Platform Security Firmware Update for the A-profile Arm Architecture

Document number	DEN0118
Document quality	BET1
Document version	1.0 BET1
Document confidentiality	Non-confidential

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.

Notice

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

Release information

Date	Version	Changes
2023/Jul/05	1.0BET1	• Add telemetry buffer definition • Define the HAFW ACPI table • Minor text clarifications • Define image directory version 2 • Introduce metadata version 2 • Move the metadata definition and all the A/B Firmware Store design guidance to Appendix A
2021/May/27	1.0BET0	• Add minor guidelines around public key management in the Trial state • Add SRA
2021/Apr/06	1.0ALP3	• Removed the trial_run state variable • Added per-image accepted flags • Renamed fwu_close to fwu_commit • Redefined fwu_commit, allowing the implementation in the Update Agent to cooperatively yield
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

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

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.

The Document consists solely of commercial items. Licensee shall be responsible for ensuring that any use, duplication or disclosure of the Document complies fully with any relevant export laws and regulations to assure that the Document or any portion thereof is not exported, directly or indirectly, in violation of such export laws.

This Licence may be translated into other languages for convenience, and Licensee agrees that if there is any conflict between the English version of this Licence and any translation, the terms of the English version of this Licence shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. No licence, express, implied or otherwise, is granted to Licensee under this

Licence, to use the Arm trade marks in connection with the Document or any products based thereon. Visit Arm's website at http://www.arm.com/company/policies/trademarks for more information about Arm's trademarks.

The validity, construction and performance of this Licence shall be governed by English Law.

Copyright © 2022, 2023 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-21585 version 4.0

Contents

Platform Security Firmware Update for the A-profile Arm Architecture

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

Conventions

Typographical conventions

The typographical conventions are:

italic

Introduces special terminology, and denotes citations.

bold

Denotes signal names, and is used for terms in descriptive lists, where appropriate.

`monospace`

Used for assembler syntax descriptions, pseudocode, and source code examples.

Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.

SMALL CAPITALS

Used for some common terms such as IMPLEMENTATION DEFINED.

Used for a few terms that have specific technical meanings, and are included in the Glossary.

Red text

Indicates an open issue.

Blue text

Indicates a link. This can be

- A cross-reference to another location within the document
- A URL, for example http://developer.arm.com

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by `0b`, and hexadecimal numbers by `0x`. In both cases, the prefix and the associated value are written in a monospace font, for example `0xFFFF0000`. To improve readability, long numbers can be written with an underscore separator between every four characters, for example `0xFFFF_0000_0000_0000`. Ignore any underscores when interpreting the value of a number.

Pseudocode descriptions

This book uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in a monospace font. The pseudocode language is described in the Arm Architecture Reference Manual.

Assembler syntax descriptions

This book contains numerous syntax descriptions for assembler instructions and for components of assembler instructions. These are shown in a `monospace` font.

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*. Arm.

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

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

[4] *Advanced Configuration and Power Interface (ACPI) Specification*. UEFI Forum, Inc.

[5] *Arm Architecture Reference Manual for A-profile architecture*. Arm.

[6] *Platform Level Data Model (PLDM) for Firmware Update Specification*. DMTF.

[7] *STRIDE chart*. See https://www.microsoft.com/security/blog/2007/09/11/stride-chart

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

If you have any comments or suggestions for additions and improvements create a ticket at https://support.developer.arm.com/. As part of the ticket include:

- The title (Platform Security Firmware Update for the A-profile Arm Architecture).
- The number (DEN0118 1.0 BET1).
- The section name to which your comments refer.
- The page number(s) to which your comments apply.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Glossary

BMC

Baseboard management controller

Client

The entity that holds the firmware images to be updated.

CPU rendezvous

Pause all CPUs but the worker CPU. Note that CPU rendezvous for the purposes of live activation implies the usage of PSCI_CPU_PAUSE.

FF-A implementation

The supervisory software in EL2, EL3, S-EL2 that implements the FF-A protocol.

Firmware componenet activation

The procedure that places a particular firmawre component instance in execution on the platform.

Firmware Store

The non-volatile memory containing the firmware images that are executed on the platform.

FM

Runtime Firmware Manager

FSM

Finite state machine

FW

Firmware

GPT

GUID Partition Table

GUID

Globally Unique Identifier

Live activation

The procedure of activating a firmware image instance, replacing a previously running instance, while the system remains in execution.

