Platform Security Firmware Update for the A-profile Arm Architecture

Non-confidential

Notice

This document is an Alpha version of a specification undergoing review by Arm partners. It is provided to give advanced information only.

Document number: DEN0118

Contents

Release information

Date	Version	Changes
2021/Apr/06	1.0ALP3	• Removed the trial_run state variable • Added per-image accepted flags.
2021/Jan/18	1.0ALP2	• Document name changed
2021/Jan/15	1.0ALP1	• First external release • Document name changed
2021/Jan/11	1.0ALP0	• First internal release

Arm Non-Confidential Document Licence (“Licence”)

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

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

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

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

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

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

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

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

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

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

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

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

Any breach of this Licence by a Subsidiary shall entitle Arm to terminate this Licence as if you were the party in breach. Any termination of this Licence shall be effective in respect of all Subsidiaries. Any rights granted to any Subsidiary hereunder shall automatically terminate upon such Subsidiary ceasing to be a Subsidiary.

LES-PRE-21585 version 4.0

About this document

Terms and abbreviations

Term	Meaning
BMC	Baseboard management controller
Client	The entity that holds the FW images to be updated.
FF-A implementation	The supervisory software in EL2, EL3, S-EL2 that implements the FF-A protocol.
FSM	Finite state machine
FW	Firmware
GPT	GUID Partition Table
GUID	Globally Unique Identifier
MBZ	Must be zero
NV	Non-volatile
Protocol-updatable bank	A collection of FW images updatable using the protocol defined in this document.
RoT	Root of trust
ROTPK	Root of trust public key
Secure State	The Arm Execution state that enables access to the Secure and Non-secure systems resources, for example memory, peripherals, and System registers.
Update Agent	The entity that receives the FW images sent from the Client and which serializes these to the NV memory.

References

This section lists publications by Arm and by third parties.

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

[1] *Platform Security Boot Guide*. (1.1) Arm.

[2] *Unified Extensible Firmware Interface*. (2.8) UEFI Forum Inc.

[3] *Arm Firmware Framework for Armv8-A*. (1.0) Arm.

Feedback

Arm welcomes feedback on its documentation.

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

- The title (Platform Security Firmware Update for the A-profile Arm Architecture).
- The document ID and version (DEN0118 1.0ALP3).

- The page numbers to which your comments apply.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

1 Overview

This document defines a firmware update protocol. The protocol assumes that the firmware images, provided by the Client, are stored in a NV memory that is managed by a separate entity, termed the Update Agent.

A common system design will place the Update Agent in the Secure World while the Client executes in the Non-secure World.

This document defines a set of primitives to transfer the FW images from the Client to the Update Agent. The document also defines the state variables used to govern the system execution.

Guidelines are provided in this document on anti-rollback counter management.

The security properties of the protocol defined in this document rely on a trusted boot procedure to be implemented. The trusted boot procedure must comply with PSBG [1].

This document defines an ABI meant for a Normal world entity to transmit the FW images to an FW Update Agent.

A common platform design would have a UEFI [2] interface, within the Non-secure State, exposed to the host OS. In those platform designs, the OS should install new FW by passing a FMP [2] formatted capsule to the capsule update abstraction[2] defined in UEFI. The ABI defined in this document should be entirely used from within the UEFI abstraction, an OS should not call any of the ABI primitives directly.

Note

This document is one of a set of resources provided by Arm that can help organisations develop products that meet the security requirements of PSA Certified on Arm-based platforms. The PSA Certified scheme provides a framework and methodology that helps silicon manufacturers, system software providers and OEMs to develop more secure products. Arm resources that support PSA Certified range from threat models, standard architectures that simplify development and increase portability, and open-source partnerships that provide ready-to-use software. You can read more about PSA Certified here: www.psacertified.org and find more Arm resources here: developer.arm.com/platform-security-resources.

---

2 System design concepts

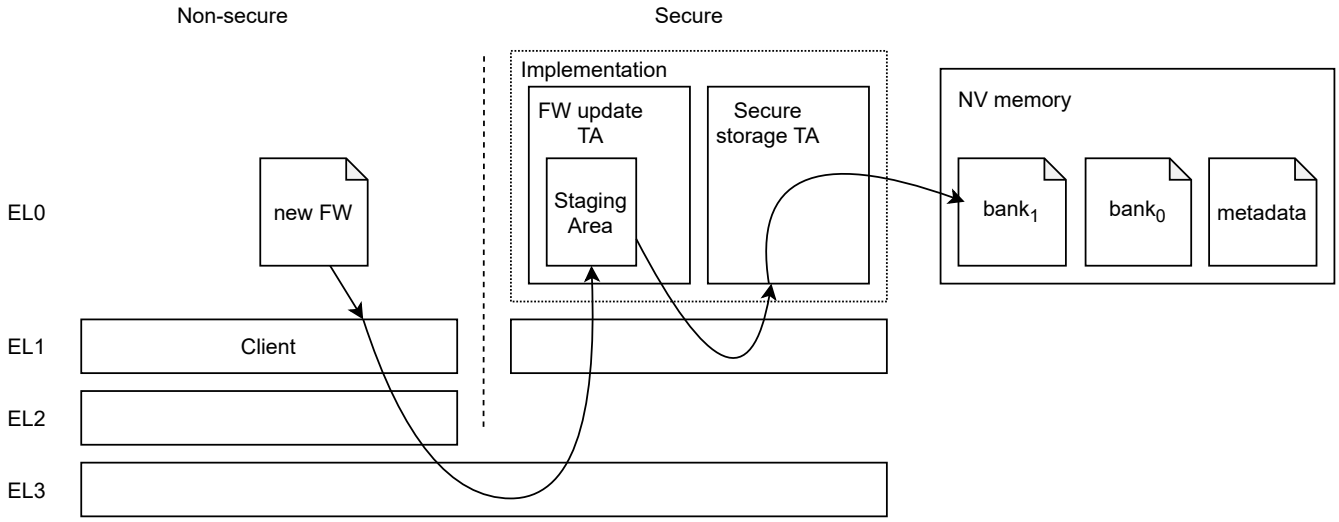

Figure 1: System diagram

The diagram in Figure 1 depicts a possible system architecture where the Client and Update Agent execute in the Non-secure and Secure World respectively. In this example system, there exist two FW image banks ($bank_0$ and $bank_1$). At any point in time there is a single *active* image bank and a single *update* image bank. The number of banks in the system is platform defined, see Section 2.1 for more information.

