
</tr>
<tr>
<td>Status code</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------</td>
</tr>
<tr>
<td>-4</td>
<td>BUSY</td>
</tr>
<tr>
<td>-5</td>
<td>INTERRUPTED</td>
</tr>
<tr>
<td>-6</td>
<td>DENIED</td>
</tr>
<tr>
<td>-7</td>
<td>RETRY</td>
</tr>
</tbody>
</table>

Table 35: SPCI_ERROR instances and conduits

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Physical SPCI instance</td>
<td>SMC, ERET</td>
</tr>
<tr>
<td>2</td>
<td>Virtual SPCI instance</td>
<td>SMC, HVC, SVC, ERET</td>
</tr>
</tbody>
</table>

Table 36: SPCI_ERROR function syntax

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Function ID</td>
<td>w0</td>
<td></td>
<td>0x84000060</td>
</tr>
<tr>
<td>uint32 Target information</td>
<td>w1</td>
<td></td>
<td>- Information to identify target SP/VM</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Only valid when SMC conduit is used. MBZ otherwise.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Bits[31:16]: ID of SP/VM.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Bits[15:0]: ID of vCPU of SP/VM to deliver error to.</td>
</tr>
<tr>
<td>int32 Error code</td>
<td>w2</td>
<td></td>
<td>SPCI function specific error code. See function definition for applicable error codes</td>
</tr>
<tr>
<td>Other Parameter registers</td>
<td>w3-w7</td>
<td>x3-x7</td>
<td>Reserved (MBZ)</td>
</tr>
</tbody>
</table>
### 8.2 SPCI_SUCCESS

**Description**

- Returns results upon successful completion of a previous invocation of a SPCI function
- Valid SPCI instances and conduits are listed in Table 38
- Syntax of this function is described in Table 39
- Figure 21 illustrates example usage of this function with the following assumptions
  - Component A makes an valid request to Component B through an SPCI function described in this specification
  - Component B uses the SPCI_SUCCESS function to return the results to Component A
  - The SPCI function used by component A can be invoked through the SMC and ERET conduits
  - Both components could be interacting at any SPCI instance support by the SPCI function. The two possible scenarios have been considered.
    - Component A is at a lower EL than component B at the SPCI instance
    - Component A is at a higher EL than component B at the SPCI instance

![SPCI_SUCCESS example from higher to lower EL](image)

![SPCI_SUCCESS example from lower to higher EL](image)

**Figure 21: Example usage of SPCI_SUCCESS**

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Physical SPCI instance</td>
<td>SMC, ERET</td>
</tr>
<tr>
<td>2</td>
<td>Virtual SPCI instance</td>
<td>SMC, HVC, SVC, ERET</td>
</tr>
</tbody>
</table>

**Table 39: SPCI_SUCCESS function syntax**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function ID</td>
<td>uint32</td>
<td>w0</td>
<td>0x84000061</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0xC4000061</td>
</tr>
</tbody>
</table>
### Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Register Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Target</td>
<td>w1</td>
<td>Information to identify target SP/VM</td>
</tr>
<tr>
<td>information</td>
<td></td>
<td>• Only valid when SMC conduit is used. MBZ otherwise.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Bits[31:16]: ID of SP/VM.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Bits[15:0]: ID of vCPU of SP/VM to deliver results to.</td>
</tr>
<tr>
<td>uint32/uint64 Result</td>
<td>w2-w7</td>
<td>SPCI function specific return results. See function definition for result</td>
</tr>
<tr>
<td>registers</td>
<td>x2-x7</td>
<td>encoding</td>
</tr>
</tbody>
</table>
8.3 SPCI_INTERRUPT

Description

- Returns control from the caller to the callee in response to an interrupt that must be:
  - Either handled by the callee
  - Or handled by another SPCI component reachable only through the callee
- Valid SPCI instances and conduits are listed in Table 41
- Syntax of this function is described in Table 42

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Non-secure physical SPCI instance</td>
<td>ERET</td>
</tr>
<tr>
<td>2</td>
<td>Virtual SPCI instance</td>
<td>ERET</td>
</tr>
<tr>
<td>3</td>
<td>Secure physical SPCI instance</td>
<td>SMC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
</table>
| Declaration
  uint32 Function ID   | Register w0 | Value 0x84000062 |
  uint32 Endpoint/vCPU IDs   | w1          | Endpoint and vCPU IDs of the caller.
  - Bits[31:16]: Endpoint ID.
  - Bits[15:0]: vCPU ID. |
  uint32 Interrupt ID      | w2          | Interrupt ID. Only valid at secure virtual SPCI instance with a S-EL0 SP as callee |
  Other parameter registers | w3-w7, x3-x7 | Reserved (MBZ) |
9 Setup and discovery interfaces

9.1 SPCI_VERSION

Description
- Returns version of the firmware framework implementation at a SPCI instance
- Valid SPCI instances and conduits are listed in Table 44
- Syntax of this function is described in Table 45
- Encoding of result parameters in the SPCI_SUCCESS function is described in Table 46
- Encoding of error code in the SPCI_ERROR function is described in Table 47

Table 44: SPCI_VERSION instances and conduits

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Physical SPCI instance</td>
<td>SMC</td>
</tr>
<tr>
<td>2</td>
<td>Virtual SPCI instance</td>
<td>SMC, HVC, SVC</td>
</tr>
</tbody>
</table>

Table 45: SPCI_VERSION function syntax

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Function ID</td>
<td>w0</td>
<td>0x84000063</td>
</tr>
<tr>
<td>Other Parameter registers</td>
<td>w1-w7</td>
<td>x1-x7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Reserved (MBZ)</td>
</tr>
</tbody>
</table>

Table 46: SPCI_SUCCESS encoding

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
</table>
| int32 Version                | w2       | • Upon a successful return, the format of the value is as follows
|                              |          |   – Bit [31]: Must be 0
|                              |          |   – Bits [30:16] Major Version: Must be 0 for this revision of SPCI
|                              |          |   – Bits [15:0] Minor Version: Must be 9 for this revision of SPCI
| Other Result registers        | w3-w7    | x3-x7                                          |
|                              |          | Reserved (MBZ)                                |

Table 47: SPCI_ERROR encoding

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32 Error code</td>
<td>w2</td>
<td>• NOT_SUPPORTED: A SPCI implementation does not exist at this SPCI instance</td>
</tr>
</tbody>
</table>
The version returned by this interface is provisionally 0.9 as the specification is under review. It will be updated to 1.0 when the specification is released publicly.

### 9.1.1 Overview

This function returns the version of the framework at the SPCI instance where it is invoked. Each SPCI instance must support this call and return its version. For this revision of SPCI, the major version is 0 and the minor version is 9.

The version number is a 31-bit unsigned integer, with the upper 15 bits denoting the major revision, and the lower 16 bits denoting the minor revision. The following rules apply to the version numbering:

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

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.

Interface invocations could need to be forwarded from one security state to another. An optional interface could be implemented by one partition manager but not the other. An invocation on such an interface must be completed with the `NOT_SUPPORTED` error code.

This interface returns the version of the framework at a SPCI instance in the system. It is possible that versions of the framework at different SPCI instances differ. These versions must be supported in accordance to the major and minor version number compatibility rules described above.

### 9.2 SPCI_FEATURES

**Description**

- Used to query whether an optional SPCI interface is implemented at a SPCI instance
- Valid SPCI instances and conduits are listed in Table 49
- Syntax of this function is described in Table 50
- Returns SPCI_SUCCESS without any further parameters upon successful completion
- Encoding of error code in the SPCI_ERROR function is described in Table 51

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Physical SPCI instance</td>
<td>SMC, ERET</td>
</tr>
<tr>
<td>2</td>
<td>Virtual SPCI instance</td>
<td>SMC, HVC, SVC</td>
</tr>
</tbody>
</table>
### Table 50: SPCI_FEATURES function syntax

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Function ID</td>
<td>w0</td>
<td>• 0x84000064</td>
</tr>
<tr>
<td>uint32 SPCI function ID</td>
<td>w1</td>
<td>• Function ID of a SPCI function</td>
</tr>
<tr>
<td>Other Parameter registers</td>
<td>w2-w7</td>
<td>• Reserved (MBZ)</td>
</tr>
<tr>
<td></td>
<td>x2-x7</td>
<td></td>
</tr>
</tbody>
</table>

### Table 51: SPCI_ERROR encoding

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32 Error code</td>
<td>w2</td>
<td>• NOT_SUPPORTED: SPCI function is not implemented or ID is invalid</td>
</tr>
</tbody>
</table>
### 9.3 SPCI_RX_RELEASE

**Description**

- Relinquish ownership of a RX buffer after reading a message from it (see Section 5.2.2.4).
- Valid SPCI instances and conduits are listed in Table 53.
- Syntax of this function is described in Table 54.
- Returns SPCI_SUCCESS without any further parameters upon successful completion.
- Encoding of error code in the SPCI_ERROR function is described in Table 55.

#### Table 53: SPCI_RX_RELEASE instances and conduits

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Physical SPCI instance</td>
<td>SMC</td>
</tr>
<tr>
<td>2</td>
<td>Virtual SPCI instance</td>
<td>SMC, HVC, SVC</td>
</tr>
</tbody>
</table>

#### Table 54: SPCI_RX_RELEASE function syntax

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Function ID</td>
<td>w0</td>
<td>0x84000065</td>
</tr>
<tr>
<td>Other Parameter registers</td>
<td>w1-w7</td>
<td>Reserved (MBZ)</td>
</tr>
<tr>
<td></td>
<td>x1-x7</td>
<td></td>
</tr>
</tbody>
</table>

#### Table 55: SPCI_ERROR encoding

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32 Error code</td>
<td>w2</td>
<td>DENIED: Caller did not have ownership of the RX buffer</td>
</tr>
</tbody>
</table>
9.4 SPCI_RXTX_MAP

**Description**
- Maps the RX/TX buffer pair in the callee's translation regime 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 60
  - Both Hypervisor and SPM must ensure the caller has exclusive access and ownership of the RX/TX buffer memory regions
- Valid SPCI instances and conduits are listed in Table 57
- Syntax of this function is described in Table 58
- Returns SPCI_SUCCESS without any further parameters upon successful completion
- Encoding of error code in the SPCI_ERROR function is described in Table 59

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Physical SPCI instance</td>
<td>SMC</td>
</tr>
<tr>
<td>2</td>
<td>Virtual SPCI instance</td>
<td>SMC, HVC, SVC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter History</th>
<th>Register Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function ID</td>
<td>w0/x0</td>
<td>0x84000066, 0xC4000066</td>
</tr>
<tr>
<td>TX address</td>
<td>w1/x1</td>
<td>Base address of the TX buffer if invoked by an endpoint or Hypervisor to register its buffer pair.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Address is a IPA or VA at the virtual SPCI instance.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Address is a PA at the physical SPCI instance.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MBZ if Hypervisor is forwarding this call on behalf of an endpoint.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Description of RX/TX buffer and identity of endpoint is specified in the TX buffer of the Hypervisor.</td>
</tr>
<tr>
<td>RX address</td>
<td>w2/x2</td>
<td>Base address of the RX buffer</td>
</tr>
<tr>
<td>RX/TX page count</td>
<td>w3/x3</td>
<td>Bit[31:6]: Reserved (MBZ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit[5:0]: Number of contiguous 4K pages allocated for each buffer.</td>
</tr>
<tr>
<td>Other Parameter registers</td>
<td>w4-w7, x4-x7</td>
<td>Reserved (MBZ)</td>
</tr>
</tbody>
</table>
Table 59: SPCI_ERROR encoding

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32 Error code</td>
<td>w2</td>
<td>- INVALID_PARAMETERS: One or more fields in input parameters is incorrectly encoded</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- NO_MEMORY:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Not enough memory to map the buffers in the callee's translation regime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Not enough memory in TX buffer of Hypervisor to describe caller's buffer pair to SPM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- DENIED: Buffer pair already registered for the SPCI component with specified ID</td>
</tr>
</tbody>
</table>

Table 60: Endpoint RX/TX descriptor

<table>
<thead>
<tr>
<th>Field</th>
<th>Byte length</th>
<th>Byte offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Endpoint ID</td>
<td>2</td>
<td>0</td>
<td>ID of endpoint that allocated the RX/TX buffer.</td>
</tr>
<tr>
<td>Reserved</td>
<td>2</td>
<td>2</td>
<td>MBZ</td>
</tr>
<tr>
<td>RX address range count</td>
<td>4</td>
<td>4</td>
<td>Count of address ranges specified using constituent memory descriptors for the RX buffer.</td>
</tr>
<tr>
<td>TX address range count</td>
<td>4</td>
<td>8</td>
<td>Count of address ranges specified using constituent memory descriptors for the TX buffer.</td>
</tr>
<tr>
<td>RX address range array</td>
<td>–</td>
<td>12</td>
<td>Array of address ranges allocated for the RX buffer that the callee must map in its translation regime. See Table 30 for how the address ranges are encoded.</td>
</tr>
<tr>
<td>TX address range array</td>
<td>–</td>
<td>–</td>
<td>Array of address ranges allocated for the TX buffer that the callee must map in its translation regime. See Table 30 for how the address ranges are encoded.</td>
</tr>
</tbody>
</table>
9.5 SPCI_RXTX_UNMAP

Description

- Unmaps an endpoint's or Hypervisor's RX/TX buffer pair from the callee's translation regime.
  - A SP invokes this interface to unmap its buffer pair from the SPM's translation regime
  - A VM invokes this interface to unmap its buffer pair from the Hypervisor's translation regime
  - Hypervisor or OS Kernel invoke this interface to unmap their buffer pair from the SPM's translation regime
  - Hypervisor forwards an invocation of this interface by a VM to the SPM
- Identity of VM is specified in w1
- Valid SPCI instances and conduits are listed in Table 62
- Syntax of this function is described in Table 63
- Returns SPCI_SUCCESS without any further parameters upon successful completion
- Encoding of error code in the SPCI_ERROR function is described in Table 64

