GlobalPlatform Technology

Virtual Primary Platform – Concepts and Interfaces

Version 1.0.1.15

Public Review

March 2021

Document Reference: GPC_SPE_142
(formerly GPC_FST_142)

Contents

Figures

Tables

1 Introduction

1.1 Audience

This document specifies concepts, rules, requirements, and interfaces related to a Virtual Primary Platform (VPP), as outlined in [IUICC Req], and is intended to ease the portability of VPP Firmwares designed for secure and Tamper Resistant Elements.

Note: Portability is defined as the capability of a program to be executed on various types of data processing systems without converting the program to a different language and with little or no modification ([ISO 2382]).

1.2 IPR Disclaimer

Attention is drawn to the possibility that some of the elements of this GlobalPlatform specification or other work product may be the subject of intellectual property rights (IPR) held by GlobalPlatform members or others. For additional information regarding any such IPR that have been brought to the attention of GlobalPlatform, please visit https://globalplatform.org/specifications/ip-disclaimers/. GlobalPlatform shall not be held responsible for identifying any or all such IPR, and takes no position concerning the possible existence or the evidence, validity, or scope of any such IPR.

1.3 References

Table 1-1: Normative References

Standard / Specification	Description	Ref
GPC_SPE_134	GlobalPlatform Technology Open Firmware Loader for Tamper Resistant Element v2.0	[OFL]
GPC_SPE_140	GlobalPlatform Technology Virtual Primary Platform – Network Protocol v2.0	[VNP]
GPC_SPE_141	GlobalPlatform Technology Virtual Primary Platform – OFL VNP Extension v2.0	[VOFL]
GPC_SPE_143	GlobalPlatform Technology Virtual Primary Platform – VPP Firmware Format v2.0	[VFF]
AIS20	Functionality classes and evaluation methodology for deterministic random number generators, Reference: AIS20, version 1, 02/12/1999, BSI	[AIS20]
AIS31	Functionality classes and evaluation methodology for physical random number generators, Reference: AIS31 version 1, 25/09/2001, BSI	[AIS31]
BSI-CC-PP-0084-2014	Security IC Platform BSI Protection Profile 2014 with Augmentation Packages	[PP-0084]
FIPS PUB 180-4	Secure Hash Standard (SHS)	[FIPS 180-4]
FIPS PUB 186-4	Digital Signature Standard (DSS)	[FIPS 186-4]
FIPS PUB 197	Advanced Encryption Standard (AES)	[FIPS 197]
FIPS PUB 198-1	The Keyed-Hash Message Authentication Code (HMAC)	[FIPS 198-1]

Standard / Specification	Description	Ref
FIPS PUB 202	SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions	[FIPS 202]
IETF RFC 2119	Key words for use in RFCs to Indicate Requirement Levels	[RFC 2119]
ISO/IEC 2382:2015	Information technology – Vocabulary	[ISO 2382]
Joint Interpretation Library	Joint Interpretation Library: “Application of Attack Potential to Smartcards, v2.9, Jan 2013”	[JIL]
NIST Special Publication 800-38A	Recommendation for Block Cipher Modes of Operation	[NIST SP800-38A]
NIST Special Publication 800-38D	Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC	[NIST SP800-38D]
NIST Special Publication 800-57 Part 1 Revision 5	Recommendation for Key Management Part 1: General	[NIST SP800-57 Pt1]
Remote BIST	Elena Dubrova, Mats Näslund, Gunnar Carlsson, John Fornehed, Ben Smeets. *Two Countermeasures Against Hardware Trojans Exploiting Non-Zero Aliasing Probability of BIST.* DOI 10.1007/s11265-016-1127-4. Springer. J Sign Process Syst. 2016	[BIST]
RFC 2104	HMAC: Keyed-Hashing for Message Authentication	[RFC 2104]
RFC 4122	A Universally Unique IDentifier (UUID) URN Namespace	[RFC 4122]
RFC 4231	Identifiers and Test Vectors for HMAC-SHA-224, HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512	[RFC 4231]
RFC 4492	Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)	[RFC 4492]
RFC 4493	The AES-CMAC Algorithm	[RFC 4493]

Table 1-2: Informative References

Standard / Specification	Description	Ref
GPC_SPE_014	GlobalPlatform Card Specification v2.3 Amendment D: Secure Channel Protocol '03' v1.2	[HLOS14]
GPC_SPE_025	GlobalPlatform Card Specification v2.3 Amendment C: Contactless Services v1.3	[HLOS25]
GPC_SPE_034	GlobalPlatform Card Specification v2.3	[HLOS34]
GPC_SPE_042	GlobalPlatform Card Specification v2.3 Amendment E: Security Upgrade for Card Content Management v1.1	[HLOS42]
GPC_SPE_093	GlobalPlatform Card Specification v2.3 Amendment F: Secure Channel Protocol '11'	[HLOS93]
ETSI TS 102 221	Smart Cards; UICC-Terminal interface; Physical and logical characteristics	[102 221]

Standard / Specification	Description	Ref
ETSI TS 102 223	Smart Cards; Card Application Toolkit (CAT)	[102 223]
ETSI TS 102 622	Smart Cards; UICC – Contactless Front-end (CLF) Interface; Host Controller Interface (HCI) (Release 13)	[102 622]
FIPS PUB 140-2	Security Requirements for Cryptographic Modules	[FIPS 140-2]
GSMA iUICC PoC PP	GSMA iUICC PoC Group Primary Platform Requirements	[IUICC Req]
GSMA SGP.02	GSMA Remote Provisioning Architecture for Embedded UICC Technical Specification v4.0	[HLOS02]
ISO/IEC 7816-4	Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange	[7816-4]

1.4 Terminology and Definitions

Selected terms used in this document are included in Table 1-3.

Table 1-3: Terminology and Definitions

Term	Definition
Abstraction	An interface layer that masks underlying implementation differences.
Access Group	A logical grouping of Programs and/or hardware functions to indicate their level of access to other Access Groups.
Accessor	(1) A Program capable of reading the data of a hardware function. (2) A hardware function capable of reading the data of another hardware function.
Address Space	The set of addresses that can be used by a particular Program or functional unit.
Application Binary Interface (ABI)	An interface between two Programs, using the processor instruction set (with details such as registers, stack organization, memory access types, CPU modes...), the sizes, layout, and alignment of basic data types the CPU can directly access. The use of an ABI targets specifically the interface between a Process and the LLOS running in different CPU modes.
Application Programming Interface (API)	A set of rules that software programs can follow to communicate with each other.
Bootstrap Program	A short Program that is permanently resident or easily loaded into a computer and whose execution brings a larger Program, such as an operating system or its loader, into memory ([ISO 2382]). In the context of this document, a Program used to load the VPP Firmware of the Primary Platform from a Remote Memory and to instantiate this Program.
COM Process	A Process of the VPP Execution Domain that enables communication between VPP Applications.

Term	Definition
Composite Identifier	Identifier having two parts: Execution Domain type and an enumerated identifier.
CPU	In the context of this document, refers to the CPU inside the TRE that executes any Program in the Primary Platform.
Embedded TRE	TRE hosted in a standalone integrated circuit and having no Memory Partition in a Remote Memory.
Exception	A notification from the kernel to a parent Process about a special condition that occurred in one of its child Processes.
Execution Domain	Property defining the membership of a Process in a logical group of Processes.
Firmware	Ordered set of instructions and associated data stored in a way that is functionally independent of main storage, usually in a ROM[1] ([ISO 2382]). See also *VPP Firmware*.
Forward Compliance	The capability to support future needs (e.g. larger VPP Firmware) within the limits of the resources outside the TRE.
Hardware Platform	Hardware architecture of the TRE.
High Level Operating System (HLOS)	An additional Program encompassed in a VPP Application that provides further Abstraction of resources and services to its HLOS Application(s).
HLOS Application	A Program or interpretable code using the HLOS to deliver a specific Use Case.
Instance	Concrete occurrence of any object at run time. The creation of an instance from the description of an object (e.g. VPP Firmware) is called instantiation.
Integrated TRE	A TRE that is integrated into, and part of, a larger SoC.
Inter Process Communication (IPC)	Communication based on a shared virtual memory between two Processes.
Interoperability	The capability to communicate, execute programs, or transfer data among various functional units in a manner that requires the user to have little or no knowledge of the unique characteristics of those units ([ISO 2382]).
IPC Descriptor	Defined in [VFF].
Kernel	See *Microkernel*.
Kernel Object	Implementation-dependent data structure within the kernel; can be a Process, Mailbox, IPC, or VRE.
Kernel Object Handle	A runtime reference to a Kernel Object instantiated by the kernel.
Kernel Object Identifier	A unique identifier that allows a VPP Application to retrieve a Kernel Object Handle. The identifier is known at development time. Some identifiers are pre-defined as part of VPP.

[1] Generically, ROM shall be understood as a Non-Volatile Memory (NVM).

Term	Definition
Key Protection Function	Hardware mechanism to separate key material from CPU access.
LIB Descriptor	Shared library metadata, as defined in [VFF].
Low Level Operating System (LLOS)	A privileged hardware-dependent Program. It includes a Microkernel, critical low-level resources, and support for security functions.
Mailbox	A component owned by a Process capable of receiving Signals from another Process or the kernel.
Mailbox Descriptor	Defined in [VFF].
MAIN Process	The first Process to run of a VPP Application.
Master Port	The access point to direct a memory transfer hardware function.
Memory Management Function (MMF)	A function in charge of translating a physical address to a virtual address and applying rules of access (e.g. read, write, execute) according to a Security Perimeter and the nature of the virtual address space.
Memory Partition	Concatenated Sub-Memory Partitions, as defined in [VFF].
MGT Process	A Process within the VPP Execution Domain responsible for managing VPP Applications.
Microkernel	Minimal Program supporting only the functionalities that are not able to run in unprivileged mode such as memory management, including their access protection settings, multi-processing support. For simplicity, the rest of this document uses “kernel” as an alias of Microkernel.
Minimum Level of Interoperability (MLOI)	The set of minimum system capabilities that are guaranteed to exist in every Primary Platform.
Mutator	(1) A Program capable of writing, modifying, or controlling the data of a hardware function. (2) A hardware function capable of writing, modifying, or controlling the data of another hardware function.
Non-Shareable Memory Space	Memory space that shall be declared by and accessed by a single Program.
One-Time Programmable (OTP)	A form of memory that can be read many times, but only written once.
Operating System	Program that controls the execution of other Programs and that provides services such as resource allocation, scheduling, input-output control, and data management ([ISO 2382]).
Physical Address	The actual, non-translated, addresses that can be used by a particular Program or functional unit.
Physical Address Space	The set of physical addresses that can be used by a particular Program or functional unit.
Pre-emption	The act of temporarily interrupting a Process being managed by a kernel, without requiring its cooperation, and with the intention of resuming the Process at a later time.
Primary Platform	A TRE along with a Low Level Operating System and VPP Services.

Term	Definition
Primary Platform Maker	The entity developing and providing the Primary Platform.
Privileged CPU Mode	CPU mode when dealing with hardware exceptions or when executing privileged instructions.
Process	Independent sequence of execution in CPU unprivileged mode and running within an independent Virtual Address Space. A Process shares memory with other Processes (e.g. IPC). A Process is scheduled by the LLOS.
Process Descriptor	Process metadata, as defined in [VFF].
Program	Independent set of instructions executed by a CPU.
Remote Audit Function	A mechanism that enables a VPP Application to challenge the authenticity of the Primary Platform.
Remote Memory	Memory located outside the TRE, which can be RAM and NVM.
Secure CPU	A CPU supporting security functions for preventing hardware attacks (e.g. DPA, DFA, Side channels).
Security Perimeter (SP)	Denotes the perimeter of a function within which rules, Access Groups, properties, and requirements apply.
Shared Memory Space	Memory space accessed by at least two Programs that declare sharing of a specific memory space.
Signal	(1). noun. A fixed bit of information representing a fixed event. (2). verb. The act of sending the signal bit between two Processes or between a Process and the kernel.
System Tick	A fixed duration periodical event that interrupts the Microkernel and is used in Process scheduling.
Tamper Resistant Element (TRE)	Hardware that supports the security and tamper resistance requirements for a Primary Platform.
Thread	A Process within another Process that uses the resources of the latter Process ([ISO 2382]).
Unprivileged CPU Mode	CPU mode when NOT able to access privileged instructions and when NOT dealing with hardware exceptions.
Use Case	A list of actions or event steps typically defining the interactions between a role and a system to achieve a goal.
Virtual	Pertains to a functional unit that appears to be real, but whose functions are accomplished by other means ([ISO 2382]).
Virtual Address Space	Set of virtual addresses that can be used by a particular Program or functional unit.
Virtual Address Space Region	A fixed-size partition of the Virtual Address Space dedicated to a given memory type of a Process, e.g. code, stack, constants, etc.
Virtual Hardware Platform	The virtualized hardware as it appears to a VPP Application.
Virtual Primary Platform (VPP)	The isolated context of the Primary Platform that executes the VPP Application.

Term	Definition
Virtual Register (VRE)	Virtual REgister is a set of virtual addresses allowing access to a hardware function.
Virtualization	Creation of a virtual (rather than actual) version of something, including virtual computer hardware platforms, storage devices, and computer network resources.
VPP Application	An instance of a VPP Firmware on a VPP; Use Case dependent and can run an HLOS and its application(s).
VPP Firmware	Firmware compliant with the VPP Firmware Format as defined in [VFF].
VPP Firmware Format	The data structure of the VPP Firmware. See [VFF].
VPP Firmware Identifier	Defined in [VFF].
VPP Firmware Impersonation	The access of the VPP Firmware Loader to the Sub-Memory Partitions of the VPP Firmware being managed (e.g. install, update).
VPP Firmware Loader	Defined in [VOFL].
VPP Process	A Process within the VPP Execution Domain.
VPP Services	The VPP Processes that abstract services such as communication and VPP Firmware management. These are provided by the VPP implementation.

1.5 Abbreviations and Notations

In this document, the following conventions apply:

- All lengths are presented in bytes, unless otherwise stated. Each byte is represented by bits b8 to b1, where b8 is the most significant bit and b1 is the least significant bit. In each representation, the leftmost bit is the most significant bit.
- Hexadecimal values are specified between single quotes, e.g. '1F'.
- All bytes specified as RFU shall be set to '00' and all bits specified as RFU shall be set to 0.
- Security Requirements established in this document are indicated by [SREQ**XX**] or by a dot in the SR column for tables in section 5.8.2. Both are used to ease traceability.
- Functional Requirements established in this document are indicated by [REQ**XX**] in order to ease their traceability.
- Commands not explicitly indicated as “optional” are mandatory.

Selected abbreviations and notations used in this document are included in Table 1-4.

Table 1-4: Abbreviations

Abbreviation	Meaning
ABI	Application Binary Interface
API	Application Programming Interface
BIST	Built-In Self Test
CPU	Central Processing Unit

Abbreviation	Meaning
DFA	Differential Fault Analysis defined in [JIL]
DPA	Differential Power Analysis defined in [JIL]
HLOS	High Level Operating System
IPC	Inter Process Communication
LIB	Library
LLOS	Low Level Operating System
MGT	Management
MLOI	Minimum Level of Interoperability
MMF	Memory Management Function
MTU	Maximum Transmission Unit
NA	Not Applicable
NVM	Non-Volatile Memory
OTP	One-Time Programmable
PFS	Perfect Forward Secrecy
PUF	Physical Unclonable Function
RAM	Random Access Memory (assumed to be volatile)
RO	Read-Only
RW	Read/Write
SoC	System on Chip
SP	Security Perimeter
SR	Security Requirement
TOE	Target Of Evaluation
TRE	Tamper Resistant Element
UUID	Universally Unique IDentifier version 5 as defined in [RFC 4122]
VPP	Virtual Primary Platform
VRE	Virtual Register
WO	Write-Only

1.6 Revision History

GlobalPlatform technical documents numbered *n*.0 are major releases. Those numbered *n*.1, *n*.2, etc., are minor releases where changes typically introduce supplementary items that do not impact backward compatibility or interoperability of the specifications. Those numbered *n.n*.1, *n.n*.2, etc., are maintenance releases that incorporate errata and precisions; all non-trivial changes are indicated, often with revision marks.

Table 1-5: Revision History

Date	Version	Description
March 2018	1.0	Public Release (with document reference GPC_FST_142)
August 2018	1.0.1	Maintenance Release
October 2019	1.0.1.5	Committee Review
December 2019	1.0.1.6	Member Review
March 2021	1.0.1.15	Public Review
TBD	2.0	Public Release (with document reference GPC_SPE_142)

2 Overview

2.1 Objectives and Use Cases

The main objectives of the Virtual Primary Platform (VPP) are described in [IUICC Req]. They can be split as follows:

- Make the VPP Application as independent as possible from the underlying Primary Platform and consequently reduce the engineering effort to adapt the VPP Application to each different Primary Platform.
- Provide at least the same level of flexibility as the legacy secure platforms for supporting all Use Cases defined in [IUICC Req].
- Ease the VPP Application certification (if required) by enabling composite certification with a pre-certified Primary Platform.

The following Primary Platform specifications shall be provided by the Primary Platform Maker:

- The architecture of the CPU and its reference
- The specification of the hardware functions accessible by the VPP Application[2]
- The Low Level Operating System (LLOS) Application Binary Interface (ABI) description adapting the interface to the LLOS implementation for a given CPU
- The Primary Platform dependent parameters defined in section 7

[2] Via the VRE as defined in section 5.8.5.5.

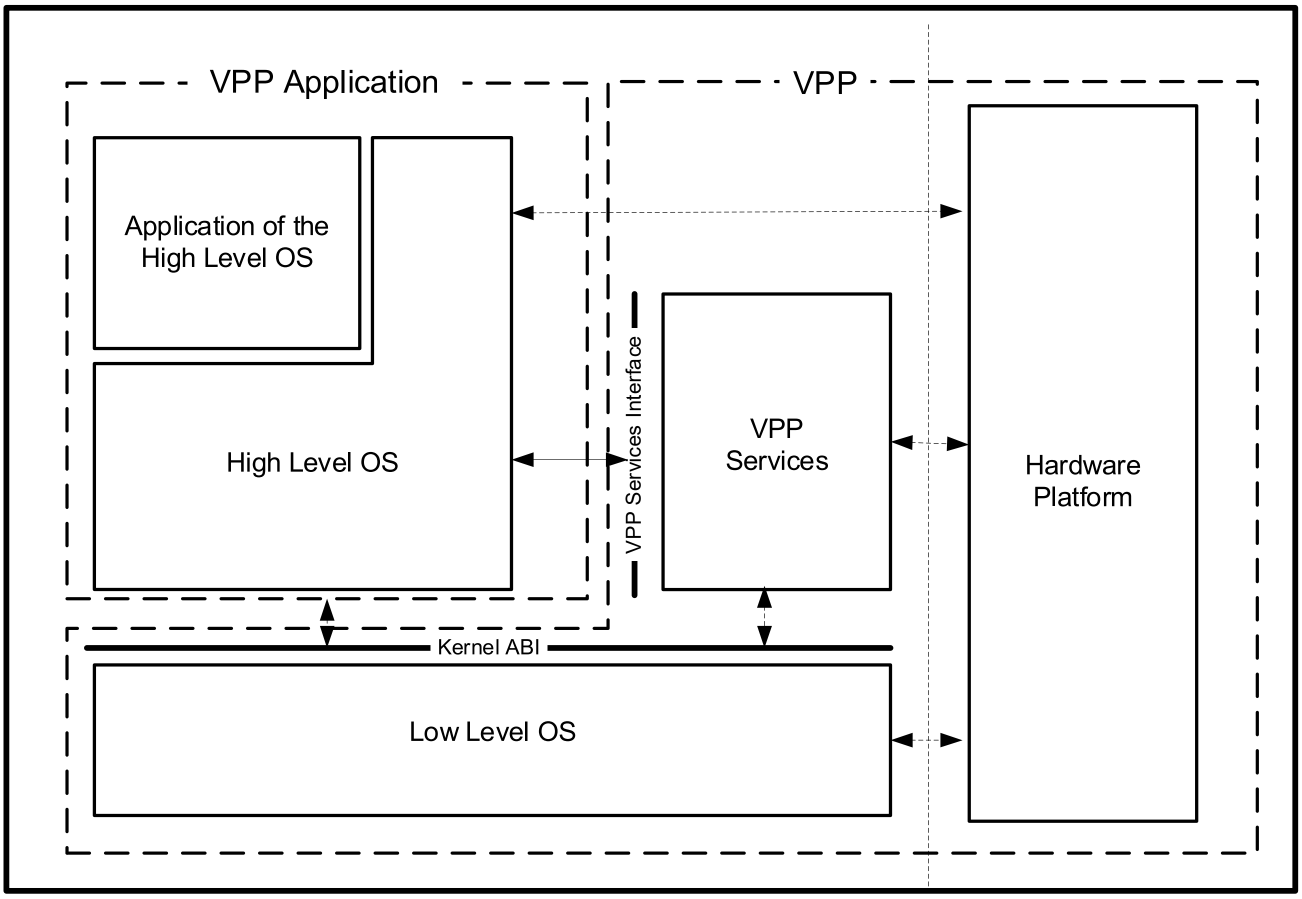
2.2 VPP Concept

The VPP concept contains three logical parts:

1. The Primary Platform consisting of the LLOS, the VPP Services, and the Hardware Platform
2. The Abstraction and Virtualization of the Primary Platform, called VPP, making the interface to any Primary Platform virtually the same[3]
3. The Use Case dependent application of the VPP (i.e. a VPP Application) consisting of a High Level Operating System (HLOS) and its application(s)

Figure 2-1 illustrates the internal architecture of the VPP concept.

Figure 2-1: VPP Concept Architecture



From the VPP Application perspective, the VPP hides the Primary Platform.

[3] It is foreseen that a VPP Firmware needs, to a certain extent, to be adapted and re-compiled for the Primary Platform.

2.3 The Content of VPP Concepts and Interfaces

- The requirements of the Primary Platform
- The Virtual Primary Platform
- The Virtualization of resources as described in section 5
- The interface with the LLOS
- The interface with the Communication Service
- The interface with the VPP Firmware Management Service
- The VPP Application
- The Minimum Level of Interoperability (MLOI)

3 Hardware Platform

3.1 Hardware Architecture

The Hardware Platform is implemented in a Tamper Resistant Element (TRE) and shall embed [SREQ**1**]:

- One or more CPU(s), which shall be specifically built for high level of security
- Random Access Memory (RAM)
- Non-Volatile Memory (NVM) of potentially different types (reprogrammable or not)
- A Random Number Generator Function
- Long-term credentials storage
- Security functions (e.g. sensors)
- A Key Protection function to restrict CPU access to secret key material
- A Memory Management Function in charge of translating the physical memory addresses to virtual memory addresses and protecting memory spaces according to their access rights.

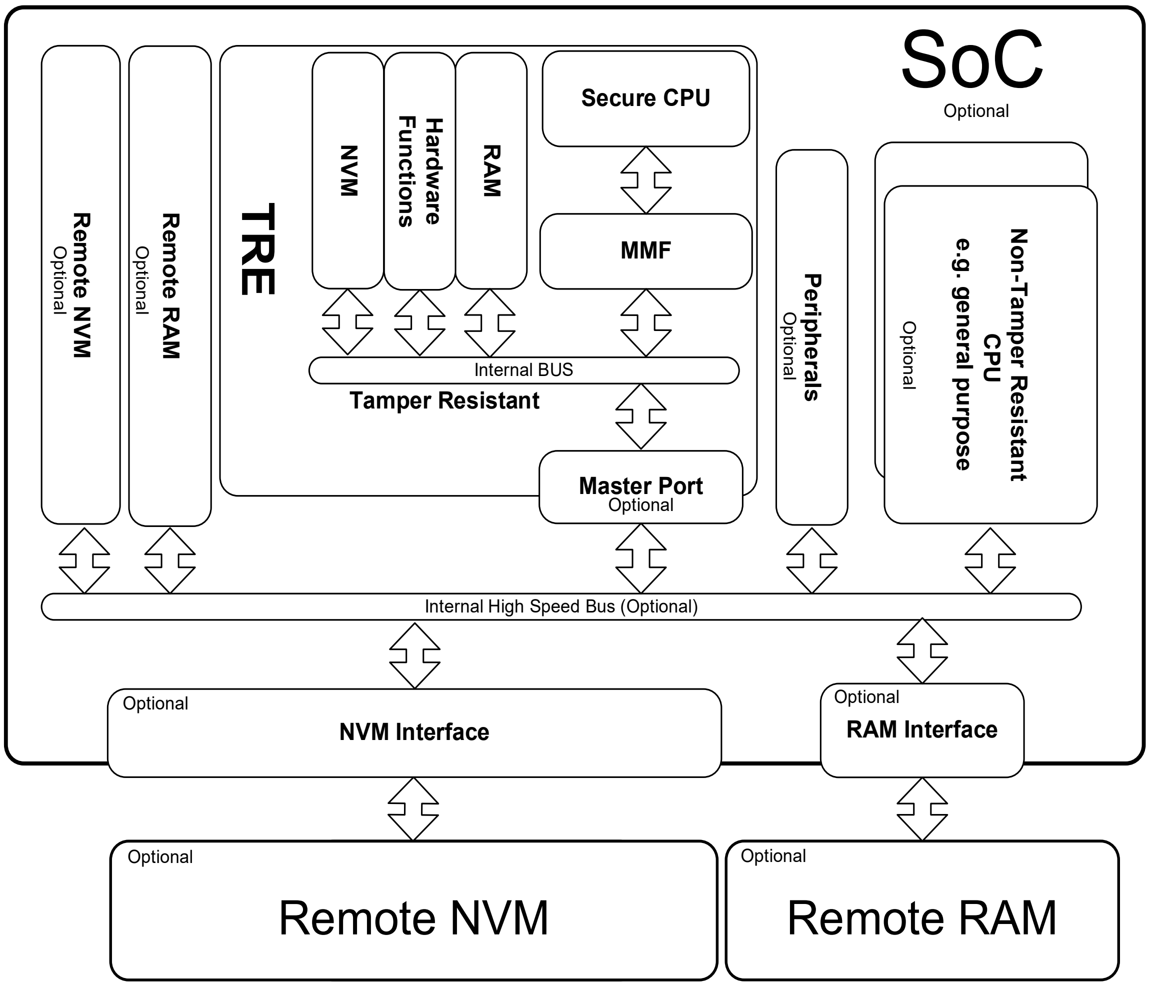
The hardware platform should furthermore embed [SREQ**2**]:

- Cryptographic Functions
- Memory Transfer Function (e.g. a Direct Memory Access controller)

Figure 3-1 shows a generic hardware platform for an Integrated TRE. For example, the SoC may be a mobile broadband modem, an application processor, a micro-controller, or a dedicated controller (e.g. an NFC controller).

Figure 3-1 shows a particular example, in the sense that the SoC may contain many additional elements that are not shown.

Figure 3-1: Example of Hardware Platform



3.2 Generic Core Features

3.2.1 CPU

The CPU shall support at least two execution modes: Privileged CPU Mode and Unprivileged CPU Mode. [SREQ**3**]

The CPU architecture shall support an Application Binary Interface (ABI) able to transfer scalar parameters between a Program running in Unprivileged CPU Mode and a Program running in Privileged CPU Mode. [SREQ**4**]

Note: ‘CPU’ in this document refers to the CPU inside the TRE that executes any Program in the Primary Platform.

3.2.2 Memory Access

The Primary Platform shall provide a means to filter memory access, for each CPU mode, based on a combination of multiple memory access (execute, read, write). [SREQ**5**]

3.2.3 Non-Volatile Memory

At least one of the following options shall be implemented in the hardware platform:

- One Time Programmable NVM (OTP) [REQ**6**]
- Reprogrammable NVM [REQ**7**]
- Non-programmable NVM (i.e. ROM) [REQ**8**]

The Hardware Platform may use a remote NVM relying on a NVM outside the Security Perimeter of the TRE. [REQ**9**]

3.2.4 Form Factor

No form factor is mandated.