A system must contain the following entities:

1. FW update client (Client)
   - Originator of the FW images to be updated.
2. FW update agent (Update Agent)
   - Execution context isolated from the Client. It receives the FW images transmitted from the Client and is responsible for serializing those to a NV storage.
   - Optionally, the Update Agent can perform FW image authentication before updating the NV storage.

Messages exchanged between the Client and the Update Agent are forwarded by FW compliant with the FF-A protocol [3] running at EL2/EL3/S-EL2/S-EL1.

The Update Agent context is identified by the *update_agent_uuid* UUID (see Table 3)

2.1 Firmware banks

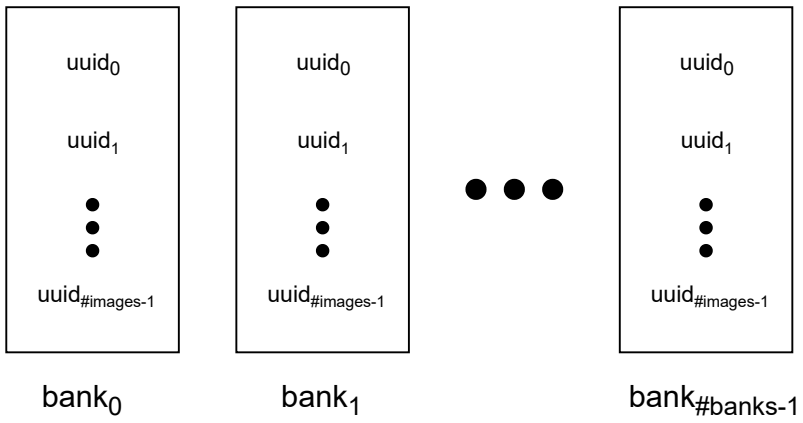

Figure 2: FW banks

The Update Agent maintains an IMPLEMENTATION DEFINED number of FW banks (*#banks*), each bank contains an IMPLEMENTATION DEFINED number of FW images (#images). All the banks contain the exact same type of images, each FW image type is identified by a UUID.

At any point in time, there exits an *active* and an *update* bank.

The *active_index* is maintained by the Update Agent in the metadata, see Section 4.1.

The *update_index* is only visible to the Update Agent. The *update_index* value is set by the Update Agent during its initialization and kept as a volatile variable.

Additionally the Update Agent records, in the metadata, the previous active bank (*previous_active_index*). The bank identified by *previous_active_index* can be used as a fallback to boot the platfrom when an updated bank fails to properly boot.

All bank indices take values in the {0, . . . , *#banks*-1} range.

The initialization of the FW banks at system provisioning is IMPLEMENTATION DEFINED.

The bank classification is determined by the *active_index* and *update_index* state variables in the following manner:

update bank: $bank_{update\_index}$

active bank: $bank_{active\_index}$

A Client can only read from or write to images in the update bank.

When coming out of a cold reset, the platform attempts to boot with $bank_{active\_index}$. For further information about banks and the boot process see Section 2.2.1.

The Update Agent must keep track of the *previous_active_index* in the metadata (see Section 4.1). This is the index of the last active bank, it is used as an indication of a potentially viable fallback FW bank.

Note: a scenario where *active_index* = *update_index* is legal if *#banks*=1. For systems where *#banks* = 1 then *active_index* = *update_index* = 0.

2.2 Boot stages

The system boot process has the following stages:

- *immutable*
  - FW present in a (generally) non-writable memory.
  - If a *secondary* stage is not present, the *immutable* stage must be aware of the *protocol-updatable* stage presence and it must be able to read and interpret the image metadata (see Table 5).
- (optional) *secondary*
  - single image FW present in a writable NV memory. This stage cannot be updated using the protocol defined in this document, its update procedure is IMPLEMENTATION DEFINED, see Section 2.3.
  - this stage must be aware of the *protocol-updatable* stage presence and it must be able to read and interpret the image metadata (see Table 5).
- *protocol-updatable*
  - The stage that is updated using the protocol defined in this document.
  - The *protocol-updatable* images can contain any other FW images not involved in the boot process.
  - The *protocol-updatable* images are duplicated. Each instance of the *protocol-updatable* images is termed a $bank$. Each $bank$ is identified by a $bank$ index, see Section 2.1.

The trusted boot procedure starts at the *immutable* stage, optionally flowing to the *secondary* stage and afterwards to the *protocol-updatable* stage.

2.2.1 Platform Boot

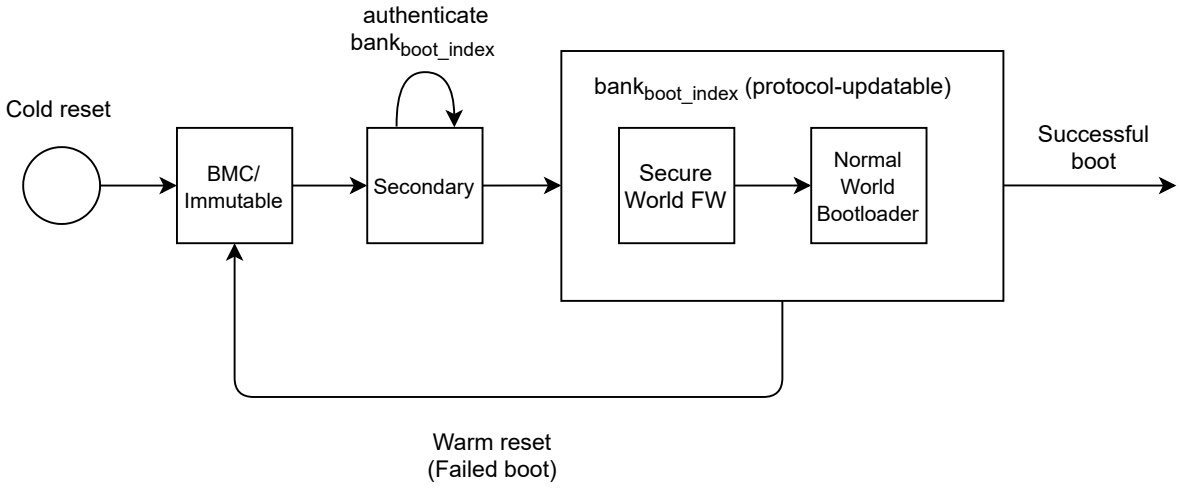

Figure 3: Boot overview

The *immutable* or secondary stage select the *protocol-updatable* bank to boot the system with ($bank_{boot\_index}$). Out of a cold reset: *boot_index=active_index*.

max_failed_boots: the maximum number of consecutive failed attempts to boot with a given bank. The immutable or secondary stages can identify a failed bank boot attempt by, for instance, inspecting the watchdog state. The mechanism to determine a failed bank boot attempt is IMPLEMENTATION DEFINED.

