OCP Layered Open-Source Cryptographic Key-management (L.O.C.K.)

NVMe™ Key Management Block

Revision 0.8

Version 0.8

Date: April 2025

Contributors

Google	Microsoft	Samsung	Solidigm	Kioxia
Andrés Lagar-Cavilla Amber Huffman Charles Kuzman Jeff Andersen Chris Sabol Srini Narayanamurthy	Lee Prewitt Michael Norris Eric Eilertson Bryan Kelly Anjana Parthasarathy Ben Keen Bharat Pillilli	Jisoo Kim Gwangbae Choi Eric Hibbard Mike Allison	Scott Shadley Gamil Cain Festus Hategekimana	John Geldman Fred Knight Paul Suhler James Borden

Revision Table

Date	Revision #	Author	Description
September 2024	0.5	Authoring Companies	Initial proposal draft based on work from the list of contributors
April 2025	0.7	Authoring Companies	Updates that include updates APIs, UML Sequence diagrams, and racheting with fuses

License

Open Web Foundation (OWF) CLA

Notes:

1. The above license does not apply to the Appendix or Appendices. The information in the Appendix or Appendices is for reference only and non-normative in nature.

NOTWITHSTANDING THE FOREGOING LICENSES, THIS SPECIFICATION IS PROVIDED BY OCP "AS IS" AND OCP EXPRESSLY DISCLAIMS ANY WARRANTIES (EXPRESS, IMPLIED, OR OTHERWISE), INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR A PARTICULAR PURPOSE, OR TITLE, RELATED TO THE SPECIFICATION. NOTICE IS HEREBY GIVEN, THAT OTHER RIGHTS NOT GRANTED AS SET FORTH ABOVE, INCLUDING WITHOUT LIMITATION, RIGHTS OF THIRD PARTIES WHO DID NOT EXECUTE THE ABOVE LICENSES, MAY BE IMPLICATED BY THE IMPLEMENTATION OF OR COMPLIANCE WITH THIS SPECIFICATION. OCP IS NOT RESPONSIBLE FOR IDENTIFYING RIGHTS FOR WHICH A LICENSE MAY BE REQUIRED IN ORDER TO IMPLEMENT THIS SPECIFICATION. THE ENTIRE RISK AS TO IMPLEMENTING OR OTHERWISE USING THE SPECIFICATION IS ASSUMED BY YOU. IN NO EVENT WILL OCP BE LIABLE TO YOU FOR ANY MONETARY DAMAGES WITH RESPECT TO ANY CLAIMS RELATED TO, OR ARISING OUT OF YOUR USE OF THIS SPECIFICATION, INCLUDING BUT NOT LIMITED TO ANY LIABILITY FOR LOST PROFITS OR ANY CONSEQUENTIAL, INCIDENTAL, INDIRECT, SPECIAL OR PUNITIVE DAMAGES OF ANY CHARACTER FROM ANY CAUSES OF ACTION OF ANY KIND WITH RESPECT TO THIS SPECIFICATION, WHETHER BASED ON BREACH OF CONTRACT, TORT (INCLUDING NEGLIGENCE), OR OTHERWISE, AND EVEN IF OCP HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Acknowledgements

The Contributors of this Specification would like to acknowledge the following companies for their feedback:

Compliance with OCP Tenets

Please describe how this Specification complies to the following OCP tenets. Compliance is required for at least three of the four tenets. The ideals behind open sourcing stipulate that everyone benefits when we share and work together. Any open source project is designed to promote sharing of design elements with peers and to help them understand and adopt those contributions. There is no purpose in sharing if all parties aren't aligned with that philosophy. The IC will look beyond the contribution for evidence that the contributor is aligned with this philosophy. The contributor actions, past and present, are evidence of alignment and conviction to all the tenets.

Openness

OCP L.O.C.K source for RTL and firmware will be licensed using the Apache 2.0 license. The specific mechanics and hosting of the code are work in progress due to CHIPS alliance timelines. Future versions of this spec will point to the relevant resources.

Efficiency

OCP L.O.C.K. is used to generate and load keys for use of encrypting user data prior to storing data at rest and decrypting stored user data at rest when read. So, it cannot yield a measurable impact on system efficiency.

Impact

OCP L.O.C.K. enables consistency and transparency to a foundational area of security of media encryption keys such that no firmware in the device ever has access to a media encryption key. Furthermore, no decrypted media encryption key exists in the device when power is removed from the device.

Scale

OCP L.O.C.K. is a committed intercept for Cloud silicon for Google and Microsoft. This scale covers both a significant portion of the Cloud market in hyperscale and enterprise.

Sustainability

The goal of OCP L.O.C.K. is to eliminate the need to destroy storage devices (e.g., SSDs) in the Cloud market by providing a mechanism that increases the confidence that a media encryption key within the device is deleted in a crypto-erase. This enables repurposing the device and or components on the device at end of use or end of life. Given the size of the Cloud market this provides a significant reduction of e-waste.

Introduction

OCP L.O.C.K. (Layered Open-source Cryptographic Key management) is a feature set conditionally compiled into Caliptra, which provides secure key management for Data-At-Rest protection in self-encrypting storage devices.

OCP L.O.C.K. was originally created as part of the Open Compute Project (OCP). The major revisions of the OCP L.O.C.K. specifications are published as part of Caliptra at OCP as OCP L.O.C.K. is an optional extension to Caliptra. The evolving source code and documentation for Caliptra are in the repository within the CHIPS Alliance Project, a Series of LF Projects, LLC.

Background

OCP L.O.C.K is being defined to improve drive security. The life of a storage device in a datacenter is that the device leaves the supplier, a customer writes user data to the device, and then the device is decommissioned. The problem is that customer data is not allowed to leave the data center. There needs to be a high confidence that the storage device leaving the datacenter is secure. The current default cloud service provider (CSP) policy to ensure this level of security is to destroy the drive. Other policies may exist that leverage drive capabilities (e.g., Sanitize), but are not generally deemed inherently trustworthy by these CSPs[^1]. This produces significant e-waste and inhibits any re-use/recycling.

OCP L.O.C.K. is solving this security issue with data encryption by defining a fuse-backed storage root key used to protect all media encryption keys used to encrypt data stored on the device. If that storage root key is deleted, then all prior media encryption keys are unable to be recovered. If that entropy is deleted, then the media encryption key is unable to be generated to decrypt the data on that storage device (i.e., no access to the plaintext behind the ciphertext).

OCP L.O.C.K. is addressing these issues by:

- Preventing leakage of media keys via firmware vulnerabilities or side channels;

- Binding media keys to a set of securely-provisioned access keys; and
- Enabling attestable “hard” purge via erasure of fuse secrets.

OCP L.O.C.K. goals

The goal of OCP L.O.C.K. is to define a Key Management Block (KMB) that:

- Isolates storage keys to a trusted hardware block
- Binds storage keys to a given set of externally-supplied access keys
- Provides replay-resistant transport security for these access keys such that they can be injected without trusting the host
- Manages a fuse-backed storage root key for sanitization
- Is able to be used in conjunction with the Opal[^2] and Key Per I/O[^3] storage device specifications

Overview

Self-encrypting drives (SEDs) store data encrypted to media encryption keys (MEKs). SEDs include the following building blocks:

- The storage media that holds data at rest.
- An Encryption Engine that performs line-rate encryption and decryption of data as it enters and exits the drive.
- A controller that manages the lifecycle of MEKs.

MEKs may be bound to user credentials, which the host must provide to the drive in order for the associated data to be readable. A given MEK may be bound to one or more credentials. This model is captured in the TCG Opal specification.

MEKs, or other keys from which MEKs are derived, may be injected into the drive and tagged with address range metadata, such that subsequent I/Os which target that address range will be encrypted to that injected MEK. This model is captured in the TCG Key Per I/O (KPIO) specification.

MEKs may be securely erased, to effectively erase all data which was encrypted to the MEK. To erase an MEK, it is sufficient for the controller to erase the MEK itself or a key from which it was derived.

In an SED that takes Caliptra with OCP L.O.C.K. features enabled, Caliptra will act as a Key Management Block (KMB). The KMB will be the only entity that can read MEKs and program them into the SED's cryptographic engine. The KMB will expose services to controller firmware which will allow the controller to transparently manage each MEK's lifecycle, without being able to access the raw MEK itself.

Threat model

The protected asset is the user data stored at rest on the drive. The adversary profile extends up to nation-states in terms of capabilities.

Adversary capabilities include:

- Interception of a storage device in the supply chain.
- Theft of a storage device from a data center.

- Destructively inspecting a stolen device.
- Running arbitrary firmware on a stolen device.
  - This includes attacks where vendor firmware signing keys have been compromised.
- Attempting to glitch execution of code running on general-purpose cores.
- Stealing debug core dumps or UART/serial logs from a device while it is operating in a data center, and later stealing the device.
- Gaining access to any class secrets, global secrets, or symmetric secrets shared between the device and an external entity.
- Executing code within a virtual machine on a multi-tenant host offered by the cloud service provider which manages an attached storage device.
- Accessing all device design documents, code, and RTL.

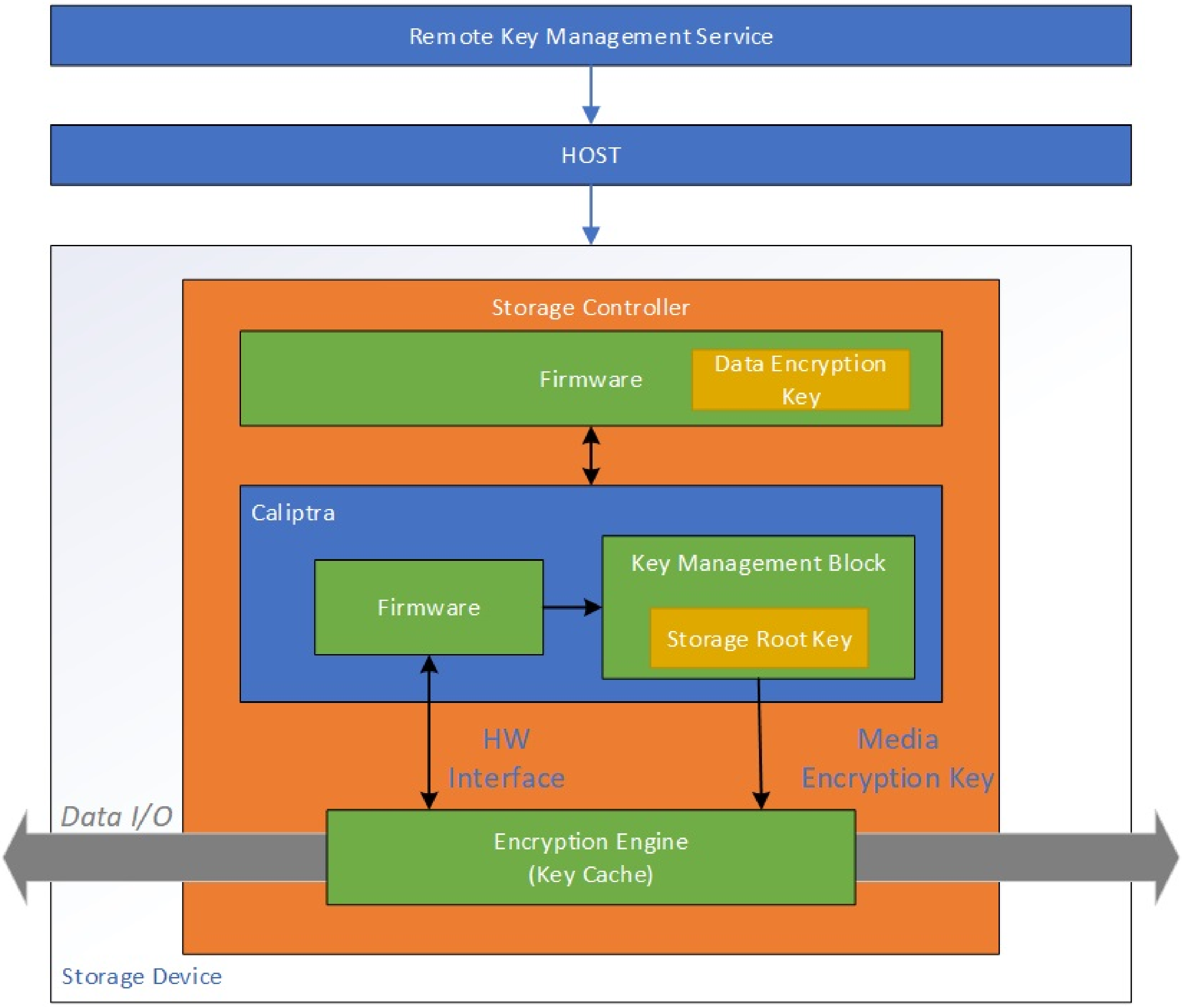
Given the above adversary profile, the following are a list of vulnerabilities that L.O.C.K. is designed to mitigate.

- Keys managed by storage controller firmware are compromised due to implementation bugs or side channels.
- Keys erased by storage controller firmware are recoverable via invasive techniques.
- MEKs are not fully bound to user credentials due to implementation bugs.
- MEKs are bound to user credentials which are compromised by a vulnerable host.
- Cryptographic erasure was not performed properly due to a buggy host.

Architecture

The following figure shows the basic high-level blocks of OCP L.O.C.K.

Figure 1: OCP L.O.C.K high level blocks



Caliptra that includes the optional OCP L.O.C.K. has a Key Management Block (KMB) that is the only entity that can derive the MEKs which protect user data and load the MEKs into the Key Cache of the Encryption Engine. The KMB derives MEKs using the following keys:

- A controller-supplied data encryption key (DEK). The DEK is the mechanism by which the controller enforces privilege separation between user credentials under TCG Opal, as well as the mechanism used to model injected MEKs under KPIO.

- A KMB-supplied storage root key, derived from secrets held in device fuses that are only accessible by the KMB. The storage root key may be rotated a small number of times, providing assurance that an advanced adversary cannot recover key material used by the drive prior to the storage root key rotation. KMB does not allow MEKs to be derived while the storage root key is erased. KMB can report whether the storage root key is erased and therefore whether the drive is clean.

- Zero or more partial MEKs (PMEKs), each of which is a cryptographically-strong value, encrypted to an externally-supplied access key. PMEKs enable multi-party authorization flows: the access key for each PMEK used to derive an MEK must be provided to the drive before the MEK can be used. Access keys are protected in transit using asymmetric encryption. This enables use-cases where the access key is served to the drive from a remote entity, without having to trust the host to which the drive is attached.

The DEK and storage root key do not require any changes to the host APIs for TCG Opal or KPIO.

Additional host APIs (i.e., Remote Key Management Services) are required to fully model storage root key rotation, PMEKs, and injectable host entropy. Such APIs are beyond the scope of the present document.

MEKs are never visible to any firmware. To load an MEK into the Key Cache of the Encryption Engine, storage controller firmware interfaces to the Key Management Block to generate or retrieve a previously generated MEK and then cause hardware to load the MEK into the Encryption Engine. Each MEK has associated vendor-defined metadata, to identify for example the namespace and LBA range to be encrypted by the MEK.

Each PMEK is encrypted as rest using an externally-injected access key where that access key is not stored persistently on the storage device. If there is more than one PMEK used to generate an MEK, then it requires multiple authorities to unlock a given range of user data. Access keys may be held at rest in a remote key management service.

Each MEK is bound for its lifetime to the Storage Root Key, the list of PMEKs, and the DEK. To generate an MEK, the access key for each PMEK must be provided. All MEKs are removed from the Encryption Engine on a power cycle or during sanitization on the storage device.

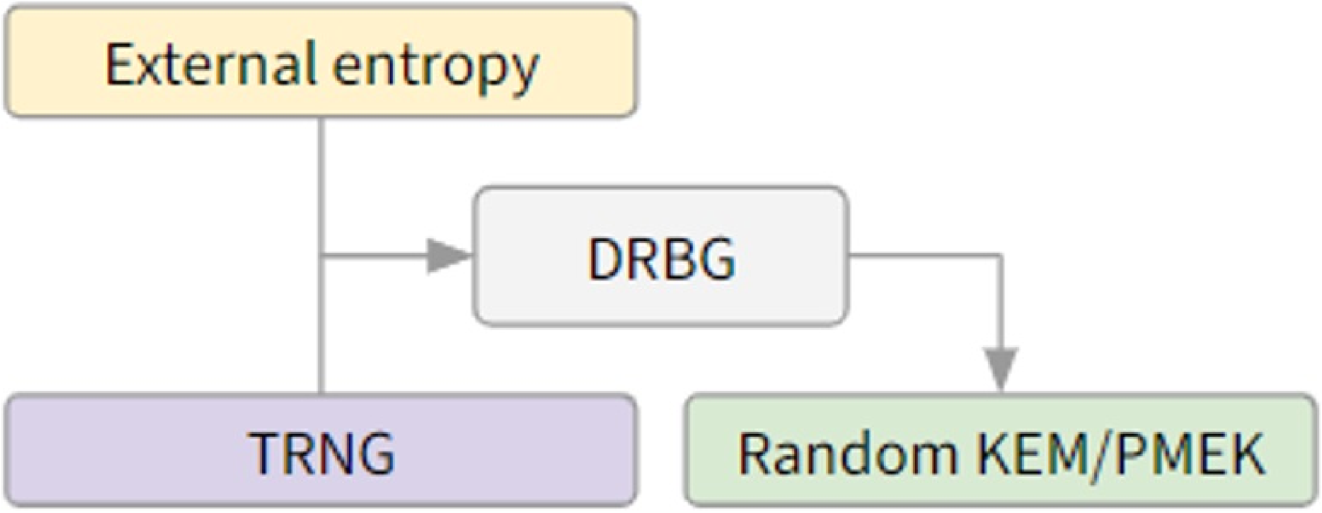
The DEK may be derived from or decrypted by a user's C_PIN to support legacy Opal. The DEK may be the imported key associated with a Key Per I/O key tag. KMB generates HPKE keypairs and stores them in internal volatile memory using a KEM algorithm such as ECDH or ML-KEM/Kyber. KMB can issue endorsements of KEM public keys allowing a Remote Key Management Services to ensure they only release access keys to authentic devices. PMEK access keys are encrypted in transit using these KEM keypairs. Upon drive reset, the access key must be re-encrypted to a new KEM for its associated PMEK to be usable.

KMB can maintain multiple active KEM keypairs. Nominally, one for each supported algorithm. KMB will automatically initialize a KEM for each supported algorithm and this may be done lazily. KMB will allow the controller to trigger KEM rotation. This can be done as part of zeroization.