3.2.5 Power

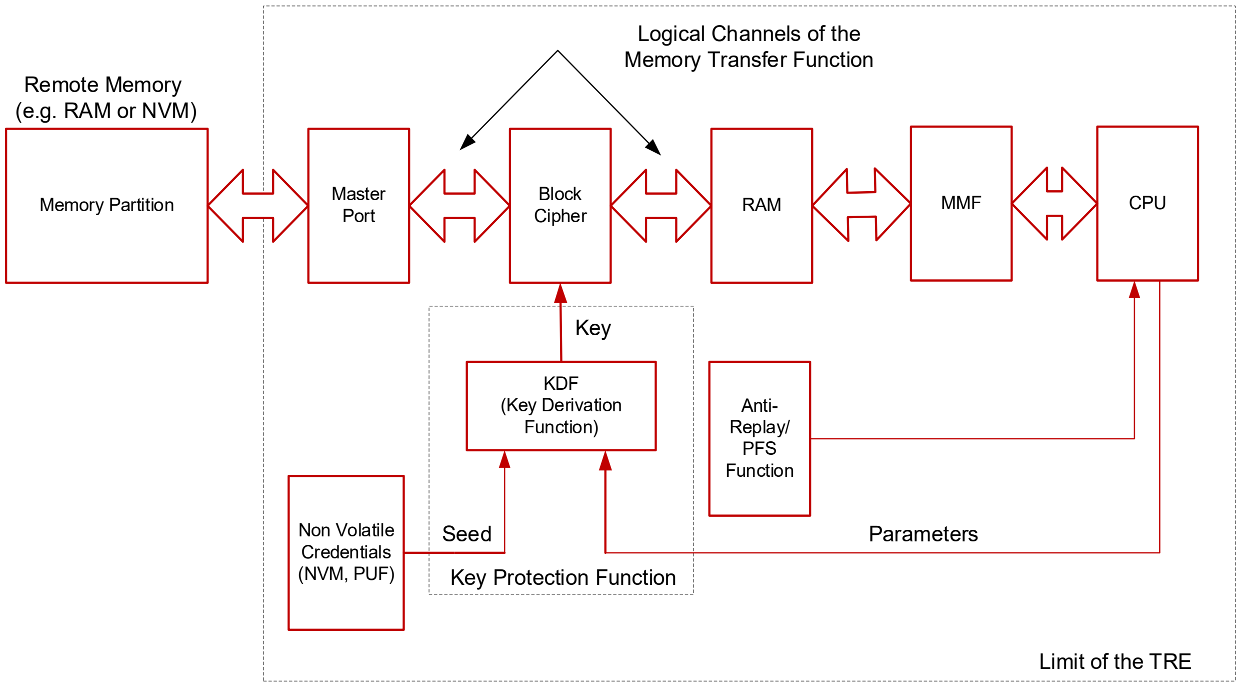
The overall power consumption of the Primary Platform is out of scope of this specification. For this document, the Primary Platform shall be considered as powered up and capable of supporting the specified behavior. The Primary Platform Maker may implement power management strategies that may not support specified behavior; for example, power down or power collapse. Such strategies and their accompanying states are also out of scope of this document.

3.2.6 Memory Transfer Function

The Hardware Platform may provide a Memory Transfer Function (e.g. DMA) to allow a direct data transfer (meaning without the use of the CPU) between two resources (i.e. hardware interface to function or memory).

Figure 3-2 illustrates the concept of Memory Transfer Function.

Figure 3-2: Memory Transfer Function Example


Logical Channels of the Memory Transfer Function
Remote Memory (e.g. RAM or NVM)
Memory Partition
Master Port
Block Cipher
RAM
MMF
CPU
Key
KDF (Key Derivation Function)
Anti-Replay/ PFS Function
Non Volatile Credentials (NVM, PUF)
Seed
Parameters
Key Protection Function
Limit of the TRE

In the above example, two logical channels of the Memory Transfer Function allow the data transfer between a Remote Memory (i.e. either RAM or NVM) and a memory area located inside the TRE via a block cipher function. The Master Port is controlled by the TRE and makes the bridge between the TRE and the SoC buses.

If a Memory Transfer Function is required for transferring data outside the TRE then:

- The Remote Memory (e.g. Memory Partition) and the TRE memory area(s) shall be isolated by a hardware function (e.g. a master port) in order to ensure the following:
  - Access control is managed by the TRE. [SREQ**10**]
  - Address translation across the different address spaces (e.g. the SoC address space and the TRE address space)

If the Memory Transfer Function is required for transferring data inside or outside the TRE, then the Primary Platform shall provide a means to protect against memory content access violations from hardware memory transfer. [SREQ**11**]

Data stored in Remote Memory must be integrity protected and confidentiality protected. [SREQ**12**]

3.2.7 Cryptographic Functions

The TRE shall execute cryptographic operations only within its Security Perimeters (i.e. the TRE SP) [SREQ**13**]. The Primary Platform may provide hardware assistance for cryptographic functions [REQ**14**].

3.2.8 Random Number Generator Function

The TRE shall execute random number generation operations only within its Security Perimeters (i.e. TRE SP). [SREQ**15**]

The Random Number Generator Function should be in conformance with [SREQ**16**]:

- AIS20 certification for a DRNG (Deterministic Random Number Generator) as defined in AIS20 version 1 ([AIS20])
- AIS31 certification for a TRNG (True Random Number Generator) as defined in AIS31 version 1 ([AIS31])

3.3 System Functions

The Hardware Platform shall embed an autonomous and independent clock and reset system confined in the TRE Security Perimeter. [SREQ**17**]

The Hardware Platform shall provide a tick counter, which has a fixed tick duration and is rising. [REQ**18**]

The Hardware Platform may embed additional system functions (e.g. interrupt controller, etc.) out of scope of this document.

3.4 Security Functions

3.4.1 General

The Primary Platform shall have an exclusive control over the mechanism that controls access to the Primary Platform from outside the TRE. [SREQ**19**]

The Primary Platform shall provide means for protecting itself against side channel attacks (hardware and Programs), Differential Power Analysis (DPA) attacks, and Differential Fault Analysis (DFA) attacks. [SREQ**20**]

3.4.2 Memory Encryption and Integrity

The TRE shall depend only on its own internal cryptographic Programs and hardware functions. [SREQ**21**]

The robustness of the memory encryption shall be equivalent to or greater than AES-256-CBC. [SREQ**22**]

Memory encryption and integrity may be combined with a hardware integrity check. [REQ**23**]

The robustness of the memory integrity shall be equivalent to or greater than HMAC-SHA-256. [SREQ**24**]

3.4.3 Key Protection Function

The Hardware Platform shall provide and use a Key Protection Function to protect specific long-term key material. [SREQ**25**]

Keys used for encryption and integrity checking of Remote Memory shall be protected by the Key Protection Function and shall not be directly accessible by the CPU. [SREQ**26**]

The Key Protection Function shall utilize a hardware key derivation function as defined in [NIST SP800-57 Pt1] section 8.2.4 [SREQ**27**]. The hardware key derivation function shall provide an interface that accepts [SREQ**28**]:

- A variable accessible only by Programs executing in Privileged CPU Mode
- A diversified persistent secret seed (e.g. from a TRE NVM or from a PUF) which is only accessible by the Key Protection Function

The Key Protection Function shall support a mode where its output shall be the key for the Cryptographic Functions. [SREQ**29**]

3.4.4 Security Sensor Function

The Hardware Platform may embed security sensor function in charge to detect abnormal operating conditions. The implementation of such function is out of scope of this document.

3.5 Memory Management Function

The Primary Platform shall embed a Memory Management Function (MMF) under the control of the LLOS [SREQ**30**] to support [SREQ**31**]:

- The Forward Compliance property, regardless of the VPP Firmware size
- A Virtual Address Space, eliminating Processes dependency on Physical Address Space of the Hardware Platform

The Memory Management Function shall enforce the access rules (i.e. Access Groups) of every Security Perimeter as defined in section 5.2. [SREQ**32**]

3.6 Memory Storage Function

The memory storage area is considered remote when it is located outside the TRE. Multiple remote memory areas may be available with different persistency (e.g. RAM, NVM) and read and write access times.

3.7 Remote Audit Function

Either the VPP Firmware Loader or the VPP Application will provide information called 'a challenge' to the Remote Audit Function, which will in return send a result. This result is sent outside the TRE and compared to a precomputed result; if the values are identical, then the integrity of the TRE is established.

The Remote Audit Function can be used for the FIPS PUB 140-2 certification for cryptographic modules ([FIPS 140-2]).

3.7.1 Remote Audit Function Requirements

The Hardware Platform should support a Remote Audit Function. [SREQ**33**]

If the Hardware Platform supports the Remote Audit Function, then the following requirements in this section apply.

The Remote Audit Function shall provide a means for a VPP Application/Firmware Loader to challenge the Hardware Platform against modification by comparing it to the one that has been certified as detailed in TOE life cycle phase 7, defined in [PP-0084]. [SREQ**34**]

The following requirements shall be fulfilled by the Hardware Platform:

- Support a Remote Audit Function that is able to detect a Hardware Platform modification with a probability greater than 0.8. [SREQ**35**]
- Prohibit access to the key components by the Remote Audit Function. [SREQ**36**]
- Provide a means for a VPP Application/Firmware Loader to provide, in accordance with the LLOS and any SoC sub-systems, a challenge needed for the Remote Audit Function in a confidential manner. [SREQ**37**]
- Provide a means for the VPP Application/Firmware Loader to retrieve the Remote Audit Function result in a confidential manner. [SREQ**38**]

Excluding the VPP Firmware used for the instantiation of the VPP Firmware Loader, a given VPP Firmware may, during its entire life cycle,[4] inject only a single Remote Audit Function challenge into the Remote Audit Function. [SREQ**39**] [5]

The duration of the Remote Audit Function operation[6] should not exceed MK_MAX_RAF_TIME_MS. [REQ**40**]

The VPP Application/Firmware Loader session shall be terminated to allow the Remote Audit Function operation to start. [SREQ**41**]

The termination of the VPP Application/Firmware Loader session is initiated by the MAIN Process or by the MGT Process as defined in section 6.3.

Any result mismatch shall indicate that the Hardware Platform has been modified. [SREQ**42**]

[4] From its initial loading to its deletion.

[5] The requirement prevents a Denial of Service attack by a VPP Application.

[6] The protocol between the TRE and other SoC sub-systems that is used to properly stop the TRE, switch the TRE hardware platform into Remote Audit Function mode, and switch it back to the operational mode is VPP implementation specific.

3.7.2 Remote Audit Function Example

The following is an example of how the Remote Audit Function is used to check the authenticity of a TRE after its certification:

- Once the compliance testing of a given TRE completes successfully, the certification laboratory generates a certain number of challenges, to be fed into the Remote Audit Function. The results are then stored in a database.[7]
- The remote auditor:
    - Requests from the certification laboratory a set of archived challenge/result pairs.
    - Establishes a secure communication channel[8] with the VPP Application/Firmware Loader.
    - Using the secure communication channel, transfers one of the requested challenges to the VPP Application/Firmware Loader.
    - Switches to the Remote Audit Function mode then back to the VPP Application/Firmware Loader.
    - Collects the corresponding result from the Remote Audit Function over the secure communication channel.
    - Compares the archived result corresponding to the collected result.

3.7.3 BIST-based Remote Audit Function

The Remote Audit Function may use a remote Built-In Self Test (BIST)[9] mechanism as described in [BIST] to challenge the Hardware Platform. In that case, the Remote Audit Function challenge is taken as part of the test initialization parameters of the Pseudo-Random Pattern Generator. The Hardware Platform isolated from other SoC sub-systems is therefore the Circuit Under Test. The Remote Audit Function result is the signature computed by the Multiple Input Signature Register.

The archived challenge/result pairs generated by the Remote Audit Function should not be predictable.

The number of possible Remote Audit Function challenge/result pairs shall be large enough to be resistant to precomputation of all possible challenge/result pairs.

3.8 Hardware Service Functions

Hardware Service Functions are implementation dependent and out of scope of this document.

One Hardware Service Function is related to the communication between the TRE and the SoC sub-systems, e.g. physical layers.

[7] The management of the database of the certification laboratory is out of scope of this document.

[8] The procedure for setting up secure communication between a remote auditor and a VPP Application may be proprietary and is out of scope of this document.

[9] E.g. L-BIST (Logic BIST)

4 Primary Platform Certification

The certification of the Primary Platform shall claim in its Security Target one of the following options [SREQ**43**]:

- Conformance with Protection Profile BSI-CC-PP-0084-2014 ([PP-0084]) [SREQ**44**]
- Conformance with Protection Profile BSI-CC-PP-0084-2014 ([PP-0084]) and the certification by composition of the Loader package 2 [SREQ**45**]
- Conformance with Protection Profile BSI-CC-PP-0084-2014 ([PP-0084]) including the Loader package 2 [SREQ**46**]

Note: Only the two last options are compliant with the requirements described in [IUICC Req].

The Primary Platform shall be encompassed by the certification Target of Evaluation (TOE) boundary. [SREQ**47**]

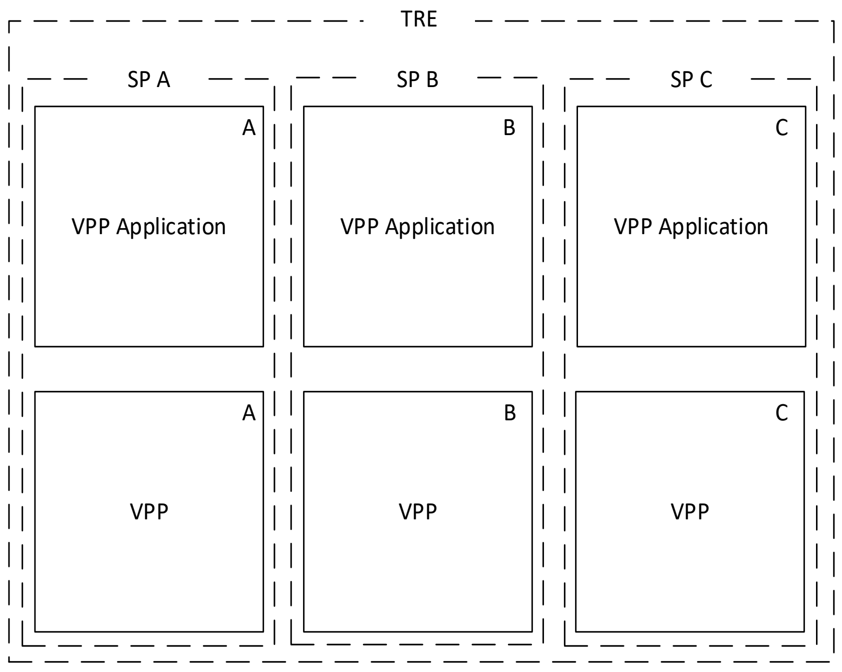
The Security Target of the Primary Platform shall cover the security requirements listed in this document. [SREQ**48**]

The certification minimum assurance level shall be at least EAL4 augmented with AVA_VAN.5 and ALC_DVS.2. [SREQ**49**]

AVA_VAN.5 tests should be performed in accordance with the JIL Application of Attack Potential to Smartcards documentation ([JIL]). [SREQ**50**]

As illustrated in Figure 4-1, a TRE may support multiple Virtual Primary Platforms as long as each isolated context is independently compliant with the requirements of this document and those of both [VFF] and [VNP].

Figure 4-1: Multiple Virtual Primary Platforms in a TRE



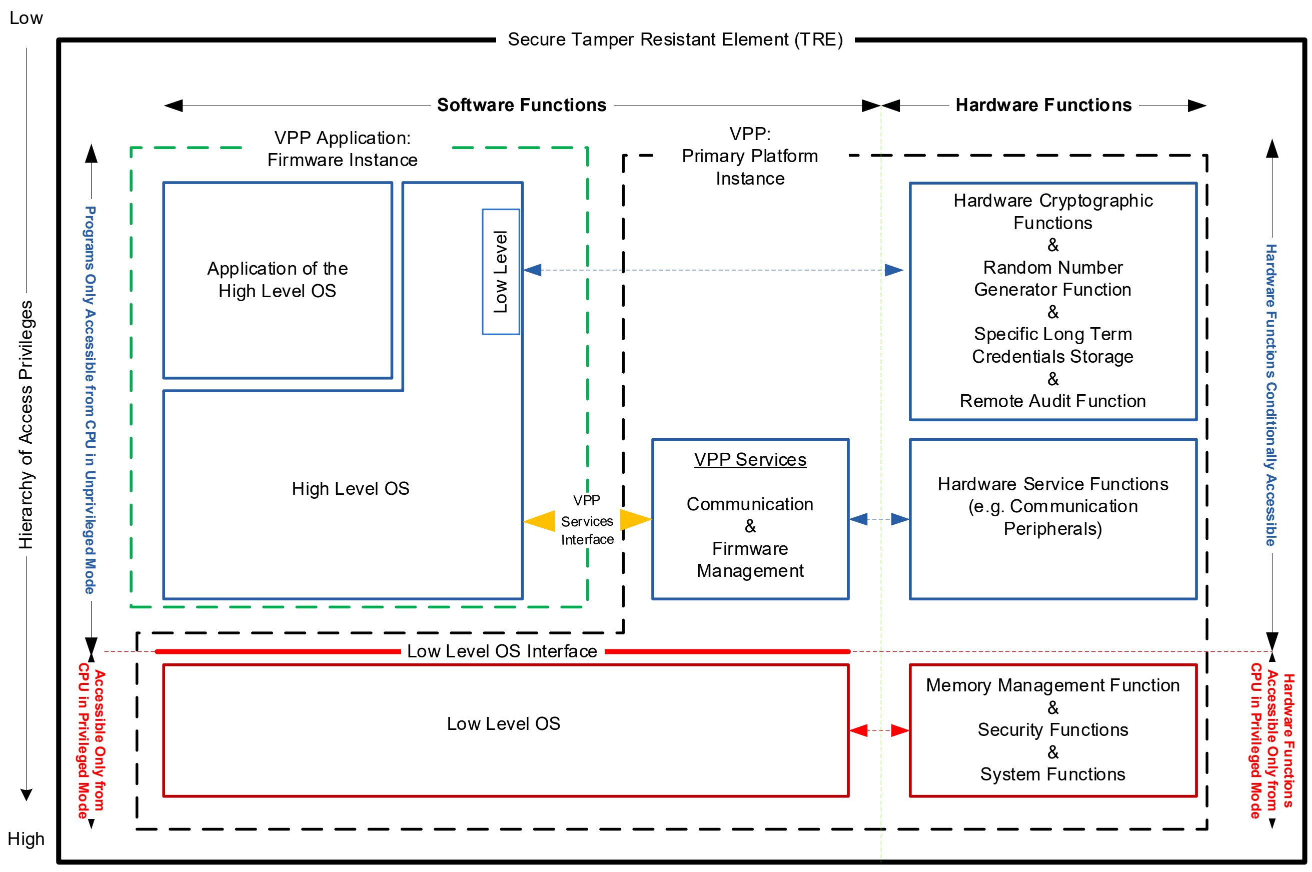
Note: A VPP Application running on one of several VPPs confronts no differences of behavior, rules, security, certifications, or MLOI compared to one running in a TRE supporting a single VPP. The VPP and the VPP Application run in their own Security Perimeter.

5 Virtual Primary Platform

5.1 Overview

Figure 5-1 describes the functional architecture of a TRE.

Figure 5-1: TRE Functional Architecture


Low
Hierarchy of Access Privileges
High
Secure Tamper Resistant Element (TRE)
Software Functions
Hardware Functions
Programs Only Accessible from CPU in Unprivileged Mode
Accessible Only from CPU in Privileged Mode
VPP Application: Firmware Instance
Application of the High Level OS
Low Level
High Level OS
VPP Services Interface
VPP: Primary Platform Instance
VPP Services
Communication & Firmware Management
Hardware Cryptographic Functions & Random Number Generator Function & Specific Long Term Credentials Storage & Remote Audit Function
Hardware Service Functions (e.g. Communication Peripherals)
Low Level OS Interface
Low Level OS
Memory Management Function & Security Functions & System Functions
Hardware Functions Conditionally Accessible
Hardware Functions Accessible Only from CPU in Privileged Mode

5.2 Access Groups

The following Access Groups are defined within the TRE:

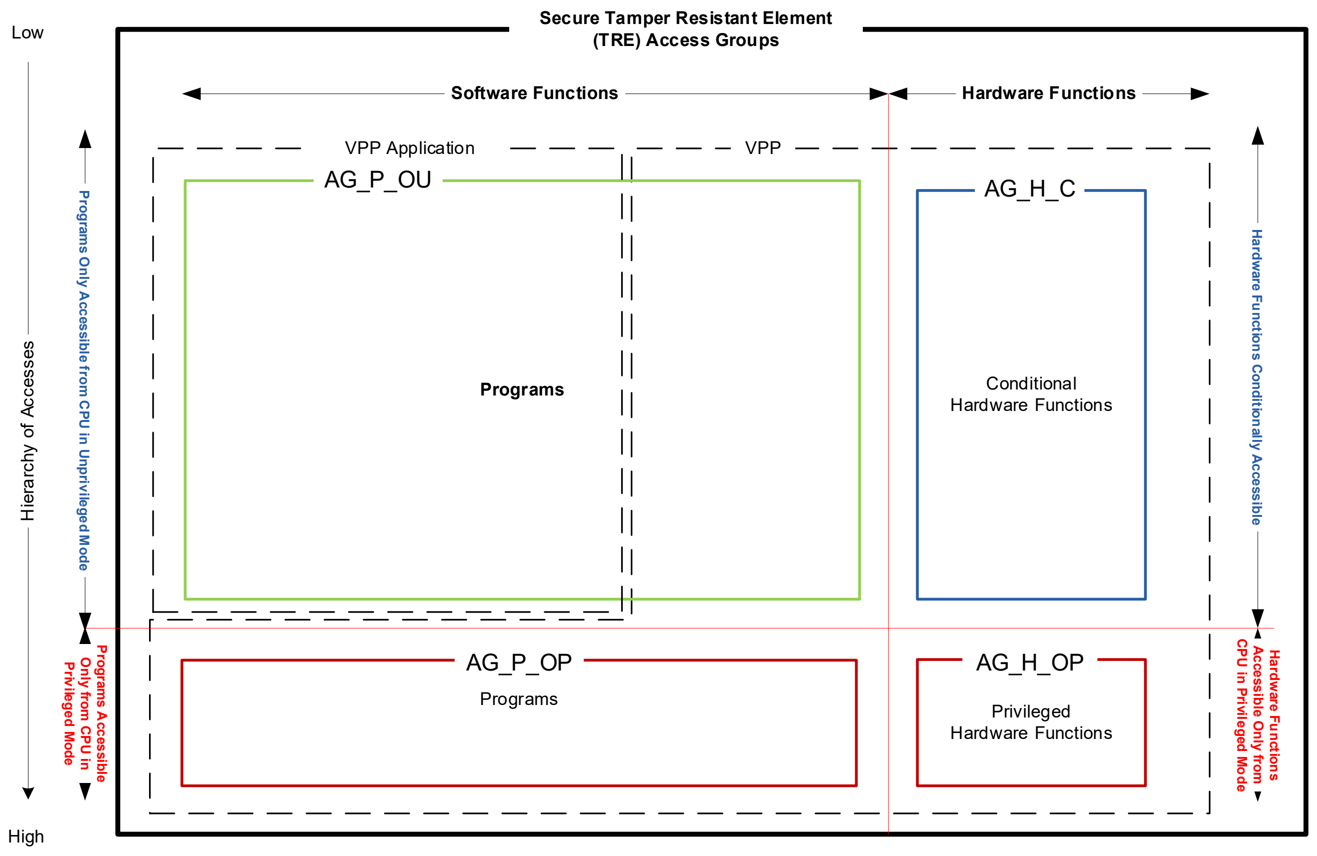
- AG_P_OU: Any Program and data only accessible from a CPU running in unprivileged mode.
- AG_H_C: Any hardware function conditionally accessible from the CPU (see sections 5.8.5.5 and 5.11).
- AG_P_OP: Any Program and data only accessible from a CPU running in privileged mode.
- AG_H_OP: Any hardware function only accessible from the CPU running in privileged mode.

In this context, “accessible” means that access to the memory by a CPU does not generate a memory access violation.

The Primary Platform (e.g. LLOS or CPU in running mode) maintains the confidentiality and the integrity of information within AG_P_OU when the CPU is executing in privileged mode (e.g. via CPU registers, VRE, MMF).

Access Groups can be implemented by any hardware/software combination, as long as they fulfil Global Requirements GR1 and GR2, defined in section 5.8.2.

Figure 5-2: TRE Access Groups


Secure Tamper Resistant Element (TRE) Access Groups
Low
High
Hierarchy of Accesses
Software Functions
Hardware Functions
VPP Application
VPP
AG_P_OU
Programs
AG_H_C
Conditional Hardware Functions
AG_P_OP
Programs
AG_H_OP
Privileged Hardware Functions
Programs Only Accessible from CPU in Unprivileged Mode
Programs Accessible Only from CPU in Privileged Mode
Hardware Functions Conditionally Accessible
Hardware Functions Accessible Only from CPU in Privileged Mode

The Programs are grouped in three different types of functions:

- The LLOS managing security-related hardware functions and native multiprocessing capabilities
- The HLOS (acting as a secondary platform) and its accompanying applications

- The VPP Services managing the hardware functions related to communication (defined in section 5.9) and VPP Firmware management (defined in section 5.10)

The hardware functions are grouped in two modules:

1. The Privileged Hardware Functions (AG_H_OP), which include:
   - System Functions
   - Security Functions
   - Memory Management Function
2. The Conditional Hardware Functions (AG_H_C), which include:
   - Hardware Service Functions
   - Cryptographic Functions
   - Remote Audit Function
   - Long-term credentials storage

The Primary Platform consists of the Hardware Platform, the VPP Services within the VPP Execution Domain, and the LLOS.

The Abstraction and Virtualization of the Primary Platform is provided by the Virtual Primary Platform (VPP), which is made of three parts:

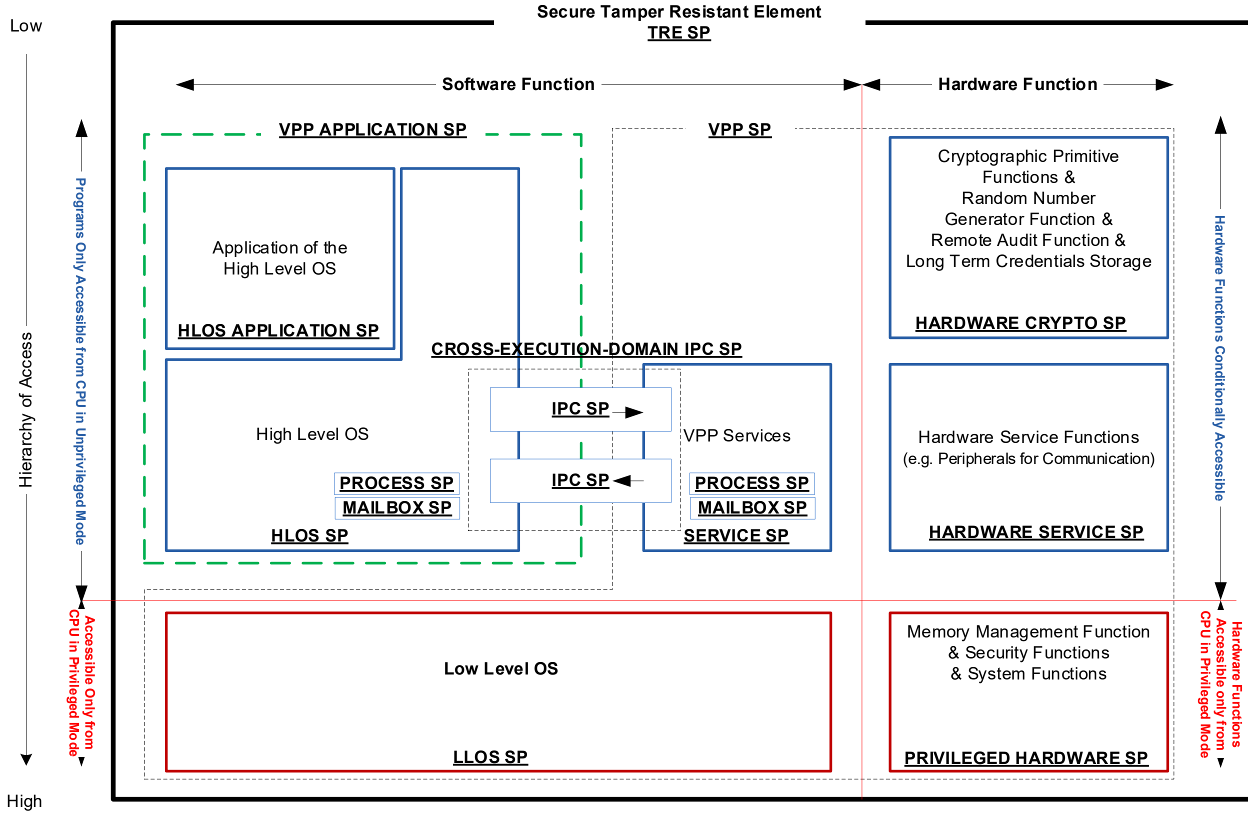
1. LLOS interface (described in section 5.8.5) and the MMF allowing the Virtualization of the physical memory (described in section 5.5)
2. The service interfaces related to communication (described in section 5.9)
3. The service interfaces related to VPP Firmware management (described in section 5.10)

5.3 Security Perimeters

The Security Perimeter (SP) defines the perimeter of a function within which rules, Access Groups, properties, and requirements shall apply.

Figure 5-3 illustrates the SPs of the TRE.