The *max_failed_boots* is a platform constant, its value is IMPLEMENTATION DEFINED.

Each *boot_index* assignment in the following list is attempted at most *max_failed_boot* times. After *max_failed_boots* consecutive warm rests, caused by a failure to boot the patform with a given assignment, the next assignment in the list must be attempted:

1. *boot_index* ← active_index
2. *boot_index* ← *previous_active_index*, if *active_index* ≠ *previous_active_index*, otherwise attempt item 3)
3. *boot_index* ← IMPLEMENTATION DEFINED bank index.

The *active_index* and *previous_active_index* are fields maintained by the Update Agent in the metadata, see Section 4.1.

The *immutable* or *secondary* stages can detect a boot failure during the *protocol-updatable* stage by inspecting a reset syndrome register. The nature of the reset syndrome register is IMPLEMENTATION DEFINED.

An authentication failure of a *protocol-updatable* bank implies a boot failure of that bank. An authentication failure is permanent until a bank is updated.

The *boot_index* must be propagated to the Update Agent. The mechanism to propagate the *boot_index* to the Update Agent is IMPLEMENTATION DEFINED.

2.3 Recovery Mode

The FW update protocol (defined in this document) allows for a fail-safe update of the *protocol-updatable* images. The FW update functionality relies on several FW components. Some of these components can themselves be updated using the FW update protocol defined in this document.

It is recommended that a new FW image bank is tested, prior to field updates, to ensure that it will be able to perform a subsequent FW update.

In rare scenarios, a system may become unable to perform FW updates. In such a scenario, or when the *secondary* stage must be updated, a recovery mode is used. The existence of a recovery mode is mandatory. The recovery mode implementation details are IMPLEMENTATION DEFINED.

The recovery mode can be implemented as:

- BMC assisted update.
- recovery mode executed from the *immutable* stage.

The recovery mode must:

- be able to write to the NV memory where the *protocol-updatable* images are stored at rest.
- be able to correctly update the FW update metadata (See Section 4.1)
- if a *secondary* stage exists, be able to write to the medium where the *secondary* stage is stored at.

2.4 Protocol UUIDs

The following UUIDs are used in the protocol definition. The UUIDs are referred to, in this document, by their UUID name.

The *update_agent_uuid* value is the identifier of the Update Agent, it can be used to bootstrap the communication between the Client and the Update Agent, as is detailed in Section 5.1.1.

The *fwu_directory_uuid* is the identifier of the image directory provided by the Update Agent. The image directory contains details about the FW images managed by the Update Agent, for more information see Section 4.2.

The *metadata_uuid* is the identifier of the metadata partition type when the metadata is stored within a GPT [2], see Section 4.1.2 for more information.

Table 3: protocol defined UUIDs

UUID	UUID name	description
6823a838-1b06-470e-9774-0cce8bfb53fd	*update_agent_uuid*	Update Agent context UUID

UUID	UUID name	description
deee58d9-5147-4ad3-a290-77666e2341a5	*fwu_directory_uuid*	The image directory UUID, see Section 4.2
8a7a84a0-8387-40f6-ab41-a8b9a5a60d23	*metadata_uuid*	The UUID of the metadata type, see Section 4.1

3 Firmware update state machine

At any given time the system can be in one of the following states:

- Regular - the system has booted from a verified FW bank.
- Staging - the procedure to update $bank_{update\_index}$ is undergoing.
- Trial - the system updated a FW bank, some images in this bank have not been accepted.

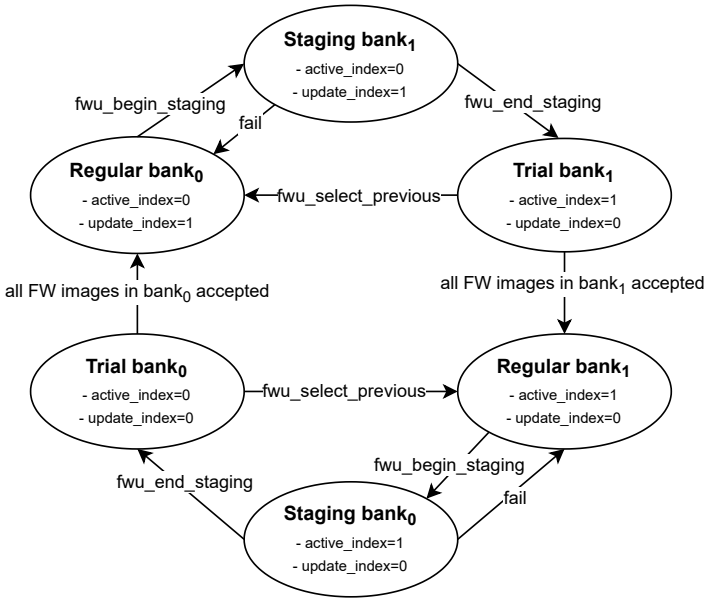

Figure 4: High level FSM

The diagram in Figure 4 depicts the state machine of a particular implementation of the firmware update protocol. In this example the Update Agent maintains 2 different FW banks ($bank_0$ and $bank_1$). For more information on the FW banks see Section 2.1.

The state transitions occur at the following boundaries:

- Regular to Staging:
  - when a fwu_begin_staging call returns successfully (see Section 5.2).
- Staging to Trial:
  - when the call fwu_end_staging returns successfully.
- Trial to Regular:
  - once all FW images in the active bank are accepted.

A detailed description of the Staging state is provided in Section 3.1, the Trial state if covered in Section 3.2.

The FW update protocol requires the following state variables:

1. *active_index*: integer indicating which FW bank is currently active. The variable is kept in the metadata region, see Table 5. Its value is updated, by the Update Agent, during the handling of a fwu_end_staging call (see Section 5.2).
2. *previous_active_index*: integer indicating which FW bank was active prior to the last FW update. The variable is kept in the metadata region, see Table 5. Its value is updated, by the Update Agent, during the handling of a fwu_end_staging call (see Section 5.2).
3. *update_index*: integer indicating which FW bank will be overwritten during a Staging state. This variable is set by the Update Agent at system boot and only visible to the Update Agent. The *update_index* must

respect the following contraint:

- if #banks > 1: *update_index* $\neq$ *active_index*.