KMB supports PMEK access key rotation where the storage controller must replace an old encrypted PMEK with a new encrypted PMEK. The end user must prove to KMB that they control both old and new access key. This is done by encrypting the new access key with the old access key. The new access key is therefore double-encrypted when provided to KMB with the old access key and with an ephemeral transport encryption key.

KMB is able to generate two kinds of keys: KEMs (used for access key transport encryption) and PMEKs (used for MEK derivation). A host is able to inject external entropy into KMB where it is held in internal volatile memory. Subsequent keys are randomly generated using both KMB's TRNG and the host's entropy as shown in figure 2.

Figure 2: KEM and PMEK Key Generation

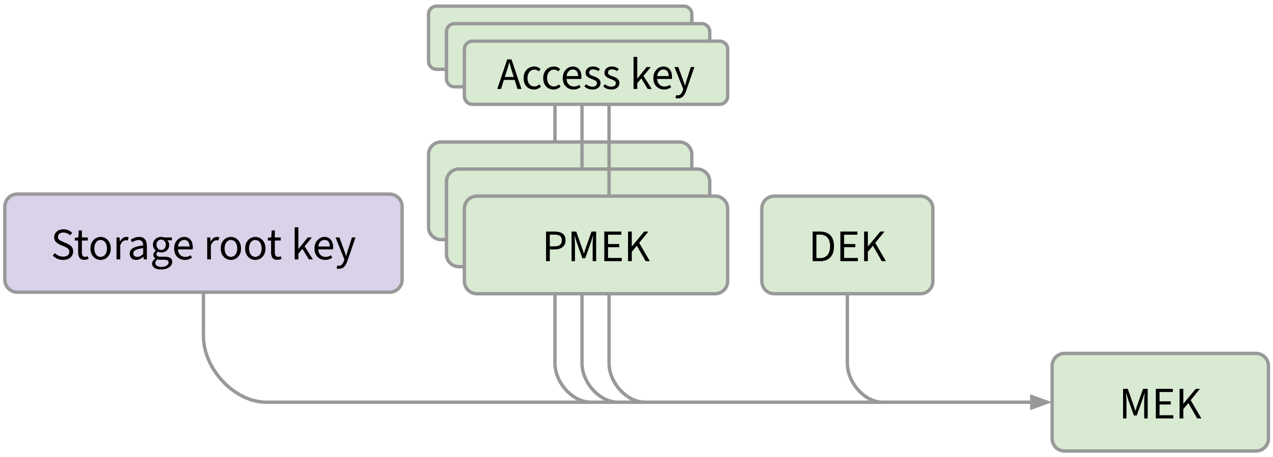


Interfaces

OCP L.O.C.K. provides two interfaces:

- The Encryption Engine interface is exposed from the vendor-implemented Encryption Engine to KMB, and defines a standard mechanism for programming MEKs and control messages.
- The mailbox interface is exposed from KMB to storage controller firmware, and enables the controller to manage MEKs.

MEK derivation



When controller firmware wishes to program an MEK to the Encryption Engine, the controller firmware performs the following steps:

1. Provides zero or more unlocked PMEKs to the KMB.

- KMB initializes the MEK seed buffer with the System Root Key and then extends that MEK seed buffer using each given unlocked PMEK.

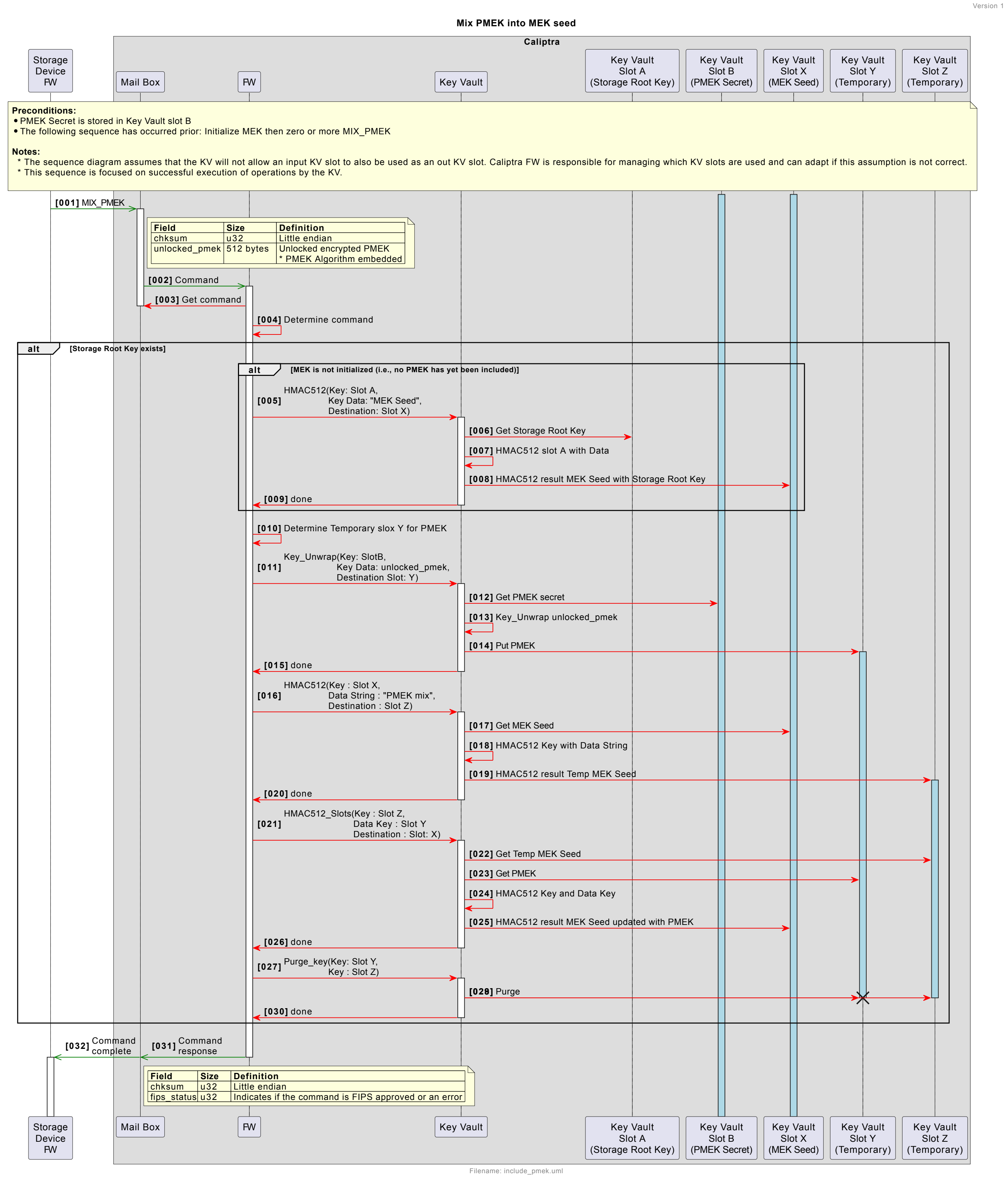
2. Provide a DEK to the KMB.

- KMB derives the MEK using the given DEK and the contents of the MEK seed buffer.

3. Provide MEK metadata to KMB, such as the MEK's associated namespace and logical block address range.

- KMB programs the derived MEK and its metadata to the Encryption Engine.

Sequence to mix a PMEK into the MEK seed


Version 1
Mix PMEK into MEK seed
Caliptra
Storage Device FW
Mail Box
FW
Key Vault
Key Vault Slot A (Storage Root Key)
Key Vault Slot B (PMEK Secret)
Key Vault Slot X (MEK Seed)
Key Vault Slot Y (Temporary)
Key Vault Slot Z (Temporary)
[001] MIX_PMEK
[002] Command
[003] Get command
[004] Determine command
alt [Storage Root Key exists]
alt [MEK is not initialized (i.e., no PMEK has yet been included)]
[005] HMAC512(Key: Slot A, Key Data: "MEK Seed", Destination: Slot X)
[006] Get Storage Root Key
[007] HMAC512 slot A with Data
[008] HMAC512 result MEK Seed with Storage Root Key
[009] done
[010] Determine Temporary slox Y for PMEK
[011] Key_Unwrap(Key: SlotB, Key Data: unlocked_pmek, Destination Slot: Y)
[012] Get PMEK secret
[013] Key_Unwrap unlocked_pmek
[014] Put PMEK
[015] done
[016] HMAC512(Key : Slot X, Data String : "PMEK mix", Destination : Slot Z)
[017] Get MEK Seed
[018] HMAC512 Key with Data String
[019] HMAC512 result Temp MEK Seed
[020] done
[021] HMAC512_Slots(Key : Slot Z, Data Key : Slot Y Destination : Slot: X)
[022] Get Temp MEK Seed
[023] Get PMEK
[024] HMAC512 Key and Data Key
[025] HMAC512 result MEK Seed updated with PMEK
[026] done
[027] Purge_key(Key: Slot Y, Key : Slot Z)
[028] Purge
[030] done
[031] Command response
[032] Command complete
Storage Device FW
Mail Box
FW
Key Vault
Key Vault Slot A (Storage Root Key)
Key Vault Slot B (PMEK Secret)
Key Vault Slot X (MEK Seed)
Key Vault Slot Y (Temporary)
Key Vault Slot Z (Temporary)
Filename: include_pmek.uml

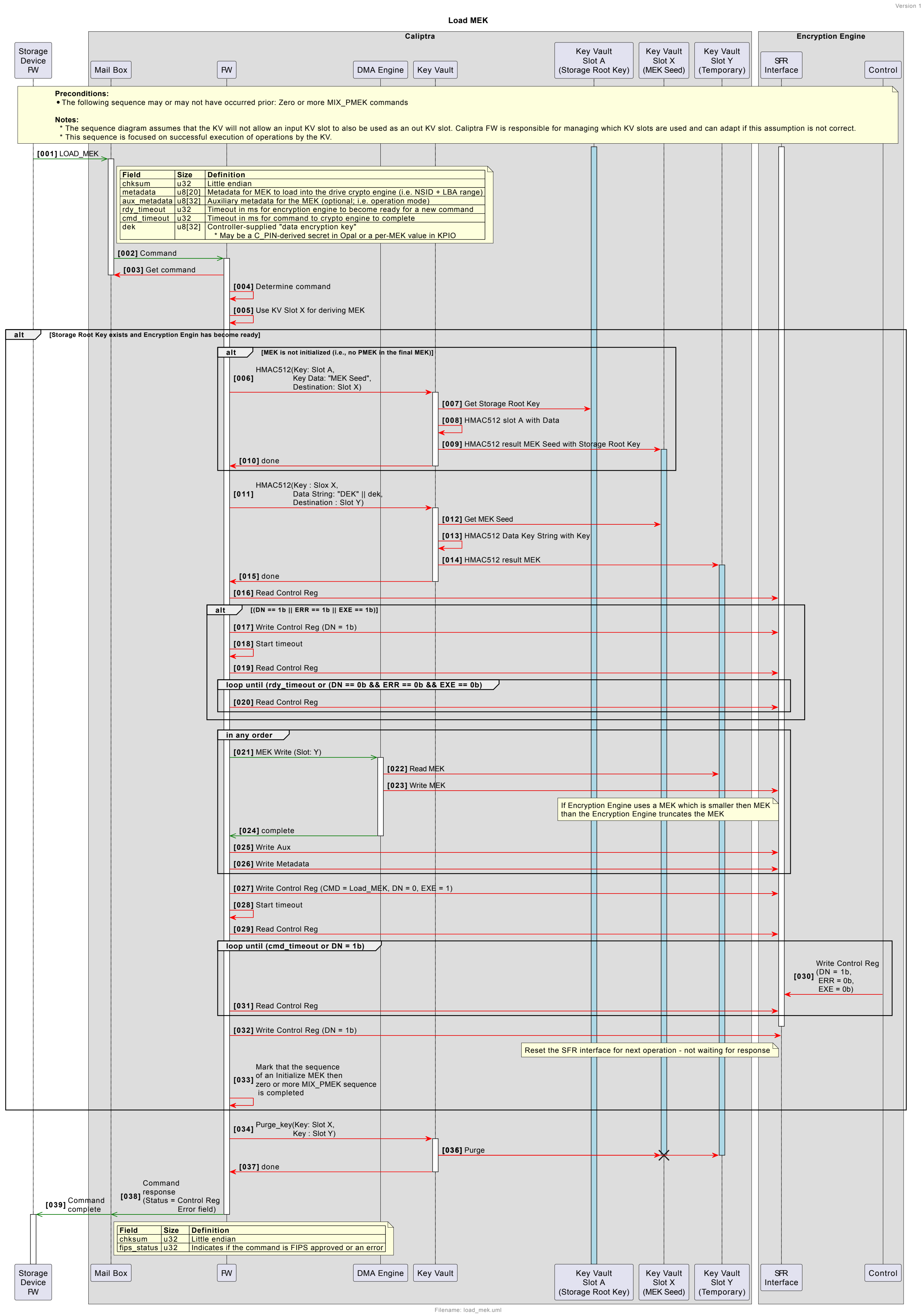
Preconditions:
- PMEK Secret is stored in Key Vault slot B
- The following sequence has occurred prior: Initialize MEK then zero or more MIX_PMEK

Notes:
* The sequence diagram assumes that the KV will not allow an input KV slot to also be used as an out KV slot. Caliptra FW is responsible for managing which KV slots are used and can adapt if this assumption is not correct.
* This sequence is focused on successful execution of operations by the KV.

Field	Size	Definition
chksum	u32	Little endian
unlocked_pmek	512 bytes	Unlocked encrypted PMEK * PMEK Algorithm embedded

Field	Size	Definition
chksum	u32	Little endian
fips_status	u32	Indicates if the command is FIPS approved or an error

Sequence to load an MEK


Version 1
Load MEK
Caliptra
Encryption Engine
Storage Device FW
Mail Box
FW
DMA Engine
Key Vault
Key Vault Slot A (Storage Root Key)
Key Vault Slot X (MEK Seed)
Key Vault Slot Y (Temporary)
SFR Interface
Control
[001] LOAD_MEK
[002] Command
[003] Get command
[004] Determine command
[005] Use KV Slot X for deriving MEK
alt [Storage Root Key exists and Encryption Engin has become ready]
alt [MEK is not initialized (i.e., no PMEK in the final MEK)]
[006] HMAC512(Key: Slot A, Key Data: "MEK Seed", Destination: Slot X)
[007] Get Storage Root Key
[008] HMAC512 slot A with Data
[009] HMAC512 result MEK Seed with Storage Root Key
[010] done
[011] HMAC512(Key : Slox X, Data String: "DEK" || dek, Destination : Slot Y)
[012] Get MEK Seed
[013] HMAC512 Data Key String with Key
[014] HMAC512 result MEK
[015] done
[016] Read Control Reg
alt [(DN == 1b || ERR == 1b || EXE == 1b)]
[017] Write Control Reg (DN = 1b)
[018] Start timeout
[019] Read Control Reg
loop until (rdy_timeout or (DN == 0b && ERR == 0b && EXE == 0b)
[020] Read Control Reg
in any order
[021] MEK Write (Slot: Y)
[022] Read MEK
[023] Write MEK
[024] complete
[025] Write Aux
[026] Write Metadata
[027] Write Control Reg (CMD = Load_MEK, DN = 0, EXE = 1)
[028] Start timeout
[029] Read Control Reg
loop until (cmd_timeout or DN = 1b)
[030] Write Control Reg (DN = 1b, ERR = 0b, EXE = 0b)
[031] Read Control Reg
[032] Write Control Reg (DN = 1b)
[033] Mark that the sequence of an Initialize MEK then zero or more MIX_PMEK sequence is completed
[034] Purge_key(Key: Slot X, Key : Slot Y)
[036] Purge
[037] done
[038] Command response (Status = Control Reg Error field)
[039] Command complete
Filename: load_mek.uml

Preconditions:
- The following sequence may or may not have occurred prior: Zero or more MIX_PMEK commands

Notes:
- * The sequence diagram assumes that the KV will not allow an input KV slot to also be used as an out KV slot. Caliptra FW is responsible for managing which KV slots are used and can adapt if this assumption is not correct.
- * This sequence is focused on successful execution of operations by the KV.