Table 62: SPCI_RXTX_UNMAP instances and conduits

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Physical SPCI instance</td>
<td>SMC</td>
</tr>
<tr>
<td>2</td>
<td>Virtual SPCI instance</td>
<td>SMC, HVC, SVC</td>
</tr>
</tbody>
</table>

Table 63: SPCI_RXTX_UNMAP function syntax

<table>
<thead>
<tr>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
</tr>
<tr>
<td>uint32</td>
</tr>
<tr>
<td>Function ID</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Other Parameter registers</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Table 64: SPCI_ERROR encoding

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32 Error code</td>
<td>w2</td>
<td>INVALID_PARAMETERS: There is no buffer pair registered on behalf of the caller</td>
</tr>
</tbody>
</table>
9.6 SPCI_PARTITION_INFO_GET

Description
• Request Hypervisor and SPM to return information about partitions instantiated in the system
  – Information can be requested for all partitions in the system by specifying the Nil UUID
  – Information can be requested for a subset of partitions in the system by specifying a non-Nil UUID
  – Information returned for each partition is described in Table 66
  – Partition information is returned in the RX buffer of the caller as an array of partition information descriptors
  – Count of partition information descriptors is returned in w2
  – If the Nil UUID is specified at the non-secure virtual SPCI instance, the Hypervisor must provide information for partitions resident in both security states
  – If the Nil UUID is specified at the secure virtual SPCI instance, the SPM must provide information for partitions resident in both security states
• Valid SPCI instances and conduits are listed in Table 67
• Syntax of this function is described in Table 68
• Encoding of result parameters in the SPCI_SUCCESS function is described in Table 69
• Encoding of error code in the SPCI_ERROR function is described in Table 70

Table 66: Partition information descriptor

<table>
<thead>
<tr>
<th>Field</th>
<th>Byte length</th>
<th>Byte offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition ID</td>
<td>2</td>
<td>0</td>
<td>• 16-bit ID of the partition</td>
</tr>
<tr>
<td>Execution context count</td>
<td>2</td>
<td>2</td>
<td>• Number of execution contexts implemented by this partition (also see Section 3.7)</td>
</tr>
<tr>
<td>Partition properties</td>
<td>4</td>
<td>4</td>
<td>• Flags to determine partition properties</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>– bit[0]: Supports receipt of direct requests. Execution context count must be 1 or equal to the number of PEs in the system (also see Section 5.4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>– bit[1]: Can send direct requests.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>– bit[2]: Supports receipt of indirect messages</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>– bit[31:2]: Reserved (MBZ).</td>
</tr>
</tbody>
</table>

Table 67: SPCI_PARTITION_INFO_GET instances and conduits

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Physical</td>
<td>SMC, ERET</td>
</tr>
<tr>
<td>2</td>
<td>Non-secure virtual</td>
<td>SMC, HVC</td>
</tr>
<tr>
<td>3</td>
<td>Secure virtual</td>
<td>SMC, HVC, SVC</td>
</tr>
</tbody>
</table>
### Table 68: SPCI_PARTITION_INFO_GET function syntax

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Function ID</td>
<td>w0</td>
<td>• 0x84000068</td>
</tr>
<tr>
<td>uint128 UUID</td>
<td>w1-w4</td>
<td>• Specified as described in Section 5.3 of [5].</td>
</tr>
<tr>
<td>Other Parameter</td>
<td>w5-w7</td>
<td>• Reserved (MBZ)</td>
</tr>
<tr>
<td>registers</td>
<td>x5-x7</td>
<td></td>
</tr>
</tbody>
</table>

### Table 69: SPCI_SUCCESS encoding

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Count</td>
<td>w2</td>
<td>• Count of partition information descriptors populated in RX buffer of caller</td>
</tr>
<tr>
<td>Other Result</td>
<td>w3-w7</td>
<td>• Reserved (MBZ)</td>
</tr>
<tr>
<td>registers</td>
<td>x3-x7</td>
<td></td>
</tr>
</tbody>
</table>

### Table 70: SPCI_ERROR encoding

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32 Error code</td>
<td>w1</td>
<td>• BUSY: Caller’s RX buffer is not free</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• INVALID_PARAMETERS: Unrecognized UUID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• NO_MEMORY: Results cannot fit in caller’s RX buffer</td>
</tr>
</tbody>
</table>
9.7 SPCI_ID_GET

Description

• Returns 16-bit ID of calling SPCI component.
• Valid SPCI instances and conduits are listed in Table 72
• Syntax of this function is described in Table 73
• Encoding of result parameters in the SPCI_SUCCESS function is described in Table 74
• Encoding of error code in the SPCI_ERROR function is described in Table 75
• This is an optional interface. The partition manager must use an implementation defined method to convey the ID of an endpoint if this interface is not supported

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Physical SPCI instance</td>
<td>SMC</td>
</tr>
<tr>
<td>2</td>
<td>Virtual SPCI instance</td>
<td>SMC, HVC, SVC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Function ID</td>
<td>w0</td>
<td>0x84000069</td>
</tr>
<tr>
<td>Other Parameter registers</td>
<td>w1-w7</td>
<td>Reserved (MBZ)</td>
</tr>
<tr>
<td></td>
<td>x1-x7</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 ID</td>
<td>w2</td>
<td>• ID of the caller.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Bit[31:16]: Reserved (MBZ)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Bit[15:0]: ID</td>
</tr>
<tr>
<td>Other Result registers</td>
<td>w3-w7</td>
<td>Reserved (MBZ)</td>
</tr>
<tr>
<td></td>
<td>x3-x7</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32 Error code</td>
<td>w2</td>
<td>• NOT_SUPPORTED: This function is not implemented</td>
</tr>
</tbody>
</table>
9.8 SPCI_MSG_POLL

Description

• Poll if a message is available in the caller’s RX buffer. Execution is returned to the caller if no message is available
  – Must not be invoked when the caller is processing a direct request
• Valid SPCI instances and conduits are listed in Table 81
• Syntax of this function is described in Table 78
• Returns SPCI_SUCCESS without any further parameters upon successful completion
• Encoding of error code in the SPCI_ERROR function is described in Table 79
• This is an optional interface

Table 77: SPCI_MSG_POLL instances and conduits

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Non-secure virtual</td>
<td>SMC, HVC</td>
</tr>
<tr>
<td>2</td>
<td>Secure virtual</td>
<td>SMC, HVC, SVC</td>
</tr>
</tbody>
</table>

Table 78: SPCI_MSG_POLL function syntax

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Function ID</td>
<td>w0</td>
<td>0x8400006A</td>
</tr>
<tr>
<td>Other Parameter registers</td>
<td>w1-w7</td>
<td>Reserved (MBZ)</td>
</tr>
<tr>
<td></td>
<td>x1-x7</td>
<td></td>
</tr>
</tbody>
</table>

Table 79: SPCI_ERROR encoding

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32 Error code</td>
<td>w1</td>
<td>• RETRY: Message is not available in the caller’s RX buffer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DENIED: Interface was invoked while handling a direct request</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• NOT_SUPPORTED: This function is not implemented</td>
</tr>
</tbody>
</table>


10 CPU cycle management interfaces

10.1 SPCI_MSG_WAIT

Description

• Block until a message is available in the caller’s RX buffer or parameter registers
  – Caller’s execution context enters the idle state (also see Section 3.11)
  – Execution is returned to the primary scheduler if no message is available
  – Caller’s execution is resumed when a message or interrupt is sent to it
• Valid SPCI instances and conduits are listed in Table 81
  – ERET conduit is used only to return execution to the primary scheduler
• Syntax of this function is described in Table 82
• Successful completion of this function is indicated through the invocation of the following functions by the callee.
  Each function ID encodes enough information for the caller to retrieve the message.
  – SPCI_INTERRUPT at any SPCI instance
  – Any framework message function at the secure physical SPCI instance
  – Message transmission functions at the secure physical SPCI instance
  – SPCI_RUN at any virtual SPCI instance
  – SPCI_MSG_SEND_DIRECT_REQ at any virtual SPCI instance if the partition supports receipt of direct messages
• Encoding of error code in the SPCI_ERROR function is described in Table 83

Table 81: SPCI_MSG_WAIT instances and conduits

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Non-secure physical</td>
<td>ERET</td>
</tr>
<tr>
<td>2</td>
<td>Secure physical</td>
<td>SMC</td>
</tr>
<tr>
<td>3</td>
<td>Non-secure virtual</td>
<td>SMC, HVC, ERET</td>
</tr>
<tr>
<td>4</td>
<td>Secure virtual</td>
<td>SMC, HVC, SVC</td>
</tr>
</tbody>
</table>

Table 82: SPCI_MSG_WAIT function syntax

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Function ID</td>
<td>w0</td>
<td>0x8400006B</td>
</tr>
</tbody>
</table>
| uint32 Endpoint/vCPU IDs | w1       | Endpoint and vCPU IDs of the caller. Only valid with the ERET conduit. Other MBZ
  – Bit[31:16]: vCPU ID
  – Bit[15:0]: Endpoint ID
| Other Parameter registers | w2-w7, x2-x7 | Reserved (MBZ)                                                        |
### 10.1.1 Component responsibilities for SPCI_MSG_WAIT

This section describes the common responsibilities that the participating SPCI components must fulfill during an invocation of the `SPCI_MSG_WAIT` interface. These components are:

1. SP or VM
2. Relayers
3. Primary scheduler

This interface is used by a VM or SP during indirect message processing to block their execution and enter the *idle* state by passing control back to the primary scheduler. Execution of the endpoint is resumed when a new direct or indirect message becomes available.

The relayers are responsible for returning control to the primary scheduler in response to an invocation of this interface. Their responsibilities in this regard are influenced by the location of the primary scheduler relative to theirs. These are described in Section 10.1.1.1.

#### 10.1.1.1 Relayer responsibilities

**Invocation from VM**

1. The Hypervisor and primary scheduler are co-resident. It must use an implementation defined mechanism to hand control to the scheduler.
2. The Hypervisor and primary scheduler are not co-resident. It must forward the `SPCI_MSG_WAIT` call to the primary scheduler through the ERET conduit on the PE where the call is made. The ID of the endpoint and its caller execution context must be passed in the `w1` parameter register.

**Invocation from SP**

1. The SPM must forward the `SPCI_MSG_WAIT` call to the primary scheduler through the ERET conduit on the PE where the call is made. The ID of the endpoint and its caller execution context must be passed in the `w1` parameter register.
2. If the Hypervisor is present, then its responsibilities are the same as those described in Section 10.1.1.1.
10.2 SPCI_YIELD

Description

• Relinquish execution back to the scheduler on current physical CPU from calling VM or SP.
  – Used by an endpoint to avoid a busy wait for a shared resource e.g. an internal lock that is not currently available.
  – Allows other software components to make progress on the PE until the shared resource is not available for the caller.
  – Must not be invoked when the caller is processing a direct request.
• Valid SPCI instances and conduits are listed in Table 85
  – ERET conduit is used only to return execution to the primary scheduler
• Syntax of this function is described in Table 86
• Successful completion of this function is indicated through the invocation of the SPCI_RUN function by the callee.
• Encoding of error code in the SPCI_ERROR function is described in Table 87
• This is an optional interface

Table 85: SPCI_YIELD instances and conduits

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Non-secure physical</td>
<td>ERET</td>
</tr>
<tr>
<td>2</td>
<td>Secure physical</td>
<td>SMC</td>
</tr>
<tr>
<td>3</td>
<td>Non-secure virtual</td>
<td>SMC, HVC, ERET</td>
</tr>
<tr>
<td>4</td>
<td>Secure virtual</td>
<td>SMC, HVC, SVC</td>
</tr>
</tbody>
</table>

Table 86: SPCI_YIELD function syntax

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Function ID</td>
<td>w0</td>
<td>0x8400006C</td>
</tr>
<tr>
<td>uint32 Endpoint/vCPU IDs</td>
<td>w1</td>
<td>Endpoint and vCPU IDs of the caller. Only valid with the ERET conduit. Else MBZ.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Bit[31:16]: vCPU ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Bit[15:0]: Endpoint ID</td>
</tr>
<tr>
<td>Other Parameter registers</td>
<td>w2-w7</td>
<td>Reserved (MBZ)</td>
</tr>
<tr>
<td></td>
<td>x2-x7</td>
<td></td>
</tr>
</tbody>
</table>

Table 87: SPCI_ERROR encoding

<table>
<thead>
<tr>
<th>Return Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
10.2.1 Component responsibilities for SPCI_MSG_YIELD

This section describes the common responsibilities that the participating SPCI components must fulfill during an invocation of the SPCI_MSG_YIELD interface. These components are:

1. SP or VM
2. Relayers
3. Primary scheduler

This interface is used by a VM or SP during indirect message processing to block their execution and pass control back to the primary scheduler when an internal dependency of caller cannot be fulfilled. Execution of the caller is resumed by the primary scheduler in response to a notification that the dependency has been fulfilled. The caller remains in the busy state during an invocation of this interface.

The relayers are responsible for returning control to the primary scheduler in response to an invocation of this interface. Their responsibilities in this regard are influenced by the location of the primary scheduler relative to theirs. These are described in Section 10.1.1.1.

10.2.1.1 Relayer responsibilities

Invocation from VM

1. The Hypervisor and primary scheduler are co-resident. It must use an implementation defined mechanism to hand control to the scheduler.
2. The Hypervisor and primary scheduler are not co-resident. It must forward the "SPCI_MSG_YIELD" call to the primary scheduler through the ERET conduit on the PE where the call is made. The ID of the endpoint and its caller execution context must be passed in the w1 parameter register.