4. image accepted status: Field recorded per-image and per-bank (see Table 7). The accepted status of all images in the $bank_{active\_index}$ determine if the system is in the Trial state, see Section 3.2.

3.1 Staging state

New FW images can only be communicated from the Client to the Update Agent during a Staging state.

The system transitions from the Regular to the Staging state once the fwu_begin_staging call successfully completes.

The Client must open an image, by invoking fwu_open, before writing to the image

Once a FW image context is open, a sequence of fwu_write_stream calls transmit the FW image to the Update Agent. The sequence diagram of the FW image transfer is shown in Figure 5.

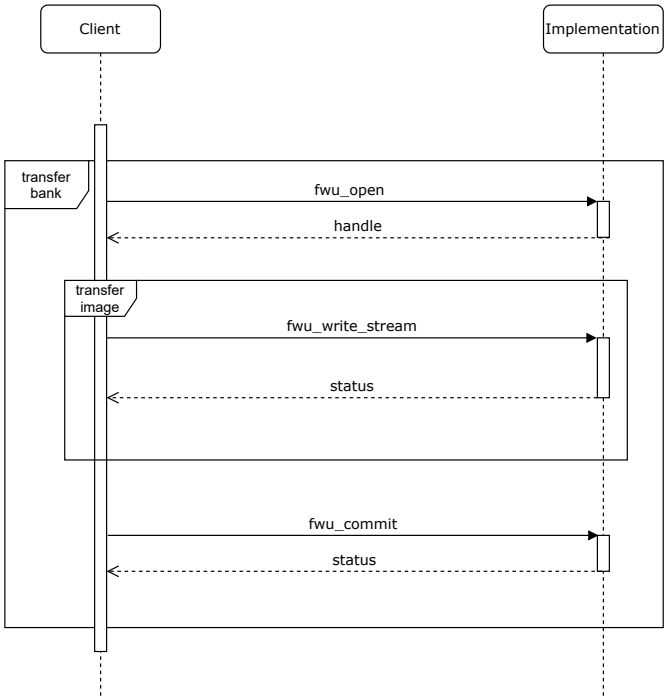

Figure 5: Staging procedure

The Update Agent can authenticate the staged FW images before committing those to the NV memory. This optional procedure is performed at the fwu_commit call. The image authentication procedure is detailed in Section 4.4.1.

The FW image authentication procedure before committing images to NV memory is:

- optional: if *#banks* > 1
- mandatory: if *#banks* = 1

The image authentication may fail, this is communicated to the Client by the fwu_commit call returning a AUTH_FAIL status code, see Section 5.2.8 for further details.

The Update Agent overwrites the images in $bank_{update\_index}$.

While handling a successful fwu_end_staging call, the Update Agent must:

- update *previous_active_index*, see Section 5.2.3.
- update *active_index*, see Section 5.2.3.

The Staging state correctly terminates when the fwu_end_staging call returns successfully.

The Staging state fails if:

1. the system resets prior to the Client calling fwu_end_staging.
2. the Client calls cancel_staging.

When the staging fails the system will transition back to the Regular state.

3.2 Trial state

The system is in the Trial state if any of the FW images in $bank_{active\_index}$ is pending acceptance, see Section 4.4.

While in the Trial state, the anti-rollback counters must not be updated.

Anti-rollback counter values must be updated while booting in the Regular state.

While booting in the Trial state, the trusted boot procedure must check that a FW image meets the version requirements of a subsequent Regular state boot. If the FW image does not meet the version requirements of a subsequent Regular state, the boot procedure fails.

The Client must invoke fwu_accept_image, for all the images currently unaccepted, in order for the Trial state to successfully terminate.

The Trial state fails if the Client calls fwu_select_previous_bank.

4 System management

4.1 FW update metadata

The metadata is a collection of fields, maintained by the Update Agent, as defined in Table 5.

The NV memory where the metadata is stored is IMPLEMENTATION DEFINED and agreed between the Update Agent and the *immutable* or *secondary* stages.

The metadata must:

- be readable by the *immutable* and *secondary* stages.
- be writable by the Update Agent.
- hold field values in a little-endian representation at the offsets defined in Table 5

The metadata is versioned using a 4 byte integer – *version* field in Table 5.

The metadata size is determined by the metadata version, the number of images per bank (#images) and the number of banks (#banks). The values *#images* and *#banks* are assumed to be constant from the point of view of this specification. The values of *#images* and *#banks* must be the same in the Update Agent and the *immutable* or *secondary* stages.

The metadata size, as a function of metadata version, is shown in Table 4.

Table 4: Metadata size per version

metadata version	*metadata_size*
1	10h + #images.(20h + #banks.18h)

The metadata representation is replicated to ensure reliable operation. The two metadata replicas must be updated in sequence.

There exists a CRC-32 field in the metadata, crc_32. The crc_32 value is updated in the following manner:

- crc_32 ← CRC32(metadata[4: metadata_size]).

During system boot, the *immutable* or *secondary* stage must use the procedure described in Section 4.1.1 to ensure that the metadata is intact.

During initialization, the Update Agent must correct a corrupted metadata region by copying the intact replica over.

The metadata replication and update in series guarantees reliability against system failures while the metadata is being updated. The replication and update in series does not detect malicious updates nor does it protect against erroneous updates to the metadata.

Note: The metadata can be maliciously crafted, it should be treated as an insecure information source.

Table 5: Metadata version 1

field	offset (bytes)	size (bytes)	Description
crc_32	0h	4h	
version	4h	4h	

field	offset (bytes)	size (bytes)	Description
active_index	8h	4h	
previous_active_index	Ch	4h	
img_entry [#images]	10h	#images.(30h + #banks.18h)	array of aggregate in Table 6

Table 6: Metadata image entry version 1

field	offset (bytes)	size (bytes)	Description
img_type_uuid	0h	10h	UUID identifying the image type
location_uuid	10h	10h	the UUID of the storage volume where the image is located
img_bank_info[#banks]	20h	18h.#banks	the properties of images with img_type_uuid in the different FW banks

Table 7: Image properties in a given FW bank version 1

field	offset (bytes)	size (bytes)	Description
img_uuid	0h	10h	the uuid of the image in this bank
accepted	10h	4h	• [0] : bit describing the image acceptance status – 1 means the image is accepted • [31:1] : MBZ
reserved	14h	4h	reserved (MBZ)

The metadata layout is defined in Table 5. The metadata contains an array of image entries (defined in Table 6) with *#images* elements.