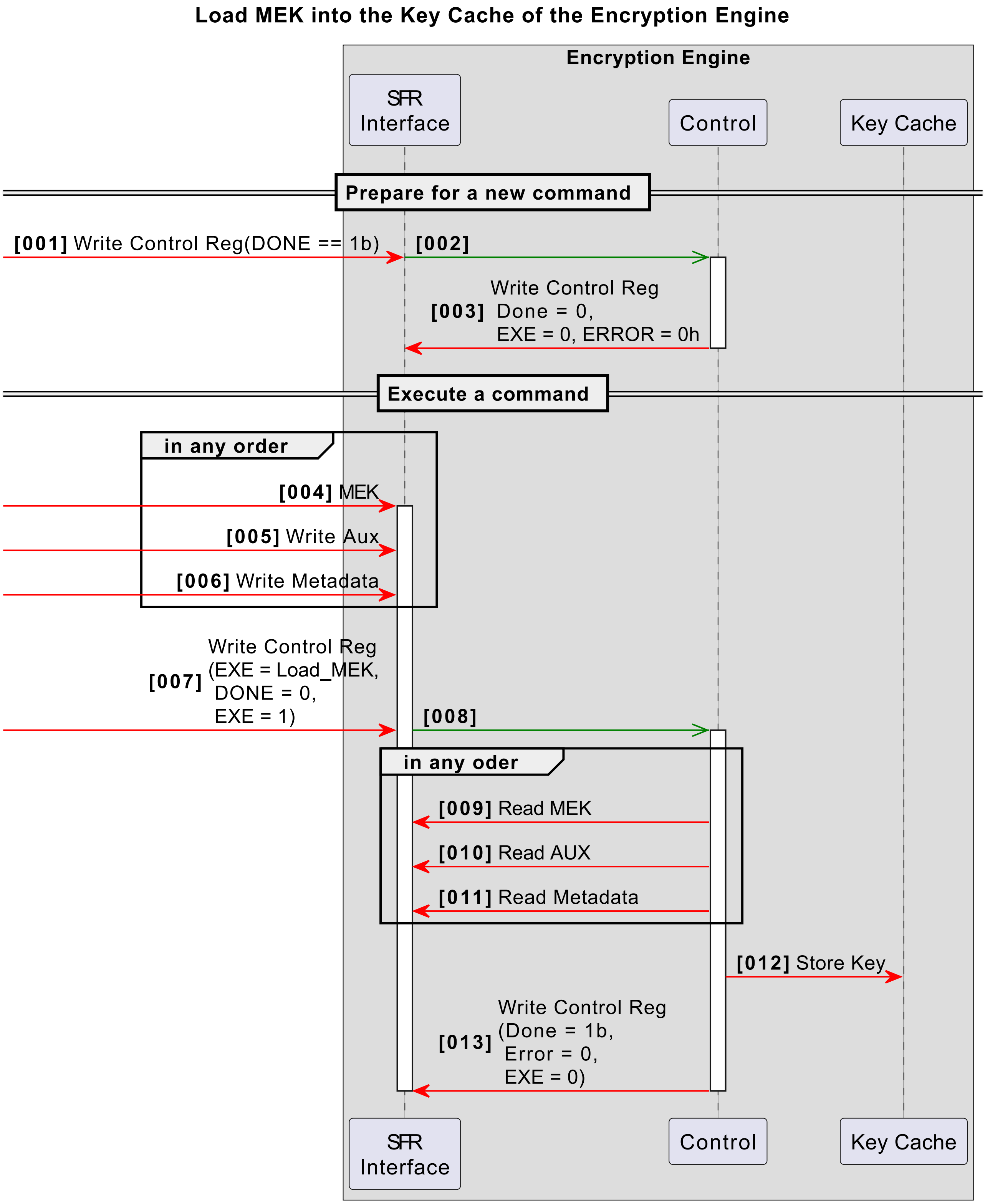
Field	Size	Definition
chksum	u32	Little endian
metadata	u8[20]	Metadata for MEK to load into the drive crypto engine (i.e. NSID + LBA range)
aux_metadata	u8[32]	Auxiliary metadata for the MEK (optional; i.e. operation mode)
rdy_timeout	u32	Timeout in ms for encryption engine to become ready for a new command
cmd_timeout	u32	Timeout in ms for command to crypto engine to complete
dek	u8[32]	Controller-supplied "data encryption key" * May be a C_PIN-derived secret in Opal or a per-MEK value in KPIO

If Encryption Engine uses a MEK which is smaller then MEK than the Encryption Engine truncates the MEK

Reset the SFR interface for next operation - not waiting for response

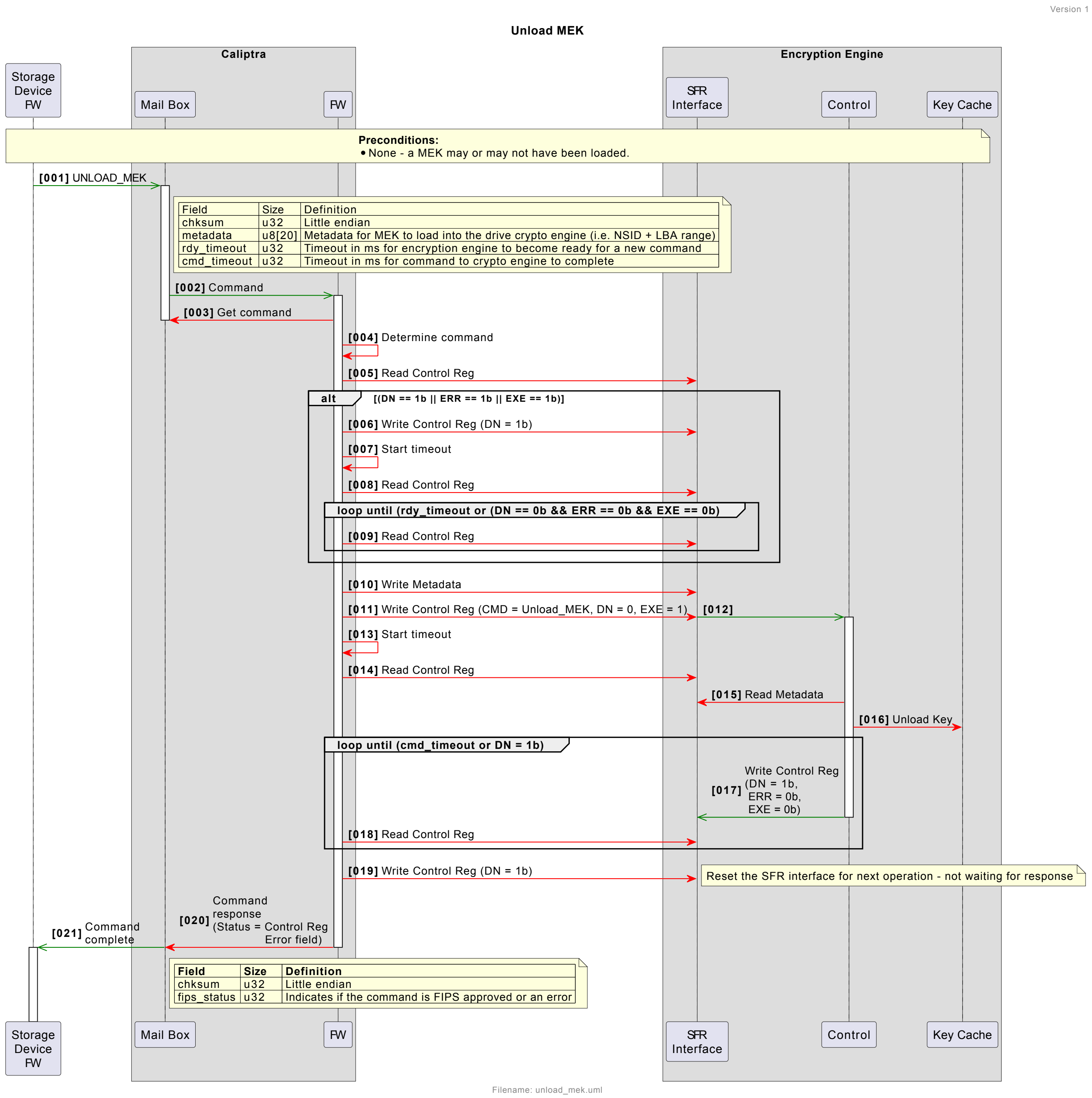
Field	Size	Definition
chksum	u32	Little endian
fips_status	u32	Indicates if the command is FIPS approved or an error

Sequence to load MEK into the Encryption Engine Key Cache


Load MEK into the Key Cache of the Encryption Engine
Encryption Engine
SFR Interface
Control
Key Cache
Prepare for a new command
[001] Write Control Reg(DONE == 1b)
[002]
[003] Write Control Reg Done = 0, EXE = 0, ERROR = 0h
Execute a command
in any order
[004] MEK
[005] Write Aux
[006] Write Metadata
[007] Write Control Reg (EXE = Load_MEK, DONE = 0, EXE = 1)
[008]
in any oder
[009] Read MEK
[010] Read AUX
[011] Read Metadata
[012] Store Key
[013] Write Control Reg (Done = 1b, Error = 0, EXE = 0)
SFR Interface
Control
Key Cache

Filename: load_mek_into_ee.uml

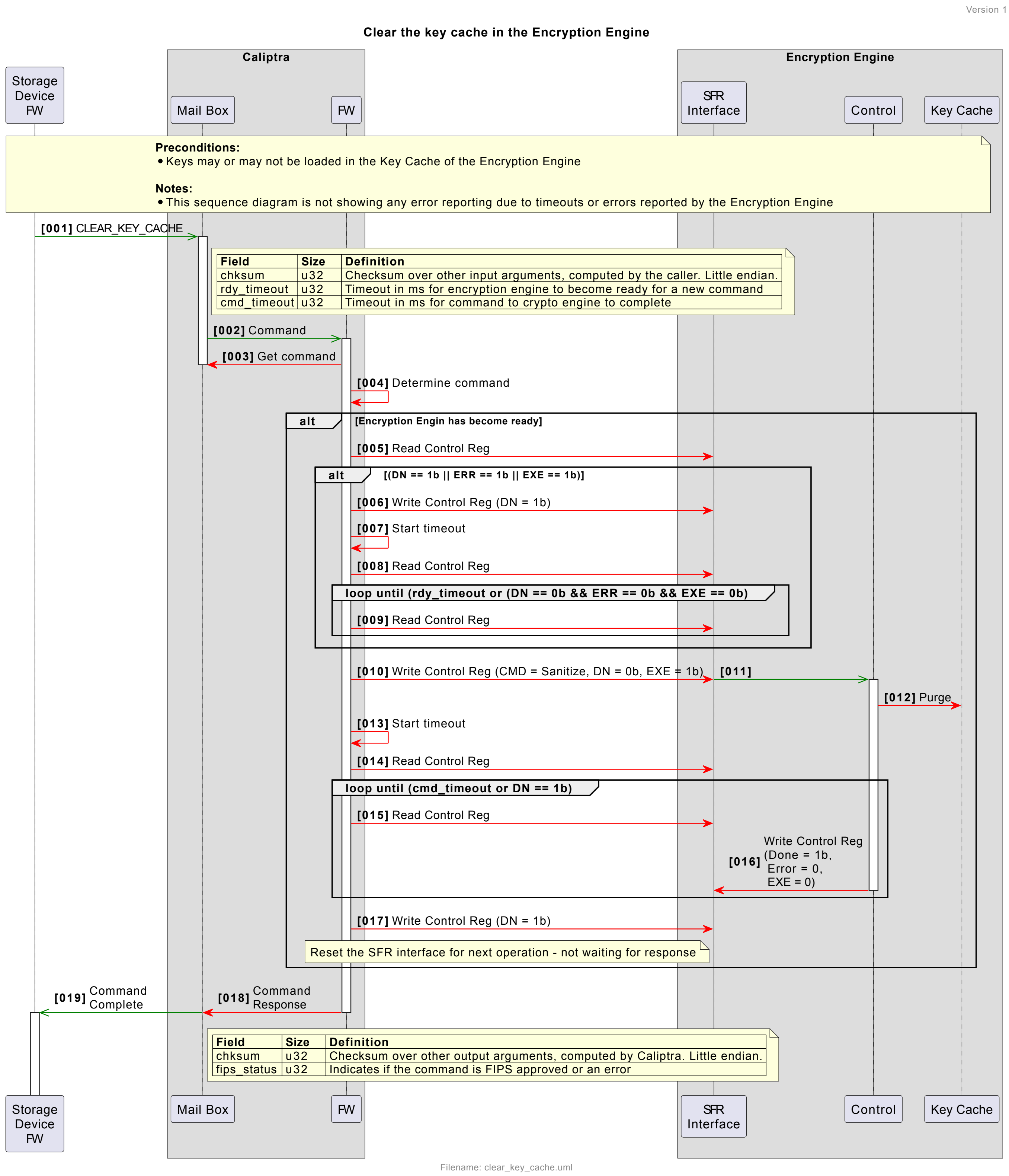
Sequence to unload an MEK from the Encryption Engine Key Cache


Version 1
Unload MEK
Calliptra
Encryption Engine
Storage Device FW
Mail Box
FW
SFR Interface
Control
Key Cache
Preconditions:
• None - a MEK may or may not have been loaded.
[001] UNLOAD_MEK
[002] Command
[003] Get command
[004] Determine command
[005] Read Control Reg
alt
[(DN == 1b || ERR == 1b || EXE == 1b)]
[006] Write Control Reg (DN = 1b)
[007] Start timeout
[008] Read Control Reg
loop until (rdy_timeout or (DN == 0b && ERR == 0b && EXE == 0b)
[009] Read Control Reg
[010] Write Metadata
[011] Write Control Reg (CMD = Unload_MEK, DN = 0, EXE = 1)
[012]
[013] Start timeout
[014] Read Control Reg
[015] Read Metadata
[016] Unload Key
loop until (cmd_timeout or DN = 1b)
[017] Write Control Reg (DN = 1b, ERR = 0b, EXE = 0b)
[018] Read Control Reg
[019] Write Control Reg (DN = 1b)
Reset the SFR interface for next operation - not waiting for response
[020] Command response (Status = Control Reg Error field)
[021] Command complete
Filename: unload_mek.uml

Field	Size	Definition
chksum	u32	Little endian
metadata	u8[20]	Metadata for MEK to load into the drive crypto engine (i.e. NSID + LBA range)
rdy_timeout	u32	Timeout in ms for encryption engine to become ready for a new command
cmd_timeout	u32	Timeout in ms for command to crypto engine to complete

Field	**Size**	**Definition**
chksum	u32	Little endian
fips_status	u32	Indicates if the command is FIPS approved or an error

Sequence to unload all MEKs (i.e., purge) from the Encryption Engine Key Cache



Version 1
Clear the key cache in the Encryption Engine
Storage Device FW
Caliptra
Mail Box
FW
Encryption Engine
SFR Interface
Control
Key Cache
Preconditions:
• Keys may or may not be loaded in the Key Cache of the Encryption Engine
Notes:
• This sequence diagram is not showing any error reporting due to timeouts or errors reported by the Encryption Engine
[001] CLEAR_KEY_CACHE
[002] Command
[003] Get command
[004] Determine command
alt [Encryption Engin has become ready]
[005] Read Control Reg
alt [(DN == 1b || ERR == 1b || EXE == 1b)]
[006] Write Control Reg (DN = 1b)
[007] Start timeout
[008] Read Control Reg
loop until (rdy_timeout or (DN == 0b && ERR == 0b && EXE == 0b)
[009] Read Control Reg
[010] Write Control Reg (CMD = Sanitize, DN = 0b, EXE = 1b)
[011]
[012] Purge
[013] Start timeout
[014] Read Control Reg
loop until (cmd_timeout or DN == 1b)
[015] Read Control Reg
[016] Write Control Reg (Done = 1b, Error = 0, EXE = 0)
[017] Write Control Reg (DN = 1b)
Reset the SFR interface for next operation - not waiting for response
[018] Command Response
[019] Command Complete
Filename: clear_key_cache.uml

Field	Size	Definition
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
rdy_timeout	u32	Timeout in ms for encryption engine to become ready for a new command
cmd_timeout	u32	Timeout in ms for command to crypto engine to complete

Field	Size	Definition
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error

Legacy MEK derivation for TCG Opal

The controller is allowed to maintain a DEK that represents a given user's media encryption key. That DEK can be encrypted at rest by the user's C_PIN.

When deriving the user's MEK, the controller can pass zero PMEKs in step 2, and the user's decrypted DEK in step 3.

Legacy MEK derivation for Key Per I/O

MEKs injected with Key Per I/O will be considered as DEKs under OCP L.O.C.K.

When deriving the associated MEK, the controller can pss zero PMEKs in step 2, and the injected DEK in step 3.

PMEK lifecycle

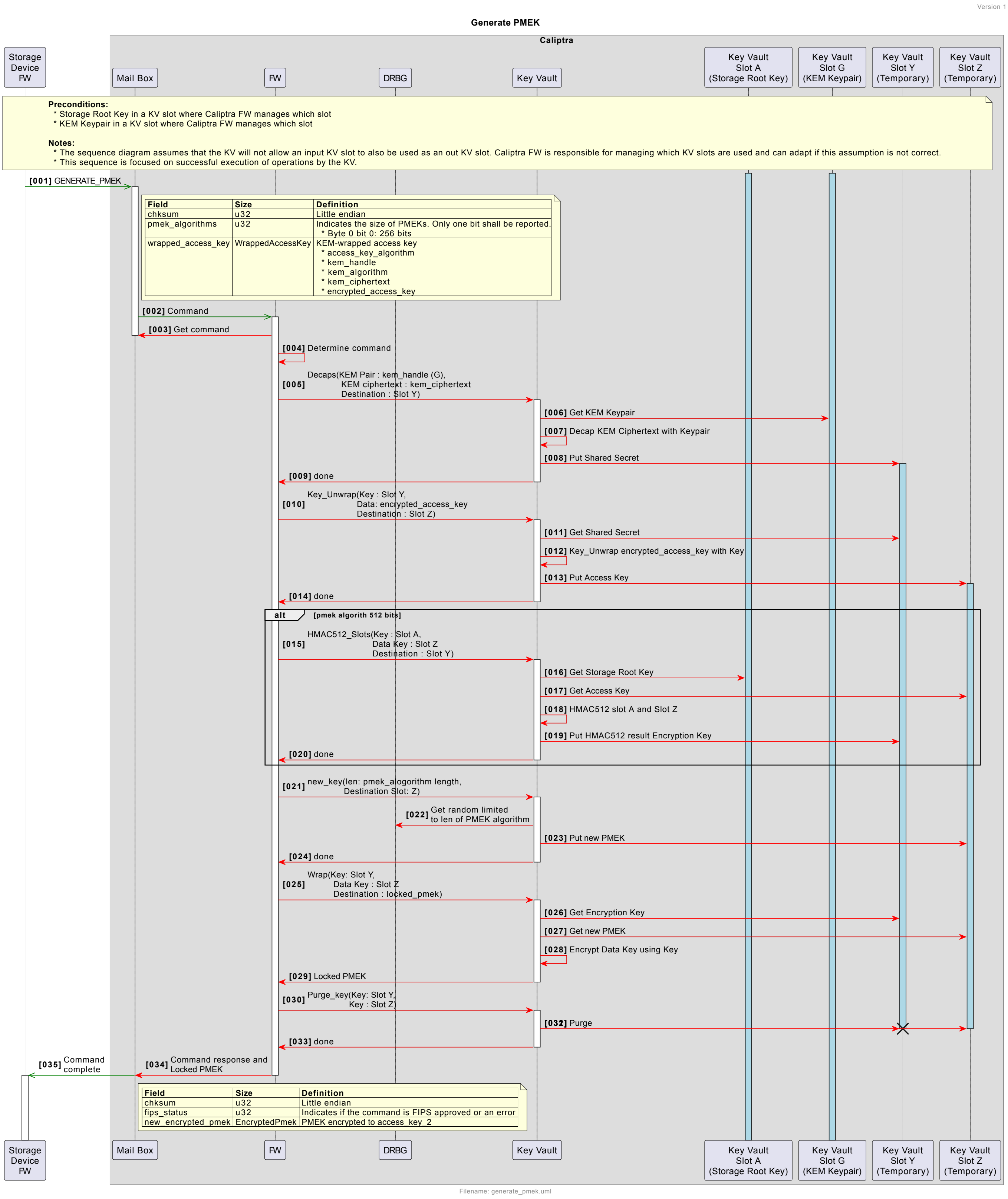
PMEK generation

Controller firmware may request that KMB generate a random PMEK, bound to a given access key. KMB performs the following steps:

1. Unwrap the given PMEK access key. See below for details on access key transport security.
2. Randomly generate a PMEK.
3. Derive a PMEK encryption key from the storage root key and the decrypted access key.
4. Encrypt the PMEK to the PMEK encryption key.
5. Return the encrypted PMEK to the controller firmware.