Invocation from SP

1. The SPM must forward the SPCI_MSG_YIELD call to the primary scheduler through the ERET conduit on the PE where the call is made. The ID of the endpoint and its caller execution context must be passed in the w1 parameter register.
2. If the Hypervisor is present, then its responsibilities are the same as those described in Section 10.2.1.1.
## 10.3 SPCI_RUN

### Description
- Run an endpoint’s execution context on the current PE.
- Initial invocation must be from the SPCI component that implements the primary scheduler or is allowed to send direct messages.
- An invocation must be forwarded to an endpoint by the relayer.
- Valid SPCI instances and conduits are listed in Table 89.
- Syntax of this function is described in Table 90.
- Successful completion of this function is indicated through the invocation of any SPCI function. SPCI_RUN is used to allocate CPU cycles to an endpoint for message processing. The endpoint could invoke any SPCI function while processing a message.
- Encoding of error code in the SPCI_ERROR function is described in Table 91.

### Table 89: SPCI_RUN instances and conduits

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Non-secure physical</td>
<td>SMC</td>
</tr>
<tr>
<td>2</td>
<td>Secure physical</td>
<td>ERET</td>
</tr>
<tr>
<td>3</td>
<td>Non-secure virtual</td>
<td>SMC, HVC, ERET</td>
</tr>
<tr>
<td>4</td>
<td>Secure virtual</td>
<td>ERET</td>
</tr>
</tbody>
</table>

### Table 90: SPCI_RUN function syntax

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Function ID</td>
<td>w0</td>
<td>• 0x8400006D</td>
</tr>
<tr>
<td>uint32 Target information</td>
<td>w1</td>
<td>• Information to identify target SP/VM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Bits[31:16]: ID of SP/VM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Bits[15:0]: ID of vCPU of SP/VM to run.</td>
</tr>
<tr>
<td>Other Parameter registers</td>
<td>w2-w7</td>
<td>• Reserved (MBZ)</td>
</tr>
<tr>
<td></td>
<td>x2-x7</td>
<td></td>
</tr>
</tbody>
</table>

### Table 91: SPCI_ERROR encoding

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32 Error code</td>
<td>w1</td>
<td>• INVALID_PARAMETERS: Unrecognized endpoint or vCPU ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DENIED: Caller is not permitted to invoke this interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• BUSY: vCPU is busy and caller must retry later</td>
</tr>
</tbody>
</table>
10.3.1 Component responsibilities for SPCI_RUN

This section describes the common responsibilities that the participating SPCI components must fulfill during an invocation of the SPCI_RUN function (also see Section 5.3.3) by the primary scheduler to run the receiver of a message. The location of the receiver relative to the primary scheduler is one of the following:

1. The primary scheduler resides in a separate VM from the receiver.
2. The primary scheduler resides in the same EL as the Hypervisor and the receiver is a SP.
3. The Hypervisor is not present. The primary scheduler resides in the OS kernel and the receiver is a SP.

Table 92 lists the valid combinations of the receiver and primary scheduler location.

<table>
<thead>
<tr>
<th>Config no.</th>
<th>Primary Scheduler location</th>
<th>Receiver location</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>VM</td>
<td>VM</td>
</tr>
<tr>
<td>2.</td>
<td>VM</td>
<td>SP</td>
</tr>
<tr>
<td>3.</td>
<td>Hypervisor</td>
<td>SP</td>
</tr>
<tr>
<td>4.</td>
<td>OS Kernel</td>
<td>SP</td>
</tr>
</tbody>
</table>

The use of the SPCI_RUN interface with the configurations listed in Table 92 is described below.

1. In configs 3 & 4, the SPCI_RUN invocation from the Hypervisor or OS kernel must be intercepted by the SPM. It must assume the responsibility of running the receiver.
2. In configs 1 & 2, the SPCI_RUN invocation from the primary scheduler must be intercepted by the Hypervisor. It must assume the responsibility of running the receiver.

   In config 1, the Hypervisor must program a return to the VM as described in the list entry 2 in Section 5.3.3.

   In config 2, the Hypervisor must invoke the SPCI_RUN interface to request the SPM to run the SP.

In both 1 & 2 above, execution of SPCI_RUN will result in the execution of the SPM. The SPM must run the SP in the same manner as the Hypervisor runs a VM as described in the list entry 2 in Section 5.3.3.

The SPM and Hypervisor must return DENIED if an endpoint invokes this interface when it only supports indirect messaging and does not implement the primary scheduler.

The SPM and Hypervisor must return BUSY if an endpoint invokes this interface when the specified vCPU of the endpoint is busy processing a request on another PE.
11 Messaging interfaces

11.1 SPCI_MSG_SEND

Overview

- Send a partition message to an endpoint 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
  - Message is copied by SPM from the TX buffer of sender S-Endpoint to the RX buffer of receiver S-endpoint
  - Message is copied by SPM from the TX buffer of sender S-Endpoint to the RX buffer of receiver NS-endpoint
  - Message is copied by SPM from the TX buffer of sender NS-Endpoint to the RX buffer of receiver S-endpoint
  - The scheduler is informed about the pending message in the receiver’s RX buffer
  - Message will be read when the receiver endpoint is scheduled to run
  - See Section 11.1.2 for caller and callee roles and responsibilities
  - Must not be invoked when the caller is processing a direct request.

- Valid SPCI instances and conduits are listed in Table 94
  - 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 95
  - Successful completion of this function call is indicated as described below
    - w0 contains \texttt{SPCI\_SUCCESS} function ID
    - w1/w1-w7/x7 are reserved and MBZ
      - Successful completion of this function does not imply that the message has been read by the receiver endpoint
  - Encoding of error code in the SPCI\_ERROR function is described in Table 96
    - See Section 11.1.1 for behavior when BUSY is returned and caller must be notified about availability of TX buffer

- This is an optional interface. It must be implemented if the partition manager does not support direct messaging.

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Physical</td>
<td>SMC, ERET</td>
</tr>
<tr>
<td>2</td>
<td>Non-secure virtual</td>
<td>SMC, HVC, ERET</td>
</tr>
<tr>
<td>3</td>
<td>Secure virtual</td>
<td>SMC, HVC, SVC</td>
</tr>
</tbody>
</table>

Table 94: SPCI_MSG_SEND instances and conduits

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function ID</td>
<td>w0</td>
<td>\texttt{0x8400006E}</td>
</tr>
<tr>
<td>Sender/Receiver IDs</td>
<td>w1</td>
<td>• Sender and receiver endpoint IDs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Bit[31:16]: Sender endpoint ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Bit[15:0]: Receiver endpoint ID</td>
</tr>
<tr>
<td>Parameter</td>
<td>Register</td>
<td></td>
</tr>
<tr>
<td>---------------------------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>uint32/uint64</td>
<td>w2/x2</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message size</td>
<td>w3</td>
<td></td>
</tr>
<tr>
<td>Attributes</td>
<td>w4</td>
<td></td>
</tr>
<tr>
<td>Sender vCPU ID</td>
<td>w5</td>
<td></td>
</tr>
<tr>
<td>Other Parameter registers</td>
<td>w6-w7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>x6-x7</td>
<td></td>
</tr>
</tbody>
</table>

- **uint32/uint64**: Reserved for future use (MBZ).
- **Message size**: Length of message payload in the RX buffer.
- **Attributes**: Message attributes
  - 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)
- **Sender vCPU ID**: 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]**: ID of SP/VM.
  - **Bits[15:0]**: ID of vCPU of SP/VM to deliver results to.
- **Other Parameter registers**: Reserved (MBZ)

### Table 96: SPCI_ERROR encoding

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32 Error code</td>
<td>w1</td>
<td>• INVALID_PARAMETERS: A field in input parameters is incorrectly encoded</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• BUSY: Receiver's RX buffer is not free</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DENIED: Interface was invoked while handling a direct request</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• NOT_SUPPORTED: This function is not implemented</td>
</tr>
</tbody>
</table>

### 11.1.1 Target availability notification

When this interface is invoked, it is possible that the callee determines that the RX buffer of the receiver endpoint cannot be written to. This can happen if either another instance of a producer is writing to the RX buffer or the receiver endpoint is reading from it as a consumer (see Section 5.2.2.4). The callee must complete the interface invocation with a **BUSY** error code in this case.

An endpoint running in EL1 in either security state can request to be notified when the RX buffer becomes available again by setting **bit[0] = 1** in the **Attributes** parameter. In this case, the relayer must:

1. Determine when the RX buffer is available as per the ownership rules described in Section 5.2.2.4.
2. Notify each caller about the RX buffer availability.

The relayer must describe the interrupt to indicate availability of the receiver endpoint’s RX buffer to each endpoint respectively through an implementation defined mechanism. This could be done through a platform discovery mechanism like ACPI or Device tree.
A consumer i.e. OS kernel, VM or SP must indicate the availability of its RX buffer by invoking the SPCI_RX_RELEASE interface (see Section 5.2.2.4).

11.1.2 Component responsibilities for SPCI_MSG_SEND

This section describes the common responsibilities that the participating SPCI components must fulfill during transmission of partition messages between endpoints through the SPCI_MSG_SEND interface. This interface is used in the scenarios listed in Section 5.1.1.

11.1.2.1 Sender responsibilities

Send from NS-Endpoint to S-Endpoint

1. Must acquire ownership of empty TX buffer (see Section 5.2.2.4).
2. Must write partition message payload to TX buffer.
3. Must specify length of partition message payload.
4. Must specify blocking behaviour in Attributes parameter.
5. Must specify sender and receiver endpoint IDs
6. Must implement support for handling all error status codes that can be returned upon completion of these interfaces.

Send from VM to VM

1. Same as sender responsibilities while sending message from NS-Endpoint to S-Endpoint as listed in Section 11.1.2.1.
2. See Section 11.1.2.2 for relayer responsibilities in this message transmission.

Send from SP to SP

1. Same as sender responsibilities while sending message from NS-Endpoint to S-Endpoint as listed in Section 11.1.2.1.
2. See Section 11.1.2.3 for relayer responsibilities in this message transmission.

Send from S-Endpoint to NS-Endpoint

1. Same as sender responsibilities while sending message from NS-Endpoint to S-Endpoint as listed in Section 11.1.2.1.
2. See Section 11.1.2.3 for relayer responsibilities in this message transmission.

11.1.2.2 Hypervisor responsibilities

Relay from VM to VM

1. Must validate sender and receiver endpoint IDs and return INVALID PARAMETER if either is invalid.
2. Must check that reserved bits are 0 in Attributes 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 lock TX buffer of Sender from concurrent accesses before copying the message.
5. Must determine availability of RX buffer of Receiver. 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.
6. Must protect RX buffer of Receiver from concurrent accesses.
7. Must copy message from Sender TX buffer to Receiver RX buffer.
8. Must unlock TX buffer of Sender after copying the message.
9. Must unlock RX buffer of Receiver after copying the message.
10. Must inform primary scheduler that Receiver has a pending message as described in Section 11.1.3.
11. Must return SUCCESS to Sender if message is successfully transmitted.

**Relay from VM to S-Endpoint**

1. Same as 1-4 in Section 11.1.2.2.
2. Invoke SPCI_MSG_SEND at physical SPCI instance with the same parameters as specified by the Sender. See Section 11.1.2.3 for SPM's responsibilities as the relayer.
3. Same as 8, 10 & 11 in Section 11.1.2.2.

**Relay from S-Endpoint to VM**

1. Same as 1-3 in Section 11.1.2.2. Invoke SPCI_ERROR with INVALID_PARAMETER as error status if any check fails.
2. Must inform primary scheduler that Receiver has a pending message as described in Section 11.1.3.

**11.1.2.3 SPM responsibilities**

**Relay from SP to SP**

1. Same as 1-9 in Section 11.1.2.2.
2. Must inform primary scheduler that Receiver has a pending message as described in Section 11.1.3.
3. Same as 11 in Section 11.1.2.2.

**Relay from S-Endpoint to NS-Endpoint**

1. Same as 1-9 in Section 11.1.2.2.
2. Same as 2 in Section 11.1.2.3.
3. Same as 11 in Section 11.1.2.2.

**Relay from NS-Endpoint to S-Endpoint**

1. Same as 1-9 & 11 in Section 11.1.2.2.

**11.1.2.4 Receiver responsibilities**

All receivers have the same responsibilities irrespective of the origin of the message and the role of the relayers in transmitting the message. These are listed below.

1. Copy message from RX buffer.
2. Transfer ownership of the RX buffer by invoking the SPCI_RX_RELEASE interface.

**11.1.3 Mechanism for scheduler notification**

This section describes how the primary scheduler must be notified depending upon its location relative to the message sender.

1. A NS-Endpoint is the sender. The primary scheduler and sender are co-resident. The sender must use an implementation defined mechanism to notify the scheduler.
2. A VM is the sender. The primary scheduler and Hypervisor are co-resident. The Hypervisor must use an implementation defined mechanism to notify the scheduler in response to the SPCI_MSG_SEND call.
3. A VM is the sender. The primary scheduler is resident in another VM.
   1. The Hypervisor must forward the SPCI_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 `SPCI_MSG_SEND` call with either a `SPCI_SUCCESS` or `SPCI_ERROR` invocation through the SMC conduit.

4. The sender is a SP. The primary scheduler is resident in the OS kernel or the Hypervisor.
   1. The SPM must forward the `SPCI_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 `SPCI_MSG_SEND` call with either a `SPCI_SUCCESS` or `SPCI_ERROR` invocation through the SMC conduit.