Figure 5-3: TRE Security Perimeters


Low
Secure Tamper Resistant Element
TRE SP
Software Function
Hardware Function
VPP APPLICATION SP
VPP SP
Programs Only Accessible from CPU in Unprivileged Mode
Hierarchy of Access
Application of the High Level OS
HLOS APPLICATION SP
Cryptographic Primitive Functions & Random Number Generator Function & Remote Audit Function & Long Term Credentials Storage
HARDWARE CRYPTO SP
Hardware Functions Conditionally Accessible
CROSS-EXECUTION-DOMAIN IPC SP
IPC SP
High Level OS
VPP Services
Hardware Service Functions (e.g. Peripherals for Communication)
PROCESS SP
MAILBOX SP
PROCESS SP
MAILBOX SP
IPC SP
HLOS SP
SERVICE SP
HARDWARE SERVICE SP
Accessible Only from CPU in Privileged Mode
Low Level OS
LLOS SP
Memory Management Function & Security Functions & System Functions
PRIVILEGED HARDWARE SP
Hardware Functons Accessible only from CPU in Privileged Mode
High

Functional description of the Security Perimeters:

- **VPP APPLICATION SP** in section 6
- **HLOS APPLICATION SP** in section 6.4.2
- **HLOS SP** in section 6.4
- **VPP SP** in section 5
- **CROSS-EXECUTION-DOMAIN IPC SP** in sections 7.4 and 7.5
- **SERVICE SP** in sections 5.9 and 5.10
- **HARDWARE SERVICE SP** in section 3.8
- **HARDWARE CRYPTO SP** in section 3.2.7
- **LLOS SP** in section 5.8
- **PRIVILEGED HARDWARE SP** in sections 3.3, 3.4, 3.5, 3.2.6, and 3.2.1
- **PROCESS SP** in sections 5.4, 5.5, and 5.6
- **MAILBOX SP** in section 5.8.5.3

- **IPC SP** in section 5.8.5.4

The following requirements shall be fulfilled:

- **TRE SP** shall contain a single VPP APPLICATION SP at any given time [SREQ**51**]. The processing of TRE data shall be performed inside the TRE SP [SREQ**52**].
- The storage of TRE data outside the TRE SP shall be protected for confidentiality, integrity, and authenticity by means solely located within the PRIVILEGED HARDWARE SP. [SREQ**53**]
- The storage of TRE data outside the TRE SP shall be resistant against replay attacks and software side channel attacks by means solely located within the PRIVILEGED HARDWARE SP. [SREQ**54]**
- The TRE code/data stored in the Remote Memory TRE SP shall be bound to the TRE. [SREQ**55**]
- **VPP APPLICATION SP** shall contain at least an HLOS APPLICATION SP and an HLOS SP and shall be in Access Group AG_P_OU. [SREQ**56**]
- **HLOS APPLICATION SP** shall be in Access Group AG_P_OU. [SREQ**57**]
- **HLOS SP** shall contain at least a PROCESS SP. [SREQ**58**]
- **HLOS SP** may contain multiple IPC SP. [SREQ**59**]
- **VPP SP** shall contain an LLOS SP, a CROSS-EXECUTION-DOMAIN IPC SP, a PRIVILEGED HARDWARE SP, multiple IPC SP, a SERVICE SP, a HARDWARE CRYPTO SP, and a HARDWARE SERVICE SP. [SREQ**60**]
- **CROSS-EXECUTION-DOMAIN IPC SP** shall be able to transfer data between the SERVICE SP and the HLOS SP. It shall be in Access Group AG_P_OU. [SREQ**61**]
- **SERVICE SP** shall contain at least two PROCESS SPs called MGT SP and COM SP and shall be in Access Group AG_P_OU. [SREQ**62**]
- **SERVICE SP** may be able to transfer data to and from the HARDWARE SERVICE SP. [SREQ**63**]
- **HARDWARE SERVICE SP** shall be in Access Group AG_H_C. [SREQ**64**]
- **HARDWARE CRYPTO SP** shall be in Access Group AG_H_C. [SREQ**65**]
- **LLOS SP** shall be able to transfer data and credentials to and from the HARDWARE SYSTEM SP and shall be in Access Group AG_P_OP. [SREQ**66**]
- **PRIVILEGED HARDWARE SP** shall be in Access Group AG_H_OP. [SREQ**67**]
- **PROCESS SP** shall run a single Process and prevent data transfer to and from any SP except via a declared IPC SP. It shall be in Access Group AG_P_OU. [SREQ**68**]
- **PROCESS SP** may handle multiple MAILBOX SP. [REQ**69**]
- **MAILBOX SP** shall have only a single PROCESS SP as receiver, shall have only a single PROCESS SP as sender, and shall be in either Access Group AG_P_OU or AG_P_OP. [SREQ**70**]
- **IPC SP** shall contain a Shared Memory Space in Access Group AG_P_OU only, shall have only a single PROCESS SP as Accessor, shall have only a single PROCESS SP as Mutator, and shall be in Access Group AG_P_OU. [SREQ**71**]

The following rules shall apply to the TRE SP:

- Data transfer between Security Perimeters in Access Group AG_P_OU and the Security Perimeter in Access Group AG_P_OP shall occur using fixed, pre-determined CPU registers. [SREQ**72**]
- Data transfer from Security Perimeters in Access Group AG_P_OU to the Security Perimeter in Access Group AG_P_OP shall be restricted to Identifiers, Handles, and Signals. [SREQ**73**]

- Data transfer from Security Perimeters in Access Group AG_P_OP to the Security Perimeter in Access Group AG_P_OU shall be restricted to memory address, Handle, Mailbox content, Errors, and Exceptions. [SREQ**74**]

5.4 Unprivileged Execution Model

A Process shall be always executed in Unprivileged CPU mode [SREQ**75**]. Each Process shall have its own Virtual Address Space [SREQ**76**].

Each Process shall run on a Virtual Hardware Platform as defined in section 6.1. [SREQ**77**]

A Process may implement a proprietary multithreading system managing its own Threads without the assistance of the LLOS.[10]

5.5 Unprivileged Virtual Address Spaces

Any Process memory shall be mapped to the two Virtual Address Spaces defined in Figure 5-4 [SREQ**78**]. The gaps between the start locations of two adjacent Virtual Address Space Regions are equal to:

- MK_VSPACE_REGION_SIZE_NVD bytes for the Virtual Address Space starting at the Virtual Address MK_BEGIN_VSPACE_NVD related to the Non-Volatile Data
- MK_VSPACE_REGION_SIZE_VD bytes for the Virtual Address Space starting at the Virtual Address MK_BEGIN_VSPACE_VD related to the Volatile Data

The gap between both Virtual Address Spaces may be null. The values of MK_VSPACE_REGION_SIZE_VD and MK_VSPACE_REGION_SIZE_NVD may be different. The Virtual Address Spaces may have any order but shall be the same for all Processes.

Each Virtual Address Space maps a Memory Partition in the Remote Memory. The representation of a VPP Firmware in the Remote Memory is a collection of Memory Partitions bound by the same VPP Firmware Identifier.

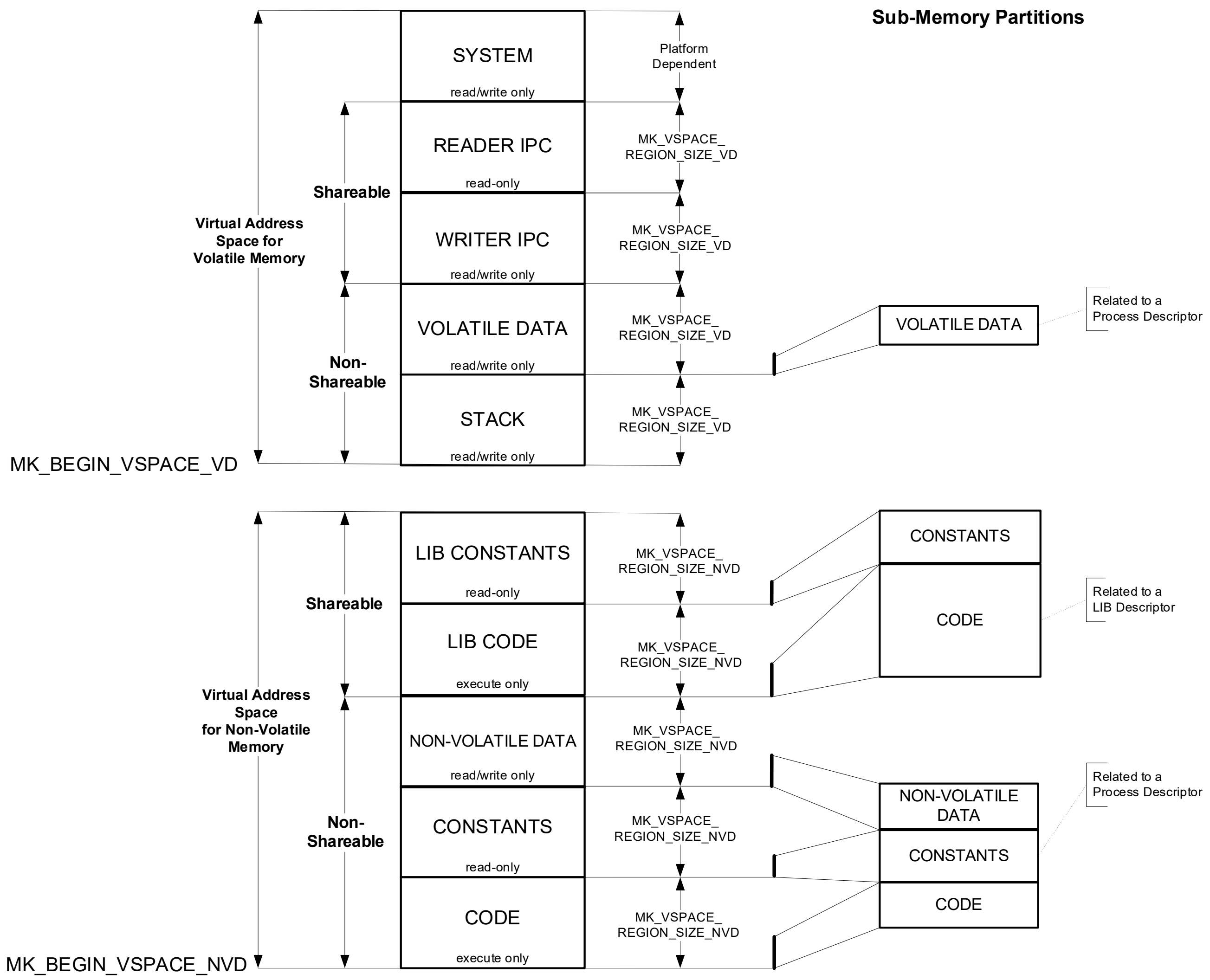
The size of the data within each Virtual Address Space Region is defined in the VPP Firmware, specifically in the VPP Firmware Header, as defined in [VFF].

A memory access performed by a Process outside the boundaries of a sub Virtual Address Space is a Security Perimeter violation.

The stack of a Process may reside in the Physical Address Space and shall have the same access rights and boundaries protection as if it resided in Virtual Address Space. [SREQ**79**]

[10] For example, the stack overflow protection is only available for the stack of the process. By implementing a proprietary multithreading system within a process, the designer of the multithreading system has no hardware assistance for stack overflow detection.

Figure 5-4: Virtual Address Spaces Mapping for Unprivileged CPU Mode



For each Process, the following regions are defined in Virtual Address Space:

- READER IPC: Used by IPC reader Process (read-only memory access)
- WRITER IPC: Used by IPC writer Process (read/write memory access)
- NON-VOLATILE DATA: Persistent storage (read/write memory access)
- VOLATILE DATA: Initialized volatile storage (read/write memory access)
- STACK: Program stack (read/write memory access)
- CONSTANTS: Process constants (read-only memory access)
- CODE: Process code (execute-only memory access)
- SYSTEM: Reserved for Primary Platform use; e.g. address of the Sub-Memory Partition as defined in section 5.8.5.6 (memory access according to use)
- LIB CONSTANTS: Constants for shared library (read-only memory access). Only Embedded TRE may have a reserved space hosting specific Programs provided by the Primary Platform Maker for performing technology dependent NVM management.

- LIB CODE: Code for shared library (execute-only memory access). Only Embedded TRE may have a reserved space hosting specific Programs provided by the Primary Platform Maker for performing technology dependent NVM management.

VPP implementations may locate the following regions in virtual memory addresses not specified by the Virtual Address Space described in Figure 5-4: READER IPC, WRITER IPC, LIB CONSTANTS, LIB CODE, and SYSTEM.

5.6 Run Time Model

The run time model is an isolated context of the Primary Platform (i.e. VPP) and the instance of the VPP Firmware (i.e. VPP Application). It includes:

- The LLOS
- The Processes belonging to the VPP Services and to the VPP Applications assigned to the Execution Domains

Depending on their functions and their Execution Domain (VPP or VPP Application), a Process and the LLOS may have access to some specific hardware functions. [REQ**80**]

Any Process shall communicate with the LLOS only by using the Kernel Functions ABI defined in section 5.8.5. [SREQ**81**]

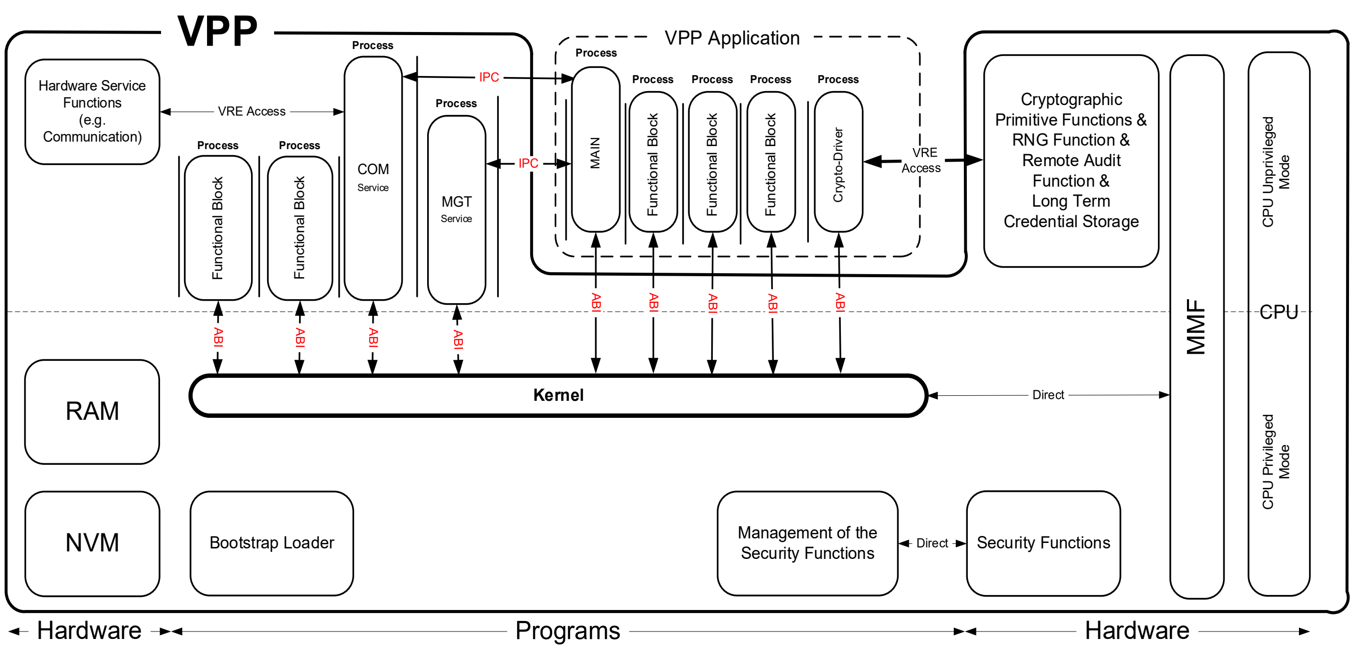
A Process shall communicate with another Process only by using IPC and Signals. [SREQ**82**]

5.6.1 Exception Handling

Exceptions are reserved for severe run-time errors that require termination of the affected Process. If the Exception is not in the MAIN Process, then the MAIN Process has a chance to shut down other Processes and then terminate itself. If the Exception is in the MAIN Process, the MAIN Process and all its descendants are terminated.

In order to recover a terminated VPP Application, it has to be restarted at the next VPP Application Session opening, as defined in section 6.3.

Figure 5-5: Runtime Model



5.7 Provisioning of VPP Firmware

The Primary Platform shall provide an interface to support the VPP Firmware Loader [REQ**83**]. The Interface shall support a VPP Firmware Loader as defined in [OFL] [SREQ**84**].

The VPP Firmware Loader shall be the only VPP Application with special privileges. [SREQ**85**]

The Primary Platform may support provisioning of VPP Firmware of the Primary Platform according to [OFL]. [REQ**86**]

5.8 Low Level Operating System (LLOS)

The Primary Platform embeds an LLOS running in Privileged CPU mode. This LLOS is minimal and contains only the functionality that cannot run in Unprivileged CPU mode. [SREQ**87**]

The LLOS shall include a kernel supporting the management of multiple Processes. [SREQ**88**]

In addition to a kernel, the LLOS shall support [SREQ**89**]:

- The initial Bootstrap Program of the Primary Platform
- The management of all the following hardware functions:
  - Security Functions, as defined in section 3.4
  - Memory Management Functions, as defined in section 3.5
  - Memory Transfer Functions, as defined in section 3.2.6
  - System Functions, as defined in section 3.3

5.8.1 Kernel Objects

The kernel manages multiple internal Kernel Objects:

- Mailbox for receiving Signals
- IPC for communication between two Processes
- Process for running a Program
- Virtual Register (VRE) to enable a Process to directly access a hardware function

In addition to the above Kernel Objects, the kernel manages the following notification mechanism:

- A Signal as a notification without additional data that may be sent from a Process to another Process or that may be sent from the kernel to a Process. A Signal is typically used to indicate that a pre-defined event has occurred. See section 7.5 for a list of such pre-defined Signals.
- An Exception as a notification without additional data that is sent by the kernel to a parent Process when a special condition occurred in one of its child Processes. A severe Exception is thrown by the kernel when a Process has caused a violation that led to its termination by the kernel.

Kernel Objects are addressed by their owner Process using a Kernel Object Identifier. Some identifiers shared between the Execution Domains are pre-defined by this document in section 7.4. All other identifiers are defined by the Firmware Makers during Firmware design stage.

Kernel Object Handles are run-time identifiers for instantiated Kernel Objects. When operating on Kernel Objects, the Kernel Functions ABI requires Kernel Object Handles. A Process shall retrieve Kernel Object Handles from the kernel by using Kernel Object Identifiers [SREQ**90**]. A Process shall use Kernel Object Handles when interacting with the kernel in order to use Kernel Objects [SREQ**91**]. A Kernel Object Handle shall be valid only within the context of its owner Process [SREQ**92**]. The kernel shall restrict Kernel Object Handle use to the Process owning the object [SREQ**93**].

5.8.2 Global Requirements and Mandatory Access Control Rules

Table 5-1 defines the requirements for any Primary Platforms.

Table 5-1: Global Requirements

Rule	SR	Description
GR1	●	The Primary Platform shall provide a mechanism to guarantee the confidentiality of any data in Non-Shareable Memory Spaces.
GR2	●	The Primary Platform shall provide a mechanism to guarantee the integrity of any data in Non-Shareable Memory Spaces.
GR3	●	The Primary Platform shall provide a mechanism to ensure that access to a hardware function, including its input and output data, is exclusive and confidential to each Accessor or Mutator.
GR4	●	The Primary Platform shall provide a mechanism to restrict access to hardware functions only to authorized Accessors or Mutators.
GR5	●	The LLOS shall have only Non-Shareable Memory Space.
GR6	●	The requirements above shall be enforced by the LLOS.
GR7	●	The kernel shall be able to manage memory assigned to any Process.
GR8	●	The kernel shall communicate with a Process only using Signals and the Kernel Functions ABI using scalars as parameters (via registers).
GR9	●	The MGT Process is the parent of the MAIN Process.
GR10	●	VRE (Virtual Register) access shall be exclusive between the access and the release of the VRE.
GR11	●	A Process accessing the VRE should clean up the hardware function related to the VRE before releasing it.
GR12		For VPP Application Processes, collaborative scheduling shall be supported.
GR13		For VPP Application Processes, pre-emptive scheduling should be supported.
GR14		For each VPP Application Process, scheduling type shall be a declared option in its VPP Firmware.
GR15		The Primary Platform shall accept a VPP Firmware only if the Primary Platform capabilities meet the VPP Firmware requirements, as described in its header.
GR16		For VPP Processes, scheduling shall be pre-emptive.
GR17	●	A VPP Process in ‘Waiting’ state may pre-empt a VPP Application Process in ‘Running’ state.
GR18	●	If collaborative scheduling is required, a pre-empted VPP Application Process shall be the next Process to execute.
GR19	●	VPP Processes shall have higher priority than VPP Application Processes.
GR20	●	A Process shall be instantiated in the ‘Suspended R’ state.
GR21	●	A Process shall be able to suspend itself or any Process in its sub-hierarchy.
GR22	●	A Process shall be able to resume any Process in its sub-hierarchy.

Rule	SR	Description
GR23	●	When a Process dies, all its resources as well as resources owned by its sub-hierarchy Processes shall be released for future use by other Process. Confidentiality of the resources of the dead Process(es) must be maintained.
GR24	●	The MGT Process shall limit its control over the MAIN Process to its suspension and resumption.
GR25	●	All Kernel Objects (e.g. Mailboxes, IPC, VRE) belonging to or used by a Process shall be instantiated before the Process is instantiated.
GR26	●	Accessing a non-existent Kernel Object shall throw a severe Exception[11] to the parent Process.
GR27	●	Any severe error[12] (e.g. kernel rules infringement, memory firewall model violation) in a Process shall throw an Exception to the parent Process.
GR28	●	Any severe error in a Process in the ‘Running’ state shall set the Process to the ‘Dead’ state.
GR29	●	Exceptions shall be cleared by the kernel only after having been read by the parent Process.
GR30	●	The current VPP Application shall be terminated prior to starting the next VPP Application.
GR31	●	All resources (Processes, Mailboxes, IPC, VRE) related to the VPP Application shall be allocated during the VPP Firmware instantiation.
GR32	●	The Handle of a Kernel Object provided to a Process shall be valid/used only for that Process.
GR33	●	The kernel shall throw a severe Exception to the parent Process of a Process that violates its Security Perimeter.
GR34	●	The kernel shall reject unknown or undefined kernel calls by throwing a severe Exception.
GR35	●	Any severe Exception in the MGT Process shall reset the TRE.
GR36		VPP shall provide VPP Applications a monotonic and rising tick counter during the VPP Application Session.

[11] MK_EXCEPTION_SEVERE as defined in section 7.3.

[12] MK_ERROR_SEVERE as defined in Table 7-13.

Table 5-2 defines the access rules to resources, granted to Processes.

Table 5-2: Mandatory Access Control Rules

Rule	SR	Description
AC1		Access to a Service, an LLOS interface, or a Hardware Function shall be denied unless explicitly allowed.
AC2		MAC rules shall be conjunctive.
AC3		The MAC rules described in this document shall be the most permissive. Primary Platform Makers and Firmware Makers may reduce the required access to kernel calls and resources, permitting only resources and kernel calls needed by a VPP Application or available in the Primary Platform.
AC4	●	A (Writer) Process shall define one or more IPCs for which that Process shall have read/write access. These IPCs shall be owned exclusively by that Process.
AC5	●	A (Reader) Process shall define one or more IPCs for which it shall have read-only access. These IPCs shall not be owned by that Process.
AC6	●	An IPC shall accept only a single Writer Process and a single Reader Process.
AC7	●	A (Receiver) Process shall declare one or more Mailboxes from which that Process shall be able to receive Signals. These Mailboxes shall be owned exclusively by that Process.
AC8	●	A (Sender) Process shall declare one or more Mailboxes to which that Process may send Signals. These Mailboxes shall not be owned by that Process.
AC9	●	A Mailbox shall accept Signals only from a single source, either a Process or the kernel.
AC10	●	Every Process shall have a specific Mailbox to which only the kernel may send Signals. This Mailbox shall be owned by the Process, and the Process shall be able to receive Signals.
AC11	●	Only the owner of a Mailbox shall be able to read its content

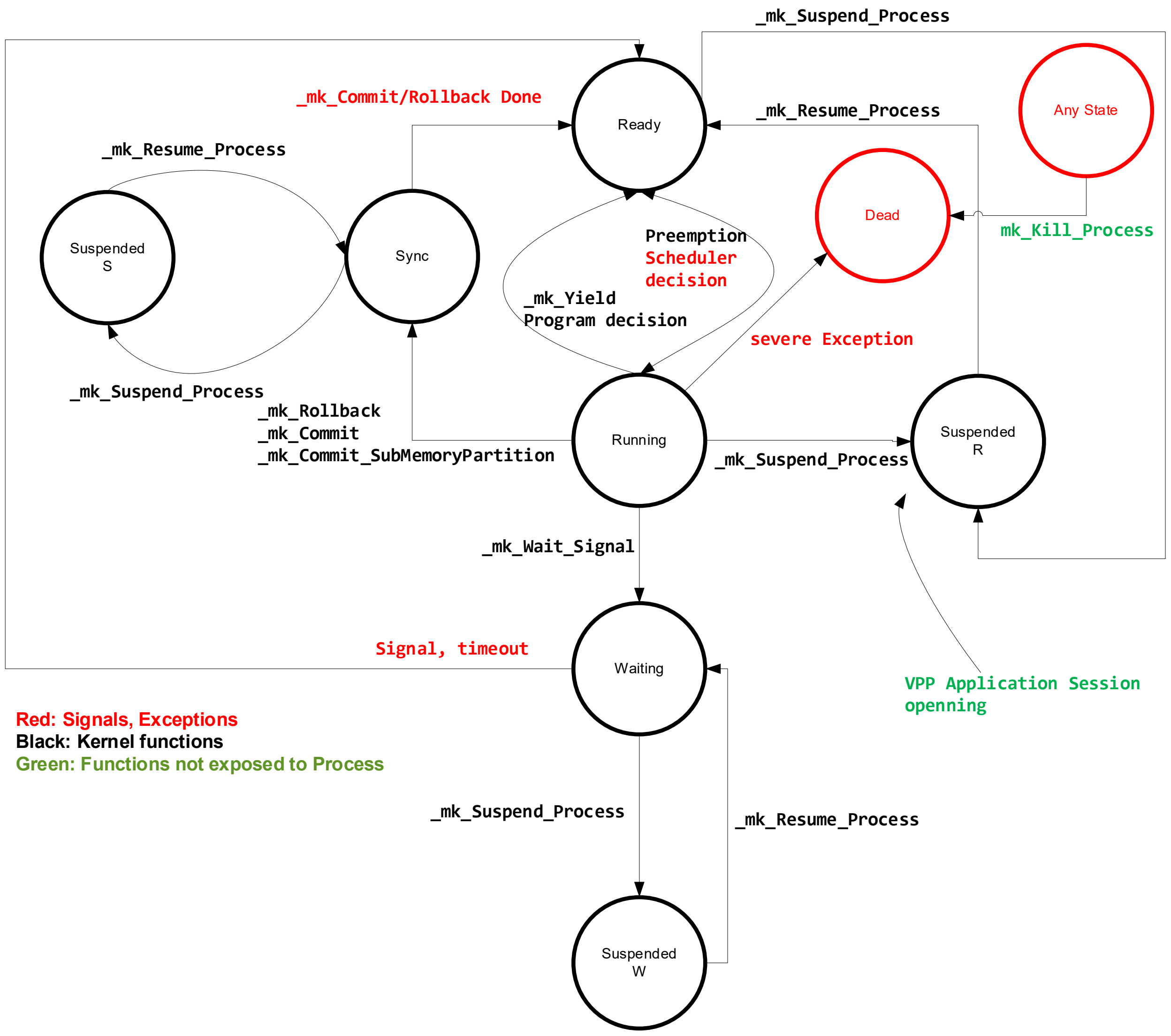
Note: All Mailboxes and IPCs shall be defined during VPP Application development and in the resulting VPP Firmware.

5.8.3 Process States Diagram

Figure 5-6 illustrates the state diagram of a Process. An event (IN or OUT) allows a Process to change states. Events may be:

- Call of a Kernel ABI function (e.g. _mk_Yield)
- A Signal from a Process via the calling of _mk_Send_Signal function
- A Signal from the kernel related to a VRE
- An Exception
- A timeout
- A pre-emption by the kernel scheduling function

Figure 5-6: Process State Diagram



This state diagram is applicable to both VPP and VPP Application Processes. Only one Process may be in the 'Running' state at any given time.

5.8.4 Definition of the Process States

Processes in 'Suspended' states (R/S/W), may receive Signals, but will handle the Signals once the Process is resumed. [REQ**94**]

When a Process transition between different states, the scheduler shall place the Process as the last position is the queue assigned to the new state with the exception of GR17. [REQ**95**]

The scheduler shall select Processes for operations based first on Process priority and then in order in the Process queue. [REQ**96**]

Table 5-3 defines in detail the states of a Process.