Controller firmware may then store the encrypted PMEK in persistent storage.

Sequence to generate a PMEK:


Version 1
Generate PMEK
Calliptra
Storage Device FW
Mail Box
FW
DRBG
Key Vault
Key Vault Slot A (Storage Root Key)
Key Vault Slot G (KEM Keypair)
Key Vault Slot Y (Temporary)
Key Vault Slot Z (Temporary)
[001] GENERATE_PMEK
[002] Command
[003] Get command
[004] Determine command
[005] Decaps(KEM Pair : kem_handle (G), KEM ciphertext : kem_ciphertext Destination : Slot Y)
[006] Get KEM Keypair
[007] Decap KEM Ciphertext with Keypair
[008] Put Shared Secret
[009] done
[010] Key_Unwrap(Key : Slot Y, Data: encrypted_access_key Destination : Slot Z)
[011] Get Shared Secret
[012] Key_Unwrap encrypted_access_key with Key
[013] Put Access Key
[014] done
alt
[pmek algorith 512 bits]
[015] HMAC512_Slots(Key : Slot A, Data Key : Slot Z Destination : Slot Y)
[016] Get Storage Root Key
[017] Get Access Key
[018] HMAC512 slot A and Slot Z
[019] Put HMAC512 result Encryption Key
[020] done
[021] new_key(len: pmek_alogorithm length, Destination Slot: Z)
[022] Get random limited to len of PMEK algorithm
[023] Put new PMEK
[024] done
[025] Wrap(Key: Slot Y, Data Key : Slot Z Destination : locked_pmek)
[026] Get Encryption Key
[027] Get new PMEK
[028] Encrypt Data Key using Key
[029] Locked PMEK
[030] Purge_key(Key: Slot Y, Key : Slot Z)
[032] Purge
[033] done
[034] Command response and Locked PMEK
[035] Command complete
Filename: generate_pmek.uml

Preconditions:
* Storage Root Key in a KV slot where Caliptra FW manages which slot
* KEM Keypair in a KV slot where Caliptra FW manages which slot

Notes:
* The sequence diagram assumes that the KV will not allow an input KV slot to also be used as an out KV slot. Caliptra FW is responsible for managing which KV slots are used and can adapt if this assumption is not correct.
* This sequence is focused on successful execution of operations by the KV.

Field	Size	Definition
chksum	u32	Little endian
pmek_algorithms	u32	Indicates the size of PMEKs. Only one bit shall be reported. * Byte 0 bit 0: 256 bits
wrapped_access_key	WrappedAccessKey	KEM-wrapped access key * access_key_algorithm * kem_handle * kem_algorithm * kem_ciphertext * encrypted_access_key

Field	Size	Definition
chksum	u32	Little endian
fips_status	u32	Indicates if the command is FIPS approved or an error
new_encrypted_pmek	EncryptedPmek	PMEK encrypted to access_key_2

PMEK unlock

Encrypted PMEKs stored at rest in persistent storage are considered "locked", and must be unlocked before they can be used to derive MEKs. Unlocked PMEKs are also encrypted when handled by controller firmware. Unlocked PMEKs do not survive across device reset.

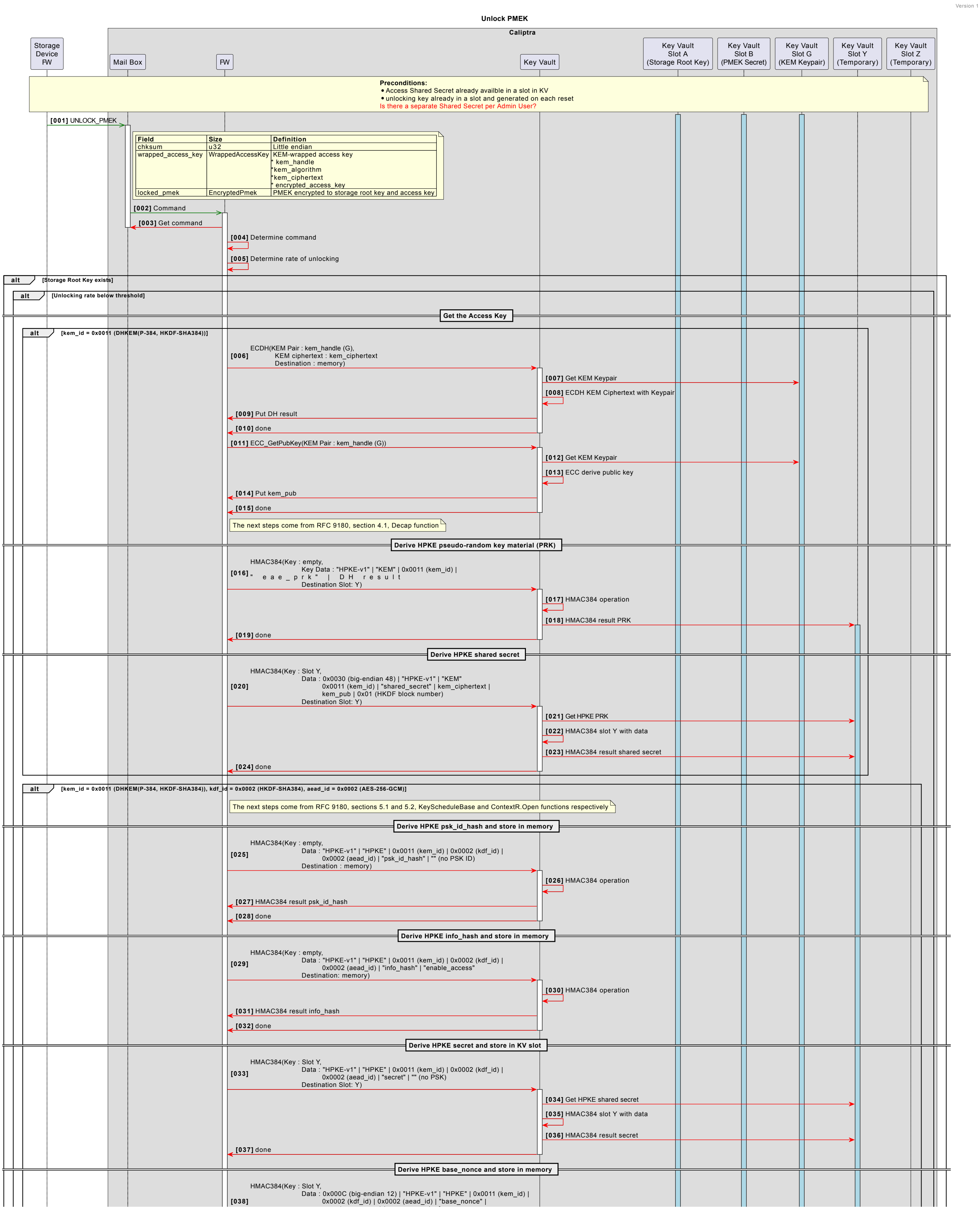
To unlock a PMEK, KMB performs the following steps:

1. Unwrap the given PMEK access key.
2. Derive the PMEK decryption key from the storage root key and the decrypted access key.

3. Decrypt the PMEK using the PMEK decryption key.
4. Encrypt the PMEK using an ephemeral export key that is randomly initialized on startup and lost on reset.
5. Return the re-encrypted "unlocked" PMEK to the controller firmware.

Controller firmware may then stash the encrypted unlocked PMEK in volatile storage, and later provide it to the KMB when deriving an MEK, as described above.

Sequence to unlock a PMEK:


Version 1
Unlock PMEK
Caliptra
Storage Device FW
Mail Box
FW
Key Vault
Key Vault Slot A (Storage Root Key)
Key Vault Slot B (PMEK Secret)
Key Vault Slot G (KEM Keypair)
Key Vault Slot Y (Temporary)
Key Vault Slot Z (Temporary)
alt
[Storage Root Key exists]
alt
[Unlocking rate below threshold]
Get the Access Key
alt
[kem_id = 0x0011 (DHKEM(P-384, HKDF-SHA384))]
Derive HPKE pseudo-random key material (PRK)
Derive HPKE shared secret
alt
[kem_id = 0x0011 (DHKEM(P-384, HKDF-SHA384)), kdf_id = 0x0002 (HKDF-SHA384), aead_id = 0x0002 (AES-256-GCM)]
Derive HPKE psk_id_hash and store in memory
Derive HPKE info_hash and store in memory
Derive HPKE secret and store in KV slot
Derive HPKE base_nonce and store in memory

Preconditions:
- Access Shared Secret already availble in a slot in KV
- unlocking key already in a slot and generated on each reset

Is there a separate Shared Secret per Admin User?

[001] UNLOCK_PMEK

Field	Size	Definition
chksum	u32	Little endian
wrapped_access_key	WrappedAccessKey	KEM-wrapped access key * kem_handle *kem_algorithm *kem_ciphertext * encrypted_access_key
locked_pmek	EncryptedPmek	PMEK encrypted to storage root key and access key

[002] Command

[003] Get command

[004] Determine command

[005] Determine rate of unlocking

```
[006]   ECDH(KEM Pair : kem_handle (G),
             KEM ciphertext : kem_ciphertext
             Destination : memory)
```

[007] Get KEM Keypair

[008] ECDH KEM Ciphertext with Keypair

[009] Put DH result

[010] done

[011] ECC_GetPubKey(KEM Pair : kem_handle (G))

[012] Get KEM Keypair

[013] ECC derive public key

[014] Put kem_pub

[015] done

The next steps come from RFC 9180, section 4.1, Decap function

```
[016]   HMAC384(Key : empty,
                Key Data : "HPKE-v1" | "KEM" | 0x0011 (kem_id) |
        "eae_prk" | DH result
                Destination Slot: Y)
```

[017] HMAC384 operation

[018] HMAC384 result PRK

[019] done

```
[020]   HMAC384(Key : Slot Y,
                Data : 0x0030 (big-endian 48) | "HPKE-v1" | "KEM"
                       0x0011 (kem_id) | "shared_secret" | kem_ciphertext |
                       kem_pub | 0x01 (HKDF block number)
                Destination Slot: Y)
```

[021] Get HPKE PRK

[022] HMAC384 slot Y with data

[023] HMAC384 result shared secret

[024] done

The next steps come from RFC 9180, sections 5.1 and 5.2, KeyScheduleBase and ContextR.Open functions respectively

```
[025]   HMAC384(Key : empty,
                Data : "HPKE-v1" | "HPKE" | 0x0011 (kem_id) | 0x0002 (kdf_id) |
                       0x0002 (aead_id) | "psk_id_hash" | "" (no PSK ID)
                Destination : memory)
```

[026] HMAC384 operation

[027] HMAC384 result psk_id_hash

[028] done

```
[029]   HMAC384(Key : empty,
                Data : "HPKE-v1" | "HPKE" | 0x0011 (kem_id) | 0x0002 (kdf_id) |
                       0x0002 (aead_id) | "info_hash" | "enable_access"
                Destination: memory)
```

[030] HMAC384 operation

[031] HMAC384 result info_hash

[032] done

```
[033]   HMAC384(Key : Slot Y,
                Data : "HPKE-v1" | "HPKE" | 0x0011 (kem_id) | 0x0002 (kdf_id) |
                       0x0002 (aead_id) | "secret" | "" (no PSK)
                Destination Slot: Y)
```

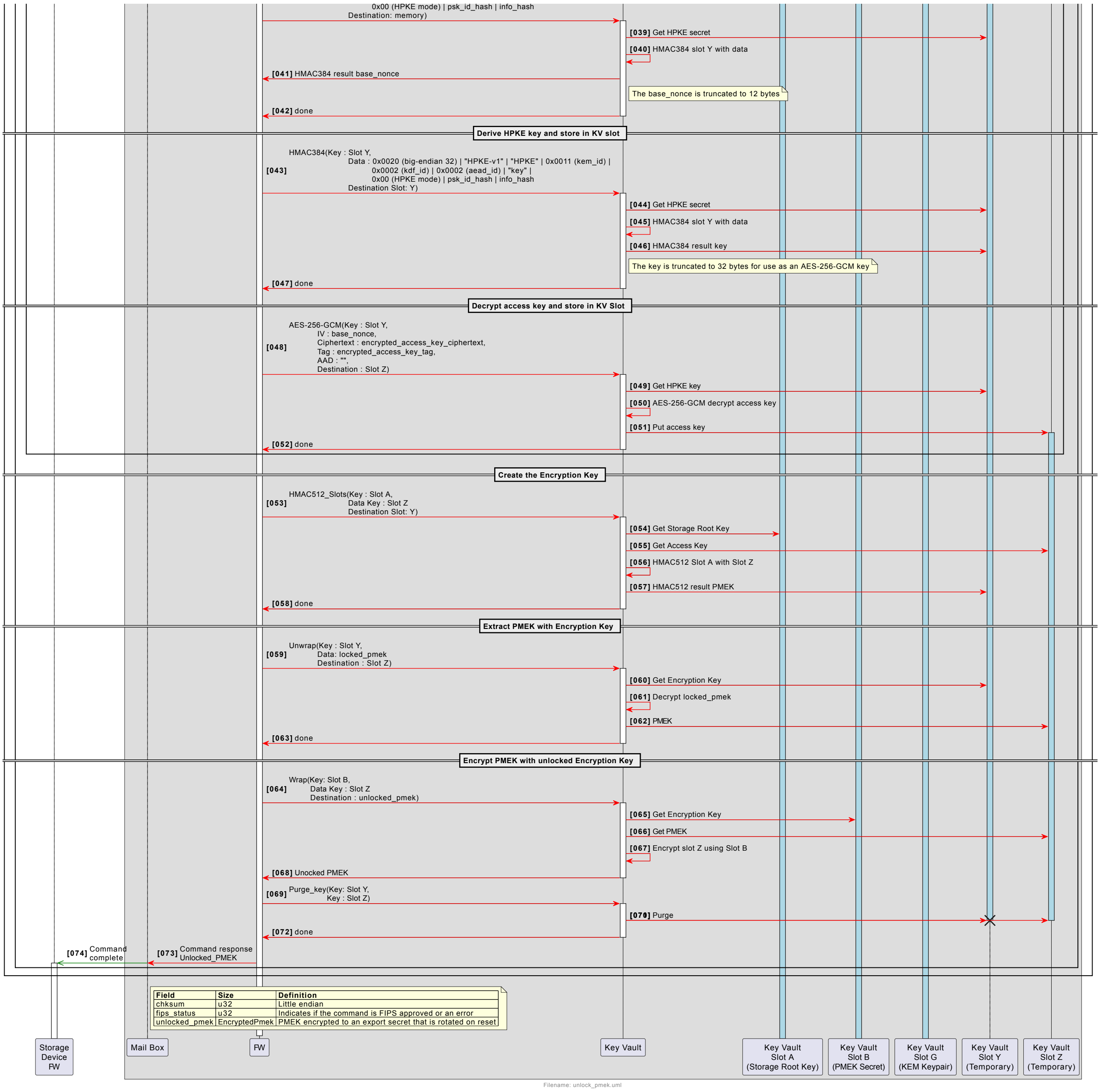
[034] Get HPKE shared secret

[035] HMAC384 slot Y with data

[036] HMAC384 result secret

[037] done

```
[038]   HMAC384(Key : Slot Y,
                Data : 0x000C (big-endian 12) | "HPKE-v1" | "HPKE" | 0x0011 (kem_id) |
                       0x0002 (kdf_id) | 0x0002 (aead_id) | "base_nonce" |
```


0x00 (HPKE mode) | psk_id_hash | info_hash
Destination: memory)
[039] Get HPKE secret
[040] HMAC384 slot Y with data
[041] HMAC384 result base_nonce
The base_nonce is truncated to 12 bytes
[042] done
Derive HPKE key and store in KV slot
[043] HMAC384(Key : Slot Y,
Data : 0x0020 (big-endian 32) | "HPKE-v1" | "HPKE" | 0x0011 (kem_id) |
0x0002 (kdf_id) | 0x0002 (aead_id) | "key" |
0x00 (HPKE mode) | psk_id_hash | info_hash
Destination Slot: Y)
[044] Get HPKE secret
[045] HMAC384 slot Y with data
[046] HMAC384 result key
The key is truncated to 32 bytes for use as an AES-256-GCM key
[047] done
Decrypt access key and store in KV Slot
[048] AES-256-GCM(Key : Slot Y,
IV : base_nonce,
Ciphertext : encrypted_access_key_ciphertext,
Tag : encrypted_access_key_tag,
AAD : "",
Destination : Slot Z)
[049] Get HPKE key
[050] AES-256-GCM decrypt access key
[051] Put access key
[052] done
Create the Encryption Key
[053] HMAC512_Slots(Key : Slot A,
Data Key : Slot Z
Destination Slot: Y)
[054] Get Storage Root Key
[055] Get Access Key
[056] HMAC512 Slot A with Slot Z
[057] HMAC512 result PMEK
[058] done
Extract PMEK with Encryption Key
[059] Unwrap(Key : Slot Y,
Data: locked_pmek
Destination : Slot Z)
[060] Get Encryption Key
[061] Decrypt locked_pmek
[062] PMEK
[063] done
Encrypt PMEK with unlocked Encryption Key
[064] Wrap(Key: Slot B,
Data Key : Slot Z
Destination : unlocked_pmek)
[065] Get Encryption Key
[066] Get PMEK
[067] Encrypt slot Z using Slot B
[068] Unocked PMEK
[069] Purge_key(Key: Slot Y,
Key : Slot Z)
[070] Purge
[072] done
[073] Command response
Unlocked_PMEK
[074] Command
complete
Storage
Device
FW
Mail Box
FW
Key Vault
Key Vault
Slot A
(Storage Root Key)
Key Vault
Slot B
(PMEK Secret)
Key Vault
Slot G
(KEM Keypair)
Key Vault
Slot Y
(Temporary)
Key Vault
Slot Z
(Temporary)

Field	Size	Definition
chksum	u32	Little endian
fips_status	u32	Indicates if the command is FIPS approved or an error
unlocked_pmek	EncryptedPmek	PMEK encrypted to an export secret that is rotated on reset

Filename: unlock_pmek.uml

PMEK access key rotation

The access key to which a PMEK is bound may be rotated. The user must prove that they have knowledge of both the old and new access key before a rotation is allowed. KMB performs the following steps:

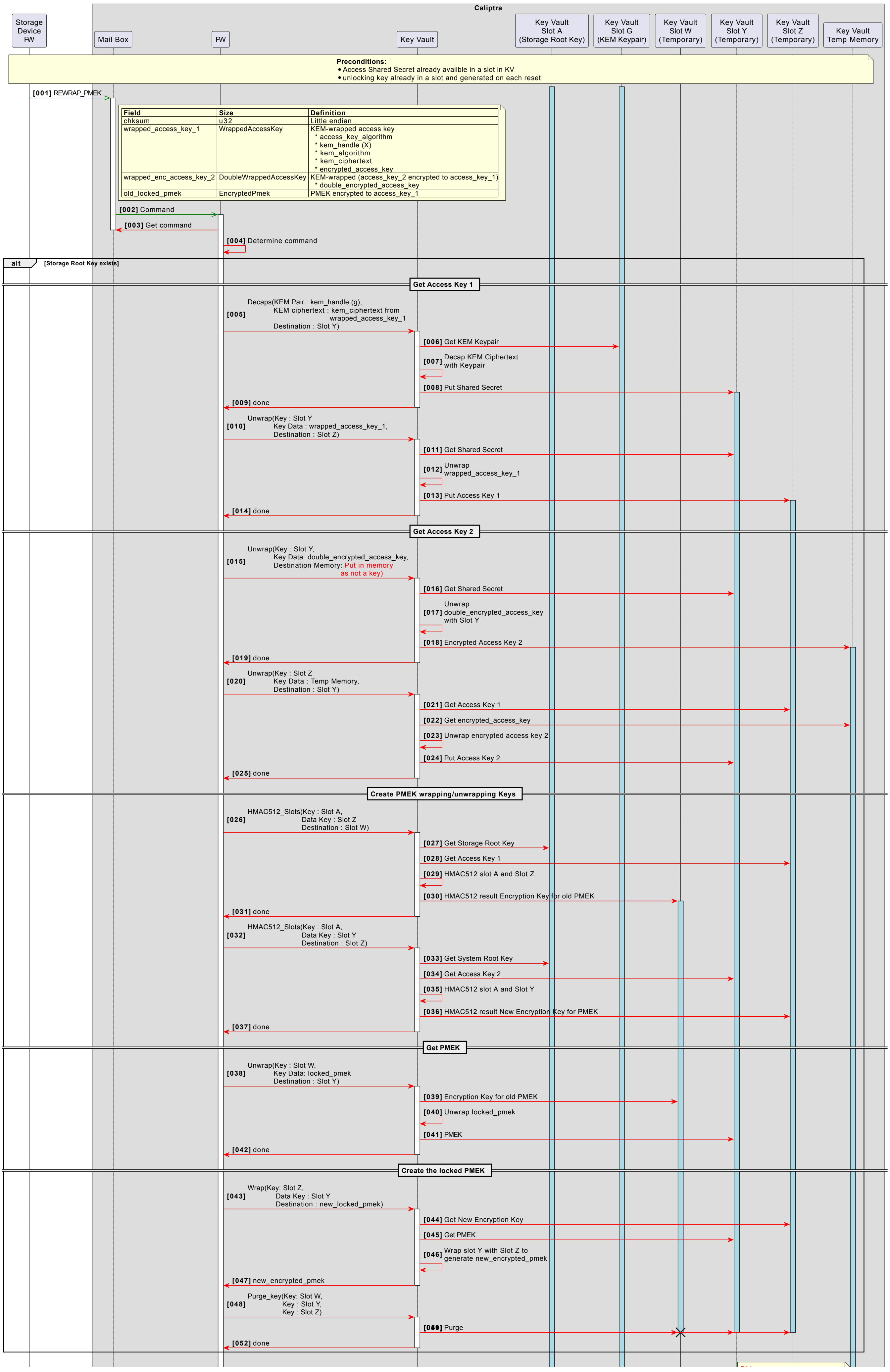
1. Unwrap the given old and new access keys.
2. Derive the old PMEK decryption key from the storage root key and the decrypted old access key.
3. Derive the new PMEK decryption key from the storage root key and the decrypted new access key.
4. Decrypt the PMEK using the old PMEK decryption key.
5. Encrypt the PMEK using the new PMEK decryption key.
6. Return the re-encrypted PMEK to the controller firmware.

Controller firmware then erases the old encrypted PMEK and stores the new encrypted PMEK in persistent storage.

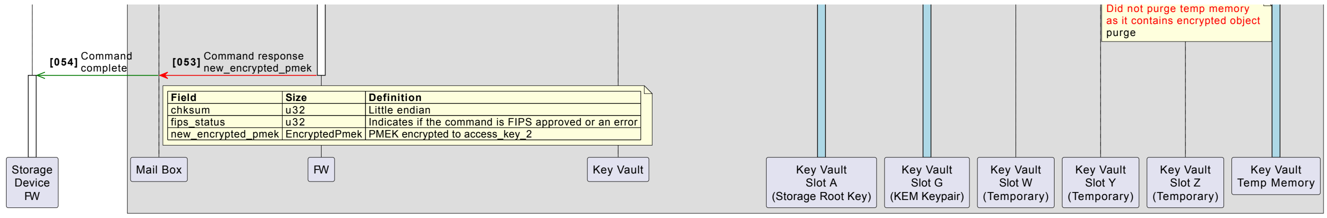
Sequence to rotate the access key of a PMEK:

Version 1

Rewrap PMEK


Caliptra
Storage Device FW
Mail Box
FW
Key Vault
Key Vault Slot A (Storage Root Key)
Key Vault Slot G (KEM Keypair)
Key Vault Slot W (Temporary)
Key Vault Slot Y (Temporary)
Key Vault Slot Z (Temporary)
Key Vault Temp Memory
Preconditions:
• Access Shared Secret already availble in a slot in KV
• unlocking key already in a slot and generated on each reset
[001] REWRAP_PMEK
[002] Command
[003] Get command
[004] Determine command
alt [Storage Root Key exists]
Get Access Key 1
[005] Decaps(KEM Pair : kem_handle (g), KEM ciphertext : kem_ciphertext from wrapped_access_key_1 Destination : Slot Y)
[006] Get KEM Keypair
[007] Decap KEM Ciphertext with Keypair
[008] Put Shared Secret
[009] done
[010] Unwrap(Key : Slot Y Key Data : wrapped_access_key_1, Destination : Slot Z)
[011] Get Shared Secret
[012] Unwrap wrapped_access_key_1
[013] Put Access Key 1
[014] done
Get Access Key 2
[015] Unwrap(Key : Slot Y, Key Data: double_encrypted_access_key, Destination Memory: Put in memory as not a key)
[016] Get Shared Secret
[017] Unwrap double_encrypted_access_key with Slot Y
[018] Encrypted Access Key 2
[019] done
[020] Unwrap(Key : Slot Z Key Data : Temp Memory, Destination : Slot Y)
[021] Get Access Key 1
[022] Get encrypted_access_key
[023] Unwrap encrypted access key 2
[024] Put Access Key 2
[025] done
Create PMEK wrapping/unwrapping Keys
[026] HMAC512_Slots(Key : Slot A, Data Key : Slot Z Destination : Slot W)
[027] Get Storage Root Key
[028] Get Access Key 1
[029] HMAC512 slot A and Slot Z
[030] HMAC512 result Encryption Key for old PMEK
[031] done
[032] HMAC512_Slots(Key : Slot A, Data Key : Slot Y Destination : Slot Z)
[033] Get System Root Key
[034] Get Access Key 2
[035] HMAC512 slot A and Slot Y
[036] HMAC512 result New Encryption Key for PMEK
[037] done
Get PMEK
[038] Unwrap(Key : Slot W, Key Data: locked_pmek Destination : Slot Y)
[039] Encryption Key for old PMEK
[040] Unwrap locked_pmek
[041] PMEK
[042] done
Create the locked PMEK
[043] Wrap(Key: Slot Z, Data Key : Slot Y Destination : new_locked_pmek)
[044] Get New Encryption Key
[045] Get PMEK
[046] Wrap slot Y with Slot Z to generate new_encrypted_pmek
[047] new_encrypted_pmek
[048] Purge_key(Key: Slot W, Key : Slot Y, Key : Slot Z)
[050] Purge
[052] done

Field	Size	Definition
chksum	u32	Little endian
wrapped_access_key_1	WrappedAccessKey	KEM-wrapped access key * access_key_algorithm * kem_handle (X) * kem_algorithm * kem_ciphertext * encrypted_access_key
wrapped_enc_access_key_2	DoubleWrappedAccessKey	KEM-wrapped (access_key_2 encrypted to access_key_1) * double_encrypted_access_key
old_locked_pmek	EncryptedPmek	PMEK encrypted to access_key_1


Did not purge temp memory
as it contains encrypted object
purge
[054] Command complete
[053] Command response new_encrypted_pmek
Storage Device FW
Mail Box
FW
Key Vault
Key Vault Slot A (Storage Root Key)
Key Vault Slot G (KEM Keypair)
Key Vault Slot W (Temporary)
Key Vault Slot Y (Temporary)
Key Vault Slot Z (Temporary)
Key Vault Temp Memory

Field	Size	Definition
chksum	u32	Little endian
fips_status	u32	Indicates if the command is FIPS approved or an error
new_encrypted_pmek	EncryptedPmek	PMEK encrypted to access_key_2

Filename: rewrap_pmek.uml

Transport encryption for PMEK access keys

In OCP L.O.C.K., the KMB maintains a set of key-encapsulation-mechanism (KEM) keypairs, one per algorithm that OCP L.O.C.K. supports. Each KEM public key is endorsed with a certificate that is generated by Caliptra and signed by Caliptra's DICE identity. KEM keypairs are randomly generated on KMB startup, may be periodically rotated, and are lost when the drive resets.

When a user wishes to unlock a PMEK (which is required prior to deriving any MEKs bound to that PMEK), the user performs the following steps:

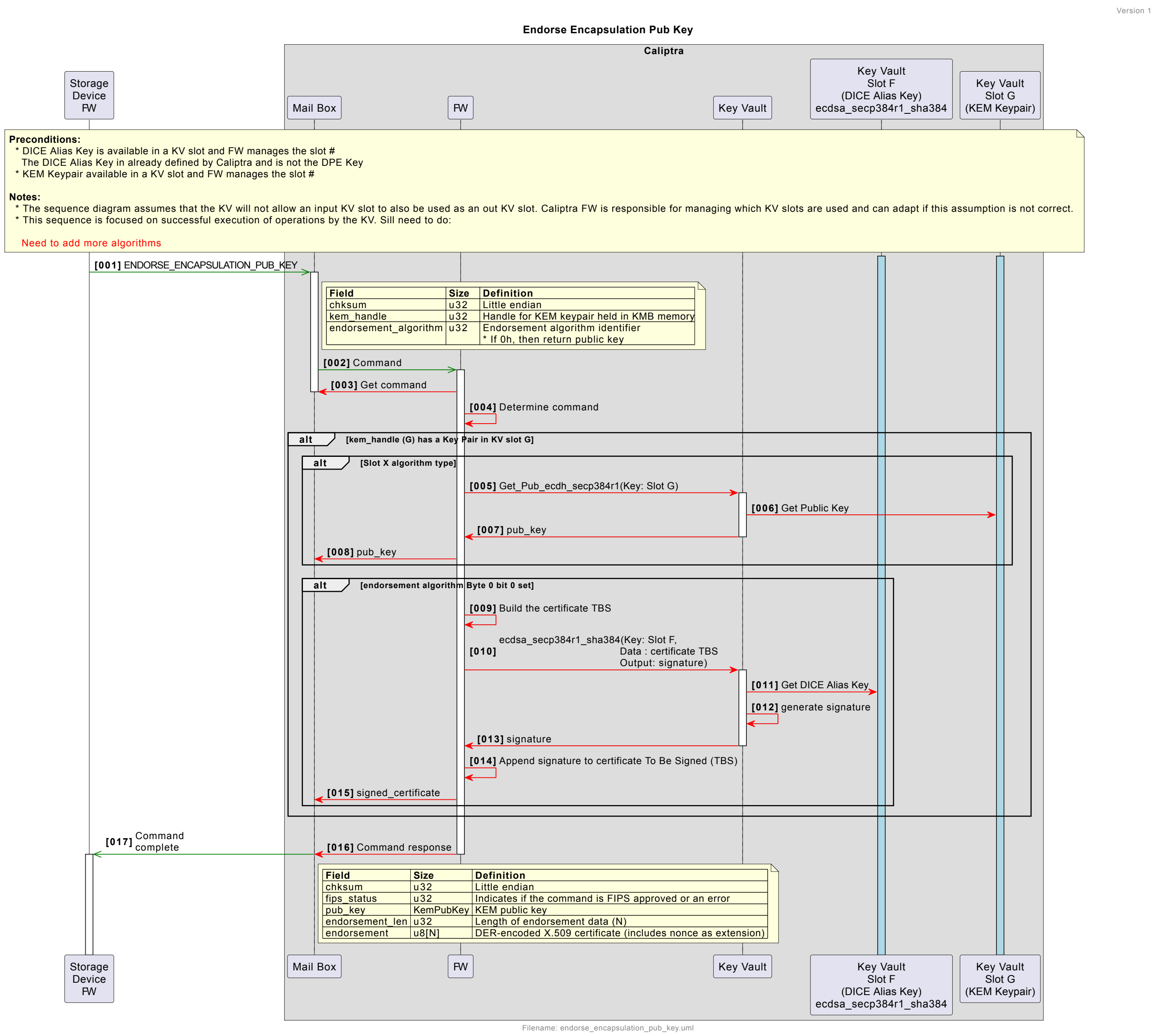
1. Obtain the KEM public key and certificate from the storage device.
2. Validate the KEM certificate and attached DICE certificate chain.
3. Run `Encaps` and encrypt their access key to the resulting shared key.
4. Transmit the KEM ciphertext and encrypted access key to the storage device.

Upon receipt, KMB will perform the following steps:

1. Run `Decaps` and decrypt the user's access key with the resulting shared secret.
2. Derive the PMEK encryption key using the storage root key and the decrypted access key.
3. Perform PMEK generation, unlock, or rotation actions detailed above.

Upon drive reset, the KEMs are regenerated, and any access keys for PMEKs that had been unlocked prior to the reset will need to be re-provisioned.

Sequence to endorse an encapsulation public key:


Version 1
Endorse Encapsulation Pub Key
Caliptra
Storage Device FW
Mail Box
FW
Key Vault
Key Vault Slot F (DICE Alias Key) ecdsa_secp384r1_sha384
Key Vault Slot G (KEM Keypair)
[001] ENDORSE_ENCAPSULATION_PUB_KEY
[002] Command
[003] Get command
[004] Determine command
alt [kem_handle (G) has a Key Pair in KV slot G]
alt [Slot X algorithm type]
[005] Get_Pub_ecdh_secp384r1(Key: Slot G)
[006] Get Public Key
[007] pub_key
[008] pub_key
alt [endorsement algorithm Byte 0 bit 0 set]
[009] Build the certificate TBS
[010] ecdsa_secp384r1_sha384(Key: Slot F, Data : certificate TBS Output: signature)
[011] Get DICE Alias Key
[012] generate signature
[013] signature
[014] Append signature to certificate To Be Signed (TBS)
[015] signed_certificate
[016] Command response
[017] Command complete
Filename: endorse_encapsulation_pub_key.uml

Preconditions:
* DICE Alias Key is available in a KV slot and FW manages the slot #
  The DICE Alias Key in already defined by Caliptra and is not the DPE Key
* KEM Keypair available in a KV slot and FW manages the slot #

Notes:
* The sequence diagram assumes that the KV will not allow an input KV slot to also be used as an out KV slot. Caliptra FW is responsible for managing which KV slots are used and can adapt if this assumption is not correct.
* This sequence is focused on successful execution of operations by the KV. Sill need to do:

  Need to add more algorithms

Field	Size	Definition
chksum	u32	Little endian
kem_handle	u32	Handle for KEM keypair held in KMB memory
endorsement_algorithm	u32	Endorsement algorithm identifier * If 0h, then return public key

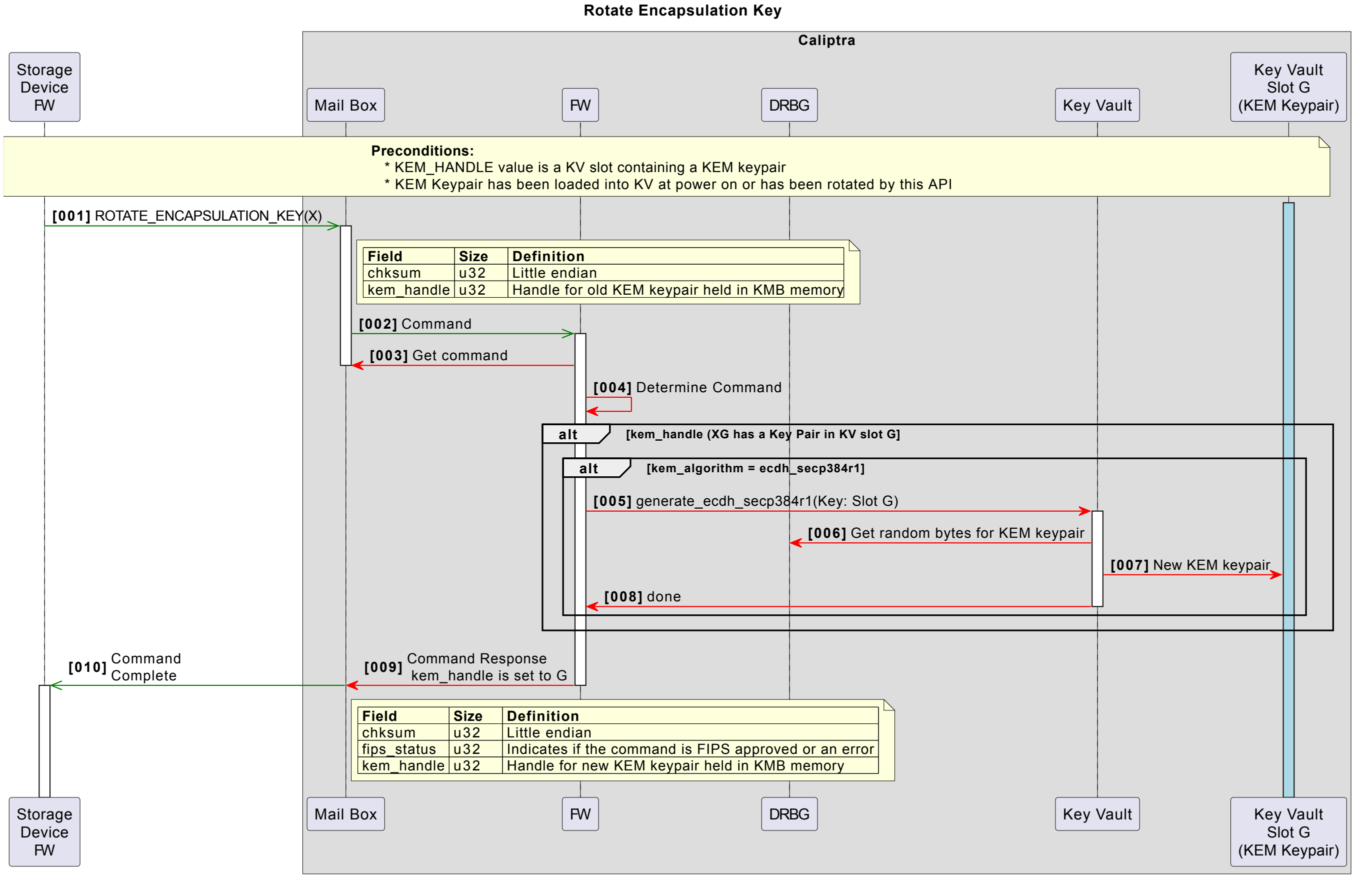
Field	Size	Definition
chksum	u32	Little endian
fips_status	u32	Indicates if the command is FIPS approved or an error
pub_key	KemPubKey	KEM public key
endorsement_len	u32	Length of endorsement data (N)
endorsement	u8[N]	DER-encoded X.509 certificate (includes nonce as extension)

Access key rotation flows

As noted above, during access key rotation the user must prove knowledge of both the old and new access keys. This is accomplished using a slight variation on the encaps-decaps flow. When a new access key is provided to KMB during a rotation, the new access key is double-encrypted: first to the old access key, and then to the shared secret obtained from the Encaps operation.