LSB

Least Significant Byte

MBZ

Must be zero

NV

Non-volatile

Protocol-updatable bank

A collection of firmware 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 firmware images sent from the Client and which serializes these to the Firmware Store.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

Chapter 1
Introduction

This document defines an architecture for firmware update on Arm A-profile systems.

The firmware update flow can be seen as being composed of two activities:

1. Update firmware images in the Firmware Store
2. Activate the newly updated firmware images.

The document defines the concept of an Update Agent that controls the Firmware Store. The Update Agent can be implemented within a Secure Partition, in the Secure World – The Firmware Store update ABI (3.4 *Firmware Store Update ABI*), defined in this document, allows for the firmware blobs to be communicated to the Update Agent by a Client in the Host OS.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

1.1 Updates to the Firmware Store

Updates to the Firmware Store involve an Update Client and an Update Agent.

The Client transfers the firmware images to the Update Agent. The latter component writes the images to the Firmware Store.

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 (Firmware Store update ABI, 3.4 *Firmware Store Update ABI*) for the Client to transfer the firmware images to the Update Agent, when the agent is in the Secure World.

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].

The Update Client in the Host OS can use the Firmware Store update ABI directly. This allows for higher OS availability because the update to the Firmware Store, and eventual flash erase requests, can be performed by the Update Agent while the OS is running. Progress information can also be provided to a user. This contrasts with other firmware blocking interfaces, such as UEFI UpdateCapsule, where progress information cannot be presented to the user and, in some instances, interrupts cannot be delivered to the Host OS.

Alternatively, the firmware can implement the UEFI [2] UpdateCapsule interface, and expose it to the Host OS as a UEFI runtime service. In this alternative, the OS installs new firmware by passing a FMP [2] formatted capsule to the capsule update abstraction[2] defined in UEFI. The UpdateCapsule implementation uses the Firmware Store update ABI to transfer the images, contained in the Capsule to the Update Agent. UpdateCapsule is blocking from the point of view of the Host OS, and hence the Firmware Store update operation will be perceived as a long-running uninterruptible procedure where the calling PE is unavailable.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

1.2 Image activation

The second activity in firmware update, image activation, is commonly conducted via a full system reset.

Advanced platforms may implement activation in a reboot-less manner, called live activation. In a live activation flow, a sub-set of the firmware images becomes active without a full system reset. This functionality is out of scope for the current specification.

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.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

Chapter 2
Firmware Store update architecture overview

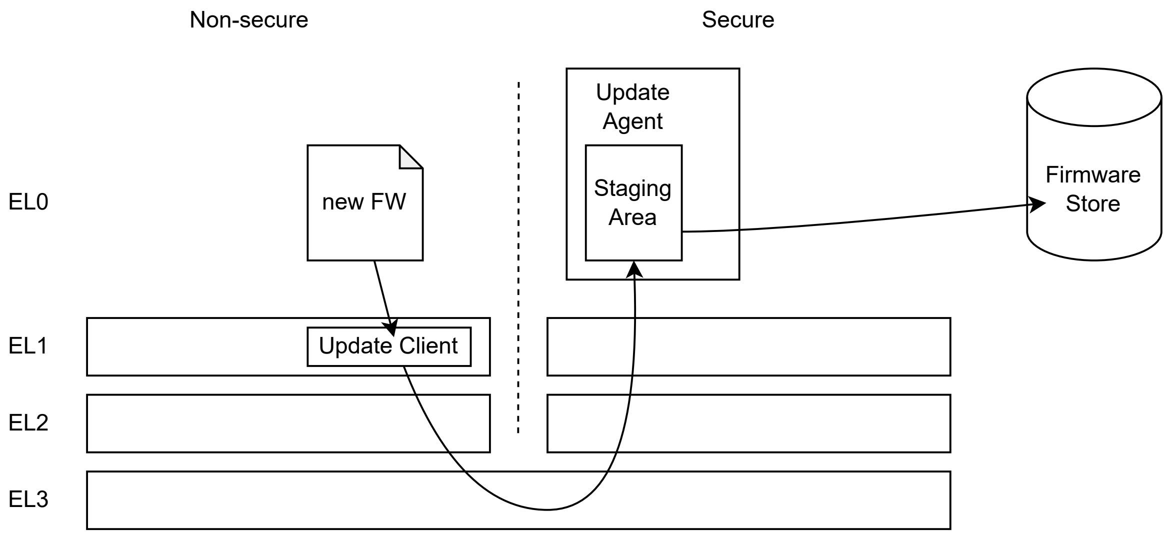

Figure 2.1: Firmware Store update system diagram

The diagram in Figure 2.1 depicts a possible system architecture where the Client and Update Agent execute in the Non-secure and Secure World respectively. The details of the Firmware Store (size, geometry, number of banks) are not visible to the Client. The Client simply interfaces with the Update Agent using the update ABI, see 3.4 *Firmware Store Update ABI*. The Update Agent manages the Firmware Store in a platform-specific manner.