5. The sender is a SP. The primary scheduler is resident in a VM. Both SPM and Hypervisor participate as relayers in this case.
   1. SPM must forward `SPCI_MSG_SEND` to the Hypervisor using the `ERET` conduit on the PE where the call is made.
   2. Hypervisor must forward `SPCI_MSG_SEND` to the primary scheduler using the `ERET` conduit on the PE where the call is made.
   3. Primary scheduler must respond to the forwarded `SPCI_MSG_SEND` call with either a `SPCI_SUCCESS` or `SPCI_ERROR` invocation through the SMC conduit.
   4. Hypervisor must forward `SPCI_SUCCESS` or `SPCI_ERROR` from the primary scheduler to the SPM through the SMC conduit.
   5. SPM must forward `SPCI_SUCCESS` or `SPCI_ERROR` from the Hypervisor to the SP through the `ERET` conduit.
11.2 SPCI_MSG_SEND_DIRECT_REQ

Description

- Send a partition message in parameter registers as a request to a target endpoint, run the endpoint and block until a response is available
- Valid SPCI instances and conduits are listed in Table 98
- Syntax of this function is described in Table 99
- Successful completion of this function is indicated through an invocation of the following interfaces by the callee:
  - `SPCI_MSG_SEND_DIRECT_RESP` to provide a response to the direct request
  - `SPCI_INTERRUPT` to indicate that the direct request was interrupted and must be resumed through the `SPCI_RUN` interface
- Encoding of error code in the SPCI_ERROR function is described in Table 100
- This is an optional interface. The `SPCI_MSG_SEND_DIRECT_REQ` interface must be implemented as well if this interface is implemented

Table 98: SPCI_MSG_SEND_DIRECT_REQ instances and conduits

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Physical</td>
<td>SMC, ERET</td>
</tr>
<tr>
<td>2</td>
<td>Non-secure virtual</td>
<td>SMC, HVC, ERET</td>
</tr>
<tr>
<td>3</td>
<td>Secure virtual</td>
<td>SMC, HVC, SVC, ERET</td>
</tr>
</tbody>
</table>

Table 99: SPCI_MSG_SEND_DIRECT_REQ function syntax

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>uint32 Function ID</td>
</tr>
<tr>
<td>Register w0</td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td>• 0x8400006F</td>
</tr>
<tr>
<td></td>
<td>• 0xC400006F</td>
</tr>
<tr>
<td>uint32 Source/Destination IDs w1</td>
<td>Source and destination endpoint IDs</td>
</tr>
<tr>
<td></td>
<td>• Bit[31:16]: Source endpoint ID</td>
</tr>
<tr>
<td></td>
<td>• Bit[15:0]: Destination endpoint ID</td>
</tr>
<tr>
<td>uint32/uint64 Reserved w2/x2</td>
<td>Reserved for future use (MBZ).</td>
</tr>
<tr>
<td>Other Parameter registers w3-w7</td>
<td>Implementation defined values</td>
</tr>
</tbody>
</table>
Table 100: SPCI_ERROR encoding

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
</table>
| int32 Error code | w1       | • INVALID_PARAMETERS: Unrecognized endpoint or vCPU ID or non-zero reserved register  
|                |          | • DENIED: Message target cannot receive a direct message             |
|                |          | • NOT_SUPPORTED: This function is not implemented                     |
|                |          | • BUSY: Message target is busy                                       |

11.2.1 Component responsibilities for SPCI_MSG_SEND_DIRECT_REQ

This section describes the common responsibilities that the participating SPCI components must fulfill during transmission of partition messages between endpoints through the SPCI_MSG_SEND_DIRECT_REQ interface. This interface is used in the scenarios listed in Table 18.

11.2.1.1 Sender responsibilities

Send from NS-Endpoint to S-Endpoint

1. Must write partition message payload to parameter registers.
2. Must specify sender and receiver endpoint IDs.
3. Must implement support for handling all error status codes that can be returned upon completion of this interface.
4. See Section 11.2.1.2 & Section 11.2.1.3 for relayer responsibilities in this message transmission.

Send from VM to VM

1. Same as sender responsibilities while sending message from NS-Endpoint to S-Endpoint as listed in Section 11.2.1.1.
2. See Section 11.2.1.2 for relayer responsibilities in this message transmission.

Send from SP to SP

1. Same as sender responsibilities while sending message from NS-Endpoint to S-Endpoint as listed in Section 11.2.1.1.
2. See Section 11.2.1.3 for relayer responsibilities in this message transmission.

Send from S-Endpoint to NS-Endpoint

1. Same as sender responsibilities while sending message from NS-Endpoint to S-Endpoint as listed in Section 11.2.1.1.
2. See Section 11.2.1.3 for relayer responsibilities in this message transmission.

11.2.1.2 Hypervisor responsibilities
**Relay from VM to VM**

1. Must validate that the sender is allowed to send direct messages. Invoke `SPCI_ERROR` with `NOT_SUPPORTED` as status if this is not the case.
2. Must validate sender and receiver endpoint IDs. Invoke `SPCI_ERROR` with `INVALID PARAMETER` as status if either is invalid.
3. Must check that reserved parameter registers are 0. Invoke `SPCI_ERROR` with `INVALID PARAMETER` as status if either is invalid.
4. Must ensure that target endpoint supports receipt of direct messages. Invoke `SPCI_ERROR` with `DENIED` as status if this is not the case.
5. Must determine availability of an idle target endpoint execution context on this PE. Invoke `SPCI_ERROR` with `BUSY` as status if not available.
6. Must ensure sender's invocation of this interface is completed only in response to an invocation of the `SPCI_MSG_SEND_DIRECT_RESP` interface.
7. Must copy parameter registers from `Sender` execution context to `Receiver` execution context.
8. Must complete the invocation of the interface the receiver had used to enter the idle state with an invocation of `SPCI_MSG_SEND_DIRECT_REQ` through the `ERET` conduit.

**Relay from VM to S-Endpoint**

1. Same as 1-3 in Section 11.2.1.2.
2. Invoke `SPCI_MSG_SEND_DIRECT_REQ` at physical SPCI instance through the SMC conduit with the same parameters as specified by the `Sender`. See Section 11.2.1.3 for SPM's responsibilities as the relayer.

**Relay from S-Endpoint to VM**

1. Same as 1-8 in Section 11.2.1.2.

11.2.1.3 SPM responsibilities

**Relay from SP to SP**

1. Same as 1-8 in Section 11.2.1.2.

**Relay from S-Endpoint to NS-Endpoint**

1. Same as 1-3 in Section 11.2.1.2.
2. Invoke `SPCI_MSG_SEND_DIRECT_REQ` at physical SPCI instance through the `ERET` conduit with the same parameters as specified by the `Sender`. See Section 11.2.1.2 for responsibilities as the relayer.

**Relay from NS-Endpoint to S-Endpoint**

1. Same as 1-8 in Section 11.2.1.2.

11.2.1.4 Receiver responsibilities

All receivers have the same responsibilities irrespective of the origin of the message and the role of the relayers in transmitting the message. These are listed below.

1. Copy message from parameter registers and process it.
2. Use the `SPCI_MSG_SEND_DIRECT_RESP` interface to return the results of message processing.
11.3 SPCI_MSG_SEND_DIRECT_RESP

Description

- Send a partition message in parameter registers as a response to a target endpoint, run the endpoint and block until a new message is available
- Valid SPCI instances and conduits are listed in Table 102
- Syntax of this function is described in Table 103
- Successful completion of this function is indicated in the same manner as that of the SPCI_MSG_WAIT function (also see Section 10.1)
- Encoding of error code in the SPCI_ERROR function is described in Table 104
- This is an optional interface. It must be implemented only if the SPCI_MSG_SEND_DIRECT_REQ interface is implemented as well

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Physical</td>
<td>SMC, ERET</td>
</tr>
<tr>
<td>2</td>
<td>Non-secure virtual</td>
<td>SMC, HVC, ERET</td>
</tr>
<tr>
<td>3</td>
<td>Secure virtual</td>
<td>SMC, HVC, SVC, ERET</td>
</tr>
</tbody>
</table>

### Table 103: SPCI_MSG_SEND_DIRECT_RESP function syntax

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Function ID</td>
<td>w0</td>
<td>• 0x84000070</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0xC4000070</td>
</tr>
<tr>
<td>uint32 Source/Destination IDs</td>
<td>w1</td>
<td>• Source and destination endpoint IDs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Bit[31:16]: Source endpoint ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Bit[15:0]: Destination endpoint ID</td>
</tr>
<tr>
<td>uint32/uint64 Reserved</td>
<td>w2/x2</td>
<td>• Reserved for future use (MBZ).</td>
</tr>
<tr>
<td>Other Parameter registers</td>
<td>w3-w7</td>
<td>• Implementation defined values</td>
</tr>
</tbody>
</table>
Table 104: SPCI_ERROR encoding

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
</table>
| int32 Error code | w1 | • INVALID_PARAMETERS: Unrecognized endpoint or vCPU ID or non-zero reserved register  
• DENIED: Message target cannot receive a direct message  
• NOT_SUPPORTED: This function is not implemented  
• BUSY: Message target is busy |

11.3.1 Component responsibilities for SPCI_MSG_SEND_DIRECT_RESP

This section describes the common responsibilities that the participating SPCI components must fulfill during transmission of partition messages between endpoints through the SPCI_MSG_SEND_DIRECT_RESP interface. This interface is used in the scenarios listed in Table 18.

11.3.1.1 Sender responsibilities

Send from NS-Endpoint to S-Endpoint
1. Must write partition message payload to parameter registers.
2. Must specify sender and receiver endpoint IDs.
3. Must implement support for handling all error status codes that can be returned upon completion of this interface.
4. See Section 11.3.1.2 & Section 11.3.1.3 for relayer responsibilities in this message transmission.

Send from VM to VM
1. Same as sender responsibilities while sending message from NS-Endpoint to S-Endpoint as listed in Section 11.3.1.1.
2. See Section 11.3.1.2 for relayer responsibilities in this message transmission.

Send from SP to SP
1. Same as sender responsibilities while sending message from NS-Endpoint to S-Endpoint as listed in Section 11.3.1.1.
2. See Section 11.3.1.3 for relayer responsibilities in this message transmission.

Send from S-Endpoint to NS-Endpoint
1. Same as sender responsibilities while sending message from NS-Endpoint to S-Endpoint as listed in Section 11.3.1.1.
2. See Section 11.3.1.3 for relayer responsibilities in this message transmission.

11.3.1.2 Hypervisor responsibilities
Relay from VM to VM

1. Must validate that the sender is allowed to send direct messages. Invoke SPCI_ERROR with NOT_SUPPORTED as status if this is not the case.
2. Must validate sender and receiver endpoint IDs. Invoke SPCI_ERROR with INVALID PARAMETER as status if either is invalid.
3. Must check that reserved parameter registers are 0. Invoke SPCI_ERROR with INVALID PARAMETER as status if either is invalid.
4. Must ensure that target endpoint supports receipt of direct messages. Invoke SPCI_ERROR with DENIED as status if this is not the case.
5. Must determine availability of an idle target endpoint execution context on this PE. Invoke SPCI_ERROR with BUSY as status if not available.
6. Must ensure sender’s invocation of this interface is completed only in response to an invocation of the SPCI_MSG_SEND DIRECT_RESP interface.
7. Must copy parameter registers from Sender execution context to Receiver execution context.
8. Must complete the invocation of the SPCI_MSG_SEND DIRECT_REQ interface that the receiver had used to send the request to which the response is being provided.

Relay from VM to S-Endpoint

1. Same as 1-3 in Section 11.3.1.2.
2. Invoke SPCI_MSG_SEND DIRECT_RESP at physical SPCI instance through the SMC conduit with the same parameters as specified by the Sender. See Section 11.3.1.3 for SPM's responsibilities as the relayer.

Relay from S-Endpoint to VM

1. Same as 1-8 in Section 11.3.1.2.

11.3.1.3 SPM responsibilities

Relay from SP to SP

1. Same as 1-7 in Section 11.3.1.2.

Relay from S-Endpoint to NS-Endpoint

1. Same as 1-3 in Section 11.3.1.2.
2. Invoke SPCI_MSG_SEND DIRECT_RESP at physical SPCI instance through the ERET conduit with the same parameters as specified by the Sender. See Section 11.3.1.2 for responsibilities as the relayer.