The KMB then performs a double decryption when unwrapping the new access key, proving that the provisioner of the new access key also knows the old access key.

Sequence to rotate an encapsulation key:


Rotate Encapsulation Key
Caliptra
Storage Device FW
Mail Box
FW
DRBG
Key Vault
Key Vault Slot G (KEM Keypair)
Preconditions:
* KEM_HANDLE value is a KV slot containing a KEM keypair
* KEM Keypair has been loaded into KV at power on or has been rotated by this API
[001] ROTATE_ENCAPSULATION_KEY(X)
[002] Command
[003] Get command
[004] Determine Command
alt [kem_handle (XG has a Key Pair in KV slot G]
alt [kem_algorithm = ecdh_secp384r1]
[005] generate_ecdh_secp384r1(Key: Slot G)
[006] Get random bytes for KEM keypair
[007] New KEM keypair
[008] done
[009] Command Response kem_handle is set to G
[010] Command Complete

Field	Size	Definition
chksum	u32	Little endian
kem_handle	u32	Handle for old KEM keypair held in KMB memory

Field	Size	Definition
chksum	u32	Little endian
fips_status	u32	Indicates if the command is FIPS approved or an error
kem_handle	u32	Handle for new KEM keypair held in KMB memory

Filename: rotate_encapsulation_key.uml

Algorithm support

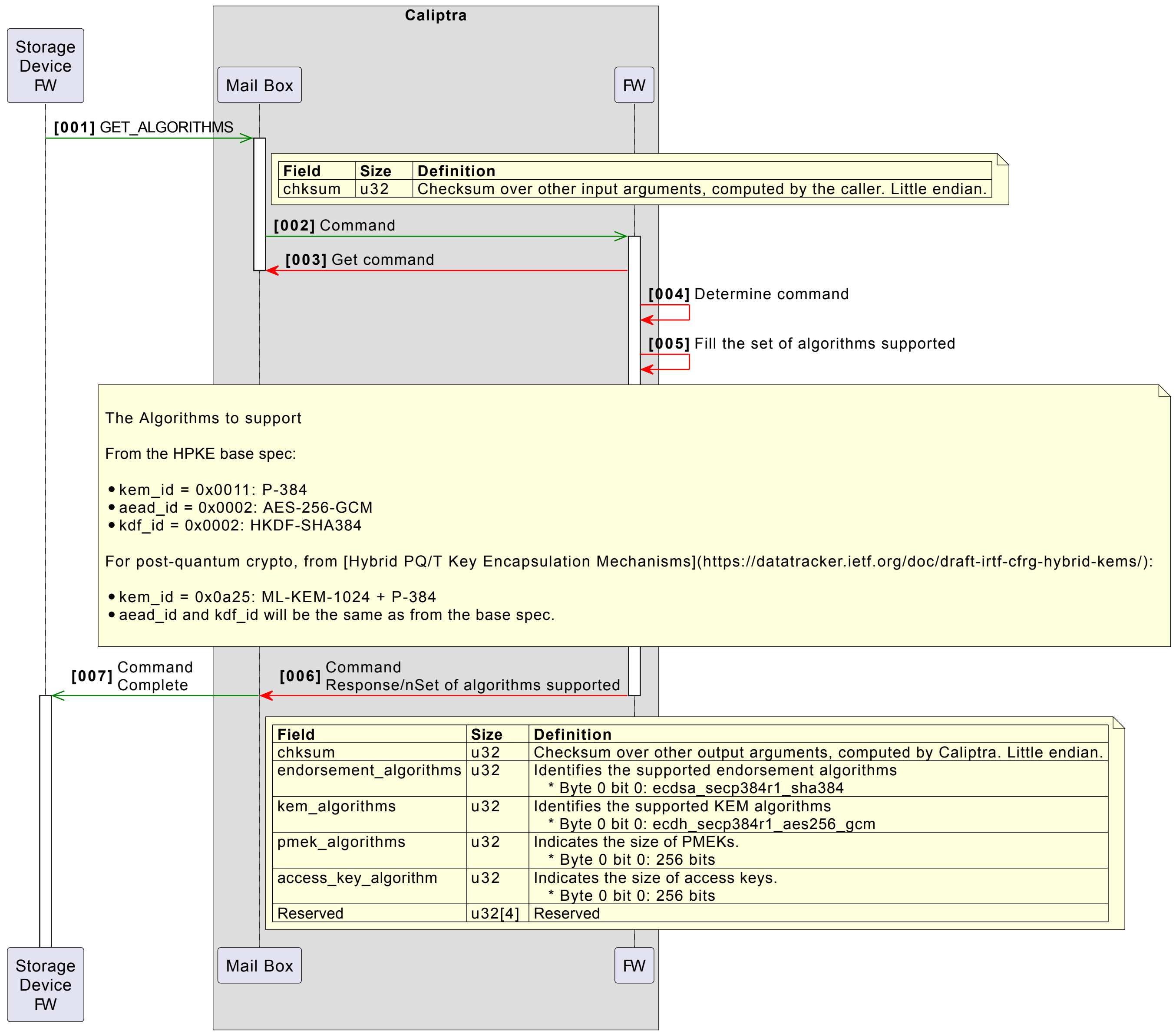
OCP L.O.C.K. supports the following KEM algorithms:

- P384 ECDH
- Hybridized ML-KEM-1024 with P-384 ECDH

Sequence to obtain the supported algorithms:

Version 1

GET_ALGORITHMS command sequence



Field	Size	Definition
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.

The Algorithms to support

From the HPKE base spec:

- kem_id = 0x0011: P-384
- aead_id = 0x0002: AES-256-GCM
- kdf_id = 0x0002: HKDF-SHA384

For post-quantum crypto, from [Hybrid PQ/T Key Encapsulation Mechanisms](https://datatracker.ietf.org/doc/draft-irtf-cfrg-hybrid-kems/):

- kem_id = 0x0a25: ML-KEM-1024 + P-384
- aead_id and kdf_id will be the same as from the base spec.

Field	Size	Definition
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
endorsement_algorithms	u32	Identifies the supported endorsement algorithms * Byte 0 bit 0: ecdsa_secp384r1_sha384
kem_algorithms	u32	Identifies the supported KEM algorithms * Byte 0 bit 0: ecdh_secp384r1_aes256_gcm
pmek_algorithms	u32	Indicates the size of PMEKs. * Byte 0 bit 0: 256 bits
access_key_algorithm	u32	Indicates the size of access keys. * Byte 0 bit 0: 256 bits
Reserved	u32[4]	Reserved

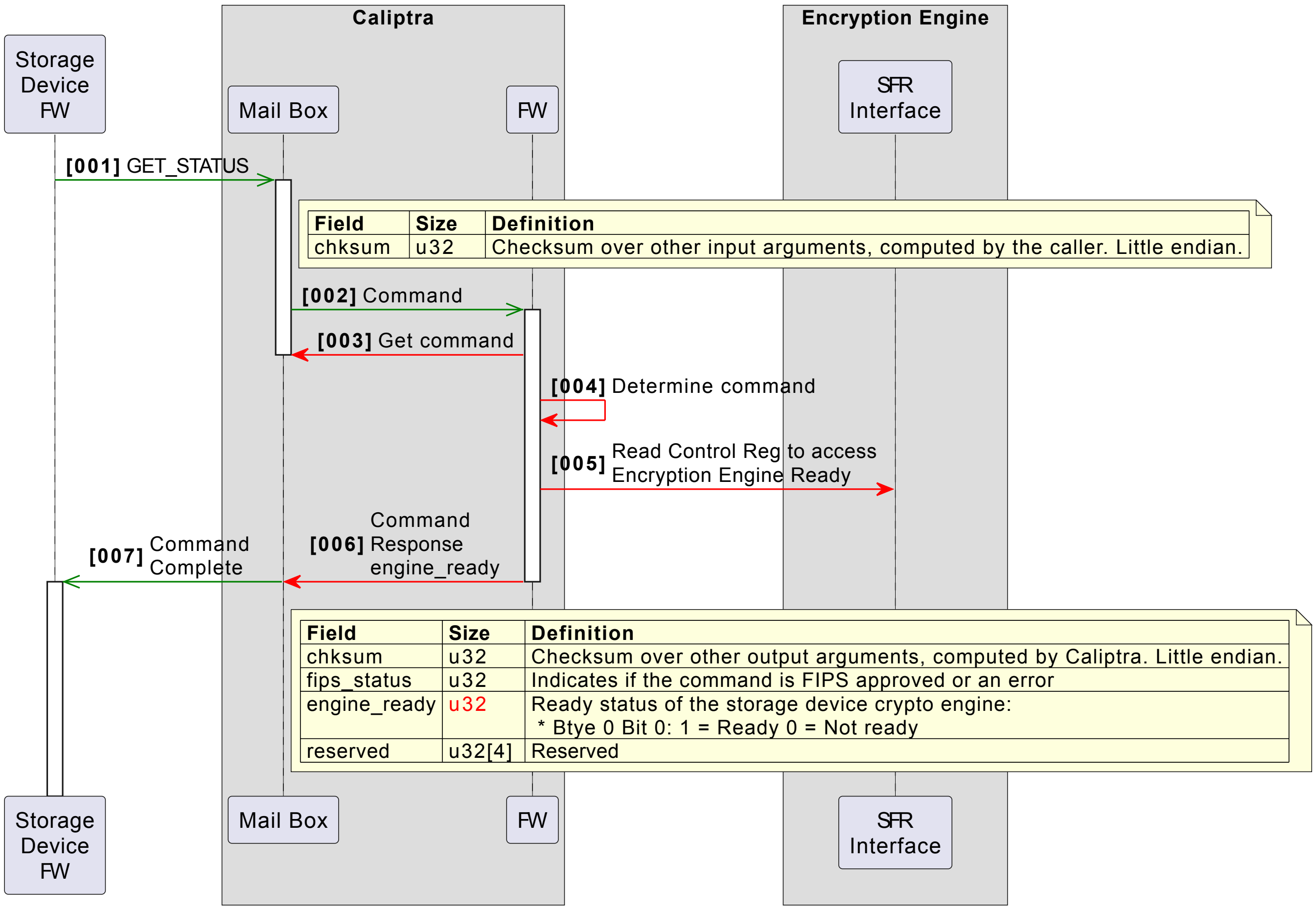
Filename: get_algorithms.uml

Status function

OCP L.O.C.K. has the following function to assist with determining the status.

Sequence to obtain the current status of OCP L.O.C.K.:

GET_STATUS command sequence



Field	Size	Definition
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.

Field	Size	Definition
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error
engine_ready	u32	Ready status of the storage device crypto engine: * Btye 0 Bit 0: 1 = Ready 0 = Not ready
reserved	u32[4]	Reserved

Filename: get_status.uml

Hardware

The following figure describes the OCP L.O.C.K. hardware.

Figure 8: KMB block diagram

This figure will be fleshed out with additional details as they become available.

Memory requirements

This section will be fleshed out with additional details as they become available.

Cryptography requirements

This section will be fleshed out with additional details as they become available.

KMB-Encryption Engine interface

This section defines the interface between the Key Management Block (KMB) and an Encryption Engine. An Encryption Engine is used to encrypt/decrypt user data and its design and implementation are vendor specific. Media Encryption Keys (MEKs) are the keys which are generated within KMB and used by Encryption Engine to encrypt and decrypt user data. This interface is used to load MEKs from KMB to Encryption Engine or to cause the Encryption Engine to unload (i.e., remove) loaded MEKs. The MEKs transferred between the KMB and the Encryption Engine shall not be accessible by the controller firmware.

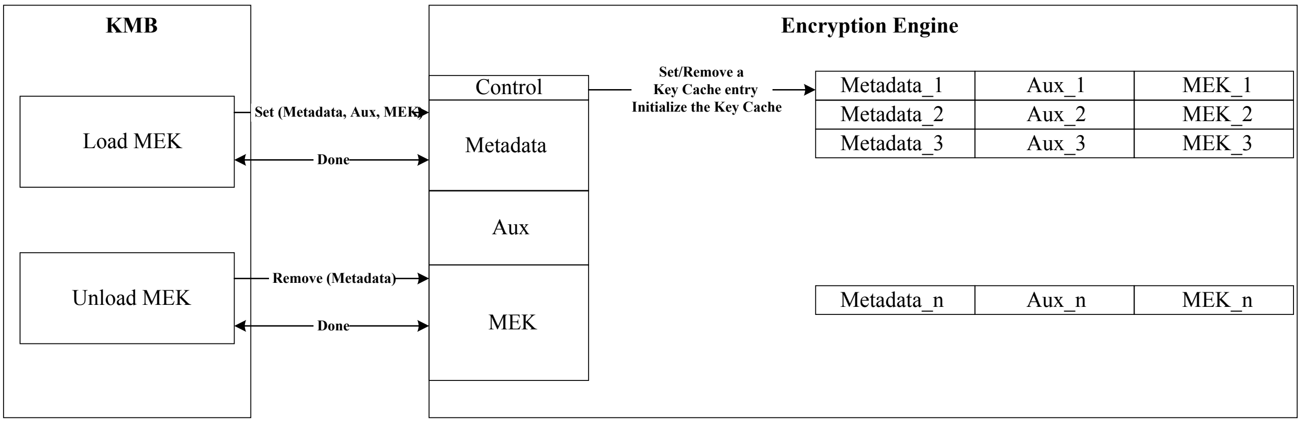
Overview

The Encryption Engine uses a stored MEK to encrypt and decrypt user data. For the purposes of this specification, the entity within the Encryption Engine used to store the MEKs is called Key Cache. Each encryption and description of user data is coupled to a specific MEK which is stored in Key Cache bound to a unique identifier, called metadata. Each (metadata, MEK) pair is also associated with additional information, called aux, which is used neither as MEK nor an identifier, but has some additional information about the pair. Therefore, the Key Cache as an entity which stores (metadata, aux, MEK) tuples.

In order to achieve the security goals for KMB, KMB is limited to be the unique component which loads an (metadata, aux, MEK) tuple into the Key Cache and unloads a tuple within the device, so that MEKs can only be exposed to KMB and Encryption Engine. Controller firmware arbitrates all operations in the KMB to Encryption Engine interface, therefore controller firmware is responsible for managing which MEK is loaded in Key Cache. Controller firmware has full control on metadata and optional aux. Figure 9 is an illustration of the KMB to Encryption Engine interface that shows:

- the tuple for loading an MEK;
- the Metadata for unloading an MEK; and
- an example of a Key Cache within the Encryption Engine.

Figure 9: KMB to Encryption SFR Interface



KMB uses Special Function Registers (SFRs) to communicate with Encryption Engine which are described in the following sections.

Special Function Registers (SFRs)

KMB uses Special Function Registers (SFRs) to communicate with Encryption Engine as shown in Table 1 and each of the following subsections describe the registers.

Table 1: KBM to Encryption Engine SFRs

Register	Address	Byte Size	Description
Control	SFR_BASE + 0h	4h	Register to handle commands

Register	Address	Byte Size	Description
Metadata	SFR_BASE + 10h	14h	Register to provide metadata
Auxiliary Data (AUX)	SFR_BASE + 30h	20h	Register to provide auxiliary values
Media Encryption Key (MEK)	SFR_BASE + 50h	40h	Register to provide MEK

SFR_BASE is an address that is configured on KMB. The integrator should make sure that KMB can access these SFRs through these addresses.

Control register

Figure 10 defines the Control register used to sequence the execution of a command and obtain the status of that command.

Figure 10: Offset SFR_Base + 0h: CTRL – Control

Bits	Type	Reset	Description
31	RO	0h	**Ready (RDY):** After an NVM Subsystem Reset, this bit is set to 1b, then the Encryption Engine is ready to execute commands. If this bit is set to 0b, then the Encryption Engine is not ready to execute commands.
30:20	RO	0h	Reserved
19:16	RO	0h	**Error (ERR):**> If the DONE bit is set to 1b by the Encryption Engine, then this field if set to a non-zero value to indicate the Encryption Engine detected an error during the execution the command specified by the CMD field. The definition of a non-zero value is vendor specific.

Value	Description
0h	Command Successful
1h to 3h	Reserved
4h to Fh	Vendor Specific

If the DONE bit is set to 1b by the Encryption Engine and this field is set to 0h, then the Encryption Engine is indicating a successful execution of a command specified by the CMD field.\n If the DONE bit is set to 1b by KMB, then the field is set to 0000b.

15:6RO0hReserved 2:5RW0h**Command (CMD):** This field specifies the command to execute or the command associated with the reported status.