Table 5-3: Definition of States

State Name	Description	Event in	Event out
Ready	The Process is eligible for running.	_mk_Resume_Process, Signal, time-out, _mk_Yield Completion of _mk_Commit or _mk_Rollback Scheduler decision, e.g. a VPP Process has received a Signal.	Scheduler decision, _mk_Suspend_Process
Running	The Process is running.	Selected for execution by the scheduler	_mk_Yield, _mk_Suspend_Process, _mk_Wait_Signal, Scheduler decision: e.g. a Signal was sent to another Process,[13] _mk_Commit, _mk_Rollback, _mk_Commit _SubMemoryPartition Severe Exception
Sync	The Process is blocked until _mk_Commit or _mk_Rollback is completed.	_mk_Commit, _mk_Rollback _mk_Commit _SubMemoryPartition _mk_Resume_Process	Completion of _mk_Commit or _mk_Rollback, _mk_Suspend_Process

[13] The process pre-emption occurs in one of the following cases:

- The Process of the VPP Execution Domain sends a Signal to another Process of the same Execution Domain which has a higher priority.
- The VPP Application has declared in its Firmware the use of pre-emptive scheduling and the Process in its Execution Domain sends a Signal to another Process of the same Execution Domain which has a higher priority.
- The VPP Application has declared in its Firmware the use of a collaborative scheduling and the process in its Execution Domain sends a Signal to a process of the VPP Execution Domain.

State Name	Description	Event in	Event out
Waiting	The Process is waiting for a Signal or an elapsed timeout.	_mk_Wait_Signal _mk_Resume_Process	Signal (via _mk_Wait_Signal), time-out, _mk_Suspend_Process
Suspended R	The Process remains in the ‘Suspended R’ state until it is resumed by its parent Process.	_mk_Suspend_Process Process instantiation (Swap IN VPP Application)[15]	_mk_Resume_Process
Suspended S	The Process remains in the ‘Suspended S’ state until it is resumed by its parent Process. When resumed, the Process is in the ‘Sync’ state.	_mk_Suspend_Process	_mk_Resume_Process
Suspended W	The Process is in ‘Suspended W’ state until it is resumed by its parent Process. When resumed, the Process is appended to the end of the list of Processes in the ‘Waiting’ state.	_mk_Suspend_Process	_mk_Resume_Process
Dead	The Process is no longer active due to a termination, a severe Exception, a rule violation, or a violation of the memory firewall model.	Process termination[14] Severe Exception	VPP Firmware instantiation[15]

[14] Internal events are expressed for easing the reading of the table but are not exposed in the ABI.

[15] VPP Firmware instantiation: The previous VPP Firmware instance is removed, then all Kernel Objects of the VPP Firmware are instantiated (i.e. Process, Mailboxes, IPC, VRE) and the Memory Partitions containing the VPP Firmware are mounted and selected.

5.8.5 Kernel Functions ABI/API

The Primary Platform Maker shall provide an ABI (Application Binary Interface) related to the implementation of the LLOS in order to map any API (Application Programming Interface) that is dependent on the programming language made available on the HLOS. [REQ**97**]

The Primary Platform Maker shall provide to the VPP Application only the ABI defined in this specification. [SREQ**98**]

The Primary Platform Maker shall provide the C language API using the above ABI and mapping it to the C function prototypes as defined in this section. [REQ**99**]

All data types and constant values are defined in section 7.

5.8.5.1 Generic Functions

5.8.5.1.1 Function _mk_Get_Exception

Brief: Retrieve an Exception.

Description: This function retrieves the last Exception thrown by the kernel. Exceptions are cleared after reading. Only exceptions of child Processes can be retrieved.

Parameter:

• _hProcess	(MK_HANDLE_t)	Handle of the Process

Return:

• bitmap	(MK_BITMAP_t)	MK_EXCEPTION_e bitmap value where each bit represents a unique Exception, as defined in Table 7-12

If bit MK_EXCEPTION_HANDLE_NOT_A_PROCESS is set in the return value MK_EXCEPTION_e bitmap, this indicates that the parameter hProcess passed in the calling Process cannot be resolved as an existing Process.

C prototype function:

```
MK_BITMAP_t _mk_Get_Exception(MK_HANDLE_t _hProcess)
```

5.8.5.1.2 Function _mk_Get_Error

Brief: Get the most recent error generated through the execution of a function within a given Process.

Description: This function retrieves an error stored by the kernel. The caller of this function can access the last error that occurred in itself or in any Process it owns. This error can be retrieved regardless of the state of the Process referenced by the hProcess Handle and is not modified by a state transition (see Table 5-3). Power off cycle and other reset will clear the stored error. The _mk_Get_Error function does not modify the last error stored for this Process during its invocation.

Parameter:

• _hProcess	(MK_HANDLE_t)	Handle of the Process

Return:

• error	(MK_ERROR_e)	value of the error (see Table 7-13)

or

- MK_ERROR_GET_ERROR_HANDLE_NOT_A_PROCESS

The error MK_ERROR_GET_ERROR_HANDLE_NOT_A_PROCESS shall be returned only if the hProcess Handle provided as a parameter of the _mk_Get_Error function does not correspond to a valid handle of a Process.

C prototype function:

```
MK_ERROR_e _mk_Get_Error(MK_HANDLE_t _hProcess)
```

5.8.5.1.3 Function _mk_Get_Time

Brief: Get the absolute time (in ticks) since the Primary Platform start up.

Description: The return value is 64 bits in length. It is important to note that whenever the Primary Platform starts or restarts, the timer is reset to zero. The returned value represents elapsed time only during the caller's VPP Application Session. Between different VPP Application Sessions there is no guarantee on elapsed time or even for the value being monotonic and increasing.

Parameter:

• None	No parameters

Return:

• time	(MK_TIME_t)	value of the current time in ticks

C prototype function:

```
MK_TIME_t _mk_Get_Time(void)
```

5.8.5.2 Process Management

5.8.5.2.1 Function _mk_Get_Process_Handle

Brief: Get the Process Handle for itself or for one of its descendants.

Description: This function gets a Process Handle through its Process identifier.

The Process retrieving the Process Handle does not inherit the rights of its owner.

Parameter:

- _eProcess_ID (MK_PROCESS_ID_u) identifier of the Process

Return:

- Handle (MK_HANDLE_t) Handle of the Kernel Object
- NULL On error, _mk_Get_Error function returns one of the following error codes:
    - MK_ERROR_UNKNOWN_ID if the ID does not exist or the Process is in 'Dead' state
    - MK_ERROR_ACCESS_DENIED if the Process is not one of the descendants of the caller Process

C prototype function:

MK_HANDLE_t _mk_Get_Process_Handle(MK_PROCESS_ID_u _eProcess_ID)

5.8.5.2.2 Function _mk_Get_Process_Priority

Brief: Get the Process priority.

Description: This function gets the priority of a Process.

Parameter:

- _hProcess (MK_HANDLE_t) Handle of the Process

Return:

- _xPriority (MK_PROCESS_PRIORITY_e) Priority of the Process or the priority reserved for error: MK_PROCESS_PRIORITY_ERROR
- On error, _mk_Get_Error function returns the following error code:
    - MK_ERROR_UNKNOWN_HANDLE if the Process Handle is unknown/invalid

C prototype function:

MK_PROCESS_PRIORITY_e _mk_Get_Process_Priority(MK_HANDLE_t _hProcess)

5.8.5.2.3 Function _mk_Set_Process_Priority

Brief: Set the Process priority.

Description: This function sets the priority of a Process. A Process may change its priority or the priority of one of its descendants. The priority can be changed anytime, independent from the state.

Parameters:

• _hProcess	(MK_HANDLE_t)	Handle of the Process
• _xPriority	(MK_PROCESS_PRIORITY_e)	Priority of the Process

Return:

• MK_ERROR_NONE	if the function is successful
• MK_ERROR_UNKNOWN_HANDLE	if the Process Handle is unknown/invalid
• MK_ERROR_UNKNOWN_PRIORITY	if the priority value is unknown/invalid

C prototype function:

MK_ERROR_e _mk_Set_Process_Priority(MK_HANDLE_t _hProcess, MK_PROCESS_PRIORITY_e _xPriority)

5.8.5.2.4 Function _mk_Suspend_Process

Brief: Suspend a Process. A Process can suspend itself or any of its descendants.

Description: This function suspends a Process. The suspended Process is no longer scheduled for execution. If a Process suspends itself, then this call will only return upon resumption by the parent Process.

Parameter:

• _hProcess	(MK_HANDLE_t)	Handle of the Process to be suspended

Return:

• MK_ERROR_NONE	if the Process suspended successfully
• MK_ERROR_UNKNOWN_HANDLE	if the Process Handle is unknown/invalid
• MK_ERROR_ACCESS_DENIED	if the Process to be suspended is not the calling Process or any of its descendants

C prototype function:

MK_ERROR_e _mk_Suspend_Process(MK_HANDLE_t _hProcess)

5.8.5.2.5 Function _mk_Resume_Process

Brief: Resume a Process

Description: This function resumes a Process. A resumed Process must be a descendant of the running Process.

Parameter:

• _hProcess	(MK_HANDLE_t)	Handle of the Process

Return:

• MK_ERROR_NONE	if the Process resumes successfully
• MK_ERROR_UNKNOWN_HANDLE	if the Process Handle is unknown/invalid
• MK_ERROR_ACCESS_DENIED	if the Process to be resumed is not a descendant of the calling Process

C prototype function:

MK_ERROR_e _mk_Resume_Process(MK_HANDLE_t _hProcess)

5.8.5.2.6 Function _mk_Request_No_Preemption

Brief: Allows a VPP Application Process to request a period of time during which it should not be pre-empted.

Description: This call requests continuous CPU processing allocation to a given Process for a certain amount of time and should help handling operations that require more exact timing. If the requested time value is 0, pre-emption will resume. Other requests with a time value shorter than MK_APP_STOP_GRACEFUL_TICKS shall be ignored. This function is optional.

Parameter:

• _uTime	(uint32_t) requested time shall not exceed MK_NO_PREEMPTION_MAX_TIMEOUT (number of ticks). If uTime is 0, then pre-emption becomes possible.

Return:

• MK_ERROR_NONE	if the request is accepted
• MK_ERROR_ILLEGAL_PARAMETER	if _uTime exceeds MK_NO_PREEMPTION_MAX_TIMEOUT

C prototype function:

MK_ERROR_e _mk_Request_No_Preemption(uint32_t _uTime)

5.8.5.2.7 Function _mk_Commit

Brief: Commits all the changes in the NVM of the caller Process.

Description: This function commits all changes to the NVM of the caller Process. No rollback is possible after calling this function. The caller Process is suspended until the completion of this operation. This operation is atomic and cannot fail (unless due to an irrecoverable error).

Parameter:

• void	No parameters

Return:

• void	No returned value

C prototype function:

void _mk_Commit(void)

5.8.5.2.8 Function _mk_RollBack

Brief: Rolls back all the changes made to the NVM of the caller Process.

Description: This function rolls back all changes to the NVM of the caller Process, back to the last commit operation. The caller Process is suspended until the completion of this operation. This operation is atomic and cannot fail (unless due to an irrecoverable error).

Parameter:

• void	No parameters

Return:

• void	No returned value

C prototype function:

void _mk_RollBack(void)

5.8.5.2.9 Function _mk_Yield

Brief: Return the control to the kernel scheduler.

Description: Let the caller Process ask the kernel to yield its execution, causing the kernel to switch the caller to ‘Ready’ state. This call will return when the Process is scheduled to run by the scheduler.

Parameter:

• void	No parameters

Return:

• void	No returned value

C prototype function:

void _mk_Yield(void)

5.8.5.3 Mailbox Management

5.8.5.3.1 Function _mk_Get_Mailbox_Handle

Brief: Get a Mailbox Handle from a Mailbox identifier.

Description: This function gets the Mailbox Handle through a Mailbox identifier.

Parameter:

• _eMailboxID	(MK_MAILBOX_ID_u)	identifier of the Mailbox

Return:

- Handle (MK_HANDLE_t) Handle of the Mailbox
- NULL On error, _mk_Get_Error function returns one of the following error codes:
  - MK_ERROR_ACCESS_DENIED if the Process is not allowed to send a Signal to the Mailbox
  - MK_ERROR_UNKNOWN_ID if the Mailbox identifier is unknown/invalid

C prototype function:

MK_HANDLE_t _mk_Get_Mailbox_Handle(MK_MAILBOX_ID_u _eMailboxID)

5.8.5.3.2 Function _mk_Get_Mailbox_ID_Activated

Brief: When waiting for Signal on any Mailbox owned by the caller Process, get the Mailbox identifier of a Process that has a pending Signal.

Description: This function retrieves the identifier of a Mailbox with a pending signal when the Process waits on any Mailbox of the caller Process.

Parameter:

- void No parameters

Return:

- identifier (MK_MAILBOX_ID_u) identifier of the Mailbox
- NULL if no Mailbox has received a Signal

C prototype function:

MK_MAILBOX_ID_u _mk_Get_Mailbox_ID_Activated(void)

5.8.5.3.3 Function _mk_Send_Signal

Brief: Send a Signal to a Mailbox.

Description: This function sends Signals to a Mailbox. The signals sent are represented as a bitmap of Signal values and there is no priority among Signals as to the order of their arrival within the Mailbox.

Parameters:

- _hMailbox (MK_HANDLE_t) Handle of the Mailbox
- _eSignal (MK_BITMAP_t) Signal value

Return:

• MK_ERROR_NONE	if the Signal(s) was/were sent successfully
• MK_ERROR_UNKNOWN_HANDLE	if the Mailbox Handle is unknown/invalid
• MK_ERROR_ACCESS_DENIED	if the caller Process is not defined as the sender Process of the Mailbox

C prototype function:

MK_ERROR_e _mk_Send_Signal(MK_HANDLE_t _hMailbox, MK_BITMAP_t _eSignal)

5.8.5.3.4 Function _mk_Wait_Signal

Brief: Wait for a Signal on a Mailbox.

Description: This function waits for a Signal on one or any Mailboxes of the caller Process, either for a given time or without a time limit. This call is blocking and will return when a signal is received or when the timeout occurs.

When a Process waits on any Mailbox, the Signals MK_SIGNAL_TIME_OUT, MK_SIGNAL_ERROR, and MK_SIGNAL_EXCEPTION are sent only to its kernel Mailbox.

When a Process waits on a Mailbox, the Signals MK_SIGNAL_TIME_OUT, MK_SIGNAL_ERROR, and MK_SIGNAL_EXCEPTION are sent to that Mailbox.

Only the owner of the Mailbox can wait on it.

Parameters:

- _hMailbox (MK_HANDLE_t) Handle of the Mailbox. If the Handle is NULL, then the Process shall wait for any Signal sent to any of the Mailboxes owned by the Process
- _uTime (uint32_t) timeout time in ticks
    - If the value is 0, then the function will not wait for a Signal and will return control to the caller Process immediately.
    - If the value is MK_ENDLESS, then the function will wait for a Signal forever and control will not be returned until a Signal is received.

Return:

• MK_ERROR_NONE	if a Signal is available or a timeout occurs
• MK_ERROR_UNKNOWN_HANDLE	if the Mailbox Handle is unknown/invalid

C prototype function:

MK_ERROR_e _mk_Wait_Signal(MK_HANDLE_t _hMailbox, uint32_t _uTime)

5.8.5.3.5 Function _mk_Get_Signal

Brief: Get a Signal from a Mailbox.

Description: This function gets a Signal on a Mailbox. A Process can only retrieve the Signal from its own Mailbox. The pending Signals are cleared once they have been read.

Parameter:

- _hMailbox (MK_HANDLE_t) Handle of the Mailbox

Return:

- bitmap (MK_BITMAP_t) a bitmap where each bit represents a Signal (MK_SIGNAL_e)
- NULL On error, _mk_Get_Error function returns the following error code:
    - MK_SIGNAL_ERROR if the Mailbox Handle is unknown/invalid, or access is denied.

C prototype function:

MK_BITMAP_t _mk_Get_Signal(MK_HANDLE_t _hMailbox)

5.8.5.4 IPC Management

The content of the IPC owned by a VPP Application is persistent while the VPP Application is instantiated.

A Process may open at least MK_MIN_CONCURRENT_IPC_LIMIT IPCs.

A VPP Application may open a maximum number of MK_IPC_LIMIT IPCs.

5.8.5.4.1 Function _mk_Get_IPC_Handle

Brief: Get the Handle of an IPC.

Description: This function gets an IPC Handle for communication between two Processes.

The size, the ownership, and the granted access of the IPC are defined in the IPC Descriptor, which is part of the VPP Firmware Header.

The owner Process (i.e. writer) of the IPC has read/write access.

The granted access Process (i.e. reader) has read-only access.

Parameter:

- _eIPC_ID (MK_IPC_ID_u) identifier of the IPC

Return:

- Handle (MK_HANDLE_t) IPC Handle
- NULL On error, _mk_Get_Error function returns the following error code:
    - MK_ERROR_ACCESS_DENIED if the Process is not allowed to use the IPC identifier
    - MK_ERROR_UNKNOWN_ID if the IPC identifier is unknown/invalid.

C prototype function:

MK_HANDLE_t _mk_Get_IPC_Handle(MK_IPC_ID_u _eIPC_ID)

5.8.5.4.2 Function _mk_Get_Access_IPC

Brief: Get access to a Shared Memory Space used by an IPC.

Description: This function returns the virtual memory address of the IPC (virtual shared memory).

The number of Processes that can concurrently access the IPC is guaranteed to be at least MK_MIN_CONCURRENT_IPC_LIMIT.

A Process can only access the IPC virtual memory address if it is the owner or the granted Process, as described in the IPC Descriptor.

Parameter:

• _hIPC	(MK_HANDLE_t)	IPC Handle

Return:

- Virtual memory address of the IPC
- NULL On error, _mk_Get_Error function returns one of the following error codes:

• MK_ERROR_UNKNOWN_HANDLE	if the IPC Handle is unknown/invalid
• MK_ERROR_ACCESS_DENIED	if the Process is not allowed to access the IPC
• MK_ERROR_IPC_LIMIT_REACHED	if IPC has reached concurrency limit

C prototype function:

void* _mk_Get_Access_IPC(MK_HANDLE_t _hIPC)

5.8.5.4.3 Function _mk_Release_Access_IPC

Brief: Release access to the IPC.

Description: This function allows releasing the access to the IPC. The Process can no longer access the virtual shared memory.

Parameter:

• _hIPC	(MK_HANDLE_t)	IPC Handle

Return:

• MK_ERROR_NONE	if the IPC releases successfully
• MK_ERROR_UNKNOWN_HANDLE	if the IPC Handle has not been acquired through _mk_Get_IPC_Handle
• MK_ERROR_HANDLE_NOT_ACCESSED	if the IPC Handle has not been accessed through _mk_Get_Access_IPC
• MK_ERROR_ACCESS_DENIED	if the Process is not allowed to access the IPC

C prototype function:

MK_ERROR_e _mk_Release_Access_IPC(MK_HANDLE_t _hIPC)

5.8.5.5 VRE Management

5.8.5.5.1 Function _mk_Get_VRE_Handle

Brief: Get the Handle of a virtual register.

Description: This function gets the VRE Handle for communication with a hardware function. A VRE is used for direct access to a hardware function.

The access policy to VREs is defined in the Process Descriptor.

Parameter:

- _eVRE_ID (MK_VRE_ID_e) identifier of the VRE

Return:

- Handle (MK_HANDLE_t) VRE Handle
- NULL On error, _mk_Get_Error function returns one of the following error codes:
  - MK_ERROR_UNKNOWN_ID if the VRE ID is unknown/invalid
  - MK_ERROR_ACCESS_DENIED if the VRE violates the MAC as defined in section 5.11

C prototype function:

MK_HANDLE_t _mk_Get_VRE_Handle(MK_VRE_ID_e _eVRE_ID)

5.8.5.5.2 Function _mk_Get_Access_VRE

Brief: Allows the caller Process to get the address of a VRE.

Description: This function gets the virtual registers base address of the hardware function (VRE) to be accessed. The operation of said hardware function is Primary Platform specific, and dependent on the hardware used and exposed by the Primary Platform Maker.

The number of different VREs that can be accessed simultaneously by a VPP Application is limited to MK_VRE_LIMIT. If that limit is exceeded, then access to new VREs shall be denied.

Parameter:

- _hVRE (MK_HANDLE_t) VRE Handle

Return:

- Virtual registers base address of the hardware function to access
- NULL On error, _mk_Get_Error function returns one of the following error codes:
  - MK_ERROR_UNKNOWN_HANDLE if the VRE Handle is unknown/invalid
  - MK_ERROR_ACCESS_DENIED if the number of VREs in use by the VPP Application has exceeded the MK_VRE_LIMIT or if the VRE is accessed from another Process

C prototype function:

void* _mk_Get_Access_VRE(MK_HANDLE_t _hVRE)

5.8.5.5.3 Function _mk_Release_Access_VRE

Brief: Release access to a VRE.

Description: This function releases the access to a VRE.

The Process is responsible for cleaning up the content of the hardware function before releasing the VRE, according to the specifications of the specific hardware function.

Parameter:

• _hVRE	(MK_HANDLE_t)	VRE Handle

Return:

• MK_ERROR_NONE	if the VRE is released successfully
• MK_ERROR_UNKNOWN_HANDLE	if the VRE Handle has not been acquired through _mk_Get_VRE_Handle
• MK_ERROR_HANDLE_NOT_ACCESSED	if the VRE Handle has not been accessed through _mk_Get_Access_VRE

C prototype function:

MK_ERROR_e _mk_Release_Access_VRE(MK_HANDLE_t _hVRE)

5.8.5.5.4 Function _mk_Attach_VRE

Brief: Attach a VRE to the kernel Mailbox of a Process.

Description: Set the callback Signals to send to the caller Process kernel Mailbox when the status of a hardware function related to a VRE changes.

This function is valid only if the Process declares the VRE access within the Process Descriptor and if the use of the VRE is allowed by the Mandatory Access Control.

Parameter:

• _hVRE	(MK_HANDLE_t) VRE Handle
• _uSignal	(MK_BITMAP_t) value of the Signal(s) to send to the kernel Mailbox when the status of the VRE changes. The Signal(s) shall be in the range of MK_SIGNAL_DOMAIN_BASE_0 to MK_SIGNAL_DOMAIN_BASE_28 (inclusive).

Return:

• MK_ERROR_NONE	if the VRE attachment is accepted
• MK_ERROR_UNKNOWN_HANDLE	if the VRE Handle is unknown/invalid or the Process Descriptor does not define a VRE access
• MK_ERROR_HANDLE_NOT_ACCESSED	if the VRE Handle has not been accessed through _mk_Get_Access_VRE
• MK_ERROR_ILLEGAL_PARAMETER	if the provided bitmap is not in the allowed range of Signals

C prototype function:

MK_ERROR_e _mk_Attach_VRE(MK_HANDLE_t _hVRE, MK_BITMAP_t _uSignal)

5.8.5.6 VPP Firmware Management

The following functions apply to the VPP Firmware Management Service Interface discussed in section 5.10.

5.8.5.6.1 Function _mk_Open_SubMemoryPartition

Brief: During VPP Firmware Impersonation only: Inform the kernel that the Sub-Memory Partition is opened for writing.

Description: This function can only be used by the VPP Firmware Loader.

This function works in conjunction with _mk_Assign_SubMemoryPartition, which must be called in order to enable writing to an individual Sub-Memory Partition as defined in [VFF]. When the Sub-Memory Partition has successfully been opened, this function needs to be followed with _mk_Close_SubMemoryPartition.

_mk_Open_SubMemoryPartition allows writing in the Sub-Memory Partition of the VPP Firmware to load or to update.

Parameter:

• _uFirmwareID	(UUID_t)	Identifier of the VPP Firmware to load or update (i.e. m_xFirmwareID in [VFF])

Return:

• MK_ERROR_NONE	if the Sub-Memory Partition is successfully opened
• MK_ERROR_UNKNOWN_UUID	if the given _uFirmwareID is unknown
• MK_ERROR_INTERNAL	if an internal error occurred; e.g. the Sub-Memory Partition was already opened
• MK_ERROR_ACCESS_DENIED	if the VPP application does not have the access rights

C prototype function:

```
MK_ERROR_e _mk_Open_SubMemoryPartition(UUID_t _uFirmwareID)
```

5.8.5.6.2 Function _mk_Close_SubMemoryPartition

Brief: During VPP Firmware Impersonation only: Inform the kernel that the Sub-Memory Partition of a VPP Firmware has been closed.

Description: This function can only be used by the VPP Firmware Loader.

Parameter:

• void	No parameters

Return:

• MK_ERROR_NONE	if the Sub-Memory Partition is successfully closed
• MK_ERROR_INTERNAL	if the Sub-Memory Partition was not open

C prototype function:

```
MK_ERROR_e _mk_Close_SubMemoryPartition(void)
```

5.8.5.6.3 Function _mk_Assign_SubMemoryPartition

Brief: During VPP Firmware Impersonation only: Inform the kernel that the caller wishes to open, in writing mode only, a Sub-Memory Partition (e.g. CODE, CONSTANTS, …) that belongs to the VPP Firmware being loaded or updated.

Description: This function allows the VPP Firmware Loader to request from the kernel the virtual memory address of a Sub-Memory Partition allocated for the VPP Firmware being loaded or updated, so that the code, the constants, and the Non-volatile Data areas of a Sub-Memory Partition may be written.

This function can only be used by the VPP Firmware Loader.

Parameter:

sub_Memory_Partition_Index	(MK_Index_t)	Index of the Sub-Memory Partition as follows:

0: Sub-Memory Partition for the CODE region

1: Sub-Memory Partition for the CONSTANTS region

2: Sub-Memory Partition for the Non-Volatile Data region

3: Sub-Memory Partition for the LIBRARY CODE region

4: Sub-Memory Partition for the LIBRARY CONSTANTS region

During VPP Firmware Impersonation of the LLOS, only indexes from 0 to 2 are valid.

Return:

- The virtual memory address of the beginning of the Sub-Memory Partition, as defined in section 5.5 and marked as MK_BEGIN_VSPACE_NVD.
- NULL On error, _mk_Get_Error function returns one of the following error codes:

• MK_ERROR_UNKNOWN_ID	if the Process identifier is unknown/invalid within the VPP Firmware to be loaded or updated
• MK_ERROR_ILLEGAL_PARAMETER	if the Sub-Memory Partition index is invalid
• MK_ERROR_ACCESS_DENIED	if no Sub-Memory Partition allocation has been performed prior to calling _mk_Assign_SubMemoryPartition
• MK_ERROR_INTERNAL	if the _mk_Assign_SubMemoryPartition is called more than once before _mk_Close_SubMemoryPartition is called

C prototype function:

```
void* _mk_Assign_SubMemoryPartition(MK_Index_t)
```

5.8.5.6.4 Function _mk_Commit_SubMemoryPartition

Brief: During VPP Firmware Impersonation only: Commit in the NVM all changes in Sub-Memory Partition.

Description: This function allows the VPP Firmware Loader to commit in the NVM the changes made in Sub-Memory Partition related to a Process. The VPP Firmware Loader may, as well, commit in the NVM the changes in Sub-Memory Partition related to a shared library or the LLOS, if the Primary Platform supports this capability. No rollback is possible after calling this function. The caller Process is suspended until the completion of this operation. This operation is atomic and cannot fail (unless due to an irrecoverable error). _mk_Commit_SubMemoryPartition may be called several times before _mk_Close_SubMemoryPartition is called.

This function can only be used by the VPP Firmware Loader.

Parameter:

• void	No parameters

Return:

• MK_ERROR_NONE	if the commit is successful
• MK_ERROR_INTERNAL	if _mk_Assign_SubMemoryPartition has not been previously called and successfully executed

C prototype function:

void _mk_Commit_SubMemoryPartition(void)

5.8.5.6.5 Function _mk_getAvailableMemoryPartition

Brief: Returns the available memory for a new Memory Partition either for Volatile or non-Volatile Memory. This function is optional.

Description: This function returns the maximum size in bytes for a new Memory Partition.

Parameter:

• memoryType_t	memoryType

Return:

- size (uint32_t) of the Memory Partition available size (in Byte)
- NULL On error, _mk_Get_Error function returns one of the following error codes:

• MK_ERROR_ILLEGAL_PARAMETER	if the memory type is invalid
• MK_ERROR_ACCESS_DENIED	if the VPP Application does not have the access rights

C prototype function:

uint32_t _mk_getAvailableMemoryPartition(memoryType_t memoryType)

5.9 Communication Service Interface

The Primary Platform Maker shall provide to the VPP Application developer all functionality required by the Communication Service Interface, as defined in this section. [SREQ**100**]

The Communication Service manages two data chunk FIFO queues as defined in section 6.1, between the VPP (COM Process) and a VPP Application (MAIN Process):

- FIFO OUT as a FIFO queue allows transferring data in sequence from a source Process (MAIN or COM) to a destination Process (COM or MAIN) via arrays of m_Size_OUT data chunks, each being m_MTU_OUT bytes long.
- FIFO IN as a FIFO queue allows transferring data in sequence from a source Process (COM or MAIN) to a destination Process (MAIN or COM) via arrays of m_Size_IN data chunks, each being m_MTU_IN bytes long.

The data chunk shall contain a packet as defined in [VNP]. [REQ**101**]

The transfer of data is based on two IPCs identified as:

- MK_IPC_COM_MAIN_ID for the data transfer from the COM Process to the MAIN Process.
- MK_IPC_MAIN_COM_ID for the data transfer from the MAIN Process to the COM Process.

Both Processes obtain access to the IPC by retrieving the virtual memory addresses using the kernel function _mk_Get_Access_IPC.