Relay from NS-Endpoint to S-Endpoint

1. Same as 1-8 in Section 11.3.1.2.

11.3.1.4 Receiver responsibilities

All receivers have the same responsibilities irrespective of the origin of the message and the role of the relayers in transmitting the message. These are listed below.
1. Copy response from parameter registers and process it.
12 Memory management interfaces

12.1 SPCI_MEM_DONATE

Description

• Starts transfer of ownership of a memory region from a sender endpoint to a receiver endpoint
• Memory region and receiver are described in a memory region descriptor (see Table 32)
• Descriptor is populated in RX or TX buffer or a separate memory region as described in Section 6.10
• Valid SPCI instances and conduits are listed in Table 106
• Syntax of this function is described in Table 107
• Encoding of result parameters in the SPCI_SUCCESS function is described in Table 108
• Encoding of error code in the SPCI_ERROR function is described in Table 109

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Physical SPCI instance</td>
<td>SMC, ERET</td>
</tr>
<tr>
<td>2</td>
<td>Virtual SPCI instance</td>
<td>SMC, HVC, SVC</td>
</tr>
</tbody>
</table>

Table 107: SPCI_MEM_DONATE function syntax

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Function ID</td>
<td>w0</td>
<td>• 0x84000071</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0xC4000071</td>
</tr>
<tr>
<td>uint32/uint64 Address</td>
<td>w1/x1</td>
<td>• Base address of a contiguous memory region distinct from the RX/TX buffers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Address is a IPA or VA at the virtual SPCI instance.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Address is a PA at the physical SPCI instance.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• MBZ if the RX or TX buffer is used.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• MBZ after the first invocation of this interface in which Fragment count was &gt; 0. See Section 6.10.</td>
</tr>
<tr>
<td>uint32 Page count</td>
<td>w2</td>
<td>• Number of pages allocated to the memory region. Page size is the same as that used for the RX/TX buffers. MBZ if the RX or TX buffer is used.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• MBZ after the first invocation of this interface in which Fragment count was &gt; 0. See Section 6.10.</td>
</tr>
<tr>
<td>uint32 Fragment count</td>
<td>w3</td>
<td>• Number of fragments of the data structure specified in Table 32 after the current one. See Section 6.10.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• MBZ if the current fragment is the last one.</td>
</tr>
<tr>
<td>uint32 Length</td>
<td>w4</td>
<td>• Total length of the data structure specified in Table 32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• MBZ after the first invocation of this interface in which Fragment count was &gt; 0.</td>
</tr>
</tbody>
</table>
### Parameter Register Value

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
</table>
| uint32 Handle | w5       | • Unique handle allocated by the caller to associate the fragments with the data being transmitted through this interface.  
• MBZ if in first invocation of this interface Fragment count is 0. |
| Other Parameter registers | w6-w7 x6-x7 | • Reserved (MBZ)                                                                                   |

### Other Parameter registers

- **w6-w7 x6-x7**
  - Reserved (MBZ)

### Table 108: SPCI_SUCCESS encoding

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
</table>
| uint32 Handle | w2       | • Globally unique handle to identify the memory region  
  - bit[31]: Handle allocator.  
  • b'0: Allocated by SPM.  
  • b'1: Allocated by Hypervisor.  
  - bit[30:0]: Implementation defined. |
| Other Result registers | w3-w7 x3-x7 | Reserved (MBZ)                                                                                   |

### Table 109: SPCI_ERROR encoding

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
</table>
| int32 Error code | w2 | • INVALID_PARAMETERS: Invalid description of memory region, attributes or receiver endpoint  
• DENIED: Memory region is not in a state to be donated (see Table 26)  
• NO_MEMORY: Not enough memory to store memory region description until receiver retrieves it  
• RETRY: Caller must retry transmission of this fragment |

### 12.1.1 Component responsibilities for SPCI_MEM_DONATE

This interface is used to initiate a transaction to donate a memory region to a single receiver endpoint (also see Section 6.5.2). Only the owner and relayer participate in this stage of the transaction. Responsibilities of the:

- Owner are listed in Section 12.1.1.1.
- Relayer are listed in Section 12.1.1.2.

#### 12.1.1.1 Owner responsibilities

1. Must ensure it is either the owner of the memory region or the proxy endpoint acting on behalf of a stream endpoint.
2. Must ensure the memory region is in an access state suitable for donation (see Table 26).
3. Must ensure the memory region fulfills the applicable rules stated in Section 6.3.1.
4. Must describe memory region in a memory region descriptor specified in Table 32 with the following constraint:
5. Must implement support for handling all error status codes that can be returned upon completion of this interface.

6. If the invocation of this interface completes successfully, then must send a request to the receiver with the following information:
   1. The type of memory management transaction i.e. donate memory
   2. Size of the memory region
   3. Attributes of the memory region as specified in the memory region attributes descriptor
   4. Globally unique handle returned by the relayer
   5. The implementation defined tag specified to the relayer

   If the receiver specified in the memory region descriptor is a SEPID, then the request must be sent to:
   • Either the proxy endpoint for the SEPID (see Section 6.2) through a partition message.
   • Or the autonomous device using the SEPID through an implementation defined mechanism.

7. If the receiver rejects the request in step 6, the sender should use the SPCI_MEM_RECLAIM interface with the handle returned by the relayer to reclaim ownership of the memory region. It must treat the memory region as being inaccessible until the SPCI_MEM_RECLAIM invocation completes.

### 12.1.1.2 Relayer responsibilities

1. Must validate that the sender is the owner of the memory region or the proxy endpoint acting on behalf of a stream endpoint. Must return DENIED in case of an error.

2. Must ensure that a request by a SP to donate secure memory to a NS-Endpoint is rejected by returning the DENIED error code.

3. Must validate that the memory region is in an ownership and access state suitable for donation (see Table 26) and return DENIED in case of an error.

4. Must ensure only a single receiver endpoint is specified by the sender and return INVALID_PARAMETER in case of an error.

5. Must validate the contents of the memory region descriptor and return INVALID_PARAMETER in case of an error e.g.
   1. Ensure that the address ranges specified in the constituent memory region descriptors do not overlap each other.
   2. The memory attributes specified in the memory region attributes descriptor are valid

6. If the receiver is a PE endpoint or a stream endpoint with a proxy endpoint, it must:
   1. Save the memory region description so that it can be validated when retrieved through invocations of the SPCI_MEM_RETRIEVE_REQ & SPCI_MEM_RETRIEVE_RESP interfaces. It must return NO_MEMORY if there is not enough memory to complete this operation.
   2. Update its internal state such that any attempts to donate, lend or share this memory region by the sender are denied.

7. Unmap the memory region from the translation regime of the owner as described below.
   1. If the sender is a PE endpoint running in EL1 or S-EL1, the memory region must be unmapped from its stage 2 translation regime (if present).
   2. If the sender is a PE endpoint running in S-EL0, the memory region must be unmapped from its stage 1 translation regime
   3. If the sender is a proxy endpoint for a stream endpoint then the memory region must be unmapped from the stage 2 translation regime corresponding to the SEPID.
The relayer must update internal state of the owner associated with the memory region to Owner-NA. This makes it possible for the owner to reclaim access to the memory region in the event of an error through an invocation of the SPCI_MEM_RECLAIM interface.

8. If the receiver is a stream endpoint without a proxy endpoint, the relayer must:
   1. Update its internal state associated with the memory region so that the stream endpoint is assigned as the owner of the memory region.
   2. Allocate an IPA range and map the memory region in the stage 2 translation regime corresponding to the SEPID.
      • The mapping must be done with the memory region and permission attributes specified in the memory region attribute descriptor.
   3. Describe the memory region to the device using the SEPID through an implementation defined mechanism.

9. If the receiver is a SP and the Hypervisor is present, it must perform only steps 1-8 and forward the memory region descriptor to the SPM through another invocation of this interface. The memory region must be described as follows:
   1. The memory region attributes must the same as those provided by the sender
   2. The memory region must be described as composed of physically addressed constituent 4K pages by using constituent memory region descriptors. This must be done by performing the VA or IPA to PA translation of the memory region described by the sender.

   The SPM must do the same if the receiver is a VM.

10. If the receiver is a SP or OS kernel then the SPM must allocate a globally unique handle and return it to the sender. If the receiver is a VM then the Hypervisor must do the same. The sender and receiver must use this handle to reference the memory region in subsequent memory management transactions involving the relayer.

11. If the call executes successfully, must ensure that:
   1. Another donate, lend, share or relinquish transaction on a subset of the memory region is rejected until the receiver calls the SPCI_MEM_RETRIEVE_REQ interface.
   2. The sender can abort the transaction by calling the SPCI_MEM_RECLAIM interface before the receiver calls the SPCI_MEM_RETRIEVE_REQ interface.
12.2 SPCI_MEM_LEND

Description
- Starts a transaction to relinquish owner’s access to a memory region and grant access to it to one or more borrowers
- Memory region and receiver are described in a memory region descriptor (see Table 32)
- Descriptor is populated in RX or TX buffer or a separate memory region as described in Section 6.10
- Valid SPCI instances and conduits are listed in Table 111
- Syntax of this function is described in Table 112
- Encoding of result parameters in the SPCI_SUCCESS function is described in Table 113
- Encoding of error code in the SPCI_ERROR function is described in Table 114

Table 111: SPCI_MEM_LEND instances and conduits

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Physical SPCI instance</td>
<td>SMC, ERET</td>
</tr>
<tr>
<td>2</td>
<td>Virtual SPCI instance</td>
<td>SMC, HVC, SVC</td>
</tr>
</tbody>
</table>

Table 112: SPCI_MEM_LEND function syntax

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Function ID</td>
<td>w0</td>
<td>• 0x84000072&lt;br&gt;• 0xC4000072</td>
</tr>
<tr>
<td>uint32/uint64 Address</td>
<td>w1/x1</td>
<td>• Base address of a contiguous memory region distinct from the RX/TX buffers.&lt;br&gt;– Address is a IPA or VA at the virtual SPCI instance.&lt;br&gt;– Address is a PA at the physical SPCI instance.&lt;br&gt;• MBZ if the RX or TX buffer is used.&lt;br&gt;• MBZ after the first invocation of this interface in which Fragment count was &gt; 0. See Section 6.10.</td>
</tr>
<tr>
<td>uint32 Page count</td>
<td>w2</td>
<td>• Number of pages allocated to the memory region. Page size is the same as that used for the RX/TX buffers. MBZ if the RX or TX buffer is used.&lt;br&gt;• MBZ after the first invocation of this interface in which Fragment count was &gt; 0. See Section 6.10.</td>
</tr>
<tr>
<td>uint32 Fragment count</td>
<td>w3</td>
<td>• Number of fragments of the data structure specified in Table 32 after the current one. See Section 6.10.&lt;br&gt;• MBZ if the current fragment is the last one.</td>
</tr>
<tr>
<td>uint32 Length</td>
<td>w4</td>
<td>• Total length of the data structure specified in Table 32&lt;br&gt;• MBZ after the first invocation of this interface in which Fragment count was &gt; 0.</td>
</tr>
<tr>
<td>uint32 Handle</td>
<td>w5</td>
<td>• Unique handle allocated by the caller to associate the fragments with the data being transmitted through this interface.&lt;br&gt;• MBZ if in first invocation of this interface Fragment count is 0.</td>
</tr>
</tbody>
</table>
**Parameter Register Value**

<table>
<thead>
<tr>
<th>Parameter registers</th>
<th>w6-w7</th>
<th>x6-x7</th>
<th>Reserved (MBZ)</th>
</tr>
</thead>
</table>

**Table 113: SPCI_SUCCESS encoding**

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Handle</td>
<td>w2</td>
<td>Globally unique handle to identify the memory region</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- bit[31]: Handle allocator.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- b'0: Allocated by SPM.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- b'1: Allocated by Hypervisor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- bit[30:0]: Implementation defined.</td>
</tr>
</tbody>
</table>

**Other Result registers**

<table>
<thead>
<tr>
<th>w3-w7</th>
<th>x3-x7</th>
<th>Reserved (MBZ)</th>
</tr>
</thead>
</table>

**Table 114: SPCI_ERROR encoding**

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32 Error code</td>
<td>w2</td>
<td>INVALID_PARAMETERS: Invalid description of memory region, attributes or receiver endpoints</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DENIED: Memory region is not in a state to be lent (see Table 27)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NO_MEMORY: Not enough memory to store memory region description until receiver retrieves it</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETRY: Caller must retry transmission of this fragment</td>
</tr>
</tbody>
</table>

### 12.2.1 Component responsibilities for SPCI_MEM_LEND

This interface is used to initiate a transaction to lend a memory region to one or more receiver endpoints (also see Section 6.5.2). Only the owner and relayer participate in this stage of the transaction. Responsibilities of the:

- **Owner are listed in Section 12.2.1.1.**
- **Relayer are listed in Section 12.2.1.2.**

#### 12.2.1.1 Owner responsibilities

1. Must ensure it is either the owner of the memory region or the proxy endpoint acting on behalf of a stream endpoint.
2. Must ensure the memory region is in an access state suitable for lending (see Table 27).
3. Must ensure the memory region fulfills the applicable rules stated in Section 6.3.1.
4. Must describe memory region in a memory region descriptor specified in Table 32.
5. Must implement support for handling all error status codes that can be returned upon completion of this interface.
6. If the invocation of this interface completes successfully, then must send a request to the receiver with the following information.
1. The type of memory management transaction i.e. lend memory
2. Size of the memory region
3. Identity of all borrowers of the memory region and attributes with which it will be mapped in the translation regime of each borrower.
4. Globally unique handle returned by the relayer
5. The implementation defined tag specified to the relayer

If the receiver specified in the memory region descriptor is a SEPID, then the request must be sent to:

- Either the proxy endpoint for the SEPID (see Section 6.2) through a partition message.
- Or the autonomous device using the SEPID through an implementation defined mechanism.

7. If the receiver rejects the request in step 6, the sender should use the SPCI_MEM_RECLAIM interface with the handle returned by the relayer to reclaim ownership of the memory region. It must treat the memory region as being inaccessible until the SPCI_MEM_RECLAIM invocation completes.

### 12.2.1.2 Relayer responsibilities

1. Must validate that the sender is the owner of the memory region or the proxy endpoint acting on behalf of a stream endpoint. Must return DENIED in case of an error.
2. Must ensure that a request by a SP to lend secure memory to a NS-Endpoint is rejected by returning the DENIED error code.
3. Must validate that the memory region is in an ownership and access state suitable for lending (see Table 27) and return DENIED in case of an error.
4. Must validate the contents of the memory region descriptor and return INVALID_PARAMETER in case of an error e.g.
   1. Ensure that the address ranges specified in the constituent memory region descriptors do not overlap each other.
   2. The memory attributes specified in the memory region attributes descriptor are valid.
5. If the receiver is a PE endpoint or a stream endpoint with a proxy endpoint, it must:
   1. Save the memory region description so that it can be validated when retrieved through invocations of the SPCI_MEM_RETRIEVE_REQ & SPCI_MEM_RETRIEVE_RESP interfaces. It must return NO_MEMORY if there is not enough memory to complete this operation.
   2. Update its internal state such that any attempts to donate, lend or share this memory region by the sender are denied.
6. Unmap the memory region from the translation regime of the owner as described below.
   1. If the sender is a PE endpoint running in EL1 or S-EL1, the memory region must be unmapped from its stage 2 translation regime (if present).
   2. If the sender is a PE endpoint running in S-EL0, the memory region must be unmapped from its stage 1 translation regime.
   3. If the sender is a proxy endpoint for a stream endpoint then the memory region must be unmapped from the stage 2 translation regime corresponding to the SEPID.

The relayer must update internal state of the owner associated with the memory region to Owner-NA. This makes it possible for the owner to reclaim access to the memory region in the event of an error through an invocation of the SPCI_MEM_RECLAIM interface.