4.1.1 Metadata integrity check

The integrity of the metadata must be verified before its information is consumed. The procedure to check the metadata integrity is detailed below:

```
metadata_check_integrity(metadata):

    if metadata.version = 1:
        metadata_size <- 10h + #images.(20h + #banks.18h)
    else:
        return False

    crc <- CRC32(metadata[4:metadata_size])
```

```
if crc != metadata.crc_32:
    return False

return True
```

4.1.2 Metadata integration with GPT [recommendation]

This is a guidance section, the aspects described in this section are not mandatory.

It is recommended that the layout of any NV medium containing FW images is defined by a GPT [2].

When embedded in a GPT, each metadata replica occupies a single partition with PartitionTypeGUID = *metadata_uuid*.

The platform may possess diferent NV mediums where FW images can be locatted at. All FW images of the same type should be located in the same NV medium. The location_uuid of each image type should match the DiskUUID [2] of the medium the image is located on.

4.2 Image directory

The Client can obtain details about the FW images handled by the Update Agent via the image directory. The Client reads the image directory by using the ABI defined in Section 5.2.

All fields in the image directory have a little-endian byte ordering.

The image directory is created by the Update Agent and reflects the information of the $bank_{boot\_index}$.

The contents of the directory are represented as an *image_directory* aggregate holding a list of *image_info_entries* with *num_images* (*#images*) elements. The *image_info* aggregate contains a subset of the information in the Metadata.

The Client opens the image directory with *handle_imgdir* = fwu_open(*fwu_directory_uuid*). The Client obtains the *image_info*, from the Update Agent, by calling fwu_read_stream(*handle_imgdir*, . . .) until the EOF.

Table 8: image_directory

field	offset (bytes)	size (bytes)	Description
directory_version	0h	4h	the version of the fields in the img_info_entry array.
num_images	4h	4h	the number of entries in the img_info_entry array.
active_index	8h	4h	the *active_index* field in the metadata.
boot_index	Ch	4h	the index of the bank that booted the platform, see Section 2.2.1.
img_info_entry[ ]	10h	–	array of Table 9 elements

The directory_version field determines the version of the image_info_entry.

Table 9: img_info_entry version 1

field	offset (bytes)	size (bytes)	Description
img_uuid	0h	10h	
client_permissions	10h	4h	bitfield specifying the access permissions that the Client has on the image: • [31:2] : MBZ • [1] : Read • [0] : Write
img_max_size	14h	4h	the maximum image size that can be installed.
lowest_accepted_version	18h	4h	the lowest version of the image that can execute on the platform (Table 6).
img_version	1Ch	4h	the image version in the $bank_{boot\_index}$ (Table 7).
accepted	20h	4h	the acceptance status of the image in the $bank_{boot\_index}$ (Table 7).
reserved	24h	4h	MBZ

4.3 Anti-rollback counter management

There exists at least one anti-rollback counter in the platform, as required in PSBG [1]. The anti-rollback counter value is monotonically increasing.

During image authentication, the image version is compared against the value of the anti-rollback counter the image is bound to. If an image has a lower value than the anti-rollback counter, then that image must not execute on the platform.

Every anti-rollback counter must:

- be readable by the *immutable* or *secondary* bootloader stages.
- be readable by the Update Agent, if the Update Agent performs the optional FW image authentication.
- be writable to by its managing entity.

The managing entity of each anti-rollback counter is IMPLEMENTATION DEFINED.

The Client can only communicate new anti-rollback counter values to the Update Agent during the Staging state. The format by which a new anti-rollback counter value is communicated to the Update Agent is IMPLEMENTATION DEFINED.

The anti-rollback counter must be updated, by its managing entity, after the end of a Trial state and before the completion of the subsequent system boot in the Regular state.

4.4 Protocol-updatable images

The protocol-updatable FW images are transferred from the Client to the Update Agent using the ABI defined in Section 5.2. The FW image format is IMPLEMENTATION DEFINED.

A FW image in a bank can have either an accepted or unaccepted status. The acceptance status of the image for a $bank_{index}$ is recorded in the following metadata image entry field:

- img_bank_info[index].accepted.

A value of 0 in the accepted field means the image is not accepted. A value of 1 in the accepted field means the image is accepted.

The Client can set the accepted status of an image by calling:

- fwu_commit: the client sets the accepted status of an image in the $bank_{update\_index}$, see Section 5.2.
- fwu_image_accept: the Client changes the accepted status of an image in the $bank_{active\_index}$ to be accepted, see Section 5.2.

A bank must have certificates for each of the FW images in the bank.

4.4.1 Image authentication

Updated firmware images must be authenticated prior to the first execution on the platform. The image authentication should be PSBG compliant [1].

The authentication procedure:

1. must happen during a PSBG compliant trusted boot procedure [1].
2. is optionally performed by the Update Agent, prior to writing the image to the NV memory, as part of the fwu_commit function handling.

The (optional) image authentication procedure, implemented in the Update Agent, requires the Update Agent to have access to the ROTPK and the entire chain of trust. The method of provisioning the ROTPK and the chain of trust to the Update Agent is IMPLEMENTATION DEFINED.

Every FW image is bound to a specific anti-rollback counter. The image to anti-rollback counter binding is IMPLEMENTATION DEFINED.

Any FW image must have a version greater or equal than its associated anti-rollback counter to be allowed execution in the platform.

The image authentication procedure is composed of the following checks:

- FW image creator authenticity check, the procedure is IMPLEMENTATION DEFINED.
- Verification that the FW image version is greater than the NV anti-rollback counter.

A failure of either check results in an image authentication failure.

Note: Prior to updating the images using the protocol described in this document, the Client may opt to perform an image authentication using a different chain of trust. This procedure is IMPLEMENTATION DEFINED.

5 Normal World Client ABI

5.1 Transport layer

The FW update ABI uses FF-A as the transport layer [3].

Prior to calling any FW update function the Client must perform a setup procedure where a shared buffer is established between the Client and the Update Agent.

5.1.1 Setup phase

The Client must trigger the following procedure with the Update Agent:

1. Client obtains the SP id of the Update Agent (*update_agent_id*) using the ffa_partition_info_get call with *update_agent_uuid* as a parameter.
2. Client shares the page pointed to by *client_buffer_va* with the Update Agent by calling ffa_mem_share passing *update_agent_id* and *client_buffer_va* as parameters. The Client receives a globally unique handle (*buffer_handle*) to the shared buffer.
3. Client sends a synchronous message to the Update Agent communicating *buffer_handle*.
4. Update Agent retrieves the VA of the page referred to by *buffer_handle* (*update_agent_buffer_va*).
5. Update Agent sends a synchronous response to the Client signaling a successful buffer exchange.