For clarity, the virtual memory addresses used by the IPC between VPP Application MAIN Process and COM Process are:

- pIPC_COM_MAIN – The COM (Writer) ➔ MAIN (Reader) IPC.
- pIPC_MAIN_COM – The MAIN (Writer) ➔ COM (Reader) IPC.

The virtual shared memory address for transferring data from one Process to another are:

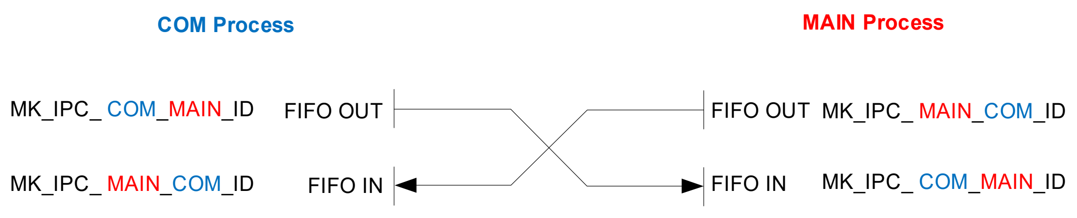
• FIFO OUT COM to MAIN:	pIPC_COM_MAIN from the COM to the MAIN Process
• FIFO OUT MAIN to COM:	pIPC_MAIN_COM from the MAIN to the COM Process

The FIFO OUT of the source Process is the FIFO IN of the destination Process:

• FIFO IN MAIN from COM:	pIPC_COM_MAIN from the COM to the MAIN Process
• FIFO IN COM from MAIN:	pIPC_MAIN_COM from the MAIN to the COM Process

Figure 5-7 illustrates the links between the FIFO IN and FIFO OUT.

Figure 5-7: FIFO IN and OUT links



The destination shall read its FIFO IN when the Signal MK_SIGNAL_IPC_UPDATED is sent by the source Process. [REQ**102**]

5.9.1 FIFO Update Procedure

Brief: Update the FIFO IN and OUT irrespective of the Process (COM or MAIN). The source Process shall send the MK_SIGNAL_IPC_UPDATED Signal to inform the destination Process that its FIFO OUT has been updated.

Description:

The parameters from the source Process MAIN or COM shall be filled with a structure pointed at by pIPC_MAIN_COM or pIPC_COM_MAIN respectively.

The source Process shall send, for each update made to its FIFO OUT, an MK_SIGNAL_IPC_UPDATED Signal to the destination Process Mailbox.

The destination Process shall read its FIFO IN only after an MK_SIGNAL_IPC_UPDATED Signal has been received on the destination Process Mailbox. [REQ**103**]

The command performs the following operations:

- The source Process signals the destination Process, using the MK_SIGNAL_IPC_UPDATED signal, that it wrote a data chunk.
- The location of the written data chunk is in m_Buff_OUT array at the index (m_Write_OUT-1) modulo m_Size_OUT.
- The destination Process signals the source Process, using the MK_SIGNAL_IPC_UPDATED signal, that it read a data chunk.
- The location of the read data chunk is in m_Buff_IN array at the index (m_Write_IN-1) modulo m_Size_IN.

Parameters:

• m_MTU_OUT	(uint16_t)	The FIFO OUT maximum transmission unit (size of a data chunk)
• m_Size_OUT	(uint16_t)	The FIFO OUT maximum number of data chunks
• m_Read_IN	(uint32_t)	The index of the last data chunk read in the FIFO IN
• m_Write_OUT	(uint32_t)	The index of the next data chunk to be written into the FIFO OUT
• m_Buff_OUT	(array)	Array of m_Size_OUT data chunks of m_MTU_OUT bytes

Return:

• m_MTU_IN	(uint16_t)	The FIFO IN maximum transmission unit (size of a data chunk)
• m_Size_IN	(uint16_t)	The FIFO IN maximum number of data chunks
• m_Read_OUT	(uint32_t)	The index of the last data chunk read in the FIFO OUT
• m_Write_IN	(uint32_t)	The index of the next data chunk to be written into the FIFO IN
• m_Buff_IN	(array)	Array of m_Size_IN data chunks of m_MTU_IN bytes

The fields in the FIFO structure are packed in IPC memory, so there is no alignment and no padding.

The algorithm supporting the management of both FIFO queues is the following:

- FIFO OUT is empty if m_Read_OUT equals m_Write_OUT.
- FIFO IN is empty if m_Read_IN equals m_Write_IN.
- FIFO OUT is full if (m_Write_OUT - m_Read_OUT) >= m_Size_OUT
- FIFO IN is full if (m_Write_IN - m_Read_IN) >= m_Size_IN.

- The next data chunk to be written in FIFO OUT is m_Buff_OUT[m_Write_OUT]. The field m_Write_OUT shall be incremented after the writing of the data chunk.
- The next data chunk to be read in FIFO IN is m_Buff_IN[m_Read_IN]. The field m_Read_IN shall be incremented after the reading of the data chunk. The content of the data chunk shall be considered as consumed, by the writer, as soon as m_Read_IN is incremented.

The indexes of the FIFO are unsigned 32-bit integers, therefore the incrementing of the indexes is modulo 2^{32}. That leads to a wrong detection of a FIFO queue full when the algorithm encounters an arithmetic overflow. The following algorithm shall be applied [REQ**104**].

```
Count_xx is an unsigned integer.

Compute Count_xx (m_Write_xx - m_Read_xx) is the number of pending data chunks in the FIFO xx.

IF Count_xx < 0 THEN

        Count_xx = complement of Count_xx +1

END IF
```

5.10 VPP Firmware Management Service Interface

5.10.1 Requirements

This Service, running in the MGT Process, may support the management of multiple VPP Firmwares regardless of their type (i.e. normal or system).

The VPP Firmware Management Service shall be accessible only by the VPP Firmware Loader. [SREQ**105**]

The VPP Firmware Loader shall be able to manage only a VPP Firmware that it originally loaded. [SREQ**106**]

The VPP Firmware Loader shall either fully write or pad with zeros ('00') the Sub-Memory Partition, according to its specified size defined in the VPP Firmware Header. [SREQ**107**]

A VPP Application is an instance of a VPP Firmware; a VPP Firmware has two states:

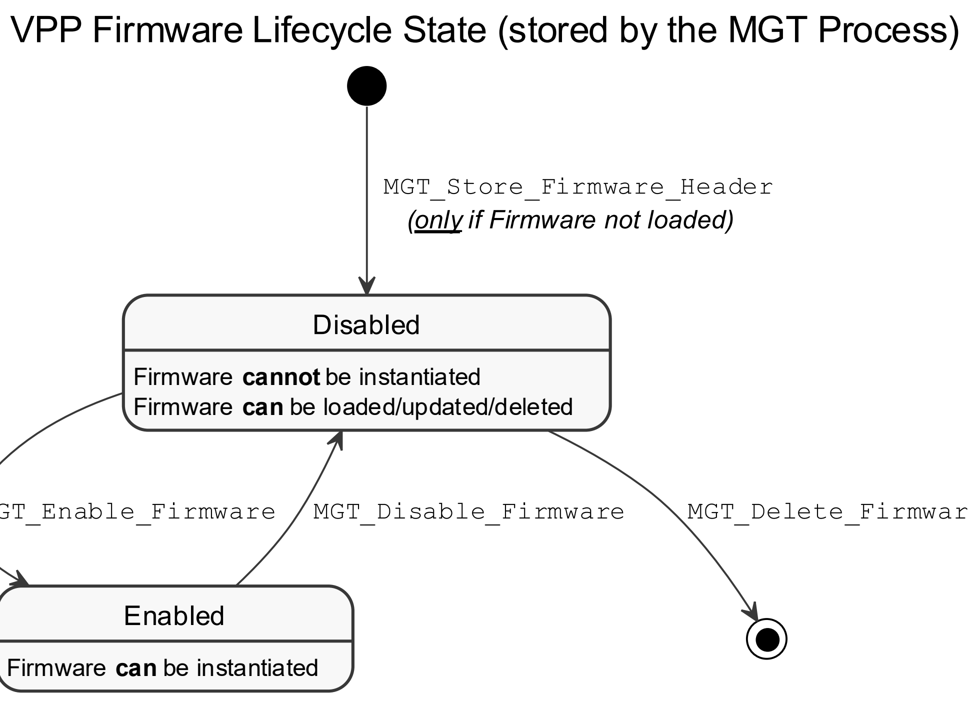
- A VPP Firmware in 'Enabled' state may be instantiated. [REQ**108**]
- A VPP Firmware in 'Disabled' state shall not be instantiated. [SREQ**109**]

VPP Firmware state shall be persistent across power cycles and is managed by the MGT Process. [SREQ**110**]

5.10.2 VPP Firmware Lifecycle State

Figure 5-8 illustrates the lifecycle state diagram in conjunction with the VPP Firmware Management Service functionality.

Figure 5-8: VPP Firmware Lifecycle State Diagram


VPP Firmware Lifecycle State (stored by the MGT Process)
MGT_Store_Firmware_Header
(only if Firmware not loaded)
Disabled
Firmware **cannot** be instantiated
Firmware **can** be loaded/updated/deleted
MGT_Enable_Firmware
MGT_Disable_Firmware
MGT_Delete_Firmware
Enabled
Firmware **can** be instantiated

When a command is sent from the MAIN Process to the MGT Process, the MAIN Process shall:

1. Fill a data structure related to the command to be executed and map it on the IPC MK_IPC_MAIN_MGT_ID. [REQ**111**]
2. Send the Signal MK_SIGNAL_IPC_UPDATED to the Mailbox MK_MAILBOX_MAIN_MGT_ID. [REQ**112**]
3. Wait for the Signal MK_SIGNAL_IPC_UPDATED on the Mailbox MK_MAILBOX_MGT_MAIN_ID. [REQ**113**]
4. Read the response from a data structure through the IPC MK_IPC_MGT_MAIN_ID. [REQ**114**]

When a response is to be returned from the MGT Process to the MAIN Process, the MGT Process shall:

1. Wait for the Signal MK_SIGNAL_IPC_UPDATED on the Mailbox MK_MAILBOX_MAIN_MGT_ID. [REQ**115**]
2. Read the command data through the IPC MK_IPC_MAIN_MGT_ID. [REQ**116**]
3. Send the response data through the IPC MK_IPC_MGT_MAIN_ID. [REQ**117**]
4. Send the Signal MK_SIGNAL_IPC_UPDATED to the Mailbox MK_MAILBOX_MGT_MAIN_ID. [REQ**118**]

Table 5-4 lists the commands.

Table 5-4: VPP Firmware Management Service Commands

Command Code	Command
'00'	MGT_Store_Firmware_Header
'01'	MGT_Retrieve_Firmware_Header
'02'	MGT_Allocate_Firmware
'03'	MGT_Delete_Firmware
'04'	MGT_Enable_Firmware
'05'	MGT_Disable_Firmware
'06'	MGT_Is_Firmware_Enabled
'07'	MGT_Open_Process_SubMemoryPartition
'08'	MGT_Close_Process_SubMemoryPartition
'09'	MGT_Open_Library_SubMemoryPartition (optional)
'0A'	MGT_Close_Library_SubMemoryPartition (optional)
'0B'	MGT_Open_LLOS_SubMemoryPartition (optional)
'0C'	MGT_Close_LLOS_SubMemoryPartition (optional)

Table 5-5 lists the response codes.

Table 5-5: VPP Firmware Management Service Response Codes

Response Code	Definition
'00'	MGT_ERROR_NONE
'01'	MGT_ERROR_ILLEGAL_PARAMETER
'02'	MGT_ERROR_INTERNAL
'03'	MGT_ERROR_UNKNOWN_UUID
'04'	MGT_ERROR_COMMAND_NOK

Table 5-6: VPP Firmware Management Service Command / Response Codes Assignment

COMMAND RESPONSE	MGT_Store_Firmware_Header	MGT_Retrieve_Firmware_Header	MGT_Allocate_Firmware	MGT_Delete_Firmware	MGT_Enable_Firmware	MGT_Disable_Firmware	MGT_Is_Firmware_Enabled	MGT_Open_Process_SubMemoryPartition	MGT_Close_Process_SubMemoryPartition	MGT_Open_Library_SubMemoryPartition	MGT_Close_Library_SubMemoryPartition	MGT_Open_LLOS_SubMemoryPartition	MGT_Close_LLOS_SubMemoryPartition
MGT_ERROR_NONE	●	●	●	●	●	●	●	●	●	●	●	●	●
MGT_ERROR _ILLEGAL_PARAMETER	●	●	●	●	●	●	●	●		●			
MGT_ERROR_INTERNAL			●	●	●	●	●	●	●	●	●	●	●
MGT_ERROR _UNKNOWN_UUID		●	●	●	●	●	●	●		●			
MGT_ERROR _COMMAND_NOK	●	●	●	●	●	●	●	●	●	●	●	●	●

Note: MGT_ERROR_COMMAND_NOK is reserved for the Primary Platform Maker, as a generic error.

The fields in the command and response structures are neither aligned nor padded.

5.10.3 VPP Firmware Header Management

5.10.3.1 MGT_Store_Firmware_Header

Brief: Store the VPP Firmware Header in the Memory Partition as defined in [VFF].

Description:

The command performs the following operations:

- Parse the firmware_header
- If parsing failed:
  - Return response_code = MGT_ERROR_ILLEGAL_PARAMETER
- Compare the firmware_header to the Primary Platform capabilities
- If firmware_header is not supported:
  - Return response_code = MGT_ERROR_ILLEGAL_PARAMETER
- Extract the VPP Firmware Identifier (UUID) from firmware_header
- Retrieve the firmware_header from the MGT Process NVM, based on its provided VPP Firmware Identifier as defined in [VFF]
- If firmware_header cannot be found:
  - Add a new VPP Firmware Header, store the provided VPP Firmware Identifier as the key to this record. The initial VPP Firmware state should be 'Disabled'
- Else:
  - Update firmware_header record
- Return response_code = MGT_ERROR_NONE

Parameters:

• '00'	(uint8_t)	MGT_Store_Firmware_Header Command
• (firmware_header data)	header as described in [VFF]	

Return:

• response_code	(uint8_t)	VPP Firmware Management Service response code as described in Table 5-5

5.10.3.2 MGT_Retrieve_Firmware_Header

Brief: Retrieve the VPP Firmware Header from the MGT Process as defined in [VFF].

Description:

The command performs the following operations:

- If firmware_identifier is not 16 bytes in size:
    - Return response_code = MGT_ERROR_ILLEGAL_PARAMETER
- Load the VPP Firmware Header from the MGT Process NVM that has the same VPP Firmware Identifier as the provided firmware_identifier.
- If the VPP Firmware Header is not found:
    - Return response_code = MGT_ERROR_UNKNOWN_UUID
- Return:
    - response_code = MGT_ERROR_NONE
    - (the VPP Firmware Header in the response)

Parameters:

• '01'	(uint8_t)	MGT_Retrieve_Firmware_Header command
• firmware_identifier	(UUID_t)	identifier of the VPP Firmware (i.e. m_xFirmwareID in [VFF]).

Return:

• response_code	(uint8_t) VPP Firmware Management Service response code as described in Table 5-5
• (VPP Firmware Header data)	The data of the VPP Firmware Header, as defined in [VFF]

5.10.4 VPP Firmware State Management

5.10.4.1 MGT_Enable_Firmware

Brief: Change VPP Firmware state to 'Enabled'.

Description:

The command performs the following operations:

- If firmware_identifier is not 16 bytes in size:
  - Return response_code = MGT_ERROR_ILLEGAL_PARAMETER
- Load the VPP Firmware Header from the MGT Process NVM that has the same VPP Firmware Identifier as the provided firmware_identifier.
- If VPP Firmware Header is not found:
  - Return response_code = MGT_ERROR_UNKNOWN_UUID
- Instruct the COM Process that the identified VPP Firmware shall be registered with the entity managing the communication to the TRE, and may now receive and be instantiated upon incoming data as defined in [VNP].
- Update the VPP Firmware state in NVM belonging to MGT Process to 'Enabled'.
- If error while enabling VPP Firmware:
  - Return response_code = MGT_ERROR_INTERNAL
- Return response_code = MGT_ERROR_NONE

Parameters:

• '04'	(uint8_t)	MGT_Enable_Firmware command
• firmware_identifier	(UUID_t)	identifier of the VPP Firmware (i.e. m_xFirmwareID in [VFF])

Return:

• response_code	(uint8_t)	VPP Firmware Management Service response code as described in Table 5-5

5.10.4.2 MGT_Disable_Firmware

Brief: Change VPP Firmware state to 'Disabled'.

Description:

The command performs the following operations:

- If firmware_identifier is not 16 bytes in size:
  - Return response_code = MGT_ERROR_ILLEGAL_PARAMETER
- Load the VPP Firmware Header from the MGT Process NVM that has the same VPP Firmware Identifier as the provided firmware_identifier.
- If VPP Firmware Header is not found:
  - Return response_code = MGT_ERROR_UNKNOWN_UUID
- Instruct the COM Process that the identified VPP Firmware shall be deregistered with the entity managing the communication to the TRE, shall not receive incoming data as defined in [VNP], and shall not be instantiated.
- Update the VPP Firmware state in NVM belonging to MGT Process to 'Disabled'.
- If error while disabling VPP Firmware:
  - Return response_code = MGT_ERROR_INTERNAL
- Return response_code = MGT_ERROR_NONE

Parameters:

• '05'	(uint8_t)	MGT_Disable_Firmware command.
• firmware_identifier	(UUID_t)	identifier of the VPP Firmware (i.e. m_xFirmwareID in [VFF]) to be disabled

Return:

• response_code	(uint8_t)	VPP Firmware Management Service response code as described in Table 5-5

5.10.4.3 MGT_Is_Firmware_Enabled

Brief: Query if the state of a VPP Firmware is ‘Enabled’.

Description:

The command performs the following operations:

- If firmware_identifier is not 16 bytes in size:
  - Return response_code = MGT_ERROR_ILLEGAL_PARAMETER
- Load the VPP Firmware Header from the MGT Process NVM that has the same VPP Firmware Identifier as the provided firmware_identifier.
- If VPP Firmware Header is not found:
  - Return response_code = MGT_ERROR_UNKNOWN_UUID
- Return:
  - response_code = MGT_ERROR_NONE
  - state =
    - TRUE if the state of the VPP Firmware is ‘Enabled’
    - FALSE otherwise

Parameters:

• '06'	(uint8_t)	MGT_Is_Firmware_Enabled command
• firmware_identifier	(UUID_t)	identifier of the VPP Firmware (i.e. m_xFirmwareID in [VFF]) to query

Return:

• response_code	(uint8_t)	VPP Firmware Management Service response code as described in Table 5-5
• state	(uint8_t)	'01' if the VPP Firmware state is ‘Enabled’, '00' if the VPP Firmware state is ‘Disabled’

5.10.4.4 MGT_Delete_Firmware

Brief: Delete an existing VPP Firmware related to a VPP Application.

Description:

The command performs the following operations:

- If firmware_identifier is not 16 bytes in size:
    - Return response_code = MGT_ERROR_ILLEGAL_PARAMETER
- Load the VPP Firmware Header from the MGT Process NVM that has the same VPP Firmware Identifier as the provided firmware_identifier.
- If VPP Firmware Header is not found:
    - Return response_code = MGT_ERROR_UNKNOWN_UUID
- If VPP Firmware state is not 'Disabled':
    - Return response_code = MGT_ERROR_INTERNAL
- Erase the Memory Partition from non-volatile data storage (NVM) and from any cache memory, based on the provided firmware_identifier.
- If error while erasing:
    - Return response_code = MGT_ERROR_INTERNAL
- Erase the VPP Firmware Header and the VPP Firmware state from the MGT Process NVM, based on the provided firmware_identifier.
- Unregister the VPP Firmware Identifier within the COM Process
- Return response_code = MGT_ERROR_NONE

Parameters:

• '03'	(uint8_t)	Code corresponding to MGT_Delete_Firmware command
• firmware_identifier	(UUID_t)	identifier of the VPP Firmware (i.e. m_xFirmwareID in [VFF]) to be deleted

Return:

• response_code	(uint8_t)	VPP Firmware Management Service response code as described in Table 5-5

5.10.5 VPP Firmware Content Management

5.10.5.1 MGT_Open_Process_SubMemoryPartition

Brief: Prepare the Sub-Memory Partition of a Process for writing by the VPP Firmware Loader.

Description:

The command performs the following operations:

- If firmware_identifier is not 16 bytes in size:
  - Return response_code = MGT_ERROR_ILLEGAL_PARAMETER
- Load the VPP Firmware Header from the MGT Process NVM that has the same VPP Firmware Identifier as the provided firmware_identifier.
- If VPP Firmware Header is not found:
  - Return response_code = MGT_ERROR_UNKNOWN_UUID
- Load the Memory Partitions of the VPP Firmware
- Initiate the instantiation of a Kernel Object for reading or writing
- Initialize the above Kernel Object with the physical memory address of the Sub-Memory Partition associated with the related content of the Process, which is itself part of the VPP Firmware being loaded
- If an error occurs:
  - Return response_code = MGT_ERROR_INTERNAL
- Return response_code = MGT_ERROR_NONE

Parameters:

• '07'	(uint8_t)	MGT_Open_Process_SubMemoryPartition command
• firmware_identifier	(UUID_t)	identifier of the VPP Firmware (i.e. m_xFirmwareID in [VFF])
• index	(MK_Index_t)	index of the Process Descriptor within the array of Process Descriptors of the VPP Firmware Header defined in [VFF]

Return:

• response_code	(uint8_t)	VPP Firmware Management Service response code as described in Table 5-5

5.10.5.2 MGT_Close_Process_SubMemoryPartition

Brief: Close the Sub-Memory Partition of a Process for reading or writing.

Description:

The command performs the following operations:

- Instruct the kernel to clear the reference related to the Sub-Memory Partition
- If error occurs:
  - Return response_code = MGT_ERROR_INTERNAL
- Return response_code = MGT_ERROR_NONE

Parameters:

• '08'	(uint8_t)	MGT_Close_Process_SubMemoryPartition command

Return:

• response_code	(uint8_t)	VPP Firmware Management Service response code as described in Table 5-5

5.10.5.3 MGT_Open_Library_SubMemoryPartition

Brief: Prepare a Sub-Memory Partition of a shared library for reading or writing by the VPP Firmware Loader as defined in [VFF]. This command is optional.

Description:

The command performs the following operations:

- If firmware_identifier is not 16 bytes in size:
    - Return response_code = MGT_ERROR_ILLEGAL_PARAMETER
- Load the VPP Firmware Header from the MGT Process NVM that has the same VPP Firmware Identifier as the provided firmware_identifier
- If VPP Firmware Header is not found:
    - Return response_code = MGT_ERROR_UNKNOWN_UUID
- Load the Memory Partitions of the VPP Firmware
- Initiate the instantiation of a Kernel Object for reading or writing
- Initialize the above Kernel Object with the physical memory address of the Sub-Memory Partition related to the specified library
- If an error occurs:
    - Return response_code = MGT_ERROR_INTERNAL
- Return response_code = MGT_ERROR_NONE

Parameters:

• '09'	(uint8_t) MGT_Open_Library_SubMemoryPartition command
• firmware_identifier	(UUID_t) identifier of the VPP Firmware (i.e. m_xFirmwareID in [VFF])
• index	(MK_Index_t) index of the LIB Descriptor within the array of LIB Descriptors of the VPP Firmware Header defined in [VFF]

Return:

• response_code	(uint8_t) VPP Firmware Management Service response code as described in Table 5-5

5.10.5.4 MGT_Close_Library_SubMemoryPartition

Brief: Close the Sub-Memory Partition of a shared library for reading or writing. This command is optional.

Description:

The command performs the following operations:

- Instruct the kernel to clear the reference related to the Sub-Memory Partition
- If an error occurs:
  - Return response_code = MGT_ERROR_INTERNAL
- Return response_code = MGT_ERROR_NONE

Parameters:

• '0A'	(uint8_t)	MGT_Close_Library_SubMemoryPartition command

Return:

• response_code	(uint8_t)	VPP Firmware Management Service response code as described in Table 5-5

5.10.5.5 MGT_Open_LLOS_SubMemoryPartition

Brief: Prepare a Sub-Memory Partition of LLOS for reading and writing by the VPP Firmware Loader as defined in [VFF]. This command is optional.

Description:

The command performs the following operations:

- Load the Memory Partition of the VPP Firmware related to the LLOS
- If an error occurs:
  - Return response_code = MGT_ERROR_INTERNAL
- Return response_code = MGT_ERROR_NONE

Parameters:

• '0B'	(uint8_t)	MGT_Open_LLOS_SubMemoryPartition command
• firmware_identifier	(UUID_t)	identifier of the VPP Firmware (i.e. m_xFirmwareID in [VFF])

Return:

• response_code	(uint8_t)	VPP Firmware Management Service response code as described in Table 5-5

5.10.5.6 MGT_Close_LLOS_SubMemoryPartition

Brief: Close the Sub-Memory Partition of the LLOS for reading or writing. This command is optional.

Description:

The command performs the following operations:

- Instruct the kernel to clear the reference related to the Sub-Memory Partition
- If an error occurs:
    - Return response_code = MGT_ERROR_INTERNAL
- Return response_code = MGT_ERROR_NONE

Parameters:

• '0C'	(uint8_t)	MGT_Close_LLOS_SubMemoryPartition command

Return:

• response_code	(uint8_t)	VPP Firmware Management Service response code as described in Table 5-5

5.10.5.7 MGT_Allocate_Firmware

Brief: Prepare the Primary Platform for a VPP Firmware loading by allocating the Memory Partitions in NVM.