A system contains the following entities:

1. Firmware update client (Client)
    - Originator of the firmware images to be updated.
2. Firmware update agent (Update Agent)
    - Entity that receives the firmware images, transmitted from the Client, and is responsible for serializing those to the Firmware Store (a NV storage).
    - Optionally, the Update Agent can perform firmware image authentication before updating the Firmware Store.

Messages exchanged between the Client and the Update Agent, use FF-A as a transport, and thus are forwarded by firmware 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_guid* GUID (see Table 3.1)

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

2.1 System boot on platforms with an updatable Firmware Store.

The system boot process has the following stages:

- *immutable*
  - Firmware 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.
- (optional) *secondary*
  - single image firmware present in a writable non-volatile memory. This stage cannot be updated using the protocol defined in this document, its update procedure is IMPLEMENTATION DEFINED, see 2.2 *Recovery Mode*.
  - this stage must be aware of the *protocol-updatable* stage presence.
- *protocol-updatable*
  - The stage that is updated using the Firmware Store update protocol, defined in this document.
  - The *protocol-updatable* images can contain any other firmware images not involved in the boot process.
  - The *protocol-updatable* images can be kept in a Firmware Store structured as a set of banks (the number of banks is IMPLEMENTATION DEFINED), see 3.1 *Firmware Store design*.
    - The immutable or secondary stage elect the bank to boot the platform with. This is IMPLEMENTATION DEFINED.

2.1.1 Platform Boot

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

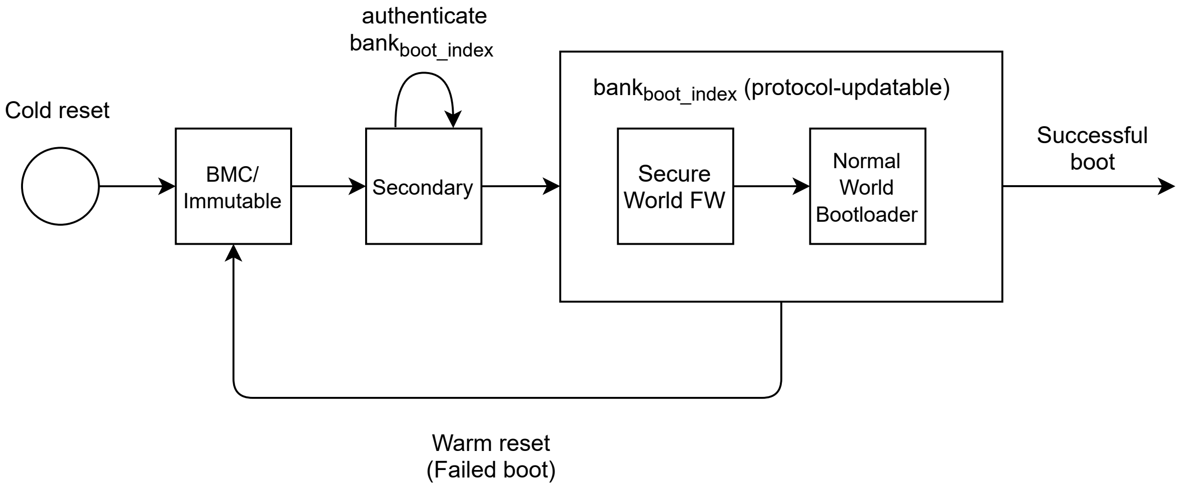

Figure 2.2: Boot overview

The *immutable* or secondary stage select the *protocol-updatable* images to boot the system with. The details of the *protocol-updatable* image selection are IMPLEMENTATION DEFINED.

The *immutable* or secondary stage may implement logic to detect failures in the boot process of the *protocol-updatable* images. This enables fail-safe boot. The details of the fail-safe procedure and adopted policies are IMPLEMENTATION DEFINED.

2.2 Recovery Mode

The Firmware Store update protocol allows for a fail-safe update of the *protocol-updatable* images. The Firmware Store update functionality relies on several firmware components. Some of these components are updatable.