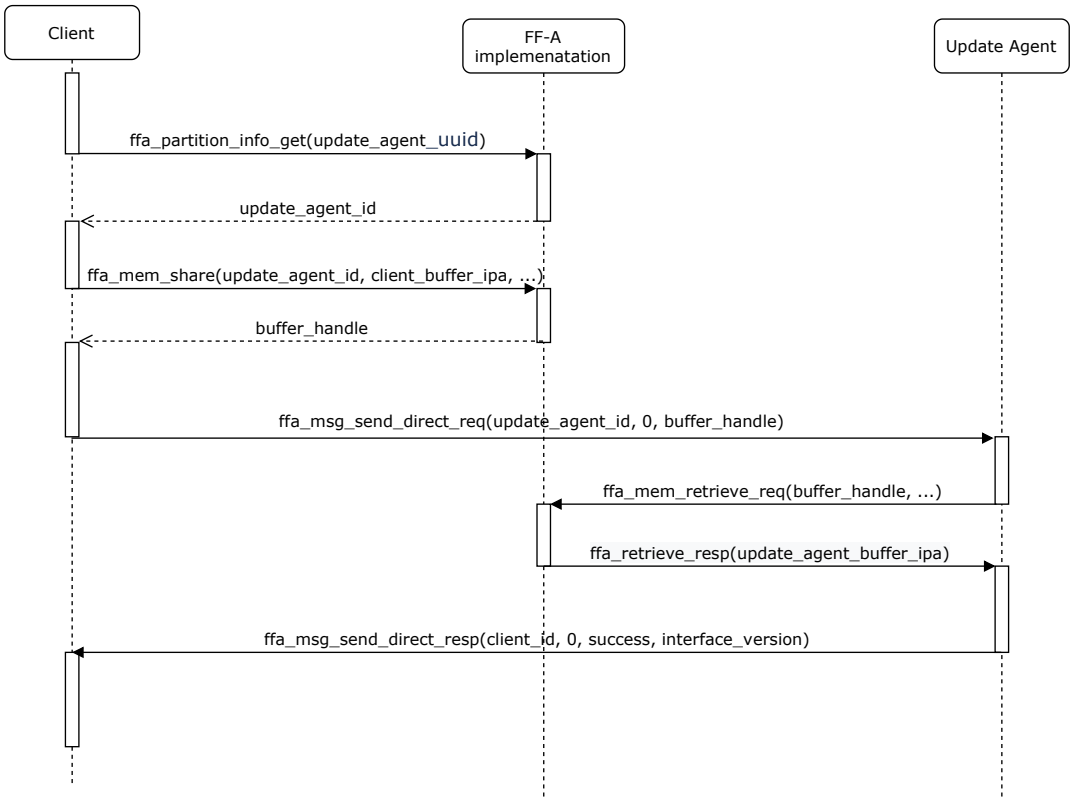

Figure 6: Transport layer setup

After a successful completion of the setup phase, the FW update ABI calls can be issued. In case of failure in the setup phase, the Client must assume the FW update protocol to be unavailable.

5.2 ABI definition

The ABI calls rely on FF-A synchronous messages and the buffer exchanged during the setup phase. The communication buffer has a size of *comm_buffer_size* bytes. The Client keeps the VA of the shared buffer in *client_buffer_va*. The Update Agent keeps the VA of the shared buffer in *update_agent_buffer_va*.

The calls defined in this ABI are a contract between the caller (Client) and the callee (Update Agent).

The caller must:

1. fill in the argument structure, as defined in the function argument definition below, onto the shared buffer.
2. call ffa_msg_send_direct_req with a *update_agent_id* destination.

The callee must:

1. fill in the return structure, as defined in the function return definition below, onto the shared buffer.
2. call ffa_msg_send_direct_resp.

The Client and the Update Agent may agree on a transport protocol level header placed at the start of the communication buffer, before the argument/result headers. This is outside the scope of this document.

Both the argument and result headers must be aligned at a 8 byte boundary.

All fields in the Argument and Return structures are little-endian.

5.2.1 fwu_discover

This call indicates the version of the ABI alongside a list of the implemented functions. The array *function_presence* contains *num_func* entries. The entry at function_presence[index] is an 8 bit integer indicating if the function is implemented and additional function features.

If function_presence[index]:

- $= 0$: function with function_id $=$ index is not implemented.
- $= 1$: function with function_id $=$ index is implemented.
- > 1: function with function_id $=$ index is implemented, additionally the returned value indicates function features specified in the function definition.

5.2.1.1 Arguments

field	offset (bytes)	size (bytes)	description
function_id=0	0	4	

5.2.1.2 Returns

field	offset (bytes)	size (bytes)	description
status	0	4	• SUCCESS
version_major	4	1	the ABI major version
version_minor	5	1	the ABI minor version
num_func	6	2	the number of entries in the function_presence array.
function_presence[]	8	num_func	array of bytes indicating functions that are implemented. The value function_presence[index] specifies the features of the function with function_id $=$ index.

5.2.2 fwu_begin_staging

This call indicates to the Update Agent that a new staging process will commence. The Client can only invoke this call during the Regular and Staging states. When the call is invoked during the Staging state, any transient state that might be held by the Update Agent is discarded.

This call is disallowed when *boot_index* $\neq$ *active_index*.

5.2.2.1 Arguments

field	offset (bytes)	size (bytes)	description
function_id=1	0	4	

5.2.2.2 Returns

field	offset (bytes)	size (bytes)	description
status	0	4	• SUCCESS • UNAVAILABLE: The system is in the Trial state or *boot_index* $\neq$ *active_index*.

5.2.3 fwu_end_staging

The Client informs the Update Agent that all the images meant to be updated have been transferred to the Update Agent and that the staging has terminated. This call can only be invoked from the Staging state. The Client must ensure that all image handles are closed before invoking this call.

During a successful call the Update Agent performs the following steps in order:

1. if *update_index* $\neq$ *active_index* then *previous_active_index* is updated: *previous_active_index* $\leftarrow$ *active_index*.
2. the *active_index* is updated: *active_index* $\leftarrow$ *update_index*.

5.2.3.1 Arguments

field	offset (bytes)	size (bytes)	description
function_id=2	0	4	

5.2.3.2 Returns

field	offset (bytes)	size (bytes)	description
status	0	4	• SUCCESS • UNAVAILABLE: The system is not in a Staging state. • DENIED: There are open image handles. • AUTH_FAIL: At least one of the images in in $bank_{update\_index}$ fails to authenticate.