Value	Description
0h	Reserved
1h	**Load MEK:** Load the key specified by the AUX field and MEK register into the Encryption Engine as specified by the METD field.
2h	**Unload MEK:** Unload the MEK from the Encryption Engine as specified by the METD field.

Value	Description
3h	**Sanitize:** Unload all of the MEKs from the Encryption Engine (i.e., Sanitize the Encryption Engine MEKs).
4h to Fh	Reserved

1RW0bDone (DN): This bit indicates the completion of a command by the Encryption Engine.\n If this bit is set to 1b by the Encryption Engine, then the Encryption Engine has completed the command specified by the CMD field.\n If the EXE bit is set to 1b and this bit is set to 1b, then the Encryption Engine has completed executing the command specified by the CMD field and the ERR field indicates the status of the execution of that command. A write of the value 1b to this bit shall cause the Encryption Engine to: - set this bit to 0b; - set the EXE bit to 0b; and - set the ERR field to 00b. 0RW0b**Execute (EXE):** A write of the value 1b to this bit specifies that the Encryption Engine is to execute the command specified by the CMD field. If the DONE bit is set to 1 by KMB, then the bit is set to 0b.

From the KMB, the Controller register is the register to write a command and receive its execution result. From its counterpart, the Encryption Engine, the Controller register is used to receive a command and write its execution result.

The expected change flow of the Controller register to handle a command is as follows:

1. If **RDY** is set to 1b, then KMB writes **CMD** and **EXE**
    1. **CMD:** either 1h, 2h or 3h
    2. **EXE:** 1b
2. The Encryption Engine writes **ERR** and **DN** 1.**ERR:** either 0b or a non-zero value depending on the execution result 2. **DN:** 1b
3. The KMB writes **DN**
    1. **DN:** 1b
4. The Encryption Engine writes **CMD**, **ERR**, **DN** and **EXE**
    1. **CMD:** 0h
    2. **ERR:** 0h
    3. **DN:** 0b
    4. **EXE:** 0b

The KMB therefore interacts with the Control register as follows in the normal circumstance:

1. The KMB writes **CMD** and **EXE**
    1. **CMD:** either 1h, 2h or 3h
    2. **EXE:** 1b
2. The KMB waits **DN** to be 1
3. The KMB writes **DN**
    1. **DN:** 1b
4. The KMB waits **DN** to be 0

Since the Controller register is in fact a part of the Encryption Engine whose implementation can be unique by each vendor, behaviors of the Control register with the unexpected flow are left for vendors. For example, a vendor who wants robustness might integrate a write-lock into the Control register in order to prevent two almost simultaneous writes on EXE bit.

Metadata Register

Figure 11 defines the Metadata register.

Figure 11: Offset SFR_Base + 10h: METD – Metadata

Bytes	Type	Reset	Description
19:00	RW	0b	**Metadata (METD):** This field specifies metadata that is vendor specific and specifies the entry in the Encryption Engine for the Encryption Key.

For the security goal of this project, the KMB and the Encryption Engine must be the only components which have access to MEKs. Each MEK must then be bound to a unique identifier, which can be accessible by other components, in order for an appropriate key to be used for any key-related operations including data I/O. In a LOCK-enabled system, the **METD** field is expected to be used as such identifier.

Instead of generating a random and unique identifier within the KMB while generating an MEK, the KMB takes an **METD** value as input from the controller firmware and write to the Metadata register without any modification for the sake of the following reasons:

1. A vendor does not need to implement an additional algorithm to map between identifiers in its own system and in the KMB
2. A vendor-unique key-retrieval algorithm can easily be leveraged into a **METD**-generation algorithm

In order to reduce ambiguity, two examples of the **METD** field will be given: Logical Block Addressing (LBA) range-based metadata; and key-tag based metadata.

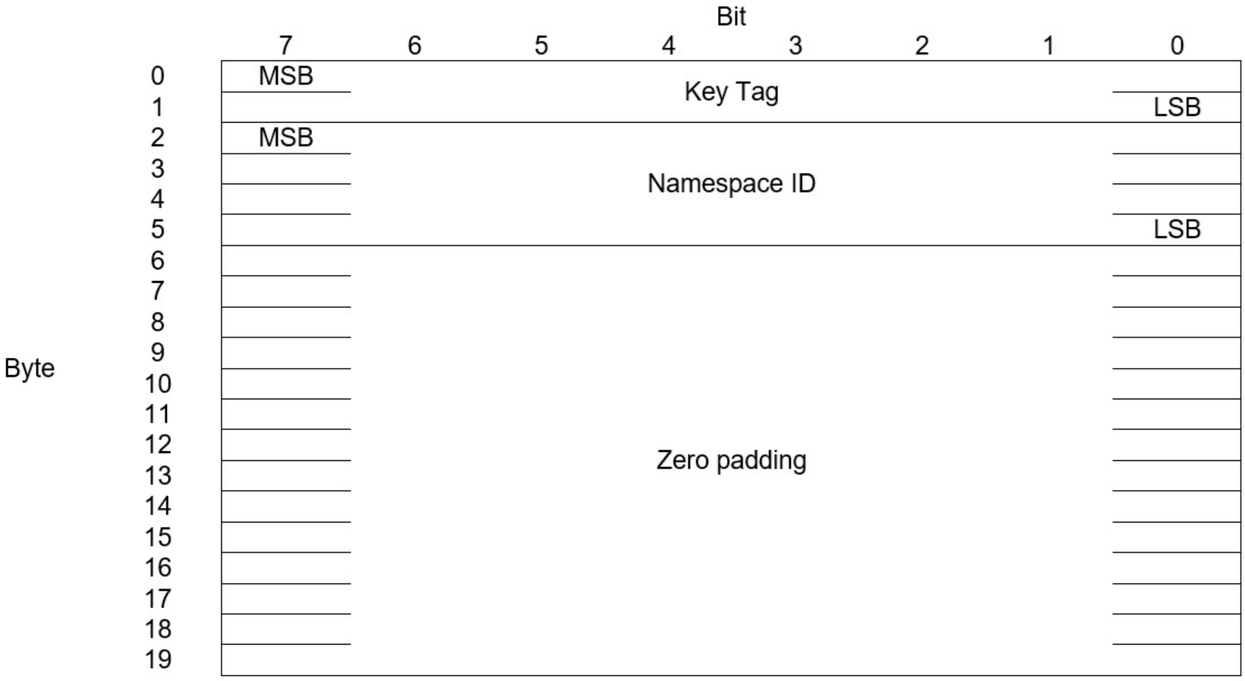
When an SSD stores data with address-based encryption, an MEK can be uniquely identified by a (LBA range, Namespace ID) pair. Then, the (LBA range, Namespace ID) pair can be leveraged into METD as on Figure 12.

Figure 12: LBA Range Based Metadata Format

	Bit								
		7	6	5	4	3	2	1	0
Byte	0	MSB	LBA Range End						
	1								
	2								
	3								
	4								
	5								
	6								
	7								LSB
	8	MSB	LBA Range Start						
	9								
	10								
	11								
	12								
	13								
	14								
	15								LSB
	16	MSB	Namespace ID						
	17								
	18								
	19								LSB

Address-based encryption is not however the only encryption mechanism in SSDs. For example, in TCG Key Per I/O, an MEK is selected by a key tag, which does not map to an address. Figure 13 shows an example of **METD** in such cases.

Figure 13: Key Tag Based Metadata Format



The above examples are not the only possible values of **METD**. Vendors are encouraged to design and use their own **METD** if it fits better to their system.

Auxiliary Data Register

Figure 14 defines the Auxiliary Data register.

Figure 14: Offset SFR_Base + 20h: AUX – Auxiliary Data

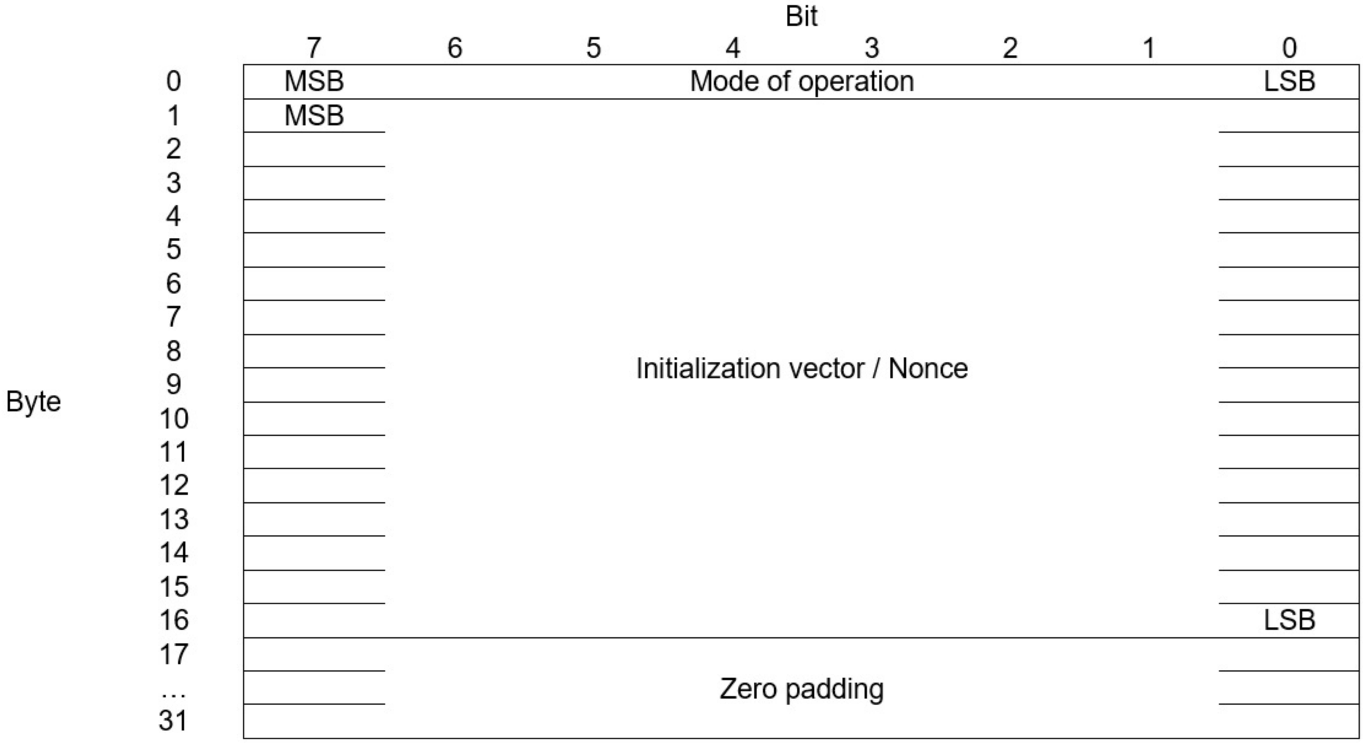
Bytes	Type	Reset	Description
19:00	RW	0h	**Auxiliary Data (AUX):** This field specifies auxiliary data associated to the MEK.

At first glance, the usage of **AUX** might not be straightforward. The intuition behind introducing the **AUX** field is to support vendor-specific features on MEKs. The KMB itself is only supporting fundamental functionalities in order to minimize attack surfaces on MEKs. Moreover, vendors are not restricted to design and implement their own MEK-related functionalities on the Encryption Engine unless they can be used to exfiltrate MEKs. In order to support these functionalities, some data may be associated and stored with an MEK and the **AUX** field is introduced to store such data with each MEK.

When the controller firmware instructs the KMB to generate a new MEK, the controller firmware is expected to provide an **AUX** value. Similar to the **METD** field, the KMB will write the **AUX** value into the Auxiliary Data register without any modification.

One simple use case of the **AUX** field is to store an offset of initialization vector or nonce. It can also be used in a more complicated use case. Here is an example. Suppose that there exists a vendor who wants to design a system which supports several modes of operation through the Encryption Engine while using the KMB. Then, a structure of **AUX** value as on Figure 15 can be used.

Figure 15: Auxiliary Data Format Example



When the controller firmware instructs KMB to generate an KMB, the controller firmware can use the **AUX** value to specify which mode of operation should be used and which value should be used as an initialization vector or a nonce with the generated MEK.

Media Encryption Key (MEK) register

Figure 16 defines the MEK register.

Figure 16: Offset SFR_Base + 40h: MEK – Media Encryption Key

Bytes	Type	Reset	Description
31:00	WO	0h	**Secret Encryption Key (SEK):** This field specifies a 256-bit Encryption Key
63:32	WO	0h	**Tweakable Key (TWK):** This is the 256-bit AES-XTS tweakable key.

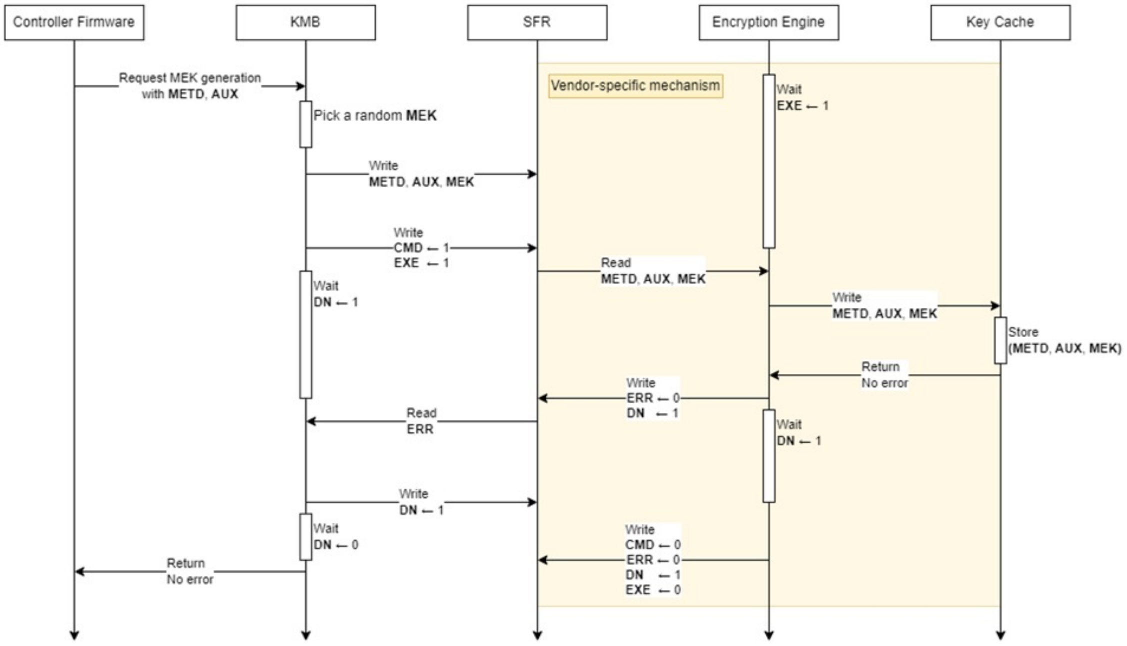
Since the AES-XTS is one of the most popular algorithms for data encryption, the MEK register is also designed with the key format of AES-XTS. The layout of the MEK register is however designed to help understanding the structure. The Encryption Engine is not restricted to only support the AES-XTS. The choice of encryption algorithm is solely dependent on vendors. When a vendor decides to use a different encryption algorithm, an MEK can be seen as a 64-byte random value rather than a (secret key, tweak key) pair and how to slice the 64-byte random value into an encryption key will be left to the vendor.

As a part of Caliptra, KMB protects MEKs as securely as any secret key in Caliptra. Within Caliptra, MEKs are only ever present in the Key Vault, so that they can be protected against any firmware-level attacks. When KMB needs to write an MEK into the MEK register, it will be accomplished by using the DMA engine. Given an index and a destination identifier, the DMA engine copies the key value stored in the key vault of given index to the destination address to which the DMA engine translates the destination identifier.

KMB command sequence

Figure 17 shows a sample command execution. This is an expected sequence when the controller firmware instructs the KMB to generate a new MEK. The internal behavior of the Encryption Engine is one of several possible mechanisms, and it can be different per vendor.

Figure 17: Command Execution Example



Storage root key fuse requirements

A storage device equipped with OCP L.O.C.K. will be equipped with N 256-bit ratchet-secret fuse banks, dubbed $R_0..R_{N-1}$. $4 \le N \le 16$. These ratchet secrets have the following requirements:

- Each ratchet secret can individually transition from all-zeroes → randomized → all-ones. R_X is only randomized once R_{X-1} has transitioned to all-ones.
- Programmable via the Caliptra fuse controller.
- Only readable by Caliptra Core, via fuse registers.
    - Internally, the fuse registers will be treated like the DICE UDS, in that their contents can only be deposited into Key Vault slots, without direct visibility by Caliptra firmware. -Caliptra Core firmware can detect which ratchet secrets are all-zeroes, randomized, or all-ones. This can be done by representing a counter in fuses, which maps to ratchet secret states. Controller firmware would be responsible for ensuring that the counter value corresponds with the ratchet secrets' current state.
      The counter values map to the following states:

State	Description

State	Description
4i + 0	Ri has begun being programmed with randomness, but is not yet considered randomized.
4i + 1	Ri has been randomized.
4i + 2	Ri has begun being programmed to all-ones, and is no longer considered randomized.
4i + 3	Ri has been programmed to all-ones.

This scheme allows ratchet state transitions to be resilient in the face of unexpected power loss. A power loss during randomization will burn the ratchet being randomized.

The counter is readable by Caliptra Core firmware and controller firmware.

Lifecycle transitions

The device will go through the following state transitions over its lifespan:

1. At the factory, R_0..R_{N-1} are all-zero.

    1. Caliptra derives a storage root key from a non-ratchetable secret derived from the DICE UDS + field entropy.
    2. Caliptra allows MEKs to be programmed to the storage encryption engine, derived from the storage root key.
    3. Caliptra firmware reports a bit indicating that it is operating in a state where any data written cannot be ratchet-erased.

2. The storage controller programs R_0 with randomness.

    1. Caliptra detects that R_0 is randomized, and derives its OCP L.O.C.K. storage root key from R_0 and a non-ratchetable secret derived from DICE UDS + field entropy.
    2. Caliptra allows MEKs to be programmed to the storage encryption engine, derived from the storage root key.

3. The storage controller programs R_0 to all-ones and resets.

    1. Upon next reset, Caliptra detects that there are no randomized ratchet secrets and does not derive a storage root key.
    2. Caliptra does not allow any MEKs to be programmed to the storage encryption engine.