Description:

The command performs the following operations:

- If firmware_identifier is not 16 bytes in size:
  - Return response_code = MGT_ERROR_ILLEGAL_PARAMETER
- Load the VPP Firmware Header from the MGT Process NVM that has the same VPP Firmware Identifier as the provided firmware_identifier
- If VPP Firmware Header is not found:
  - Return response_code = MGT_ERROR_UNKNOWN_UUID
- Allocate Memory Partitions for VPP Firmware according to the Process Descriptors of the VPP Firmware Header
- Allocated Memory Partitions are uninitialized. Therefore, the VPP Firmware Loader shall fully write into each Virtual Address Space Region, first with all content of each provided VPP Firmware Sub-Memory Partition, then padding the rest with zeros ('00').
- If error while allocating:
  - Return response_code = MGT_ERROR_INTERNAL
- Return response_code = MGT_ERROR_NONE

Parameters:

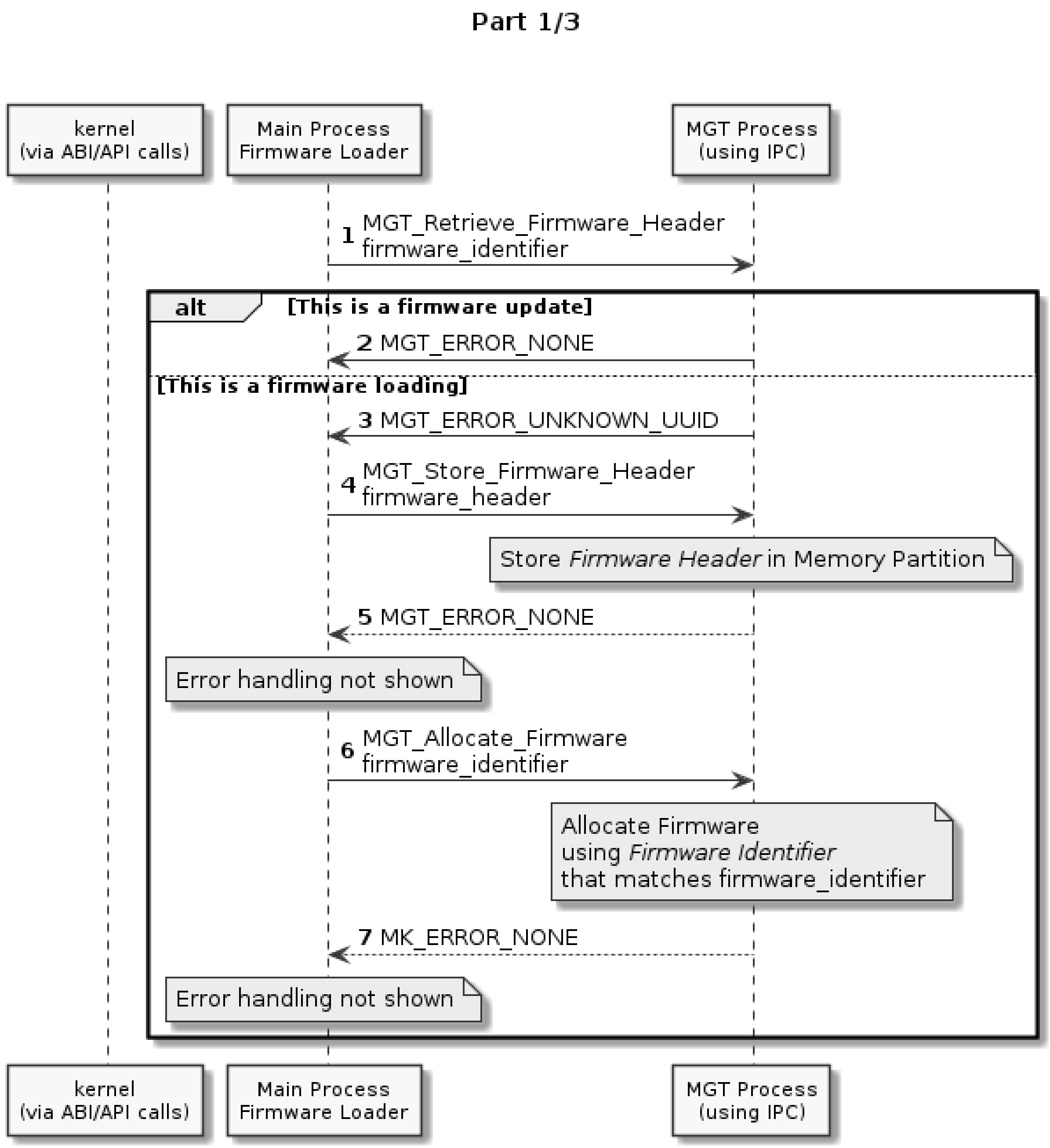
• '02'	(uint8_t)	MGT_Allocate_Firmware command
• firmware_identifier	(UUID_t)	identifier of the VPP Firmware (i.e. m_xFirmwareID in [VFF])

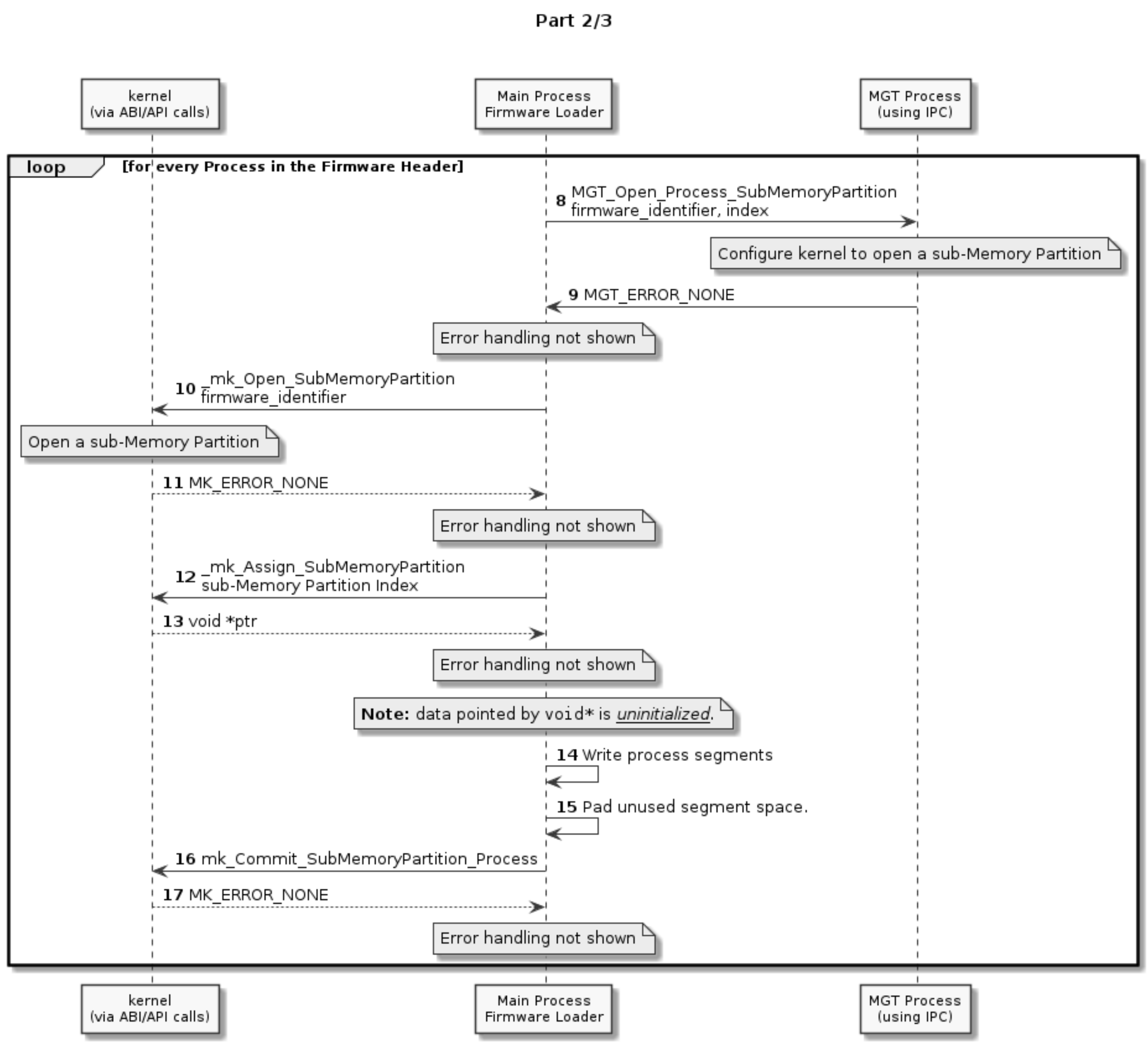
Return:

• response_code	(uint8_t)	VPP Firmware Management Service response code as described in Table 5-5

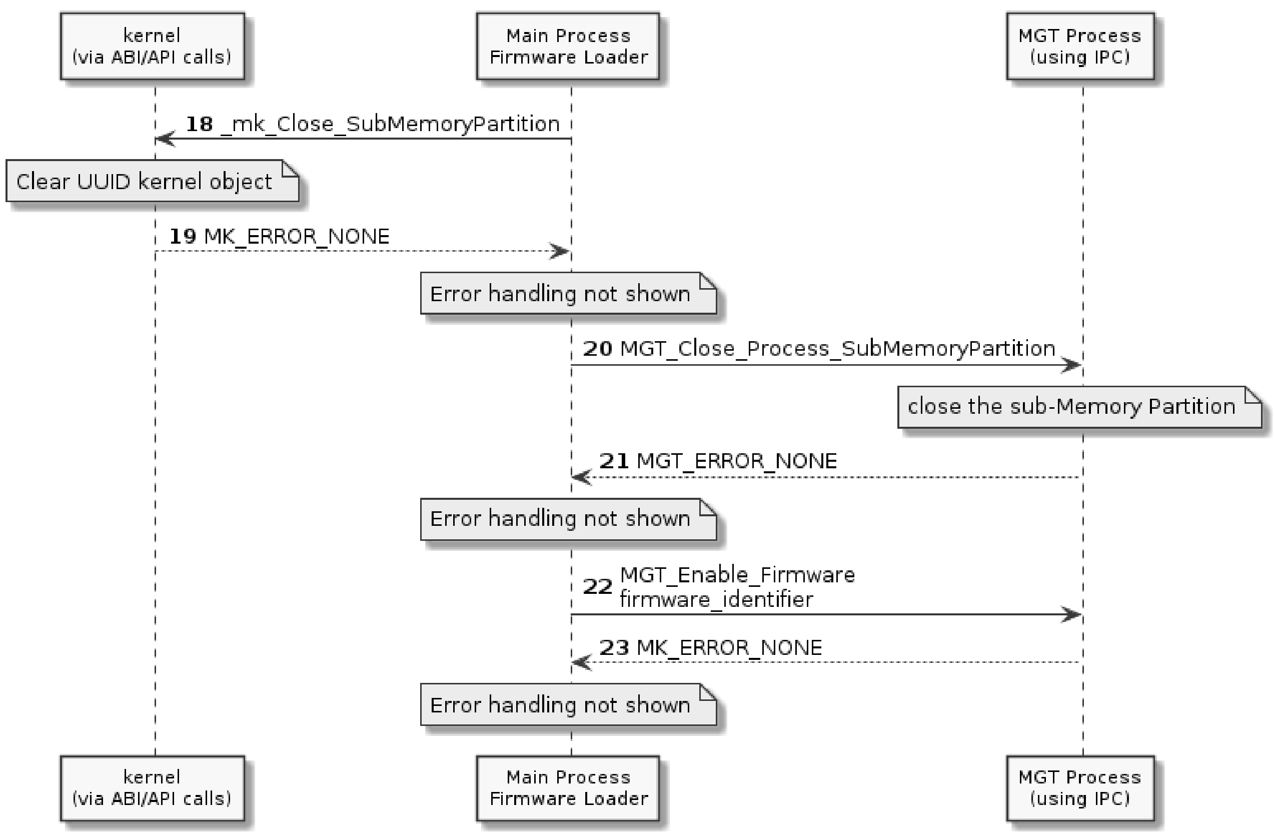
Figure 5-9 illustrates the installation of a VPP Firmware.

Figure 5-9: VPP Firmware Loading and Update


Part 1/3
kernel (via ABI/API calls)
Main Process Firmware Loader
MGT Process (using IPC)
1 MGT_Retrieve_Firmware_Header firmware_identifier
alt [This is a firmware update]
2 MGT_ERROR_NONE
[This is a firmware loading]
3 MGT_ERROR_UNKNOWN_UUID
4 MGT_Store_Firmware_Header firmware_header
Store *Firmware Header* in Memory Partition
5 MGT_ERROR_NONE
Error handling not shown
6 MGT_Allocate_Firmware firmware_identifier
Allocate Firmware using *Firmware Identifier* that matches firmware_identifier
7 MK_ERROR_NONE
Error handling not shown


Part 2/3
kernel (via ABI/API calls)
Main Process Firmware Loader
MGT Process (using IPC)
loop [for every Process in the Firmware Header]
8 MGT_Open_Process_SubMemoryPartition firmware_identifier, index
Configure kernel to open a sub-Memory Partition
9 MGT_ERROR_NONE
Error handling not shown
10 _mk_Open_SubMemoryPartition firmware_identifier
Open a sub-Memory Partition
11 MK_ERROR_NONE
Error handling not shown
12 _mk_Assign_SubMemoryPartition sub-Memory Partition Index
13 void *ptr
Error handling not shown
Note: data pointed by void* is uninitialized.
14 Write process segments
15 Pad unused segment space.
16 mk_Commit_SubMemoryPartition_Process
17 MK_ERROR_NONE
Error handling not shown
kernel (via ABI/API calls)
Main Process Firmware Loader
MGT Process (using IPC)

Part 3/3


kernel
(via ABI/API calls)
Main Process
Firmware Loader
MGT Process
(using IPC)
18 _mk_Close_SubMemoryPartition
Clear UUID kernel object
19 MK_ERROR_NONE
Error handling not shown
20 MGT_Close_Process_SubMemoryPartition
close the sub-Memory Partition
21 MGT_ERROR_NONE
Error handling not shown
22 MGT_Enable_Firmware
firmware_identifier
23 MK_ERROR_NONE
Error handling not shown
kernel
(via ABI/API calls)
Main Process
Firmware Loader
MGT Process
(using IPC)

5.11 Mandatory Access Control

The Mandatory Access Control (MAC) controls access to:

- Cross-Execution-Domain Mailboxes and IPCs as defined in section 7.4
- Groups of kernel functions as defined in section 5.11.3

If a VPP Application violates the Mandatory Access Control and the violation is not handled by the Kernel Functions API, the VPP generates a severe Exception (i.e. MK_EXCEPTION_SEVERE) causing the VPP Application to terminate.

Cross-Execution-Domain Mailboxes and IPCs are only defined for the following Processes:

- MAIN Process of a VPP Application
- COM Process of VPP
- MGT Process of VPP

VPP Firmware Loader has access rights in addition to those provided to VPP Applications.

5.11.1 VPP Application

For VPP Applications, Table 5-7 details the availability of Mailbox and IPC between two Processes as listed above.

Table 5-7: Standard Access to Cross-Execution-Domain Mailboxes and IPCs

		Sender Process					
		COM Process		**MGT Process**		**MAIN Process**	
		IPC	**MAILBOX**	**IPC**	**MAILBOX**	**IPC**	**MAILBOX**
Receiver Process	**COM**					MK_IPC _MAIN_COM_ID	MK_MAILBOX _MAIN_COM_ID
	MGT						MK_MAILBOX _MAIN_MGT_ID
	MAIN	MK_IPC _COM_MAIN_ID	MK_MAILBOX _COM_MAIN_ID		MK_MAILBOX _MGT_MAIN_ID		

The VPP Application shall be able to access the following VREs [SREQ**119**]:

• MK_VRE_RNG	Random Number Generation hardware function

The VPP Application may access the following VREs:

• MK_VRE_ECC	ECC accelerator hardware function
• MK_VRE_RSA	RSA accelerator hardware function
• MK_VRE_AES	AES accelerator hardware function
• MK_VRE_HASH	HASH accelerator hardware function
• MK_VRE_RAF	Remote Audit Function

Note: The Primary Platform Maker may provide additional, platform-specific VREs, to be accessible by the VPP Application.

5.11.2 VPP Firmware Loader

As mentioned above, VPP Firmware Loader has access rights in addition to those provided to VPP Applications.

For VPP Firmware Loader, Table 5-8 details the availability of Mailbox and IPC between two allowed Processes as listed above.

Table 5-8: VPP Firmware Loader Access to Cross-Execution-Domain Mailboxes and IPCs

		Sender Process					
		COM Process		MGT Process		MAIN Process	
		IPC	MAILBOX	IPC	MAILBOX	IPC	MAILBOX
Receiver Process	COM					MK_IPC _MAIN_COM_ID	MK_MAILBOX _MAIN_COM_ID
	MGT					MK_IPC _MAIN_MGT_ID	MK_MAILBOX _MAIN_MGT_ID
	MAIN	MK_IPC _COM_MAIN_ID	MK_MAILBOX _COM_MAIN_ID	MK_IPC _MGT_MAIN_ID	MK_MAILBOX _MGT_MAIN_ID		

The VPP Firmware Loader shall be able to access the following VRE [SREQ**120**]:

• MK_VRE_RNG	Random Number Generation hardware function

The VPP Firmware Loader may optionally access the following VREs:

• MK_VRE_ECC	ECC accelerator hardware function
• MK_VRE_RSA	RSA accelerator hardware function
• MK_VRE_ROT	Long-term credentials storage as defined in section 3.1
• MK_VRE_AES	AES accelerator hardware function
• MK_VRE_HASH	HASH accelerator hardware function
• MK_VRE_RAF	Remote Audit Function

- Any VRE reserved by the Primary Platform Maker

5.11.3 Kernel Functions Groups

Table 5-9 defines the groups of kernel functions.

Table 5-9: Groups of Kernel Functions

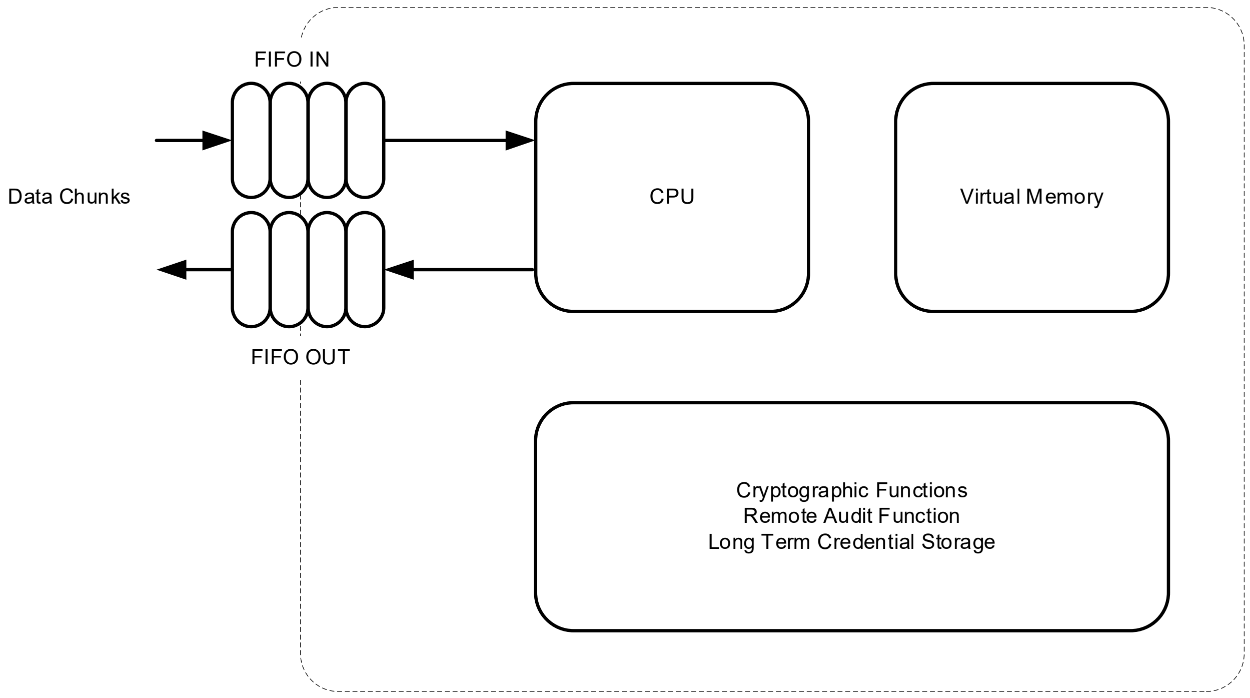
Groups	Description	Allowed Kernel Functions
MK_MAC_GA	General ABI for any Process Available to all Applications, including VPP Processes	_mk_Get_Error _mk_Get_Exception _mk_Get_Process_Priority _mk_Set_Process_Priority _mk_Suspend_Process _mk_Resume_Process _mk_Request_No_Preemption _mk_Commit _mk_RollBack _mk_Yield _mk_Get_Mailbox_Handle _mk_Get_IPC_Handle _mk_Get_Access_IPC _mk_Release_Access_IPC _mk_Get_Mailbox_ID_Activated _mk_Send_Signal _mk_Wait_Signal _mk_Get_Signal _mk_Get_VRE_Handle _mk_Attach_VRE _mk_Get_Access_VRE _mk_Release_Access_VRE _mk_Get_Time _mk_Get_Process_Handle
MK_MAC_SYS_APP	System ABI for VPP Firmware Loader Processes Restricted to the VPP Firmware Loader	_mk_Open_SubMemoryPartition _mk_Close_SubMemoryPartition _mk_Assign_SubMemoryPartition _mk_Commit_SubMemoryPartition

6 Virtual Primary Platform Application

6.1 The Virtual Hardware Platform

Figure 6-1 illustrates the virtual hardware platform on which the VPP Application is built.

Figure 6-1: Virtual Hardware Platform



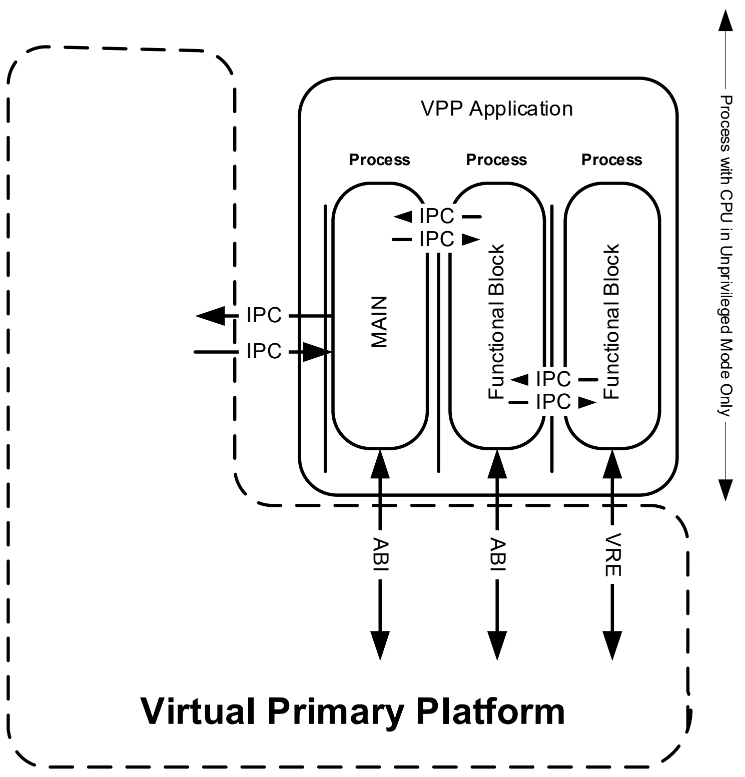
Each Process of the VPP Application runs on top of a virtual hardware platform.

Only the MAIN Process of the VPP Application shall access a FIFO of data chunks. [SREQ**121**]

6.2 Structure

Figure 6-2 illustrates the structure of the VPP Application.

Figure 6-2: Structure of the VPP Application



The VPP Application may be a collection of Processes. The VPP Application shall run at least a Process named MAIN [SREQ**122**].

6.3 VPP Application Session

A VPP Application session represents the period of time from the VPP Firmware instantiation to its termination.

The following operations shall be performed during the VPP Firmware instantiation [SREQ**123**]:

- All VPP Application Processes are instantiated:
  - The content of CODE, CONSTANTS, and the optional LIB CODE and LIB CONSTANTS of each Process is initialized with the data previously written by the VPP Firmware Loader.
  - The NVM of each Process is initialized by initial values provided by the VPP Firmware Loader then by the content of the last successful _mk_Commit_SubMemoryPartition operation.
  - The content of Volatile Data of each Process is zeroed.
  - The Virtual/Physical Memory[16] Space content of each STACK is zeroed.
  - All Writer IPCs are cleared with zeros.
  - All Processes are instantiated in the 'Suspended R' state with the default priority *MK_PROCESS_PRIORITY_NORMAL*. The entry point address for each Process is provided in the VPP Firmware, as defined in [VFF].
- The VPP COM Process shall [SREQ**124**]:
  - Reset the m_Read_IN field of its FIFO OUT.
  - Write the incoming data chunks, if any, in its FIFO OUT.
  - Notify the MAIN Process that its FIFO IN has been updated
- The MGT Process resumes the MAIN Process.
- The MAIN Process copies:
  - The values m_MTU_IN and m_Size_IN from its FIFO IN within the IPC referenced by MK_IPC_COM_MAIN_ID
  - The values in m_MTU_OUT and m_Size_OUT to its FIFO OUT within the IPC referenced by MK_IPC_MAIN_COM_ID

During execution, each Process initializes the shareable or non-shareable Volatile Data it is responsible for.

A VPP Application may restart itself (and therefore its session) by sending the Signal MK_SIGNAL_APP_RESTART to the Mailbox identified as MK_MAILBOX_MAIN_MGT_ID in the MGT Process.

A VPP Application session shall end when VPP Application termination begins. [REQ**125**]

A VPP Application termination shall begin [SREQ**126**]:

- When the Mailbox identified as MK_MAILBOX_MGT_MAIN_ID in the MAIN Process receives the MK_SIGNAL_KILL_REQUESTED
- Or when the Mailbox identified as MK_MAILBOX_MAIN_MGT_ID in the MGT Process receives the MK_SIGNAL_KILL_ITSELF

VPP shall complete VPP Application termination [REQ**127**]:

[16] The stack of the process may be mapped to the Virtual Address Space or to the Physical Address Space (Primary Platform implementation dependent).

- When the Mailbox identified as MK_MAILBOX_MAIN_MGT_ID in the MGT Process receives the MK_SIGNAL_KILL_ACCEPTED or MK_SIGNAL_KILL_ITSELF
- Or after VPP Application MK_APP_STOP_GRACEFUL_TICKS time elapsed. [SREQ**128**]

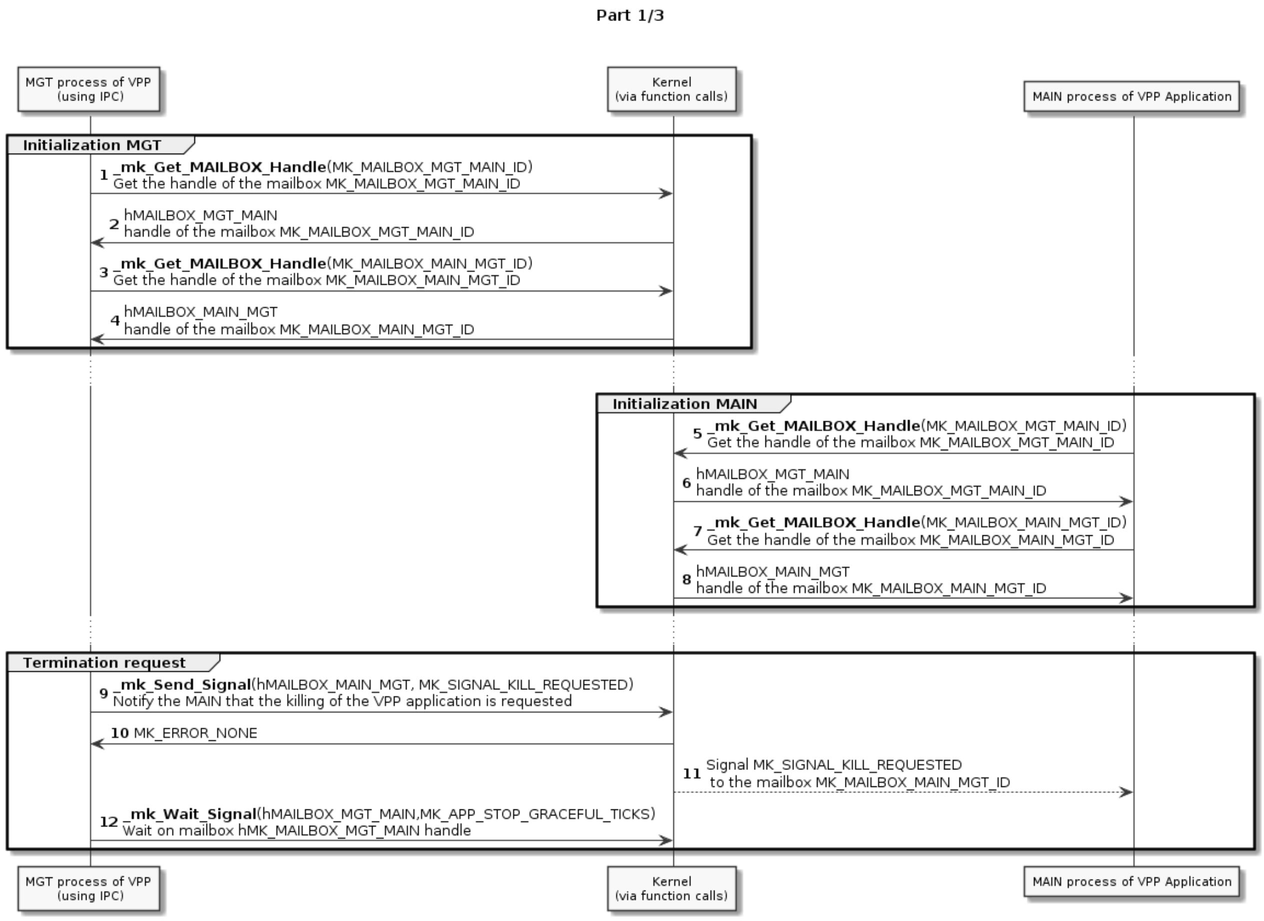
The following applies when the VPP Application terminates [SREQ**129**]:

- All data chunks in the FIFO OUT of the COM and MAIN Processes that have not been read are lost.
- Only changes committed to NVM Memory Partitions are preserved.

VPP shall ensure that VPP Application restart does not override VPP Application termination. [SREQ**130**]

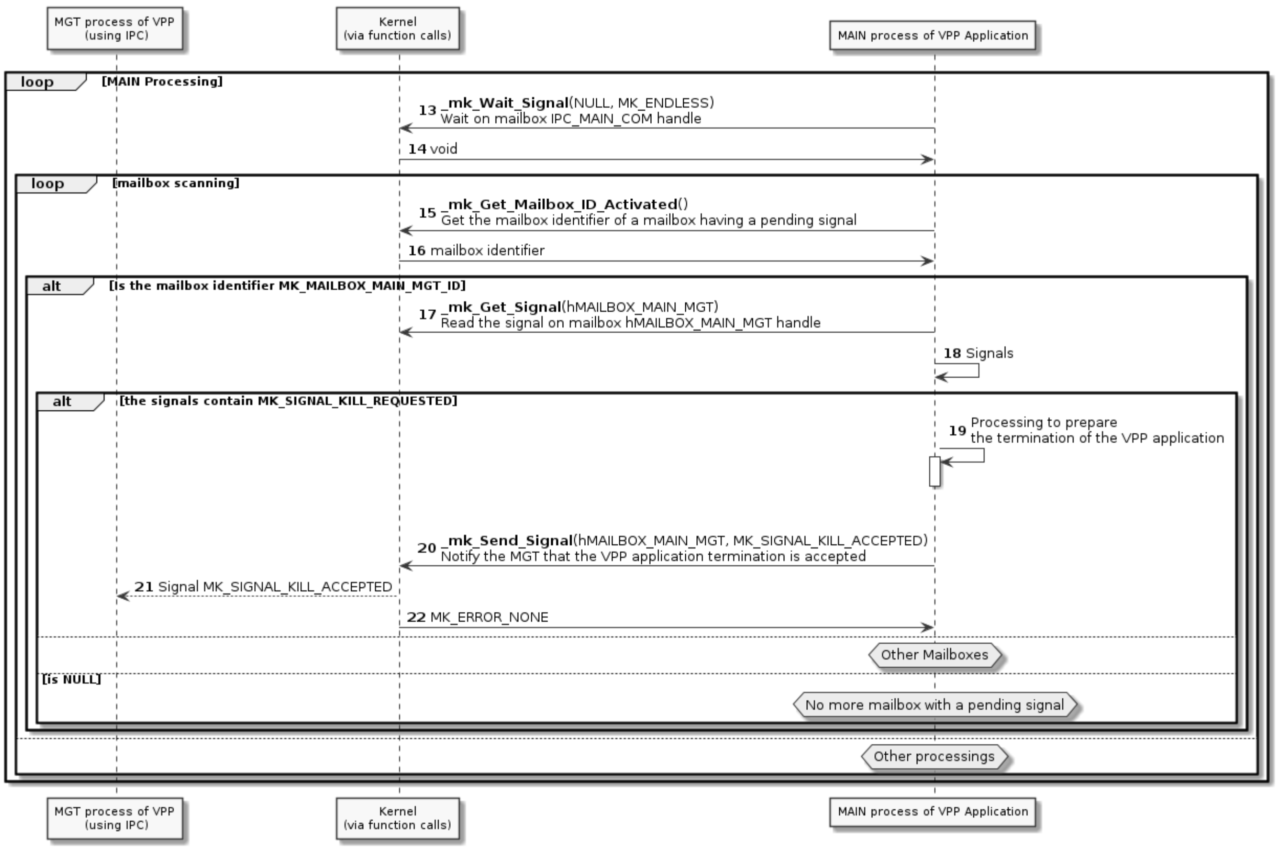
Figure 6-3 illustrates the termination of the VPP Application.