5.2.4 fwu_cancel_staging

The Client cancels the staging procedure and the system transitions back to the Regular state. This call can only be invoked from the Staging state.

5.2.4.1 Arguments

field	offset (bytes)	size (bytes)	description
function_id=3	0	4	

5.2.4.2 Returns

field	offset (bytes)	size (bytes)	description
status	0	4	• SUCCESS • UNAVAILABLE: The system is not in a Staging state

5.2.5 fwu_open

The open call returns a handle to the image with UUID=image_uuid. The Client uses the handle in subsequent calls to read from or write to the image. An image can have a single active handle. If multiple fwu_open calls are performed on a given UUID, only the last returned handle is valid. This call cannot be invoked in the trial state.

5.2.5.1 Arguments

field	offset (bytes)	size (bytes)	description
function_id=4	0	4	
image_uuid	4	16	UUID of the image to be opened

5.2.5.2 Returns

field	offset (bytes)	size (bytes)	description
status	0	4	• SUCCESS: Call completed correctly. Remaining return fields are valid • UNKNOWN: image with UUID=image_uuid does not exist • UNAVAILABLE: The system is in the Trial state
handle	4	4	staging context identifier

5.2.6 fwu_write_stream

The call writes at most *max_payload_size* bytes to the Update Agent context pointed to by handle, where *max_payload_size* = *comm_buffer_size* - offset_of(fwu_write_stream_arguments, payload). The data to be written is passed in the payload present in the shared buffer, after the end of the arguments header. A Client can only invoke the call during a Staging state.

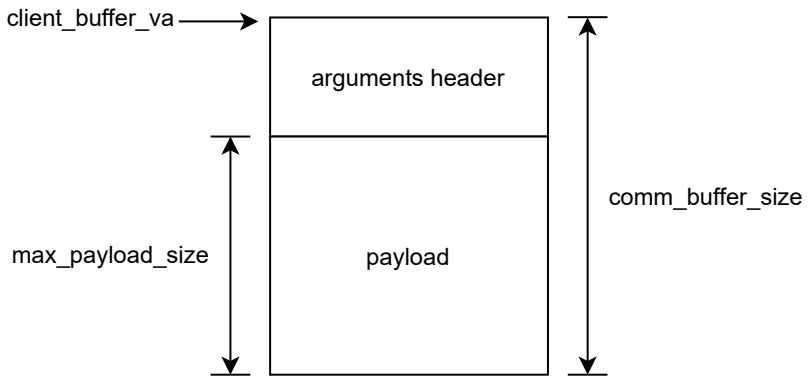

Figure 7: fwu_write_stream arguments in shared buffer

5.2.6.1 Arguments

field	offset (bytes)	size (bytes)	description
function_id=5	0h	4	
handle	4h	4	The handle of the context being written to.
data_len	8h	4	Size of the data present in the payload
payload	Ch	–	The data to be transferred

5.2.6.2 Returns

field	offset (bytes)	size (bytes)	description
status	0	4	• SUCCESS • UNKNOWN: unrecognized handle • OUT_OF_BOUNDS: less than data_len bytes available in the image. • NO_PERMISSION: The image cannot be written to. • UNAVAILABLE: The system is not in a Staging state

5.2.7 fwu_read_stream

The call reads at most max_payload_size bytes from the Update Agent context pointed to by handle.

The data to be read is passed in the payload contained in the shared buffer, after the end of the returns header.

- The field total_bytes, in the return, can be used by the Client after a first invocation to reserve enough memory to store the file being read.
- The field total_bytes can also be used by the Client to track when EOF is reached.
- EOF is also detected by a read_stream if $0 \leqslant$ *read_bytes* $<$ *max_payload_size*, where *max_payload_size* = *comm_buffer_size* - offset_of(read_stream_return, payload).

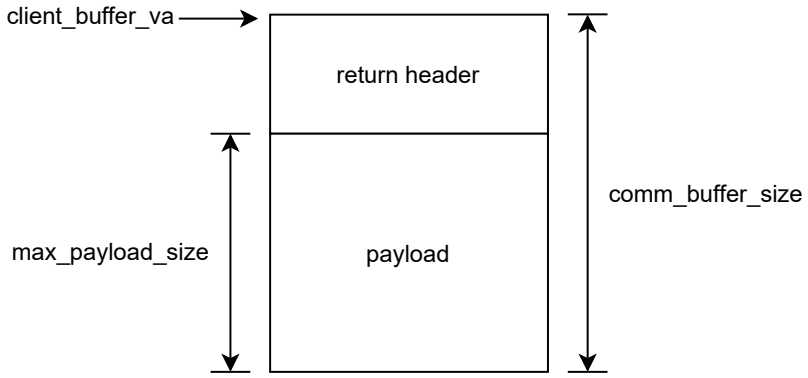

Figure 8: fwu_read_stream returns in shared buffer

5.2.7.1 Arguments

field	offset (bytes)	size (bytes)	description
function_id=6	0	4	
handle	4	4	The handle of the context being read from.

5.2.7.2 Returns

field	offset (bytes)	size (bytes)	description
status	0h	4	• SUCCESS: remaining return fields are valid. • UNKNOWN: handle is not recognized. • NO_PERMISSION: The image cannot be read from. • UNAVAILABLE: The image cannot be temporarily read from.
read_bytes	4h	4	
total_bytes	8h	4	
payload	Ch	–	

5.2.8 fwu_commit

The call closes the image pointed to by handle. A return of AUTH_FAIL signals an image authentication failure in the Update Agent. As with SUCCESS, an AUTH_FAIL return status implies that the handle is closed.

The Update Agent must set the image entry metadata field img_bank_info[update_index].accepted to:

- 0: if acceptance_req $>$ 0;
- 1: if acceptance_req $=$ 0;

The Client passes the *max_atomic_len* hint argument, specifying the length of time (in ns) that the Client can widstand the Update Agent to execute continuously without yielding back. If *max_atomic_len*=0 then the Client can tolerate an unbounded execution time by the Update Agent. The Update Agent should yield back to the Client before *max_atomic_len* ns elapse.

When the Update Agent yields before completing the call, it must return the RESUME status. If the Update Agent returns the RESUME status, then it must return the *total_work* and *progress* fields.

Note: The ratio of *progress* and *total_work* gives the proportion of outstanding work.