7. If the receiver is a stream endpoint without a proxy endpoint, the relayer must:
   1. Update its internal state associated with the memory region to indicate that the stream endpoint has been granted access to the memory region.
2. Allocate an IPA range and map the memory region in the stage 2 translation regime corresponding to the SEPID.
   - The mapping must be done with the memory region and permission attributes specified in the memory region attribute descriptor.

3. Describe the memory region to the stream endpoint through an implementation defined mechanism.

8. If the receiver is a SP and the Hypervisor is present, it must perform only steps 1-7 and forward the memory region descriptor to the SPM through another invocation of this interface. The memory region must be described as follows:
   1. The memory region attributes must the same as those provided by the sender
   2. The memory region must be described as composed of physically addressed constituent 4K pages by using constituent memory region descriptors. This must be done by performing the VA or IPA to PA translation of the memory region described by the sender.

   The SPM must do the same if the receiver is a VM.

9. If the receiver is a SP or OS kernel then the SPM must allocate a globally unique handle and return it to the sender. If the receiver is a VM then the Hypervisor must do the same. The sender and receiver must use this handle to reference the memory region in subsequent memory management transactions involving the relayer.

10. If the call executes successfully, must ensure that:
   1. Another donate, lend, share or relinquish transaction on a subset of the memory region is rejected until the receiver calls the \texttt{SPCI\_MEM\_RETRIEVE\_REQ} interface.
   2. The sender can abort the transaction by calling the \texttt{SPCI\_MEM\_RECLAIM} interface before the receiver calls the \texttt{SPCI\_MEM\_RETRIEVE\_REQ} interface.
12.3 SPCI_MEM_SHARE

Description

- Starts a transaction to grant access to a memory region to one or more borrowers
- Memory region and receivers are described in a memory region descriptor (see Table 32)
- Descriptor is populated in RX or TX buffer or a separate memory region as described in Section 6.10
- Valid SPCI instances and conduits are listed in Table 116
- Syntax of this function is described in Table 117
- Encoding of result parameters in the SPCI_SUCCESS function is described in Table 118
- Encoding of error code in the SPCI_ERROR function is described in Table 119

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Physical SPCI instance</td>
<td>SMC, ERET</td>
</tr>
<tr>
<td>2</td>
<td>Virtual SPCI instance</td>
<td>SMC, HVC, SVC</td>
</tr>
</tbody>
</table>

Table 117: SPCI_MEM_SHARE function syntax

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Function ID</td>
<td>w0</td>
<td>0x84000073 / 0xC4000073</td>
</tr>
<tr>
<td>uint32/uint64 Address</td>
<td>w1/x1</td>
<td>Base address of a contiguous memory region distinct from the RX/TX buffers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Address is a IPA or VA at the virtual SPCI instance.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Address is a PA at the physical SPCI instance.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MBZ if the RX or TX buffer is used.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MBZ after the first invocation of this interface in which Fragment count was &gt; 0. See Section 6.10.</td>
</tr>
<tr>
<td>uint32 Page count</td>
<td>w2</td>
<td>Number of pages allocated to the memory region. Page size is the same as that used for the RX/TX buffers. MBZ if the RX or TX buffer is used.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MBZ after the first invocation of this interface in which Fragment count was &gt; 0. See Section 6.10.</td>
</tr>
<tr>
<td>uint32 Fragment count</td>
<td>w3</td>
<td>Number of fragments of the data structure specified in Table 32 after the current one. See Section 6.10.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MBZ if the current fragment is the last one.</td>
</tr>
<tr>
<td>uint32 Length</td>
<td>w4</td>
<td>Total length of the data structure specified in Table 32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MBZ after the first invocation of this interface in which Fragment count was &gt; 0.</td>
</tr>
<tr>
<td>uint32 Handle</td>
<td>w5</td>
<td>Unique handle allocated by the caller to associate the fragments with the data being transmitted through this interface.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MBZ if in first invocation of this interface Fragment count is 0.</td>
</tr>
</tbody>
</table>
### Parameter Register Value

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other Parameter registers</td>
<td>w6-w7</td>
<td>• Reserved (MBZ)</td>
</tr>
<tr>
<td></td>
<td>x6-x7</td>
<td></td>
</tr>
</tbody>
</table>

#### Table 118: SPCI_SUCCESS encoding

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Handle</td>
<td>w2</td>
<td>• Globally unique handle to identify the memory region</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• bit[31]: Handle allocator.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• b'0: Allocated by SPM.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• b'1: Allocated by Hypervisor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• bit[30:0]: Implementation defined.</td>
</tr>
</tbody>
</table>

| Other Result registers | w3-w7 | x3-x7 | Reserved (MBZ) |

#### Table 119: SPCI_ERROR encoding

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>int32 Error code</td>
<td>w2</td>
<td>• INVALID_PARAMETERS: Invalid description of memory region, attributes or receiver endpoints</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DENIED: Memory region is not in a state to be shared (see Table 28)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• NO_MEMORY: Not enough memory to store memory region description until receiver retrieves it</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• RETRY: Caller must retry transmission of this fragment</td>
<td></td>
</tr>
</tbody>
</table>

### 12.3.1 Component responsibilities for SPCI_MEM_SHARE

This interface is used to initiate a transaction to share a memory region with one or more receiver endpoints (also see Section 6.5.2). Only the owner and relayer participate in this stage of the transaction. Responsibilities of the:

- Owner are listed in Section 12.3.1.1.
- Relayer are listed in Section 12.3.1.2.

#### 12.3.1.1 Owner responsibilities

1. Must ensure it is either the owner of the memory region or the proxy endpoint acting on behalf of a stream endpoint.
2. Must ensure the memory region is in an access state suitable for sharing (see Table 28).
3. Must ensure the memory region fulfills the applicable rules stated in Section 6.3.1.
4. Must describe memory region in a memory region descriptor specified in Table 32.
5. The owner could request the memory region to be mapped with different attributes in its stage 1 or 2 translation regime. It must specify these attributes and its endpoint ID in a memory region attributes descriptor in Table 32.
6. Must implement support for handling all error status codes that can be returned upon completion of this interface.
7. If the invocation of this interface completes successfully, then must send a request to the receiver with the following information:
   1. The type of memory management transaction i.e. share memory
   2. Size of the memory region
   3. Identity of all borrowers (including the owner) of the memory region and the attributes with which it will be mapped in the translation regime of each borrower.
   4. Globally unique handle returned by the relayer
   5. The implementation defined tag specified to the relayer

If the receiver specified in the memory region descriptor is a SEPID, then the request must be sent to:
   • Either the proxy endpoint for the SEPID (see Section 6.2) through a partition message.
   • Or the autonomous device using the SEPID through an implementation defined mechanism.

8. If the receiver rejects the request in step 6, the sender should use the `SPCI_MEM_RECLAIM` interface with the handle returned by the relayer to reclaim ownership of the memory region. It must treat the memory region as being inaccessible until the `SPCI_MEM_RECLAIM` invocation completes.

### 12.3.1.2 Relayer responsibilities

1. Must validate that the sender is the owner of the memory region or the proxy endpoint acting on behalf of a stream endpoint. Must return `DENIED` in case of an error.

2. Must ensure that a request by a SP to share secure memory to a NS-Endpoint is rejected by returning the `DENIED` error code.

3. Must validate that the memory region is in an ownership and access state suitable for sharing (see Table 28) and return `DENIED` in case of an error.

4. Must validate the contents of the memory region descriptor and return `INVALID_PARAMETER` in case of an error e.g.
   1. Ensure that the address ranges specified in the constituent memory region descriptors do not overlap each other.
   2. The memory attributes specified in the memory region attributes descriptor are valid.

5. If the receiver is a PE endpoint or a stream endpoint with a proxy endpoint, it must:
   1. Save the memory region description so that it can be validated when retrieved through invocations of the `SPCI_MEM_RETRIEVE_REQ` & `SPCI_MEM_RETRIEVE_RESP` interfaces. It must return `NO_MEMORY` if there is not enough memory to complete this operation.
   2. Update its internal state such that any attempts to donate, share or share this memory region by the sender are denied.

6. If the owner has specified different memory region attributes for its stage 1 or 2 translation regime, then the translation tables must be updated to reflect the new attributes.

7. The relayer must maintain the internal state of the owner associated with the memory region as Owner-EA. This makes it possible for the owner to reclaim access to the memory region in the event of an error through an invocation of the `SPCI_MEM_RECLAIM` interface.

8. If the receiver is a stream endpoint without a proxy endpoint, the relayer must:
   1. Update its internal state associated with the memory region to indicate that the stream endpoint has been granted access to the memory region.
   2. Allocate an IPA range and map the memory region in the stage 2 translation regime corresponding to the SEPID.
      • The mapping must be done with the memory region and permission attributes specified in the memory region attribute descriptor.
3. Describe the memory region to the stream endpoint through an implementation defined mechanism.

9. If the receiver is a SP and the Hypervisor is present, it must perform only steps 1-8 and forward the memory region descriptor to the SPM through another invocation of this interface. The memory region must be described as follows:

   1. The memory region attributes must the same as those provided by the sender
   2. The memory region must be described as composed of physically addressed constituent 4K pages by using constituent memory region descriptors. This must be done by performing the VA or IPA to PA translation of the memory region described by the sender.

   The SPM must do the same if the receiver is a VM.

10. If the receiver is a SP or OS kernel then the SPM must allocate a globally unique handle and return it to the sender. If the receiver is a VM then the Hypervisor must do the same. The sender and receiver must use this handle to reference the memory region in subsequent memory management transactions involving the relayer.

11. If the call executes successfully, must ensure that:

   1. Another donate, lend, share or relinquish transaction on a subset of the memory region is rejected until the receiver calls the SPCI_MEM_RETRIEVE_REQ interface.
   2. The sender can abort the transaction by calling the SPCI_MEM_RECLAIM interface before the receiver calls the SPCI_MEM_RETRIEVE_REQ interface.
12.4 SPCI_MEM_RETRIEVE_REQ

Description

• Requests completion of a donate, lend or share memory management transaction
  – Caller uses this interface to:
    ▪ Verify the transaction details provided by the initiator of the transaction/owner of the memory region
    ▪ Request the memory region to be mapped in its translation regime
  – Callee uses this interface to:
    ▪ Verify that the transaction details provided by the caller match those provided by the initiator of the transaction
    ▪ Map the memory region in the caller’s translation regime
• Transaction details are described in a descriptor specified in Table 124
• Descriptor is populated in RX or TX buffer or a separate memory region as described in Section 6.10
• Valid SPCI instances and conduits are listed in Table 121
• Syntax of this function is described in Table 122
• Usage of the SPCI_SUCCESS function is described below
  – SPCI_SUCCESS function is used only:
    ▪ If the descriptor specified in Table 124 is fragmented and the fragment that was transmitted is not the last fragment, an invocation of this function indicates that the corresponding fragment was transmitted successfully.
    ▪ If the memory region was mapped by the callee using the address ranges specified by the caller. This implies that the callee did not have to allocate address ranges on behalf of the caller and describe them through an invocation of the SPCI_MEM_RETRIEVE_RESP interface.
  – No parameters apart from the SPCI_SUCCESS function ID in w0 are used to indicate success.
• Encoding of error code in the SPCI_ERROR function is described in Table 123
• Successful transmission of the last fragment is indicated by an invocation of the SPCI_MEM_RETRIEVE_RESP function (see Section 12.5).

Table 121: SPCI_MEM_RETRIEVE_REQ instances and conduits

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Physical SPCI instance</td>
<td>SMC, ERET</td>
</tr>
<tr>
<td>2</td>
<td>Virtual SPCI instance</td>
<td>SMC, HVC, SVC</td>
</tr>
</tbody>
</table>

Table 122: SPCI_MEM_RETRIEVE_REQ function syntax

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Function ID</td>
<td>w0</td>
<td>• 0x840000074</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0xC40000074</td>
</tr>
<tr>
<td>uint32/uint64 Address</td>
<td>w1/x1</td>
<td>• Base address of a contiguous memory region distinct from the RX/TX buffers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Address is a IPA or VA at the virtual SPCI instance.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>– Address is a PA at the physical SPCI instance.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• MBZ if the RX or TX buffer is used.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• MBZ after the first invocation of this interface in which Fragment count was &gt; 0. See Section 6.10.</td>
</tr>
</tbody>
</table>
Secure Partition Client Interface Specification

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
</table>
| uint32 Page count | w2       | • Number of pages allocated to the memory region. Page size is the same as that used for the RX/TX buffers. MBZ if the RX or TX buffer is used.  
• MBZ after the first invocation of this interface in which Fragment count was > 0. See Section 6.10. |
| uint32 Fragment count | w3      | • Number of fragments of the data structure specified in Table 124 after the current one. See Section 6.10.  
• MBZ if the current fragment is the last one. |
| uint32 Length     | w4       | • Total length of the data structure specified in Table 124  
• MBZ after the first invocation of this interface in which Fragment count was > 0. |
| uint32 Handle     | w5       | • Unique handle allocated by the caller to associate the fragments with the data being transmitted through this interface.  
• MBZ if in first invocation of this interface Fragment count is 0. |
| Other Parameter   | w6-w7 x6-x7 | • Reserved (MBZ)                                                     |

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
</table>
| int32 Error code | w2   | • INVALID_PARAMETERS: Invalid handle  
• DENIED: Memory region is being accessed by one or more borrowers  
• NO_MEMORY: Not enough memory to map memory region in caller’s translation regime  
• RETRY: Caller must retry transmission of this fragment. |