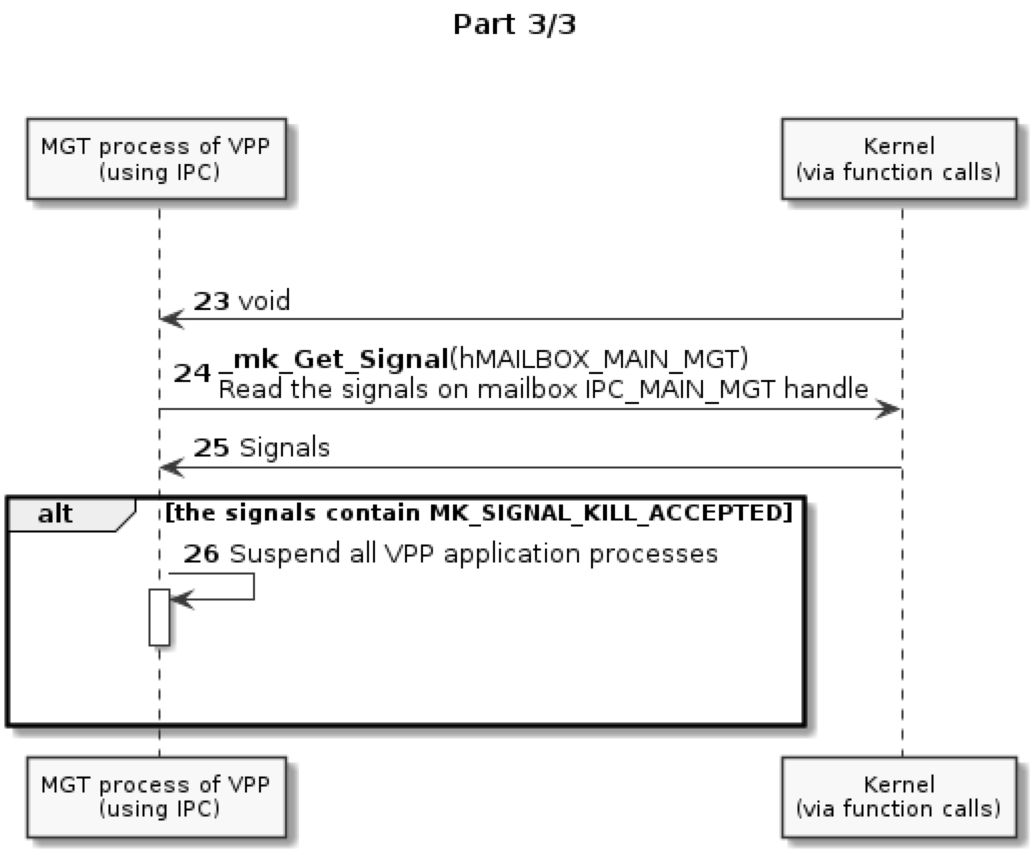
Figure 6-3: VPP Application Termination



Part 1/3
MGT process of VPP (using IPC)
Kernel (via function calls)
MAIN process of VPP Application
Initialization MGT
1 _mk_Get_MAILBOX_Handle(MK_MAILBOX_MGT_MAIN_ID) Get the handle of the mailbox MK_MAILBOX_MGT_MAIN_ID
2 hMAILBOX_MGT_MAIN handle of the mailbox MK_MAILBOX_MGT_MAIN_ID
3 _mk_Get_MAILBOX_Handle(MK_MAILBOX_MAIN_MGT_ID) Get the handle of the mailbox MK_MAILBOX_MAIN_MGT_ID
4 hMAILBOX_MAIN_MGT handle of the mailbox MK_MAILBOX_MAIN_MGT_ID
Initialization MAIN
5 _mk_Get_MAILBOX_Handle(MK_MAILBOX_MGT_MAIN_ID) Get the handle of the mailbox MK_MAILBOX_MGT_MAIN_ID
6 hMAILBOX_MGT_MAIN handle of the mailbox MK_MAILBOX_MGT_MAIN_ID
7 _mk_Get_MAILBOX_Handle(MK_MAILBOX_MAIN_MGT_ID) Get the handle of the mailbox MK_MAILBOX_MAIN_MGT_ID
8 hMAILBOX_MAIN_MGT handle of the mailbox MK_MAILBOX_MAIN_MGT_ID
Termination request
9 _mk_Send_Signal(hMAILBOX_MAIN_MGT, MK_SIGNAL_KILL_REQUESTED) Notify the MAIN that the killing of the VPP application is requested
10 MK_ERROR_NONE
11 Signal MK_SIGNAL_KILL_REQUESTED to the mailbox MK_MAILBOX_MAIN_MGT_ID
12 _mk_Wait_Signal(hMAILBOX_MGT_MAIN,MK_APP_STOP_GRACEFUL_TICKS) Wait on mailbox hMK_MAILBOX_MGT_MAIN handle
MGT process of VPP (using IPC)
Kernel (via function calls)
MAIN process of VPP Application

Part 2/3


MGT process of VPP (using IPC)
Kernel (via function calls)
MAIN process of VPP Application
loop [MAIN Processing]
13 _mk_Wait_Signal(NULL, MK_ENDLESS) Wait on mailbox IPC_MAIN_COM handle
14 void
loop [mailbox scanning]
15 _mk_Get_Mailbox_ID_Activated() Get the mailbox identifier of a mailbox having a pending signal
16 mailbox identifier
alt [Is the mailbox identifier MK_MAILBOX_MAIN_MGT_ID]
17 _mk_Get_Signal(hMAILBOX_MAIN_MGT) Read the signal on mailbox hMAILBOX_MAIN_MGT handle
18 Signals
alt [the signals contain MK_SIGNAL_KILL_REQUESTED]
19 Processing to prepare the termination of the VPP application
20 _mk_Send_Signal(hMAILBOX_MAIN_MGT, MK_SIGNAL_KILL_ACCEPTED) Notify the MGT that the VPP application termination is accepted
21 Signal MK_SIGNAL_KILL_ACCEPTED
22 MK_ERROR_NONE
Other Mailboxes
[is NULL]
No more mailbox with a pending signal
Other processings


Part 3/3
MGT process of VPP (using IPC)
Kernel (via function calls)
23 void
24 _mk_Get_Signal(hMAILBOX_MAIN_MGT)
Read the signals on mailbox IPC_MAIN_MGT handle
25 Signals
alt [the signals contain MK_SIGNAL_KILL_ACCEPTED]
26 Suspend all VPP application processes
MGT process of VPP (using IPC)
Kernel (via function calls)

6.4 High Level Operating System (HLOS)

6.4.1 Overview

The HLOS may support one or several Applications. [REQ**131**]

The HLOS may claim conformance to [HLOS02], [HLOS34], [HLOS25], [HLOS14], [HLOS42], and [HLOS93].

6.4.2 HLOS Application

An application running on an HLOS supporting a suitable configuration based on [HLOS02], [HLOS34], [HLOS25], [HLOS14], [HLOS42], and [HLOS93] may claim conformance with [102 221], [102 223], [7816-4], and [102 622].

6.4.3 Remote Application Management

The HLOS may support Remote Application Management of non-native applications running on an application framework (e.g. Java Card) of the HLOS.

7 Minimum Level of Interoperability (MLOI)

Note: Interoperability is the capability to communicate, execute programs, or transfer data among various functional units in a manner that requires the user to have little or no knowledge of the unique characteristics of those units ([ISO 2382]).

Note: Some constants and data types defined in this section are used in other documents defining VPP, e.g. [VFF].

7.1 Basic Data Types

The Primary Platform shall at least be based on 32-bit architecture CPU [REQ**132**]. The endianness is Primary Platform Dependent.

Note: In Table 7-1, XLEN indicates the size in bytes of a memory address. It is a platform dependent value: For 32-bit platforms, XLEN equals 4 bytes; for 64-bit platforms, XLEN equals 8 bytes.

Table 7-1 defines the basic data types used within the VPP Firmware Header.

Table 7-1: Basic Data Types

Type	Description	Size (byte)
uint8_t	8-bit unsigned integer	1
uint16_t	16-bit unsigned integer	2
uint32_t	32-bit unsigned integer	4
uint64_t	64-bit unsigned integer	8
VPP_FRW_TYPE_e	Enumerated VPP Firmware type as defined in Table 7-6	1
memoryType_t	8-bit unsigned integer which can be one of the values in Table 7-9	1
MK_Index_t	Index of an element in a typed array	2
MK_IPC_ID_u	Composite Identifier of an IPC as defined in Table 7-2	2
MK_MAILBOX_ID_u	Composite Identifier of a Mailbox as defined in Table 7-2	2
MK_PROCESS_ID_u	Composite Identifier of a Process as defined in Table 7-2	2
MK_PROCESS_PRIORITY_e	Enumerated type for priority of a Process as defined as defined in Table 7-4	2
MK_LIB_ID_u	Composite Identifier of a shared library as defined in Table 7-2	2
MK_VRE_ID_e	Enumerated VRE Identifier as defined in Table 7-5	4
PPROCESS_Function_t	Memory address of a Process entry point	XLEN
PLLOS_Function_t	Memory address of an LLOS entry point	XLEN
StackType_t	Stack element	XLEN
UUID_t	Universally Unique IDentifier	16
MK_ERROR_e	Enumerated type for errors as defined in Table 7-13	2
MK_EXCEPTION_e	Enumerated type for Exceptions as defined in Table 7-12	2
MK_BITMAP_t	32-bit bitmap for Exception, Signal, or LIB Descriptor conveyor	4
MK_SIGNAL_e	Enumerated Signal type as defined in Table 7-8	4
MK_HANDLE_t	Handle to a Kernel Object	XLEN
MK_TIME_t	Time unsigned 64-bit integer (uint64_t)	8
VPP_SCHEDULING_TYPE_e	Enumerated scheduling type as defined in Table 7-7	1

Table 7-2: Composite Type Identifiers

Composite Type Identifier	Execution Domain type (unsigned integer bit field) Bit [15-14]	Enumerated Identifier (unsigned integer bit field) Bit [13-0]	Description
MK_IPC_ID_u	MK_EXECUTION_DOMAIN_TYPE_e as defined in Table 7-3	MK_IPC_ID_e	Identifier of an IPC
MK_MAILBOX_ID_u		MK_MAILBOX_ID_e	Identifier of a Mailbox
MK_PROCESS_ID_u		MK_PROCESS_ID_e	Identifier of a Process
MK_LIB_ID_u		MK_LIB_ID_e	Identifier of a shared library

Table 7-3: Execution Domain Types MK_EXECUTION_DOMAIN_TYPE_e

Type	Enumerated Identifier Bit [1-0]	Domain	Value
MK_EXECUTION_DOMAIN_TYPE_e	MK_EXECUTION_DOMAIN_TYPE_VPP	VPP Firmware Loader Execution Domain	b10
	MK_EXECUTION_DOMAIN_TYPE_APP	VPP Application Execution Domain	b01

Table 7-4: Priority Values of a Process

Priority	Definition	Value
MK_PROCESS_PRIORITY_LOW	Lowest priority	'0000'
MK_PROCESS_PRIORITY_NORMAL	Normal priority (Default)	'0004'
MK_PROCESS_PRIORITY_HIGH	Highest priority	'0008'
MK_PROCESS_PRIORITY_ERROR	Indicates error in retrieving Process priority	'FFFF'

Table 7-5 defines the interoperable identifiers for accessing hardware resources.[17]

Table 7-5: VRE Identifiers

Identifier	Definition	Value
MK_VRE_AES	Access to the interface of the AES function	'01'
MK_VRE_ECC	Access to the interface of the ECC function	'04'
MK_VRE_RSA	Access to the interface of the RSA function	'08'
MK_VRE_ROT	Access to the interface of the Long-term credentials storage	'10'
MK_VRE_HASH	Access to the interface of the Hash function	'20'
MK_VRE_RNG	Access to the interface of the RNG function	'40'
MK_VRE_RAF	Access to the interface of the Remote Audit Function	'80'
MK_VRE_DOMAIN_BASE	Access to the interfaces of additional Execution Domain hardware functions	'100'

Note: This table enumerates VRE Identifiers for various possibly optional hardware functions.

Table 7-6 details the different types of VPP Firmware handled by the VPP Firmware Loader.

Table 7-6: VPP Firmware Types

Identifier	Definition	Value
FIRMWARE_TYPE_APP	The VPP Firmware for a VPP Application	'01'
FIRMWARE_TYPE_VPP	The VPP Firmware for the Primary Platform excluding the VPP Firmware of the LLOS	'02'
FIRMWARE_TYPE_SYSAPP	The VPP Firmware for a VPP Firmware Loader	'04'
FIRMWARE_TYPE_LLOS	The VPP Firmware for the LLOS	'08'

Table 7-7: Scheduling Types

Identifier	Definition	Value
MK_SCHEDULING_TYPE_COLLABORATIVE	Collaborative scheduling	'01'
MK_SCHEDULING_TYPE_PREEMPTIVE	Pre-emptive scheduling	'02'

[17] VRE usage is hardware/implementation dependent. For example, two Primary Platform implementations may use different RNG hardware and thus require different handling of the MK_VRE_RNG.

Table 7-8: Signal Identifiers

MK_SIGNAL_e	Description	Value
MK_SIGNAL_TIME_OUT	Timeout notification	'00000001'
MK_SIGNAL_ERROR	_mk_Get_Signal function or VRE has generated an error	'00000002'
MK_SIGNAL_EXCEPTION	Notification for an Exception from a child Process	'00000004'
MK_SIGNAL_DOMAIN_BASE_0	Mailbox defined Signals for generic use within the scope of the Execution Domains MK_EXECUTION_DOMAIN_TYPE_VPP and MK_EXECUTION_DOMAIN_TYPE_APP as defined in Table 7-3	'00000008'
MK_SIGNAL_DOMAIN_BASE_1		'00000010'
MK_SIGNAL_DOMAIN_BASE_2		'00000020'
MK_SIGNAL_DOMAIN_BASE_3		'00000040'
MK_SIGNAL_DOMAIN_BASE_4		'00000080'
MK_SIGNAL_DOMAIN_BASE_5		'00000100'
MK_SIGNAL_DOMAIN_BASE_6		'00000200'
MK_SIGNAL_DOMAIN_BASE_7		'00000400'
MK_SIGNAL_DOMAIN_BASE_8		'00000800'
MK_SIGNAL_DOMAIN_BASE_9		'00001000'
MK_SIGNAL_DOMAIN_BASE_10		'00002000'
MK_SIGNAL_DOMAIN_BASE_11		'00004000'
MK_SIGNAL_DOMAIN_BASE_12		'00008000'
MK_SIGNAL_DOMAIN_BASE_13		'00010000'
MK_SIGNAL_DOMAIN_BASE_14		'00020000'
MK_SIGNAL_DOMAIN_BASE_15		'00040000'
MK_SIGNAL_DOMAIN_BASE_16		'00080000'
MK_SIGNAL_DOMAIN_BASE_17		'00100000'
MK_SIGNAL_DOMAIN_BASE_18		'00200000'
MK_SIGNAL_DOMAIN_BASE_19		'00400000'
MK_SIGNAL_DOMAIN_BASE_20		'00800000'
MK_SIGNAL_DOMAIN_BASE_21		'01000000'
MK_SIGNAL_DOMAIN_BASE_22		'02000000'
MK_SIGNAL_DOMAIN_BASE_23		'04000000'
MK_SIGNAL_DOMAIN_BASE_24		'08000000'
MK_SIGNAL_DOMAIN_BASE_25		'10000000'
MK_SIGNAL_DOMAIN_BASE_26		'20000000'
MK_SIGNAL_DOMAIN_BASE_27		'40000000'
MK_SIGNAL_DOMAIN_BASE_28		'80000000'

Table 7-9: Memory Types

Identifier	Definition	Value
MK_MEMORY_TYPE_VOLATILE	Volatile Memory	'01'
MK_MEMORY_TYPE_NON_VOLATILE	Non-volatile Memory	'02'

7.2 Constants and Limits

Table 7-10 defines the constants and limits related to the VPP Firmware Format or a VPP Firmware or the VPP Application as the runtime instance of the VPP Firmware. Other limits may apply to the Primary Platform.

The Primary Platform shall support the Constants and Limits as described in Table 7-10. [REQ**133**]

Primary Platform-dependent values, as described in Table 7-11, shall be provided by the Primary Platform Maker. [REQ**134**]

Table 7-10: Constants and Limits for Any Primary Platform

Name	Description	Value
MK_APP_STOP _GRACEFUL_TICKS	A grace period, in ticks, given to a VPP Application, so it may shut down gracefully	10
MK_IPC_DOMAIN_BASE_ID	Minimum enumerated IPC identifier within the scope of an Execution Domain	'100'
MK_IPC_COM_LENGTH	Length of the IPC identified by MK_IPC_MAIN_COM_ID and MK_IPC_COM_MAIN_ID	4KB
MK_IPC_LIMIT	Maximum number of IPC Descriptors per VPP Firmware	64
MK_IPC_MAX_ID	Maximum enumerated IPC identifier value within the scope of an Execution Domain (including MAX_ID)	'3FFF'
MK_IPC_MGT_LENGTH	Length of the IPC identified by MK_IPC_MAIN_MGT_ID and MK_IPC_MGT_MAIN_ID	6KB
MK_IPC_SIZE_LIMIT	Maximum IPC length	32KB
MK_LIB_DOMAIN_BASE_ID	Minimum enumerated library identifier within the scope of an Execution Domain	'100'
MK_LIB_LIMIT	Maximum number of LIB Descriptors in the VPP Firmware	32
MK_LIB_MAX_ID	Maximum enumerated shared library identifier value within the scope of an Execution Domain (including MAX_ID)	'3FFF'

Name	Description	Value
MK_LIB_CODE_RESERVED_MAX	Maximum size of the reserved memory in the LIB CODE region for NVM technology dependent Programs. This parameter shall be set to 0 for integrated TRE.	< MK_MIN _SUPPORTED _MEMORY _PARTITION _SIZE_NVD
MK_LIB_CONSTANT _RESERVED_MAX	Maximum size of the reserved memory in the LIB CONSTANTS region for NVM technology dependent Programs. This parameter shall be set to 0 for integrated TRE.	< MK_MIN _SUPPORTED _MEMORY _PARTITION _SIZE_NVD
MK_MAILBOX_DOMAIN_BASE_ID	Minimum enumerated Mailbox identifier within the scope of an Execution Domain	'100'
MK_MAILBOX_LIMIT	Maximum number of Mailbox Descriptors per VPP Firmware excluding the kernel Mailbox	64
MK_MAILBOX_MAX_ID	Maximum enumerated Mailbox identifier value within the scope of a domain (including MAX_ID)	'3FFF'
MK_MIN _CONCURRENT_IPC_LIMIT	Minimum number of IPCs accessible concurrently by a Process	6
MK_MIN_STACKS_SUM _SUPPORTED	The minimum size in bytes that the Primary Platform supports for the sum of all stack memory used by all Processes in a VPP Application	24K
MK_MIN_SUPPORTED _MEMORY_PARTITION_SIZE	Minimum size in bytes of Memory Partition (as defined in [VFF]) supported by the Primary Platform	8MB
MK_MIN_SUPPORTED _MEMORY_PARTITION_SIZE_NVD	For Embedded TRE, minimum size in bytes of Non-Volatile Data in the Memory Partition (as defined in [VFF]) supported by the Primary Platform	1MB
MK_MIN_SUPPORTED _MEMORY_PARTITION_SIZE_VD	For Embedded TRE, minimum size in bytes of Volatile Data in the Memory Partition (as defined in [VFF]) supported by the Primary Platform	64KB
MK_MIN_VIRTUAL _MEMORY_SIZE[18]	Minimum size of the Virtual Memory that the MMF shall manage	1KB
MK_PROCESS_DOMAIN_BASE_ID	Minimum enumerated Process identifier within the scope of an Execution Domain	'100'
MK_PROCESS_LIMIT	Maximum number of Process Descriptors in the VPP Firmware	32

[18] This information allows the VPP Application designer to prevent some software side channel attacks.

Name	Description	Value
MK_MAX_PROCESS_ID	Maximum enumerated Process identifier value within the scope of an Execution Domain	'3FFF'
MK_MIN_SUPPORTED_STACK	Minimum stack size supported for a Process, given in StackType_t units	512 StackType_t units (2KB if 32-bit)
MK_ENDLESS	Value indicating that a Kernel Object shall not time out but shall wait until a response is received	2^{32}-1

Table 7-11: Primary Platform Dependent Constants and Limits

Name	Description
MK_MEMORY_PARTITION_SIZE	The size in bytes of the Memory Partition. Shall be greater than MK_MIN_SUPPORTED_MEMORY _PARTITION_SIZE and shall be equal to the sum of MK_MEMORY_PARTITION_SIZE_NVD and MK_MEMORY_PARTITION_SIZE_VD.
MK_MEMORY_PARTITION_SIZE_NVD	For Embedded TRE, the size in bytes of the Memory Partition containing Non-Volatile Data. Shall be greater than MK_MIN_SUPPORTED_MEMORY _PARTITION_SIZE_NVD.
MK_MEMORY_PARTITION_SIZE_VD	For Embedded TRE, the size in bytes of the Memory Partition containing Volatile Data. Shall be greater than MK_MIN_SUPPORTED_MEMORY _PARTITION_SIZE_VD.
MK_AVERAGE_COMMIT_TIME	Average NVM transaction time (tick unit).
MK_BEGIN_VSPACE_NVD	Virtual memory address of the beginning of the Virtual Address Space for Non-Volatile Data.
MK_BEGIN_VSPACE_VD	Virtual memory address of the beginning of the Virtual Address Space for Volatile Data.
MK_IS_LITTLE_ENDIAN	0 – false 1 – true
MK_IS_PREEMPTIVE _SCHEDULING_SUPPORTED	Define the type of scheduling being supported for VPP Applications. 0 – false 1 – true
MK_MAX_STACKS_SUM_SUPPORTED	The maximum size in bytes that the Primary Platform supports for the sum of all stack memory used by all Processes in a VPP Application.
MK_MAX_CONTEXT_SWITCH_TIME	Maximum time in ticks for switching between two Processes.
MK_MAX_RAF_TIME_MS	Maximum time in milliseconds for performing a Remote Audit Function operation.
MK_NO_PREEMPTION_MAX_TIMEOUT	Maximum time in ticks during which the VPP Application Process cannot be pre-empted by a VPP Process. 0 if _mk_Request_No_Preemption is not supported.
MK_PERFORMANCE_INDEX_METHOD	Performance benchmark method, to be declared by Primary Platform Maker.
MK_PERFORMANCE_INDEX	The performance of the Primary Platform based on the given benchmark method.
MK_MS_PER_TICK	Value of the kernel tick time in milliseconds.
MK_VRE_LIMIT	Maximum number of VREs that could be accessed simultaneously.

Name	Description
MK_VSPACE_REGION_SIZE	The size in bytes of the Virtual Address Space Region. Shall be equal to or greater than MK_MEMORY_PARTITION_SIZE.
MK_VSPACE_REGION_SIZE_NVD	For Embedded TRE, the size in bytes of the Virtual Address Space Region containing Non-Volatile Data. Shall be equal to or greater than MK_MEMORY_PARTITION_SIZE_NVD.
MK_VSPACE_REGION_SIZE_VD	For Embedded TRE, the size in bytes of the Virtual Address Space Region containing Volatile Data. Shall be equal to or greater than MK_MEMORY_PARTITION_SIZE_VD.

7.3 Errors and Exceptions

Table 7-12 defines the Exception values.

Table 7-12: Exceptions

Exception Name	Description	Rank Value
MK_EXCEPTION_ERROR	An error has occurred in a child of the Process.	0
MK_EXCEPTION_SEVERE	A severe Exception has occurred (e.g. memory violation).	1
MK_EXCEPTION _CHILD_PROCESS_DIED	A child Process has died.	2
MK_EXCEPTION_HANDLE _NOT_A_PROCESS	A pseudo exception, indicating that the handle passed as parameter is invalid; e.g. does not refer to a Process.	15
MK_EXCEPTION_VENDOR_BASE	Start of the vendor specific exceptions.	16
MK_EXCEPTION_MAX	Maximum Exception rank value allowed. This rank value cannot be assigned by any implementation.	31

The Exception type is MK_EXCEPTION_e as an enumerated value representing the bit rank (power of 2) within a 32-bit bitmap (MK_BITMAP_t).

Table 7-13 defines the error values.

Table 7-13: Errors

Error Name	Description	LSB Value
MK_ERROR_NONE	No error	0
MK_ERROR_UNKNOWN_UUID	Unknown UUID	1
MK_ERROR_SEVERE	Severe error	2
MK_ERROR_ILLEGAL_PARAMETER	Illegal parameter	3
MK_ERROR_UNKNOWN_ID	Unknown identifier	4
MK_ERROR_UNKNOWN_HANDLE	Unknown Handle	5
MK_ERROR_UNKNOWN_PRIORITY	Unknown priority	6
MK_ERROR_ACCESS_DENIED	Access denied	7
MK_ERROR_INTERNAL	Internal error	8
MK_ERROR_HANDLE_NOT_ACCESSED	Handle not accessed	9
MK_ERROR_GET_ERROR_HANDLE _NOT_A_PROCESS	Error specific to function _mk_Get_Error	10
MK_ERROR_IPC_LIMIT_REACHED	IPC concurrency limit reached	11
MK_ERROR_VENDOR_BASE	Reserved for VPP implementation-specific	32
MK_ERROR_MAX	Maximum error rank value. This rank value cannot be assign by any implementation.	255

The Error type is MK_ERROR_e (16 bit) where the eight most significant bits (MSBs) are the complementary bits of the eight least significant bits (LSBs).

7.4 Cross-Execution-Domain Identifiers

Table 7-14 defines the Cross-Execution-Domain Composite Identifiers.

Table 7-14: Cross-Execution-Domain Composite Identifier

Identifiers	Domain Type Bit [15-14]	Enumerated Identifier Bit [13-0]	Description
MK_PROCESS_COM_VPP_ID	MK_EXECUTION_DOMAIN_TYPE_VPP	0	VPP COM Process
MK_PROCESS_MGT_VPP_ID	MK_EXECUTION_DOMAIN_TYPE_VPP	1	MGT Process
MK_PROCESS_MAIN_APP_ID	MK_EXECUTION_DOMAIN_TYPE_APP	0	MAIN Process
MK_MAILBOX_COM_MAIN_ID	MK_EXECUTION_DOMAIN_TYPE_VPP	0	COM Process Mailbox (receiver: MAIN Process)
MK_MAILBOX_MGT_MAIN_ID	MK_EXECUTION_DOMAIN_TYPE_VPP	1	MGT Process Mailbox (receiver: MAIN Process)
MK_MAILBOX_MAIN_COM_ID	MK_EXECUTION_DOMAIN_TYPE_APP	0	MAIN Process Mailbox (receiver: COM Process)
MK_MAILBOX_MAIN_MGT_ID	MK_EXECUTION_DOMAIN_TYPE_APP	1	MAIN Process Mailbox (receiver: MGT Process)
MK_IPC_MAIN_COM_ID	MK_EXECUTION_DOMAIN_TYPE_APP	0	IPC from the MAIN Process to the COM Process
MK_IPC_COM_MAIN_ID	MK_EXECUTION_DOMAIN_TYPE_VPP	0	IPC from the COM Process to the MAIN Process
MK_IPC_MAIN_MGT_ID[19]	MK_EXECUTION_DOMAIN_TYPE_APP	1	IPC from the MAIN Process to the MGT Process
MK_IPC_MGT_MAIN_ID[19]	MK_EXECUTION_DOMAIN_TYPE_VPP	1	IPC from the MGT Process to the MAIN Process

Cross-Execution-Domain IPC Descriptors and Mailbox Descriptors are automatically instantiated by the kernel. As such, they cannot be defined in the VPP Firmware. Their ID and the IPC size are fixed.