The Update Agent must continue calling fwu_commit, while the return is RESUME. The acceptance_req argument is ignored, by the Update Agent, for any subsequent fwu_commit call following a RESUME return status.

5.2.8.1 Arguments

field	offset (bytes)	size (bytes)	description
function_id=7	0h	4	
handle	4h	4	The handle of the context being closed.
acceptance_req	8h	4	If positive, the Client requests the image to be marked as unaccepted.
max_atomic_len	Ch	4	Hint, maximum time (in ns) that the Update Agent can execute continuously without yielding back to the Client. A value of 0 means that the Update Agent can execute for an unbounded time.

5.2.8.2 Returns

field	offset (bytes)	size (bytes)	description
total_work	0	4	• Units of work the Update Agent must perform until fwu_commit returns successfully.
progress	0	4	• Units of work already completed by the Update Agent.

field	offset (bytes)	size (bytes)	description
status	0	4	• SUCCESS • UNKNOWN: unrecognized handle. • AUTH_FAIL: image closed, authentication failed. • RESUME: the Update Agent yielded, the Client must invoke the call again.

5.2.9 fwu_accept_image

The call sets the metadata img_bank_info[active_index].accepted=1 for the image with type = image_type_uuid. This call can only be invoked if the system booted with $bank_{active\_index}$.

5.2.9.1 Arguments

field	offset (bytes)	size (bytes)	description
function_id=9	0	4	
reserved	4	4	MBZ
image_type_uuid	8	10h	

5.2.9.2 Returns

field	offset (bytes)	size (bytes)	description
status	0	4	• SUCCESS • UNKNOWN: image with type=image_type_uuid is not managed by the Update Agent. • UNAVAILABLE: the system has not booted with $bank_{active\_index}$.

5.2.10 fwu_select_previous

The call sets the *active_index* in the metadata to *previous_active_index*. This call is only available when:

- the system is in the Trial state.
- *boot_index* $=$ *previous_active_index*.

While handling this call:

- the *active_index* is updated: *active_index* $\leftarrow$ *previous_active_index*.
- the *previous_active_index* is updated: IMPLEMENTATION DEFINED assignment.

5.2.10.1 Arguments

field	offset (bytes)	size (bytes)	description
function_id=10	0	4	

5.2.10.2 Returns

field	offset (bytes)	size (bytes)	description
status	0	4	• SUCCESS • UNAVAILABLE: the system is not in the Trial state. • AUTH_FAIL: At least one of the images in in $bank_{previous\_update\_index}$ fails to authenticate.

6 Return status

status	value
SUCCESS	0
UNKNOWN	-1
UNAVAILABLE	-2
OUT_OF_BOUNDS	-3
AUTH_FAIL	-4
NO_PERMISSION	-5
DENIED	-6
RESUME	-7

Part I

In-band updates on systems with a BMC

Some systems have a BMC between the NV FW storage and the Host SoC.

The BMC can restrict the Host from accessing NV FW storage.

The Host may be unable to directly update the FW images in the NV storage.

There are two possible models (B1 and B2) with respect to how the BMC obstructs the view that the Host has to the NV storage:

	B1	B2
Host has direct R/W access to NV storage	Y	N
Host accesses the NV storage indirectly via the BMC	Y	Y

- B1: Host has read and write access to the entire NV storage.
- B2: Host can only indirectly read and write to the NV storage by delegating to the BMC.

In B1 the Host must synchronize with the BMC to ensure that the BMC will not concurrently access the update bank.

In B2 the Host must send the FW images to the BMC using the ABI previously defined.

ABI implementation in B2

The ABI implementation between the BMC and the Host requires:

- a shared buffer between the Host and the BMC;
- an event triggered by the Host, delivered to the BMC which the BMC must acknowledge back to the Host.
- an event triggered by the BMC, delivered to the Host, where the BMC signals request termination.

Host and BMC synchronization in both models

Whether the Host can access the NV storage directly (B1) or indirectly via the BMC (B2), the Host must inform the BMC when the Host intends to enter a phase where it will write to the NV storage.

When transitioning to the Staging state, the Host performs an IMPLEMENTATION DEFINED synchronization with the BMC. This synchronization mechanism gives the Host full permission to directly, or indirectly via the BMC, access the NV storage.

The synchronization mechanism requires:

- an event triggered by the Host and delivered to the BMC, which the BMC must acknowledge.

While the Host is in the Staging state, the BMC can only write to the NV storage if the Host commands it to.

The BMC can deny the Host entrance into the Staging state via an IMPLEMENTATION DEFINED return to the request from the Host.

The Staging state terminates:

- if the Host resets
- if the Host explicitly calls fwu_end_staging.

Once the Staging state terminates the BMC regains the right to access the NV storage.

If the Host takes too long in the Staging state, the BMC can send an IMPLEMENTATION DEFINED termination event. That event signals to the Host that the BMC can resume writing to NV storage and that the Host must cease any direct accesses or that indirect write requests, via the BMC, will be denied. Once the BMC has sent this event it can resume writing to NV storage immediately.

Host boot

In B2 the BMC can create the illusion that there exists a single FW bank in the NV storage. In this case, the platform does not require a FWU metadata exposed to the Host.

In B1 the BMC must maintain a FWU Metadata such that the Host bootloader knows which bank to boot with.

Part II

Normal World controlled FW store

Some platform designs assign the NV memory, where the FW store resides, to the direct control of the Normal World. In these platform designs the Client may execute from within the context that has read/write access to the FW store. In that case, the Client takes the role of the Update Agent and is responsible for writing the FW images to the FW store.

State machine

The FW update state machine is composed only of the Regular and Trial states. The state transitions occur at the following boundaries:

- Regular to Trial: Once the Client updates the active_index field in the metadata and the new $bank_{active\_index}$ has any un-accepted FW images.
- Trial to Regular: Once the Client has marked all images in the $bank_{active\_index}$ as accepted in the metadata.

After writing each FW image, the Client must set the image entry metadata field img_bank_info[update_index].accepted to:

- 0: if the Client intends to defer the image acceptance;
- 1: if the Client intends to accept the image immediately.

The system is in the Trial state while any image in the current $bank_{active\_index}$ is not accepted.

FW directory information

The Client must be able to obtain the data otherwise provided by the FW directory (see Section 4.2).

The following fields are present in the metadata:

- *active_index*
- per-image *img_uuid*
- per-image $bank_{active\_index}$ *accepted* flag.

The remaing FW directory fields must be obtained by the Client via an IMPLEMENTATION DEFINED procedure.