<table>
<thead>
<tr>
<th>Table 123: SPCI_ERROR encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field</td>
</tr>
<tr>
<td>-----------------------------</td>
</tr>
<tr>
<td>Handle</td>
</tr>
</tbody>
</table>
| Transaction type            | 4           | 4           | • Function ID of ongoing transaction. It must be one of the following.  
  – SPCI_MEM_DONATE  
  – SPCI_MEM_LEND  
  – SPCI_MEM_SHARE |
| Tag                         | 4           | 8           | • Implementation defined value associated with the receiver and the memory region. |

Table 124: Descriptor to retrieve a donated, lent or shared memory region
Secure Partition Client Interface Specification

### Field

<table>
<thead>
<tr>
<th>Field</th>
<th>Byte length</th>
<th>Byte offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count of global memory region attributes descriptors</td>
<td>4</td>
<td>12</td>
<td>• Number of descriptors that specify the attributes with which the memory region will be mapped in the translation regime of each borrower other than the caller or the stream endpoints the caller is a proxy for.</td>
</tr>
<tr>
<td>Offset of global memory region attributes descriptors</td>
<td>4</td>
<td>16</td>
<td>• Offset from the base address of this descriptor to the array of global memory region attributes descriptors. See Table 31.</td>
</tr>
<tr>
<td>Retrieve properties descriptor count</td>
<td>4</td>
<td>20</td>
<td>• Count of descriptors with properties to retrieve a memory region on behalf of the caller or one or stream endpoints the caller is a proxy for.</td>
</tr>
</tbody>
</table>
| Retrieve properties descriptors                 | –           | 24          | • Array of descriptors with properties to retrieve a memory region (see Table 125). Each descriptor describes the identity of the receiver, memory attributes and address ranges with which the memory region should be mapped in the specified endpoint.  
  - Caller can specify more than one receiver only if it is a proxy endpoint for more than one stream endpoints  
  - The attributes & receiver ID must match those provided to the relayer by the initiator of this transaction.  
  - The endpoints specified in the descriptors must be one or more of the following depending upon the transaction type  
    - The PE endpoint making the SPCI_MEM_RETRIEVE call  
    - One or more stream endpoints that the PE endpoint making the SPCI_MEM_RETRIEVE call is a proxy for |

### Table 125: Descriptor with properties to retrieve a memory region

<table>
<thead>
<tr>
<th>Field</th>
<th>Byte length</th>
<th>Byte offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory region attributes descriptor</td>
<td>4</td>
<td>0</td>
<td>• Memory region attributes descriptor as specified in Table 31.</td>
</tr>
<tr>
<td>Total page count</td>
<td>4</td>
<td>4</td>
<td>• Size of the memory region described as the count of 4K pages. Must be equal to the sum of page counts specified in each constituent memory descriptor.</td>
</tr>
</tbody>
</table>

Page 127 of 143 Copyright © 2019 Arm Limited or its affiliates. All rights reserved. DEN0077A Non-confidential 1.0
<table>
<thead>
<tr>
<th>Field</th>
<th>Byte length</th>
<th>Byte offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address range count</td>
<td>4</td>
<td>8</td>
<td>• Count of address ranges specified using constituent memory descriptors.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The count can be zero if the receiver expects the relayer to allocate the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>address ranges.</td>
</tr>
<tr>
<td>Address range array</td>
<td>4</td>
<td>12</td>
<td>• Array of address ranges that the relayer must use to map the memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>region specified in this transaction in the translation regimes of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>receiver specified in the memory region attributes descriptor.</td>
</tr>
</tbody>
</table>

12.4.1 Component responsibilities for SPCI_MEM_RETRIEVE_REQ

This interface is used to initiate completion of a transaction to donate, lend or share a memory region with the caller endpoint. The transaction is completed when the relayer:

- Either invokes the `SPCI_MEM_RETRIEVE_RESP` function with a description of the address ranges used to map the memory region in the receiver’s translation regime.
- Or invokes the `SPCI_SUCCESS` function to indicate that the memory region was mapped with the address ranges provided by the receiver.

Only a receiver endpoint and relayer participate in this stage of the transaction. Responsibilities of the:

- Receiver are listed in Section 12.4.1.1.
- Relayer are listed in Section 12.4.1.2.

12.4.1.1 Receiver responsibilities

1. Must ensure it has obtained and validated the following information sent by the initiator of the transaction.
   1. The type of memory management transaction i.e. donate, lend, or share memory
      1. For a donate memory transaction, the identity of the receiver endpoint and the memory attributes with which the memory region will be mapped in its translation regime.
      2. For a lend memory transaction, list of all borrowers of the memory region and attributes with which it will be mapped in the translation regime of each borrower.
      3. For a share memory transaction, list of all borrowers (including the owner) of the memory region and attributes with which it will be mapped in the translation regime of each borrower.
   2. Size of the memory region
   3. Globally unique handle returned by the relayer
   4. The implementation defined tag specified to the relayer

   If any of the above information is invalid then must reject the transaction by sending a partition or implementation defined message to the initiator.

2. If the receiver is a PE endpoint, it:
   1. Could allocate an address range to map the memory region in its translation regime. Each address range must be specified in a constituent memory region descriptor (see Table 30).
   2. Must populate a descriptor (see Table 124) with properties to retrieve the memory region.

3. If the receiver is a proxy endpoint for one or more stream endpoints, it must perform step 2 for each stream endpoint.

4. Must use the type of memory management transaction, globally unique handle, tag and one or more descriptors from step 2 and/or 3 to populate a descriptor to retrieve the memory region (see Table 124).
5. Must implement support for handling all error status codes that can be returned upon completion of this interface.

6. If an error is returned by the relayer, must send a message to the initiator of the transaction indicating inability to receive the memory region.

### 12.4.1.2 Relayer responsibilities

1. Must validate the information provided in the descriptor to retrieve the memory region provided by the receiver to ensure it matches that provided by the initiator of the transaction. This includes the following.
   1. The global handle is valid and identical.
   2. The transaction type is valid and identical.
   3. The implementation defined tag is the identical.
   4. The caller is a PE endpoint and/or the proxy endpoint for one or more stream endpoints specified by the initiator.
   5. The memory region size and attributes are valid and identical
   6. List of borrowers provided by the receiver and the memory attributes with which the memory region must be mapped in their translation regime matches that provided by the initiator of a lend or share transaction.

   Must return `INVALID_PARAMETER` in case of an error.

2. Must return `NO_MEMORY` if there is not enough memory to complete this operation.

3. Must clear the contents of the memory region if `bit[0]` is set in the `Flags` parameter specified by the owner.

4. Must map the memory region in the translation regime of the receiver endpoint as described below.
   1. If the receiver is a PE endpoint running in EL1 or S-EL1, the memory region must be mapped in its stage 2 translation regime (if present) using the IPA ranges provided by it.
   2. If the receiver is a PE endpoint running in S-EL0, the memory region must be mapped in its stage 1 translation regime using the VA ranges provided by it.
   3. If the receiver is a proxy endpoint for one or more stream endpoints then the memory region must be mapped in the stage 2 translation regime corresponding to each SEPID. The memory region must not be mapped in the translation regime of the proxy endpoint.

5. Must mark the transaction as complete once one of the following events happen.
   1. All receivers specified by the initiator of the transaction successfully retrieve the memory region through an invocation of this interface. The ownership and access state of the memory region for each borrower must be updated as described below.
      1. `Owner-EA` if the transaction type is `SPCI_MEM_DONATE`.
      2. `!Owner-EA` or `!Owner-SA` if the transaction type is `SPCI_MEM_LEND`.
      3. `!Owner-SA` if the transaction type is `SPCI_MEM_SHARE`.
   2. The memory region is not mapped in the translation regime of any borrower/receiver endpoint and the initiator of the transaction calls the `SPCI_MEM_RECLAIM` interface to abort the transaction.

6. If the call executes successfully, must ensure that:
   1. A donate, lend or share transaction on a subset of the memory region is rejected until the owner has successfully reclaimed access to the memory region
   2. Only a receiver that has invoked the `SPCI_MEM_RETRIEVE_REQ` interface successfully can invoke the `SPCI_MEM_RELINQUISH` interface.
   3. The owner of the memory region can invoke the `SPCI_MEM_RECLAIM` interface only if it is not mapped in the translation regime of any borrower/receiver endpoint.
7. It is possible that both SPs and NS-Endpoints participate in a memory management transaction. If the Hypervisor is present then it is likely that it will track the state of memory regions in tandem with the SPM. The states maintained by both components must be kept in sync. This should be done through an implementation defined mechanism and/or by forwarding an invocation of this interface from the Hypervisor to the SPM and vice versa.

If the descriptor specified in Table 124 is transmitted in fragments, then the call must be forwarded only after the entire descriptor has been successfully transmitted.
12.5 SPCI_MEM_RETRIEVE_RESP

Description

- Completes a donate, lend or share memory management transaction
  - Caller uses this interface to specify the address ranges where the memory region has been mapped in the callee's translation regime
  - Caller uses this interface to successfully complete an invocation of the SPCI_MEM_RETRIEVE_REQ interface by the callee
  - Callee uses this interface to obtain the address ranges corresponding to the memory region referenced by the transaction
- Address ranges used to map the memory region in the translation regime of the callee or one or more stream endpoints the callee is a proxy for are encoded as specified in Table 130
- Descriptor must be populated using the same method that was used in the invocation of the SPCI_MEM_RETRIEVE_REQ interface that is being completed. Also see Section 6.10
  - If the TX buffer was used in the SPCI_MEM_RETRIEVE_REQ interface, then the RX buffer of the callee must be used
  - If a separate memory region was used in the SPCI_MEM_RETRIEVE_REQ interface, then the same memory region must be used
- Valid SPCI instances and conduits are listed in Table 127
- Syntax of this function is described in Table 128
- Usage of the SPCI_SUCCESS function is described below
  - SPCI_SUCCESS function is used only:
    - If the descriptor specified in Table 130 is fragmented.
    - The fragment that was transmitted is not the last fragment.
  - An invocation of this function indicates that the corresponding fragment was transmitted successfully.
  - No parameters apart from the SPCI_SUCCESS function ID in w0 are used to indicate success.
- Encoding of error code in the SPCI_ERROR function is described in Table 129
- Successful transmission of the last fragment is indicated by an invocation of any SPCI function by the callee

Table 127: SPCI_MEM_RETRIEVE_RESP instances and conduits

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Physical SPCI instance</td>
<td>SMC, ERET</td>
</tr>
<tr>
<td>2</td>
<td>Virtual SPCI instance</td>
<td>ERET</td>
</tr>
</tbody>
</table>

Table 128: SPCI_MEM_RETRIEVE_RESP function syntax

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Function ID</td>
<td>w0</td>
<td>0x84000075</td>
</tr>
<tr>
<td>uint32/uint64 Parameter</td>
<td>w1/x1</td>
<td>Reserved (MBZ).</td>
</tr>
<tr>
<td>uint32/uint64 Parameter</td>
<td>w2/x1</td>
<td>Reserved (MBZ).</td>
</tr>
</tbody>
</table>
| uint32 Fragment count | w3          | Number of fragments of the data after the current one. See Section 6.10. MBZ if the current fragment is the last one.
### Parameter Register Value

**uint32 Length**
- **w4**
- Total length of the data structure specified in Table 130
- MBZ after the first invocation of this interface in which **Fragment count** was > 0.

**uint32 Handle**
- **w5**
- Unique handle allocated by the caller to associate the fragments with the data being transmitted through this interface.
- MBZ if in first invocation of this interface **Fragment count** is 0.

**Other Parameter registers**
- **w6-w7 x6-x7**
- Reserved (MBZ)

### Table 129: SPCI_ERROR encoding

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
</table>
| int32 Error code | w2 | • INVALID_PARAMETERS: Invalid handle
  • NO_MEMORY: Not enough memory to map memory region in callee’s translation regime
  • RETRY: Caller must retry transmission of this fragment. |

### Table 130: Encoding of mapped address ranges of retrieved memory region

<table>
<thead>
<tr>
<th>Field</th>
<th>Byte length</th>
<th>Byte offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count of receiver address range descriptors</td>
<td>4</td>
<td>0</td>
<td>Count of descriptors that specify the address range used to map the donated, lent or shared memory region with the callee endpoint or one or more stream endpoints the callee is a proxy endpoint for.</td>
</tr>
<tr>
<td>Array of receiver address range descriptors</td>
<td>4</td>
<td>4</td>
<td>Array of descriptors that specify the address range used to map the donated, lent or shared memory region with the callee endpoint or one or more stream endpoints the callee is a proxy endpoint for (see Table 131).</td>
</tr>
</tbody>
</table>

### Table 131: Descriptor with address ranges of retrieved memory region

<table>
<thead>
<tr>
<th>ID of receiver endpoint</th>
<th>2</th>
<th>0</th>
<th>16-bit ID of receiver endpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total page count</td>
<td>4</td>
<td>2</td>
<td>Size of the memory region described as the count of 4K pages. Must be equal to the sum of page counts specified in each constituent memory descriptor.</td>
</tr>
<tr>
<td>Address range count</td>
<td>4</td>
<td>6</td>
<td>Count of address ranges specified using constituent memory descriptors.</td>
</tr>
</tbody>
</table>
Table 131: Descriptor with address ranges of retrieved memory region

<table>
<thead>
<tr>
<th>Address range array</th>
<th>4</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Array of address ranges that the relayer has used to map the memory region specified in this transaction in the translation regimes of the callee or one or more stream endpoints the callee is a proxy endpoint for. See Table 30.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

12.5.1 Component responsibilities for SPCI_MEM_RETRIEVE_RESP

This interface is used to complete a transaction to donate, lend or share a memory region with the callee endpoint. This interface is used by the relayer to signal success in response to the latest invocation of the SPCI_MEM_RETRIEVE_REQ interface by the callee. Only a receiver endpoint and relayer participate in this stage of the transaction.

The relayer must fulfill the responsibilities listed in Section 12.4.1.2. It must then use the descriptor specified in Table 131 to encode the address ranges used to map the memory region in each receiver's translation regime.