4. The storage controller programs R_1 to a random value and resets.

    1. Upon next reset, Caliptra detects that R_1 is randomized, and derives its OCP L.O.C.K. storage root key from R_1 and a non-ratchetable secret derived from the DICE UDS + field entropy.
    2. Caliptra allows MEKs to be programmed to the storage encryption engine, derived from the storage root key.

5. The storage controller programs R_1 to all-ones and resets.

6. The storage controller programs R_2 to a random value and resets.

7. The storage controller programs R_2 to all-ones and resets.

8. The storage controller programs R_3 to a random value and resets.

    1. See steps 2 and 3 for Caliptra's behavior in each state.

...

9. The storage controller programs R_{N-1} to all-ones and resets.

    1. Upon next reset, Caliptra detects that there are no randomized ratchet secrets, and no all-zeroes ratchet secrets.
    2. Caliptra derives a storage root key from a non-ratchetable secret derived from the DICE UDS + field entropy.
    3. Caliptra allows MEKs to be programmed to the storage encryption engine, derived from the storage root key.
    4. Caliptra firmware reports a bit indicating that it is operating in a perma-dirty state, where no future ratchets are possible.

Storage root key fuse programming

OCP L.O.C.K contains a fused block that contains the ability to program N number of Storage Root Keys one at a time. A device out of manufacturing does not program a System Root Key and the device behaves as existing devices do today where the SSD firmware manages the key. This is known as the EMPTY state. OCP L.O.C.K supports a PROGAM_NEXT_ROOT_KEY API to then program the initial System Root Key. That System Root Key is then used to derive all MEKs. That System Root Key can be erased by using the ERASE_CURRENT_ROOT_KEY Api. Once that System Root Key is erased, no System Root Key exists in OCP L.O.C.K. and no MEKs can be generated by OCP L.O.C.K until the PROGAM_NEXT_ROOT_KEY API causes a new System Root Key to be programmed.

Once all of the N Storage Root Keys have been programmed and erased, no System Root Key exists in OCP L.O.C.K. and no MEKs can be generated by OCP L.O.C.K. The ENABLE_IO_WITHOUT_RATCHET API can be used to permanently allow the device to behave as existing devices do today where the SSD firmware manages the key.

There can be errors programming a System Root Key and erasing a System Root Key. If these errors occur, the APIs can be use to retry the operation. If there still is an error, then the device is in an unusable state.

The diagram below has an example flow where the number of System Rook Keys available to be programmed is 3 (N=3):

State transition	active_slot	slot_state	next_action
Factory	0	EMPTY	PROGRAM
Program first entry	0	PROGRAMMED	ERASE
Erase first entry (recoverable failure)	0	PARTIALLY_ERASED	ERASE
Erase first entry	0	ERASED	PROGRAM

State transition	active_slot	slot_state	next_action
Program next entry (recoverable failure)	1	PARTIALLY_PROGRAMMED	PROGRAM
Program next entry (unrecoverable failure)	1	PARTIALLY_PROGRAMMED	ERASE
Erase failed entry	1	ERASED	PROGRAM
Program next entry	2	PROGRAMMED	ERASE
Erase final entry	2	ERASED	ENABLE_IO_WITHOUT_RATCHET
Enable perma dirty	2	NON_FUSE_RATCHETABLE	NONE

Error reporting and handling

This section describes OCP L.O.C.K error reporting and handling.

This section will be fleshed out with additional details as they become available.

Fatal errors

This section will be fleshed out with additional details as they become available.

Non-fatal errors

This section will be fleshed out with additional details as they become available.

Runtime firmware environment

This section provides an extension to the runtime firmware environment defined for OCP Caliptra runtime firmware specification due to the support of OCP L.O.C.K.

Boot and initialization

This section defines additional boot and initialization flows needed to support OCP L.O.C.K.

The Runtime Firmware main function SHALL cause the generation of the Storage Root Key and store this into the Key Vault. The Storage Root key is generated from the Storage Root Key fuses.

The following sections define the additional Caliptra mailbox commands due to supporting OCP L.O.C.K.

GET_STATUS

Exposes a command that allows the SoC to determine if the Encryption Engine is ready to process commands as well vendor-defined drive crypto engine status data.

Command Code: 0x4753_5441 (“GSTA”)

Table: GET_STATUS input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.

Table: GET_STATUS output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error

engine_readyu32 Ready status of the storage device crypto engine:

- OCP L.O.C.K. defines low range, vendor defines high range
- Byte 0:
    - Bit 0: 1 = Ready 0 = Not ready

reservedu32[4]Reserved

GET_ALGORITHMS

Exposes a command that allows the SoC to determine the types of algorithms supported by KBM for endorsement, KEM, PMEK, and access key generation.

Command Code: 0x4743_4150 (“GCAP”)

Table: GET_ALGORITHMS input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.

Table: GET_ALGORITHMS output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
endorsement_algorithms	u32	Identifies the supported endorsement algorithms: • Byte 0 ◦ bit 0: ecdsa_secp384r1_sha384[^2]
hpke_algorithms	u32	Identifies the supported HPKE algorithms: • Byte 0 ◦ bit 0: kem_id = 0x0011, aead_id = 0x0002, kdf_id = 0x0002[^5] ◦ bit 1: kem_id = 0x0a25, aead_id = 0x0002, kdf_id = 0x0002[^6]

pmek_algorithms	u32	Indicates the size of PMEKs: • Byte 0 ○ bit 0: 256 bits
access_key_algorithm	u32	Indicates the size of access keys: • Byte 0 ○ bit 0: 256 bits, with a 128-bit truncated SHA384 ID
Reserved	u32[4]	Reserved

CLEAR_KEY_CACHE

This command unloads all MEKs in the Encryption Engine and deletes all keys in KMB.

Command Code: 0x4353_4543 (“CSEC”)?????

Table: CLEAR_KEY_CACHE input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
rdy_timeout	u32	Timeout in ms for encryption engine to become ready for a new command
cmd_timeout	u32	Timeout in ms for command to crypto engine to complete

Table: CLEAR_KEY_CACHE output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error

ENDORSE_ENCAPSULATION_PUB_KEY

This command generates a signed certificate for the specified KEM using the specified endorsement algorithm.

Command Code: 0x4E45_505B (“EEPK”)

Table: ENDORSE_ENCAPSULATION_PUB_KEY input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian
kem_handle	u32	Handle for KEM keypair held in KMB memory
endorsement_algorithm	u32	Endorsement algorithm identifier. If 0h, then return public key

Table: ENDORSE_ENCAPSULATION_PUB_KEY output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian
fips_status	u32	Indicates if the command is FIPS approved or an error
pub_key	KemPubKey	KEM public key
endorsement_len	u32	Length of endorsement data (N)
endorsement	u8[N]	DER-encoded X.509 certificate (includes nonce as extension)

ROTATE_ENCAPSULATION_KEY

This command rotates the KEM keypair indicated by the specified handle and stores the new KEM keypair in volatile memory within KMB.

Command Code: 0x5245_4E4B ("RENK")

Table: ROTATE_ENCAPSULATION_KEY input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian
kem_handle	u32	Handle for old KEM keypair held in KMB memory

Table: ROTATE_ENCAPSULATION_KEY output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian
fips_status	u32	Indicates if the command is FIPS approved or an error
kem_handle	u32	Handle for new KEM keypair held in KMB memory

GENERATE_PMEK

This command unwraps the specified access key, generates a random PMEK, then uses the Storage Root Key and access key to encrypt the PMEK which is returned for the Storage Controller to persistently store.

Command Code: 0x5245_4E4B ("RENK")

Table: GENERATE_PMEK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
pmek_algorithms	u32	Indicates the size of PMEKs. Only one bit shall be

		reported: - Byte 0 - bit 0: 256 bits
wrapped_access_key	WrappedAccessKey	KEM-wrapped access key: - access_key_algorithm - kem_handle - kem_algorithm - kem_ciphertext - encrypted_access_key

Table: GENERATE_PMEK output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error
new_encrypted_pmek	EncryptedPmek	PMEK encrypted to access_key_2

REWRAP_PMEK

This command Unwraps access_key_1 and enc_access_key_2. Then access_key_1 is used to decrypt enc_access_key_2. The specified PMEK is decrypted using KDF(Storage root key, "PMEK", access_key_1). A new PMEK is encrypted with the output of KDF(Storage root key, "PMEK", access_key_2). The new encrypted PMEK is returned.

The Storage Controller stores the returned new encrypted PMEK. The Storage Controller may attempt to do a decryption the new PMEK without an error before deleting old PMEK. Controller firmware erases the old encrypted PMEK.

Command Code: 0x5245_5750 (“REWP”)

Table: REWRAP_PMEK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
wrapped_access_key_1	WrappedAccessKey	KEM-wrapped access key: - access_key_algorithm - kem_handle (X)m - kem_algorith - kem_ciphertext - encrypted_access_key

wrapped_enc_access_key_2	DoubleWrappedAccessKey	KEM-wrapped (access_key_2 encrypted to access_key_1): • double_encrypted_access_key
old_locked_pmek	EncryptedPmek	PMEK encrypted to access_key_1

Table: REWRAP_PMEK output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error
new_encrypted_pmek	EncryptedPmek	PMEK encrypted to access_key_2

UNLOCK_PMEK

This command Unwraps wrapped_access_key. Then the unwrapped access_key is used to decrypt locked_pmek using KDF(Storage root key, "PMEK", access_key). An "unlocked" PMEK is encrypted with the the ephemeral export secret. The encrypted unlocked PMEK is returned.

Command Code: 0x554E_4C50 (“UNLP”)

Table: UNLOCK_PMEK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
wrapped_access_key	WrappedAccessKey	KEM-wrapped access key: • kem_handle • kem_algorithm • kem_ciphertext • encrypted_access_key
locked_pmek	EncryptedPmek	PMEK encrypted to storage root key and access key

Table: UNLOCK_PMEK output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error
unlocked_pmek	EncryptedPmek	PMEK encrypted to an export secret that is rotated on reset

MIX_PMEK

This command initializes the MEK seed if not already initialized, decrypts the specified PMEK with the with the ephemeral export secret, and then updates the MEK seed in KMB by performing a KDF with the MEK seed, the decrypted PMEK, and the string PMEK mix".

When generating an MEK, one or more MIX_PMEK commands are processed to modify the MEK seed.

Command Code: 0x4D50_4D4B ("MPMK")

Table: MIX_PMEK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
unlocked_pmek	EncryptedPmek	PMEK encrypted to an export secret that is rotated on reset

Table: MIX_PMEK output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error

LOAD_MEK

This command causes the specified controller data encryption key to be combined with the MEK seed. The final MEK seed, specified metadata, and aux_metadata are loaded into the Encryption Engine Key Cache. The metadata is specific to the storage controller and specifies the information to the Encryption Engine on where within the Key Cache, the MEK is loaded.

The storage controller specified data encryption key may be a C_PIN-derived secret for Opal or a per-MEK value in KPIO.

The final MEK is generated by performing a KDF on the existing MEK seed in the KMB, the dek, and the string "MEK".

When generating an MEK, the MEK seed is initialized if no PMEK has already been inserted into the MEK seed.

Command Code: 0x4C4D_454B ("LMEK")

Table: LOAD_MEK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
metadata	u8[20]	Metadata for MEK to load into the drive crypto engine (i.e. NSID + LBA range)
aux_metadata	u8[32]	Auxiliary metadata for the MEK (optional; i.e. operation mode)

rdy_timeout	u32	Timeout in ms for encryption engine to become ready for a new command
cmd_timeout	u32	Checksum over other input arguments, computed by the caller. Little endian.
dek	u8[32]	Controller-supplied "data encryption key: • May be a C_PIN-derived secret in Opal or a per-MEK value in KPIO

Table: LOAD_MEK output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error

UNLOAD_MEK

This command causes the MEK associated to the specified metadata to be unloaded for the Key Cache of the Encryption Engine. The metadata is specific to the storage controller and specifies the information to the Encryption Engine on where within the Key Cache, the MEK is loaded.

Command Code: 0x554D_454B (“UMEK”)

Table: UNLOAD_MEK input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
metadata	u8[20]	Metadata for MEK to load into the drive crypto engine (i.e. NSID + LBA range)
rdy_timeout	u32	Timeout in ms for encryption engine to become ready for a new command
cmd_timeout	u32	Timeout in ms for command to crypto engine to complete

Table: UNLOAD_MEK output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian
fips_status	u32	Indicates if the command is FIPS approved or an error

ENUMERATE_KEM_HANDLES

This command returns a list of all currently-active KEM handles for resources held by KMB.

Command Code: 0x4548_444C (“EHDL”)

Table: ENUMERATE_KEM_HANDLES input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.

Table: ENUMERATE_KEM_HANDLES output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error
kem_handle_count	u32	Number of KEM handles (N)
kem_handles	KEMHandle[N]	List of (KEM handle value, KEM algorithm) tuples

ERASE_CURRENT_ROOT_KEY

This command program all un-programmed bits in the current root key slot, so all bits are programmed. May resume a previously-failed erase operation.

Command Code: 0x4543_524B ("ECRK")

Table: ERASE_CURRENT_ROOT_KEY input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.

Table: ERASE_CURRENT_ROOT_KEY output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error

PROGRAM_NEXT_ROOT_KEY

This command generates a random key and program it into the next-available root key slot. May resume a previously-failed program operation, if HW supports that.

Command Code: 504E_524B ("PNRK")

Table: PROGRAM_NEXT_ROOT_KEY input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.

Table: PROGRAM_NEXT_ROOT_KEY output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error

ENABLE_IO_WITHOUT_RATCHET

This command enables a perma-dirty state where I/O is permitted, but no root key slots are left to program.

Command Code: 4550_4443 (“EPDS”)

Table: ENABLE_IO_WITHOUT_RATCHET input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.

Table: ENABLE_IO_WITHOUT_RATCHET output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error

REPORT_ROOT_KEY_STATE

This command reports the state tof the System Rook Key.

Command Code: 5252_4B53 (“RRKS”)

Table: REPORT_ROOT_KEY_STATE input arguments

Name	Type	Description
chksum	u32	Checksum over other input arguments, computed by the caller. Little endian.
nonce	u8[16]	Freshness nonce

Table: REPORT_ROOT_KEY_STATE output arguments

Name	Type	Description
chksum	u32	Checksum over other output arguments, computed by Caliptra. Little endian.
fips_status	u32	Indicates if the command is FIPS approved or an error
total_slots	u16	Total number of root-key slots
active_slot	u16	Currently-active root-key slots
slot_state	SlotState (u16)	State of the currently-active slot

Value	Description
0h	EMPTY (Able to load MEKs)
1h	PARTIALLY_PROGRAMMED (Not to load MEKs)

Value	Description
2h	PROGRAMMED (Able to load MEKs)
3h	PARTIALLY_ERASED (Not to load MEKs)
4h	ERASED (Not to load MEKs)
5h	NON_FUSE_RATCHETABLE (Able to load MEKs)
6h to FFFFh	Reserved

next_actionNextAction (u16) Next action that can be taken on the active slot

Value	Description
0h	NONE
1h	PROGRAM
2h	ERASE
3h	ENABLE_IO_WITHOUT_RATCHET
4h to FFFFh	Reserved

eat_lenu16Total length of the IETF EAT eatu8[eat_len]CBOR-encoded and signed IETF EAT

Fault handling

A mailbox command can fail to complete in the following ways due to OCP L.O.C.K.:

- An ill-formed command
- Encryption Engine timeout
- Encryption Engine reported error

In all of these cases, the error is reported in the command returned status.

Depending on the type of fault, the SoC may to resubmit the mailbox command.

Each mailbox command that causes a command to execute on the Encryption Engine includes a timeout value is specified by the command. Caliptra aborts the command executing on the Encryption Engine if the Encryption Engine does not complete the command within the specified timeout and reports a LOCK_ENGINE_TIMEOUT result code.

Table 3 defines the additional Caliptra result codes due to supporting OCP L.O.C.K.

Table 3: OCP L.O.C.K. mailbox command result codes

Table 2: OCP L.O.C.K. mailbox command result codes

Name	Value	Description
LOCK_ENGINE_TIMEOUT	0x4C45_544F (“LETO”)	Timeout occurred when communicating with the drive crypto engine to execute a command
LOCK_ENGINE_CODE + u16	0x4443_xxxx (“ECxx”)	Vendor-specific error code in the low 16 bits

Name	Value	Description
LOCK_BAD_ALGORITHM	0x4C42_414C (“LBAL”)	Unsupported algorithm, or algorithm does not match the given handle
LOCK_BAD_HANDLE	0x4C42_4841 (“LBHA”)	Unknown handle
LOCK_NO_HANDLES	0x 4C4E_4841 (“LNHA”)	Too many extant handles exist
LOCK_KEM_DECAPSULATION	0x4C4B_4445 (“LKDE”)	Error during KEM decapsulation
LOCK_ACCESS_KEY_UNWRAP	0x4C41_4B55 (“LAKU”)	Error during access key decryption
LOCK_PMEK_DECRYPT	0x4C50_4445 (“LPDE”)	Error during PMEK decryption

Terminology

The following acronyms and abbreviations are used throughout this document.

Abbreviation	Description
AES	Advanced Encryption Standard
CSP	Cloud Service Provider
DEK	Data Encryption Key
DICE	Device Identifier Composition Engine
DRBG	Deterministic Random Bit Generator
ECDH	Elliptic-curve Diffie–Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
HMAC	Hash-Based Message Authentication Code
KDF	Key Derivation Function
KEM	Key Encapsulation Mechanism
KMB	Key Management Block
L.O.C.K.	Layered Open-Source Cryptographic Key-management
MEK	Media Encryption Key
ML-KEM	Module-Lattice-Based Key-Encapsulation Mechanism
NIST	National Institute of Standards and Technology

Abbreviation	Description
OCP	Open Compute Project
PMEK	Partial Media Encryption Key
RTL	Register Transfer Level
SED	Self-encrypting drive
SSD	Solid-state drive
UART	Universal asynchronous receiver-transmitter
XTS	XEX-based tweaked-codebook mode with ciphertext stealing

Acknowledgements

The Caliptra Workgroup acknowledges the following individuals for their contributions to this specification:

Currently this list is identical to the list contributors.

References

1. Self-encrypting deception: weaknesses in the encryption of solid state drives by Carlo Meijer and Bernard van Gastel
2. TCG Opal
3. TCG Key Per I/O
4. RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3
5. RFC 9180: Hybrid Public Key Encryption
6. Hybrid PQ/T Key Encapsulation Mechanisms