[19] The IPCs between MAIN and MGT Processes are valid for the VPP Firmware Loader only.

7.5 Cross-Execution-Domain Signals

Table 7-15 defines the Cross-Execution-Domain Signals.

Table 7-15: Cross-Execution-Domain Signals

Identifiers	Value	Description
MK_SIGNAL_IPC_UPDATED	MK_SIGNAL_DOMAIN_BASE_0	The IPC updated.
MK_SIGNAL_KILL_REQUESTED	MK_SIGNAL_DOMAIN_BASE_1	MGT signaled the MAIN Process to terminate itself.
MK_SIGNAL_KILL_ACCEPTED	MK_SIGNAL_DOMAIN_BASE_1	The MAIN Process signaled MGT that it has accepted the kill request.
MK_SIGNAL_APP_RESTART	MK_SIGNAL_DOMAIN_BASE_2	The MAIN Process signaled MGT to restart the VPP Application.
MK_SIGNAL_KILL_ITSELF	MK_SIGNAL_DOMAIN_BASE_3	The MAIN Process committed suicide.

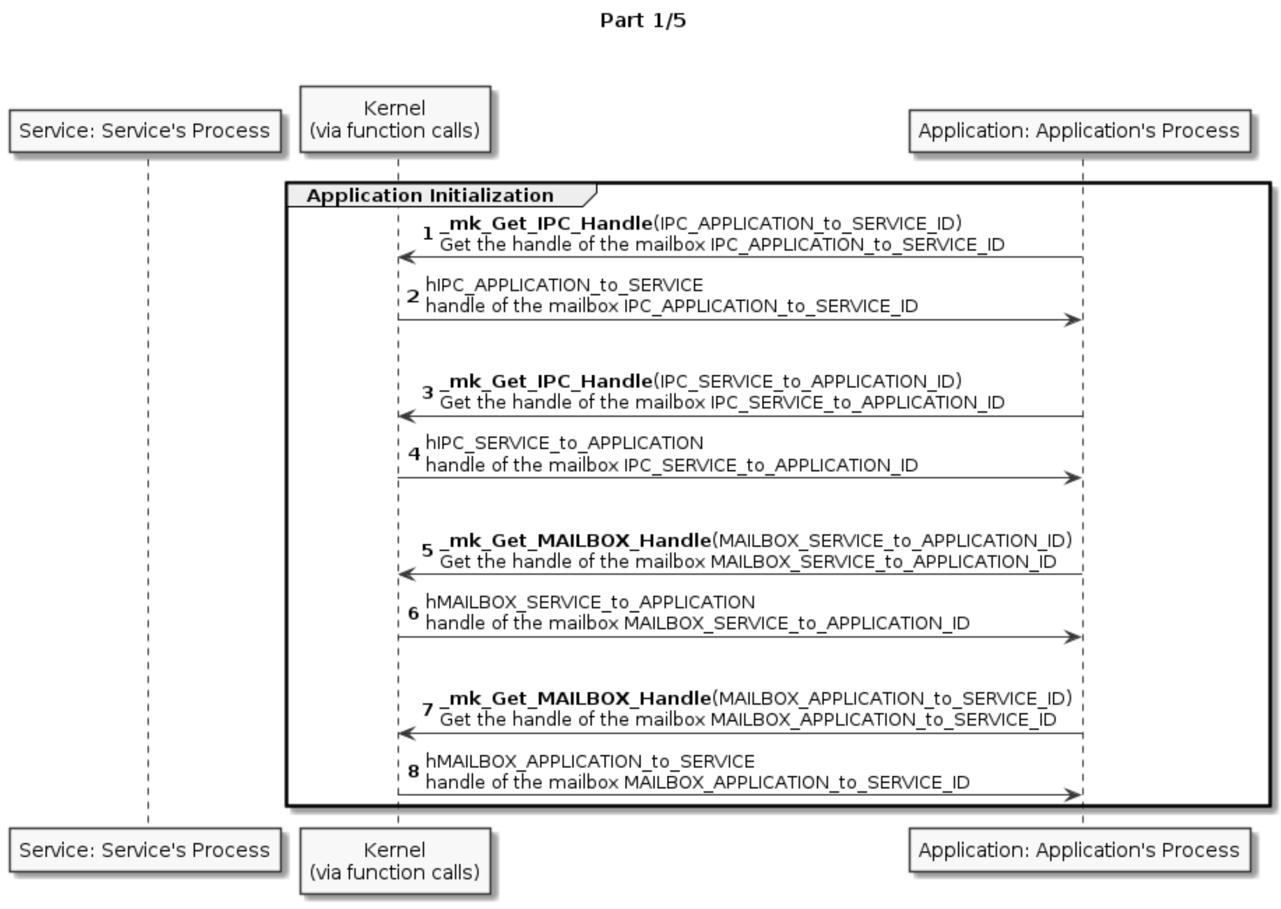
As a Mailbox has only a single reader and writer, Cross-Execution-Domain Signals use the same values when different Signals are used by different Mailboxes. For example, MK_SIGNAL_KILL_REQUESTED has the same value as MK_SIGNAL_KILL_ACCEPTED, but they are being used on different mailboxes.

Annex A Services

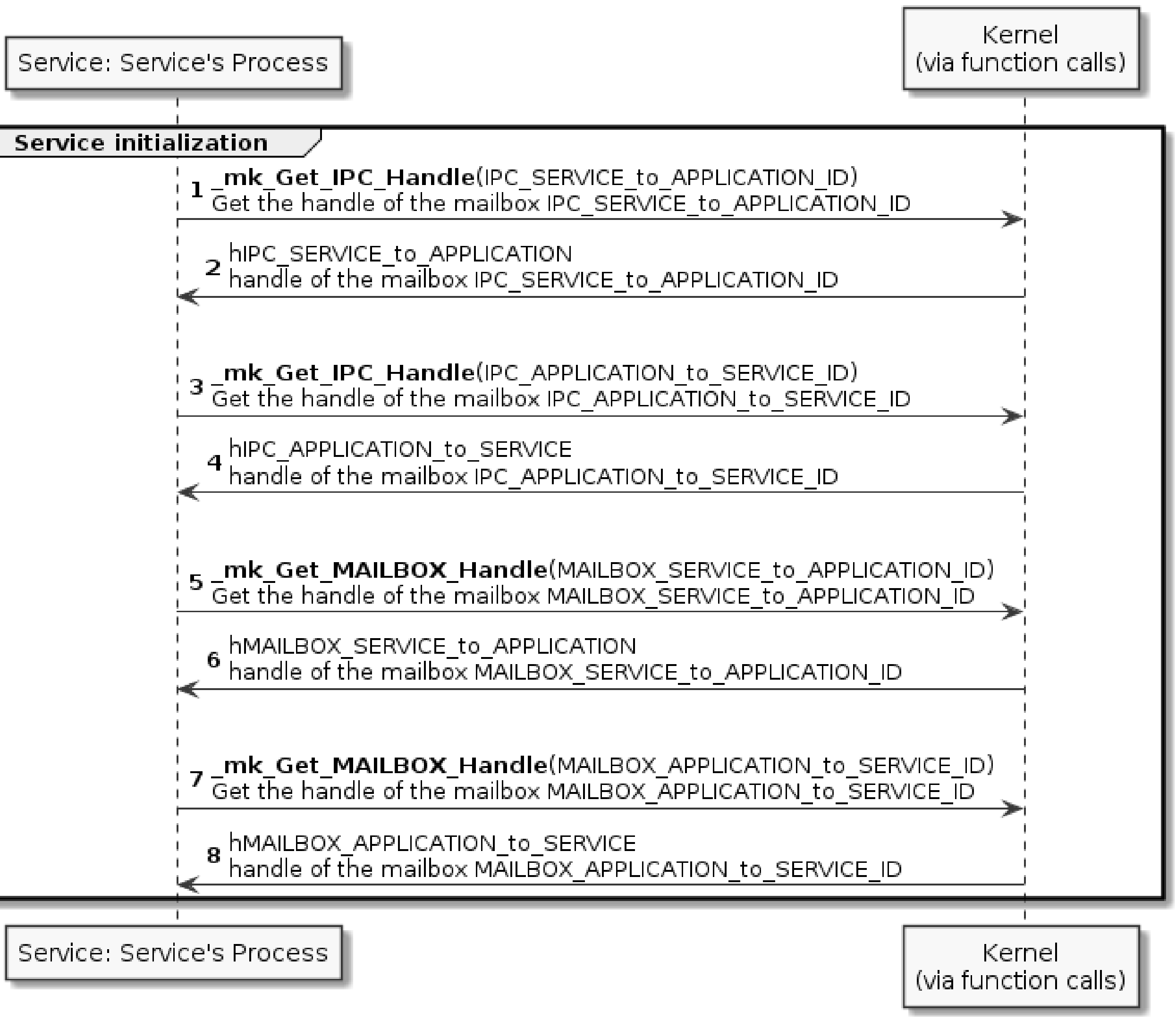
A.1 Service Generic Message Flow

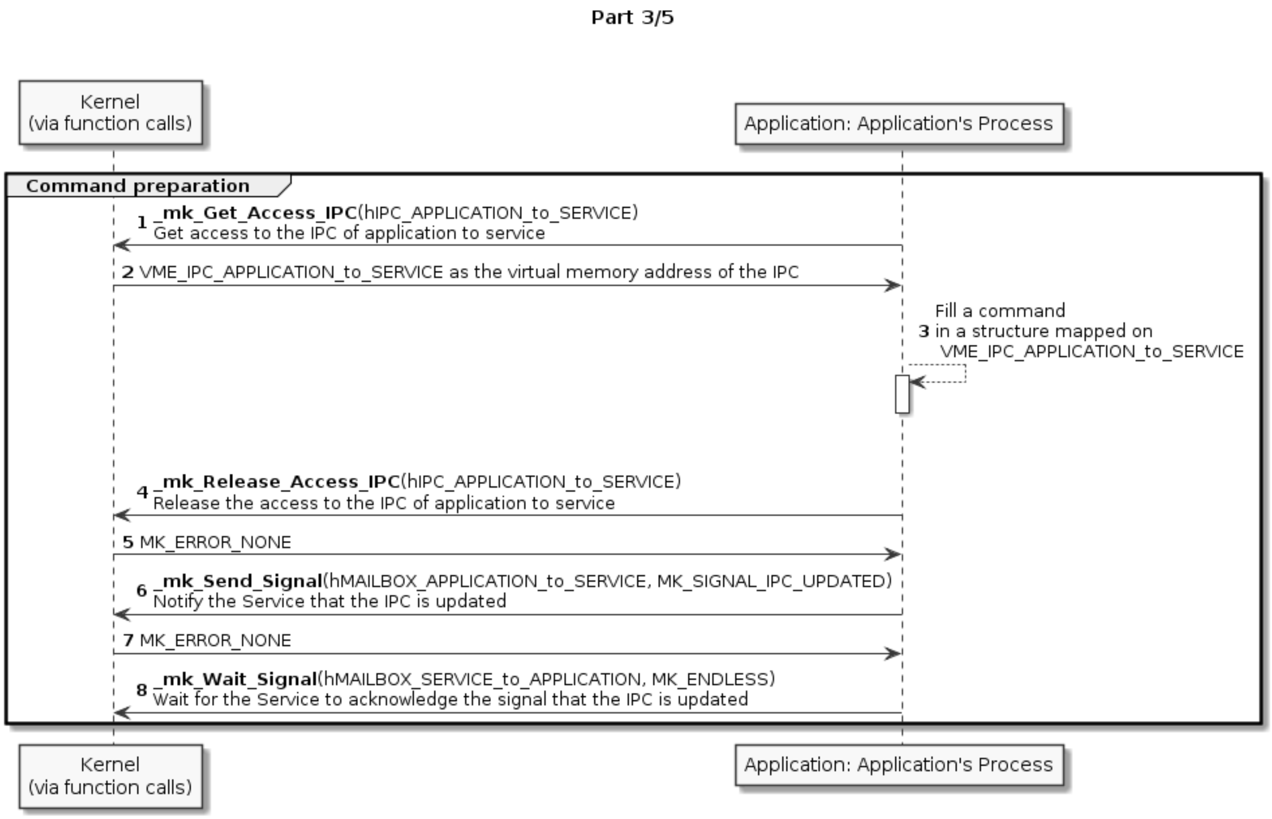
Figure A-1 illustrates a generic data flow between a service providing a function and an Application consuming that function.

Figure A-1: Generic Data Flow

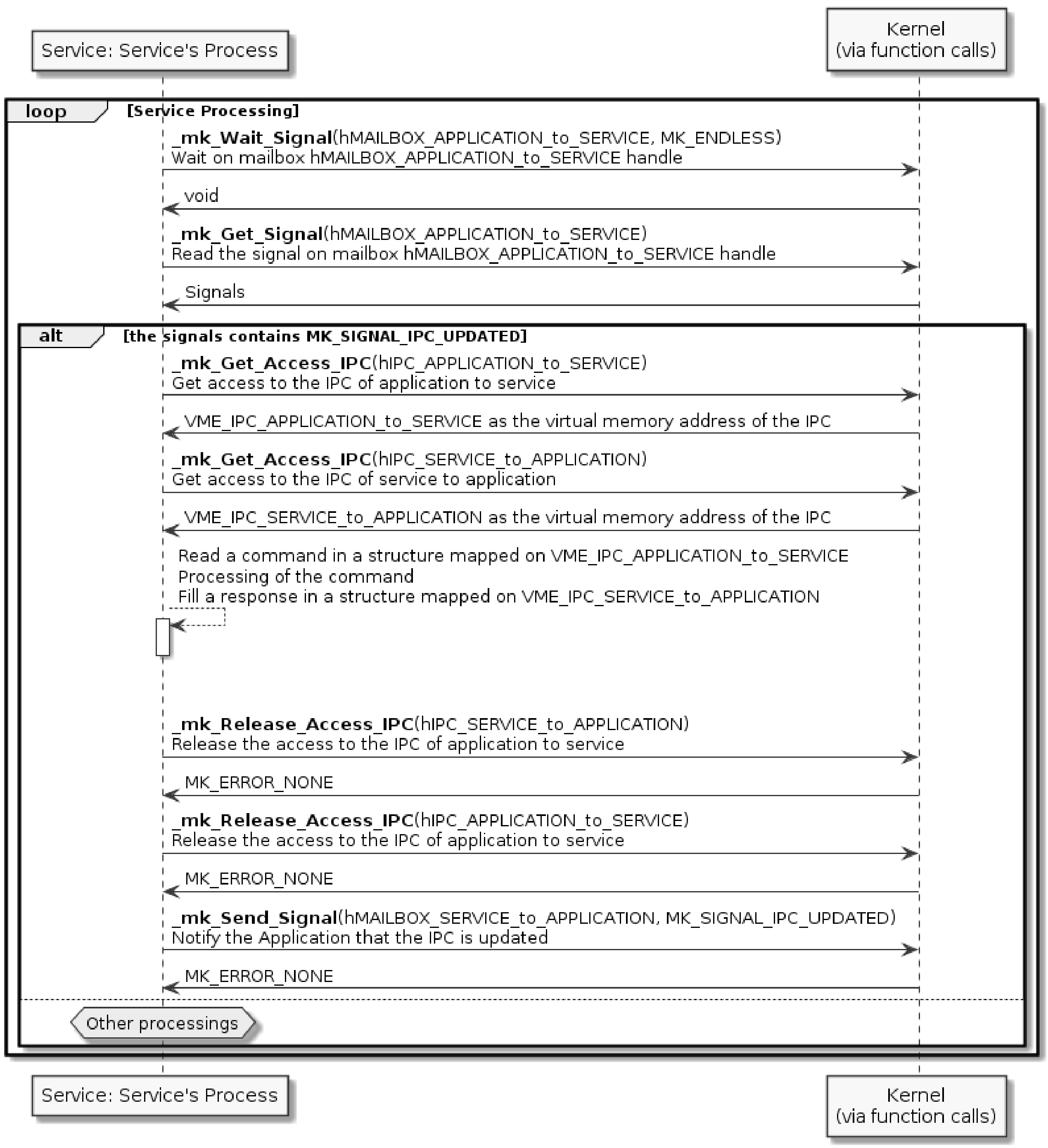

Part 1/5
Service: Service's Process
Kernel (via function calls)
Application: Application's Process
Application Initialization
1 _mk_Get_IPC_Handle(IPC_APPLICATION_to_SERVICE_ID) Get the handle of the mailbox IPC_APPLICATION_to_SERVICE_ID
2 hIPC_APPLICATION_to_SERVICE handle of the mailbox IPC_APPLICATION_to_SERVICE_ID
3 _mk_Get_IPC_Handle(IPC_SERVICE_to_APPLICATION_ID) Get the handle of the mailbox IPC_SERVICE_to_APPLICATION_ID
4 hIPC_SERVICE_to_APPLICATION handle of the mailbox IPC_SERVICE_to_APPLICATION_ID
5 _mk_Get_MAILBOX_Handle(MAILBOX_SERVICE_to_APPLICATION_ID) Get the handle of the mailbox MAILBOX_SERVICE_to_APPLICATION_ID
6 hMAILBOX_SERVICE_to_APPLICATION handle of the mailbox MAILBOX_SERVICE_to_APPLICATION_ID
7 _mk_Get_MAILBOX_Handle(MAILBOX_APPLICATION_to_SERVICE_ID) Get the handle of the mailbox MAILBOX_APPLICATION_to_SERVICE_ID
8 hMAILBOX_APPLICATION_to_SERVICE handle of the mailbox MAILBOX_APPLICATION_to_SERVICE_ID
Service: Service's Process
Kernel (via function calls)
Application: Application's Process

Part 2/5

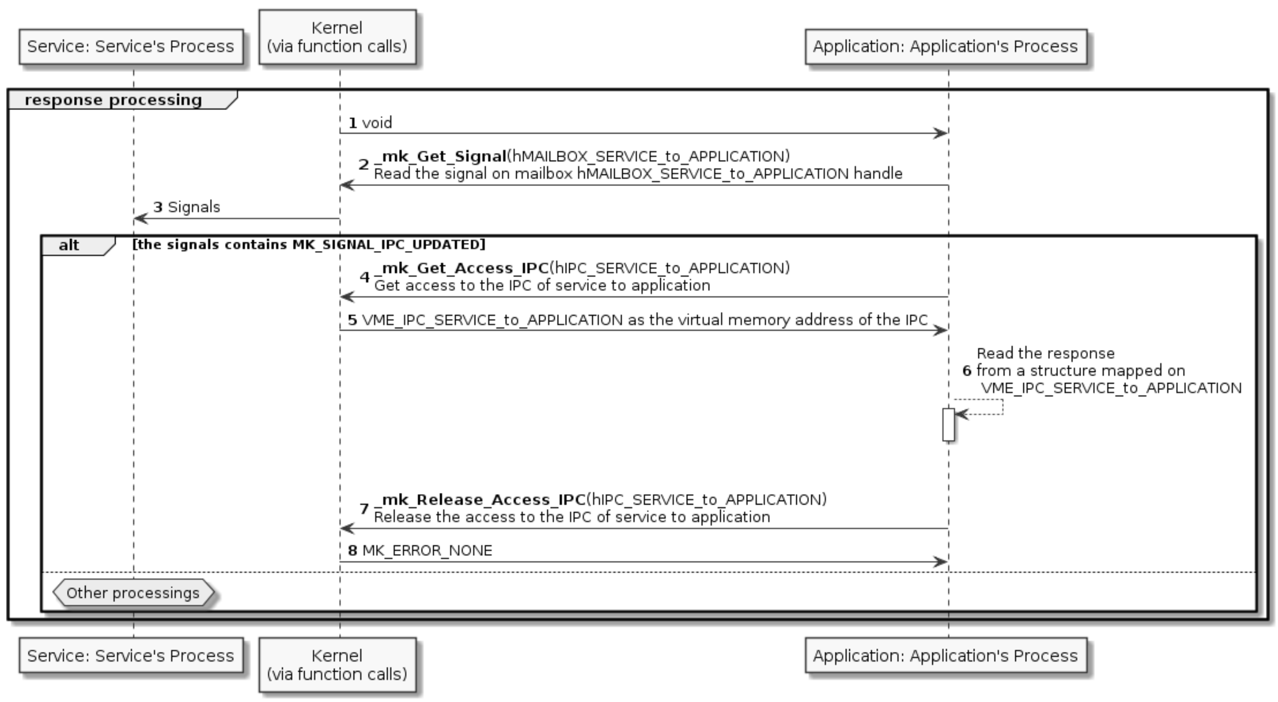

Service: Service's Process
Kernel (via function calls)
Service initialization
1 _mk_Get_IPC_Handle(IPC_SERVICE_to_APPLICATION_ID) Get the handle of the mailbox IPC_SERVICE_to_APPLICATION_ID
2 hIPC_SERVICE_to_APPLICATION handle of the mailbox IPC_SERVICE_to_APPLICATION_ID
3 _mk_Get_IPC_Handle(IPC_APPLICATION_to_SERVICE_ID) Get the handle of the mailbox IPC_APPLICATION_to_SERVICE_ID
4 hIPC_APPLICATION_to_SERVICE handle of the mailbox IPC_APPLICATION_to_SERVICE_ID
5 _mk_Get_MAILBOX_Handle(MAILBOX_SERVICE_to_APPLICATION_ID) Get the handle of the mailbox MAILBOX_SERVICE_to_APPLICATION_ID
6 hMAILBOX_SERVICE_to_APPLICATION handle of the mailbox MAILBOX_SERVICE_to_APPLICATION_ID
7 _mk_Get_MAILBOX_Handle(MAILBOX_APPLICATION_to_SERVICE_ID) Get the handle of the mailbox MAILBOX_APPLICATION_to_SERVICE_ID
8 hMAILBOX_APPLICATION_to_SERVICE handle of the mailbox MAILBOX_APPLICATION_to_SERVICE_ID
Service: Service's Process
Kernel (via function calls)


Part 3/5
Kernel (via function calls)
Application: Application's Process
Command preparation
1 _mk_Get_Access_IPC(hIPC_APPLICATION_to_SERVICE) Get access to the IPC of application to service
2 VME_IPC_APPLICATION_to_SERVICE as the virtual memory address of the IPC
3 Fill a command in a structure mapped on VME_IPC_APPLICATION_to_SERVICE
4 _mk_Release_Access_IPC(hIPC_APPLICATION_to_SERVICE) Release the access to the IPC of application to service
5 MK_ERROR_NONE
6 _mk_Send_Signal(hMAILBOX_APPLICATION_to_SERVICE, MK_SIGNAL_IPC_UPDATED) Notify the Service that the IPC is updated
7 MK_ERROR_NONE
8 _mk_Wait_Signal(hMAILBOX_SERVICE_to_APPLICATION, MK_ENDLESS) Wait for the Service to acknowledge the signal that the IPC is updated
Kernel (via function calls)
Application: Application's Process

part 4/5


Service: Service's Process
Kernel
(via function calls)
loop
[Service Processing]
_mk_Wait_Signal(hMAILBOX_APPLICATION_to_SERVICE, MK_ENDLESS)
Wait on mailbox hMAILBOX_APPLICATION_to_SERVICE handle
void
_mk_Get_Signal(hMAILBOX_APPLICATION_to_SERVICE)
Read the signal on mailbox hMAILBOX_APPLICATION_to_SERVICE handle
Signals
alt
[the signals contains MK_SIGNAL_IPC_UPDATED]
_mk_Get_Access_IPC(hIPC_APPLICATION_to_SERVICE)
Get access to the IPC of application to service
VME_IPC_APPLICATION_to_SERVICE as the virtual memory address of the IPC
_mk_Get_Access_IPC(hIPC_SERVICE_to_APPLICATION)
Get access to the IPC of service to application
VME_IPC_SERVICE_to_APPLICATION as the virtual memory address of the IPC
Read a command in a structure mapped on VME_IPC_APPLICATION_to_SERVICE
Processing of the command
Fill a response in a structure mapped on VME_IPC_SERVICE_to_APPLICATION
_mk_Release_Access_IPC(hIPC_SERVICE_to_APPLICATION)
Release the access to the IPC of application to service
MK_ERROR_NONE
_mk_Release_Access_IPC(hIPC_APPLICATION_to_SERVICE)
Release the access to the IPC of application to service
MK_ERROR_NONE
_mk_Send_Signal(hMAILBOX_SERVICE_to_APPLICATION, MK_SIGNAL_IPC_UPDATED)
Notify the Application that the IPC is updated
MK_ERROR_NONE
Other processings
Service: Service's Process
Kernel
(via function calls)

part 5/5


Service: Service's Process
Kernel (via function calls)
Application: Application's Process
response processing
1 void
2 _mk_Get_Signal(hMAILBOX_SERVICE_to_APPLICATION) Read the signal on mailbox hMAILBOX_SERVICE_to_APPLICATION handle
3 Signals
alt [the signals contains MK_SIGNAL_IPC_UPDATED]
4 _mk_Get_Access_IPC(hIPC_SERVICE_to_APPLICATION) Get access to the IPC of service to application
5 VME_IPC_SERVICE_to_APPLICATION as the virtual memory address of the IPC
6 Read the response from a structure mapped on VME_IPC_SERVICE_to_APPLICATION
7 _mk_Release_Access_IPC(hIPC_SERVICE_to_APPLICATION) Release the access to the IPC of service to application
8 MK_ERROR_NONE
Other processings

Annex B VPP for Embedded Devices (Informative)

VPP can be implemented on Integrated TREs, Embedded TREs, or TREs that are in between. For simplicity, in this annex the Embedded TRE is considered to be a device that does not rely on volatile / non-volatile memory residing outside of the TRE.

The above difference may lead to different VPP implementations on Embedded TREs; the following sections introduce a few examples of such adaptations.

B.1 NVM Management

Non-Volatile Memory in Embedded TRE can be implemented by different technologies, but for simplicity it is assumed that the NVM technology is Flash.

Flash management in Embedded TRE requires specific operations due to the physical nature of the internal Flash and to the Harvard architecture sometimes present in Embedded TRE.

In Harvard architecture, addressing spaces of *Code* and *Data* are separated. Typically, the Flash memory is assigned to the Code address space, while the Volatile Memory is assigned to the Data space. In specific Embedded TREs, this limitation applies both to physical and virtual address space. This implies that specific data cannot be moved from Flash to RAM (or vice-versa) keeping the same virtual address

As an Embedded TRE has a limited internal Volatile Memory, it could be useful for VPP to move data from Flash to RAM and vice versa. For example, to increase the amount of RAM visible to the VPP Application beyond the physical memory, RAM Virtual Memory in Flash can be used. To perform the Virtualization, copy operations need to be performed from RAM to Flash, but as they involve access to VPP application data (AG_P_OU), these copy operations need to be performed in unprivileged mode. An implementation option could be to have the copy operation as part of a library (LIB CODE), in order to be executed in unprivileged mode. The library would be provided by the Primary Platform Issuer and, therefore, the interface between the library and the LLOS, and between the VPP Application and the library are out of scope of the document. Another option to implement such a mechanism could be to define a VPP Process to transfer the corresponding memory from Volatile to Non-Volatile Memory. Finally, other implementations might rely on dedicated hardware to perform the transfer operation (like DMA), etc.

For the internal NVM, the physical update of Flash memory requires some operations that are strictly connected to the hardware, such as performing a back-up of a memory page, erasing the memory page, updating the memory page, etc. All those operations typically require access to both the hardware resources (that can be performed through the VRE) and to the VPP Application data (that needs to be done in unprivileged mode). Such operations may also be provided by the VPP implementation and loaded in the LIB CODE space at VPP Application execution time.

B.2 Multiple VPP Applications

Certain Embedded TRE Use Cases require that multiple VPP Applications each maintain their own state.

For example, one VPP Application may manage the Telecom application while another manages NFC transactions; in this scenario, when an NFC transaction is received – typically with strict timing requirements – the Telecom application may be pre-empted in order to serve the NFC transaction; however, after the NFC transaction is completed, the Telecom application should be restored with the same context. The pre-emption mechanism in such a scenario is hence required and can pre-empt all the Processes of a VPP Application, such that several Processes from several VPP Applications are in 'Suspended' state.

To allow this, multiple Virtual Primary Platforms will be present in the TRE, as indicated in section 4. Note that to allow the co-existence of multiple VPP Applications, the TRE must have several Virtual Primary Platforms, as each Virtual Primary Platform shall contain only one VPP Application. As indicated in section 5.3, TRE SP shall contain only one VPP Application SP. However, this does not mean that several physical Embedded TREs are required; rather, when a VPP Application is in execution, the VPP Application can access only its own data and each Virtual Primary Platform can access only data related to its specific VPP Application. The two VPP Applications may also share the same physical memory space but secure mechanisms and memory protection mechanisms shall prevent unauthorized accesses as indicated above. The same isolation is requested also for other resources, such as Virtual Registers, etc.