A mechanism described in Section 6.10 must be used if the size of the descriptor specified in Table 131 is greater than the size of the memory region being used to transfer it. The relayer (caller) and receiver (callee) responsibilities are described in the same section.
12.6 SPCI_MEM_RELINQUISH

Description
• Indicates to the relayer that access to a memory region can be removed from the translation regime of a borrower.
• Valid SPCI instances and conduits are listed in Table 134.
• Syntax of this function is described in Table 135.
• Successful completion of this function is indicated through the invocation of the SPCI_SUCCESS function by the callee.
• Encoding of error code in the SPCI_ERROR function is described in Table 136.

Table 133: Descriptor to relinquish a memory region

<table>
<thead>
<tr>
<th>Field</th>
<th>Byte length</th>
<th>Byte offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handle</td>
<td>4</td>
<td>0</td>
<td>Globally unique handle to identify a memory region</td>
</tr>
</tbody>
</table>
| Flags            | 4           | 4           | bit[0]: Clear memory region contents after unmapping it from the borrower's translation regime  
|                  |             |             | bit[31:1]: Reserved (MBZ).                                   |
| Endpoint count   | 4           | 8           | Count of endpoints                                           |
| Endpoint array count | –       | 12          | Array of endpoint IDs. Each entry contains a 16-bit ID          |

Table 134: SPCI_MEM_RELINQUISH instances and conduits

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Physical SPCI instance</td>
<td>SMC, ERET</td>
</tr>
<tr>
<td>2</td>
<td>Virtual SPCI instance</td>
<td>SMC, HVC, SVC</td>
</tr>
</tbody>
</table>

Table 135: SPCI_MEM_RELINQUISH function syntax

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Function ID</td>
<td>w0</td>
<td>0x84000076</td>
</tr>
<tr>
<td>Other Parameter registers</td>
<td>w1-w7, x1-x7</td>
<td>Reserved (MBZ)</td>
</tr>
</tbody>
</table>

Table 136: SPCI_ERROR encoding

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
</table>
| int32 Error code | w2       | INVALID_PARAMETERS: Invalid description of handle, flags or borrower endpoints  
|              |          | DENIED: Memory region is not in an access state to be relinquished by the borrower |
12.6.1 Component responsibilities for SPCI_MEM_RELINQUISH

This interface is used by a borrower endpoint to inform the relayer that a memory region can be unmapped from its translation regime.

- The memory region is identified by the handle parameter.
- The handle and list of borrower endpoints is specified in descriptor described in Table 133 in the TX buffer of the caller.
- More than one endpoint can be specified if the caller is a proxy endpoint acting on behalf of one or more stream endpoints.
- The caller can also specify if the contents of the memory buffer must be cleared after being unmapped.

Responsibilities of the:
- Borrower are listed in Section 12.6.1.1.
- Relayer are listed in Section 12.6.1.2.

12.6.1.1 Borrower responsibilities

1. Must ensure it has access to the memory region identified by the handle parameter.
2. Must implement support for handling all error status codes that can be returned upon completion of this interface.

12.6.1.2 Relayer responsibilities

1. Must ensure that the global handle provided by the borrower is valid and associated with a memory region it can access. Must return INVALID_PARAMETER in case of an error.
2. Must ensure that the Flags parameter is correctly encoded in the descriptor and the identities of borrower endpoints are valid. Must return INVALID_PARAMETER in case of an error.
3. Must ensure that the memory region is in the iOwner-SA or iOwner-EA state (see Table 22) for all borrower endpoints specified by the caller. Must return DENIED in case of an error.
4. Must unmap the memory region from the translation regime of the borrower as described below.
   1. If the borrower is a PE endpoint running in EL1 or S-EL1, the memory region must be unmapped from its stage 2 translation regime (if present).
   2. If the borrower is a PE endpoint running in S-EL0, the memory region must be unmapped from its stage 1 translation regime.
   3. If the caller is a proxy endpoint for a stream endpoint then the memory region must be unmapped from the stage 2 translation regime corresponding to the SEPID.

The relayer must update internal state of the borrower associated with the memory region to iOwner-NA.
5. Must clear the contents of the memory region if bit[0] is set in the Flags parameter.
6. It is possible that a NS-Endpoint is the owner and a SP is the borrower of a memory region specified in an invocation of this interface. If the Hypervisor is present then the state of the memory region tracked by it and the SPM must remain in sync. This is important since an invocation of the SPCI_MEM_RELINQUISH interface from the SP could happen in parallel to an invocation of the SPCI_MEM_RECLAIM interface by the owner. The invocations must be serialized. This should be done through an implementation defined mechanism and/or by forwarding an invocation of this interface from the Hypervisor to the SPM and vice versa.
12.7 SPCI_MEM_RECLAIM

Description

- Restores exclusive access to a memory region back to its owner
- Valid SPCI instances and conduits are listed in Table 138
- Syntax of this function is described in Table 139
- Successful completion of this function is indicated through the invocation of the SPCI_SUCCESS function by the callee.
- Encoding of error code in the SPCI_ERROR function is described in Table 140

Table 138: SPCI_MEM_RECLAIM instances and conduits

<table>
<thead>
<tr>
<th>Config No.</th>
<th>SPCI instance</th>
<th>Valid conduits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Physical SPCI instance</td>
<td>SMC, ERET</td>
</tr>
<tr>
<td>2</td>
<td>Virtual SPCI instance</td>
<td>SMC, HVC, SVC</td>
</tr>
</tbody>
</table>

Table 139: SPCI_MEM_RECLAIM function syntax

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint32 Function ID</td>
<td>w0</td>
<td>0x84000077</td>
</tr>
<tr>
<td>uint32 Handle</td>
<td>w1</td>
<td>Globally unique handle to identify the memory region</td>
</tr>
<tr>
<td>uint32 Flags</td>
<td>w2</td>
<td>bit[0]: Clear memory region contents before mapping it in owner's translation regime.</td>
</tr>
<tr>
<td>Other Parameter registers</td>
<td>w3-w7 x3-x7</td>
<td>Reserved (MBZ)</td>
</tr>
</tbody>
</table>

Table 140: SPCI_ERROR encoding

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32 Error code</td>
<td>w2</td>
<td>INVALID_PARAMETERS: Invalid description of handle or flags parameter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DENIED: Memory region is not in an access state to be reclaimed by the owner</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NO_MEMORY: Not enough memory to map the memory region in owner's translation regime</td>
</tr>
</tbody>
</table>

12.7.1 Component responsibilities for SPCI_MEM_RECLAIM

This interface is used in the following ways.

1. To complete a transaction to relinquish a memory region owned by the caller endpoint. Borrowers use the SPCI_MEM_RELINQUISH interface to relinquish access to the memory region. The owner uses this interface to reclaim exclusive access to the memory region.
2. To abort an in-progress transaction to donate, lend or share a memory region owned by the caller endpoint. If any receiver endpoint is unable to accept the transaction and the memory region is not mapped into the translation regime of any other receiver endpoint, the owner can use this transaction to reclaim exclusive access to the memory region.

Responsibilities of the:
- Owner are listed in Section 12.7.1.1.
- Relayer are listed in Section 12.7.1.2.

12.7.1.1 Owner responsibilities
1. Must ensure it is the owner of the memory region identified by the handle parameter.
2. Must ensure that access to the memory region has been relinquished by all borrowers.
3. Must implement support for handling all error status codes that can be returned upon completion of this interface.

12.7.1.2 Relayer responsibilities
1. Must ensure that the global handle provided by the owner is valid and associated with a memory region it owns. Must return INVALID_PARAMETER in case of an error.
2. Must ensure that the Flags parameter is correctly encoded. Must return INVALID_PARAMETER in case of an error.
3. Must ensure that the memory region is in the Owner-NA state (see Table 22) for all the receiver endpoints associated with the memory region. If one or more borrowers are stream endpoints without a proxy endpoint. In this case:
   1. Each borrower must relinquish access to the memory region through an implementation defined mechanism.
   2. The relayer must unmap the memory region from the translation regime of the SEPID.
   Must return DENIED in case of an error.
4. Must clear the contents of the memory region if bit[0] is set in the Flags parameter.
5. If the state of the memory region for the owner is Owner-NA i.e. the region was lent, then must map the memory region in the translation regime of the owner as described below. The address range used to add the mapping must be the same as that specified by the owner during the latest transaction to donate, lend or share the memory region.
   1. If the owner is a PE endpoint running in EL1 or S-EL1, the memory region must be mapped in its stage 2 translation regime (if present).
   2. If the owner is a PE endpoint running in S-EL0, the memory region must be mapped in its stage 1 translation regime.
   3. If the caller is a proxy endpoint for a stream endpoint that owns the memory region then the memory region must be mapped in the stage 2 translation regime corresponding to the SEPID.
   Must return NO_MEMORY in case there is not enough memory to create the mapping in the owner's translation regime.
6. If the state of the memory region for the owner is Owner-SA i.e. the region was shared, then must ensure that it is mapped in the translation regime of the owner at the same address range and with the same memory attributes as those when the SPCI_MEM_SHARE interface was invoked.
   Must return NO_MEMORY in case there is not enough memory to change the mapping in the owner's translation regime.
7. It is possible that a NS-Endpoint is the owner and a SP was the borrower in the transaction to relinquish access to the memory region. If the Hypervisor is present then the state of the memory region tracked by it and the SPM must remain in sync. This is important since an invocation of the SPCI_MEM_RELINQUISH interface from the SP could happen in parallel to an invocation of this interface by the owner. The transactions must be serialized. This should be done through an implementation defined mechanism and/or by forwarding an invocation of this interface from the Hypervisor to the SPM and vice versa.

8. If the call executes successfully, the state of the memory region for the owner must transition to Owner-EA.
13 Appendix

13.1 S-EL0 partitions

S-EL0 partitions are used to achieve isolation amongst secure services on Armv8.3 and earlier architecture versions. A S-EL0 partition is expected to host one or more device drivers to control hardware that is only accessible from the Secure world. Normal world will access these drivers through the message passing interfaces described in this specification. Some examples of S-EL0 partition use cases are described below.

13.1.1 UEFI PI Standalone Management Mode partitions

Standalone management mode (STMM) is described in [10] as a processor architecture agnostic, sandboxed secure execution environment. It is meant to be used for device drivers that cannot be implemented in the OS kernel but are required during run time.

On Armv8-A systems, STMM is implemented in a S-EL0 partition to constraint its visibility of the system address map and physical interrupts. This isolation enables a more robust secure firmware implementation. This design is better from a security perspective than a design where STMM drivers are implemented in EL3.

Furthermore, execution in EL3 always runs to completion. Isolation of STMM drivers in a 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.

13.1.1.1 SPCI 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 SPCI interfaces. This guidance is based upon certain assumptions about the Standalone management mode that are listed below.

- A STMM driver is neither reentrant nor thread safe but its single execution context can run on any PE in the system. Hence, a STMM S-EL0 partition is considered to be a UP migrate capable partition.
- STMM services are accessed from the UEFI runtime environment in the Normal world through direct partition messages (see Section 5.4). 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 i.e. one partition does not access another partition’s services.
- 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 [11] specifies the MM_COMMUNICATE interface that enables the Normal world to access driver services implemented in a single STMM S-EL0 partition. SPCI interfaces can be used to access such services implemented in more than one STMM S-EL0 partitions as described below.

- SPCI_MSG_SEND_DIRECT_REQ & SPCI_MSG_SEND_DIRECT_RESP interfaces must be used to send and receive partition messages directly.
- SPCI_RUN interface must be used by the MM communication driver to resume a preempted STMM SP.
- A STMM SP can use any interface defined in Section 7 apart from the following:
  - SPCI_MSG_POLL
  - SPCI_YIELD
  - SPCI_MSG_SEND
This implies that STMM SPs must neither use indirect messaging for communication nor relinquish execution to the primary scheduler.

Based upon the assumptions about Standalone management mode, the following runtime models are applicable to a STMM partition.

- A STMM SP implements services that are invoked in response to interrupts targeted to the SPM. The execution of these services must run to completion.
- A STMM SP implements services that are invoked in response to SPCI interface invocations. The execution of these services could be required to:
  - Either run to completion
  - Or allowed to be transparently preempted and resumed

For simplicity and due to architectural limitations of S-EL0, a STMM SP that implements services that can be preempted must not implement services that are invoked through interrupts i.e. they must be implemented in separate STMM SPs. Each SP's resource description must specify the expected runtime behavior when one of its services is invoked (see Section 4.1.1).

The SPM must enable run to completion semantics through an implementation defined mechanism. These semantics imply that once a service has been invoked, the STMM SP must run at a priority level such that:

- It cannot be preempted by a non-secure interrupt
- It cannot be preempted by a secure interrupt of the same or lower priority
- It can be transparently preempted and resumed by the SPM due to an interrupt of higher priority

The SPM must also ensure that a STMM SP process only a single request at a time to preserve its non reentrant and non-thread safe run time model. This is implicitly guaranteed if the request processing runs to completion.

Some example flows to illustrate common aspects of interaction with a STMM SP based upon the above concepts are described below.

- Figure 22 describes how the MM communication driver can discover presence of STMM SPs and their properties. It is assumed that:
  - All STMM SPs share a MM service UUID. This UUID is used by MM communication driver to discover all the STMM SPs.
  - Each STMM SP specifies this UUID, its run time model, memory regions, devices etc in its partition manifest.
  - The MM communication buffer for each STMM SP is allocated by the EFI MM communication driver.
- Figure 23 describes how the MM communication driver and a STMM SP can communicate using direct partition messages and the communication buffer shared between them.
- Figure 24 describes how the STMM SP can be invoked in response to an interrupt.
Figure 22: MM communication driver initialization
Figure 23: Message exchange between a STMM SP and MM communication driver
Figure 24: Invocation of a STMM SP in response to an interrupt