It is recommended that a new set of firmware images is tested, prior to field updates, to ensure that the new set of firmware images can still perform a subsequent update to the Firmware Store.

In rare scenarios, a system may become unable to perform updates to the Firmware Store. 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 Firmware Store where the *protocol-updatable* images are stored at rest.
- be able to correctly configure the Firmware Store (details are IMPLEMENTATION DEFINED).
- if a *secondary* stage exists, be able to write to the medium where the *secondary* stage is stored at.

Chapter 3
Firmware Store update protocol

A Firmware Store is controlled by an Update Agent in the Secure World. On some platform designs, the Update Agent can alternatively be in the Normal World.

3.1 Firmware Store design

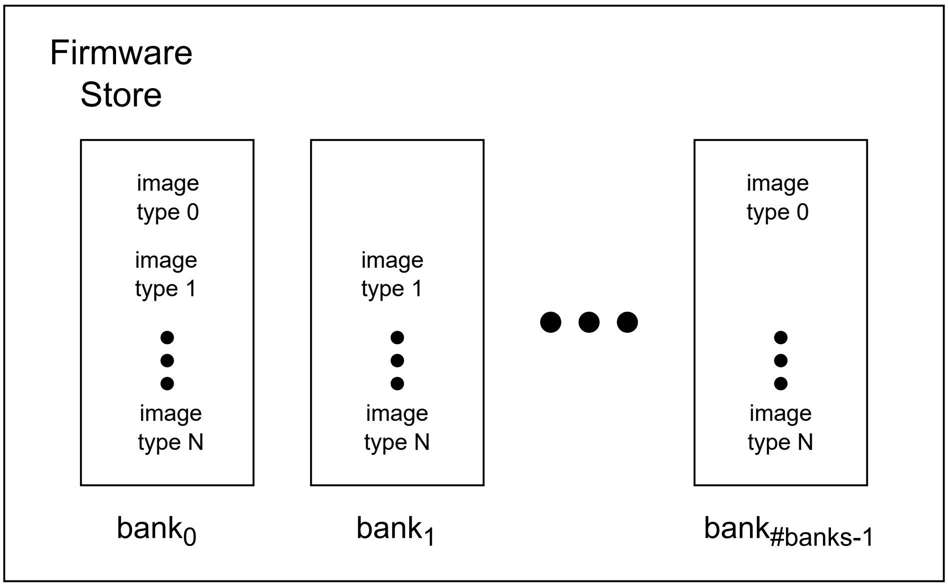

Figure 3.1: Firmware Store

The Firmware Store may have an IMPLEMENTATION DEFINED number of firmware banks (*#banks*), each bank contains an IMPLEMENTATION DEFINED number of firmware images (#images). Although the types of firmware images, on each bank, are allowed to differ between banks, it is expected that most banks contain the same image types.

At any point in time there is an active bank, denoted as $bank_{active\_index}$. The platform is expected to always boot using the firmware in the $bank_{active\_index}$. A situation where the platform boots with a firmware bank different than the active bank constitutes a system failure scenario. The situation can only happen if the firmware in the active bank is inoperative.

The Update Agent can implement a scheme where the active bank is kept intact while an update bank is being overwritten. This scheme is commonly referred to as A/B updates. Any details around the number of banks and the policy that the Update Agent uses to maintain the firmware banks are IMPLEMENTATION DEFINED.

3.2 Firmware Store update protocol GUIDs

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

The *update_agent_guid* 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 3.4.1.1 *Dynamic shared buffer setup phase*.

The *fwu_directory_guid* 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 3.3.1 *Image directory*.

The *metadata_guid* is the identifier of the metadata partition type when the metadata is stored within a GPT [2], see A3.2.4.1 *Metadata integration with GPT* for more information.

Table 3.1: protocol defined GUIDs

GUID	GUID name	description
6823a838-1b06-470e-9774-0cce8bfb53fd	*update_agent_guid*	Update Agent context GUID
deee58d9-5147-4ad3-a290-77666e2341a5	*fwu_directory_guid*	The image directory GUID, see 3.3.1 *Image directory*
8a7a84a0-8387-40f6-ab41-a8b9a5a60d23	*metadata_guid*	The GUID of the metadata type, see A3.2.1 *Firmware update metadata*

3.2.1 Firmware Store state machine

The Firmware Store can be in one of the following states:

- Regular - all the images in the active bank have been accepted.
- Staging - the procedure to update images in the Firmware Store is undergoing.
- Trial - the Firmware Store has been updated, at least one of the firmware images in the active bank has not been accepted yet.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

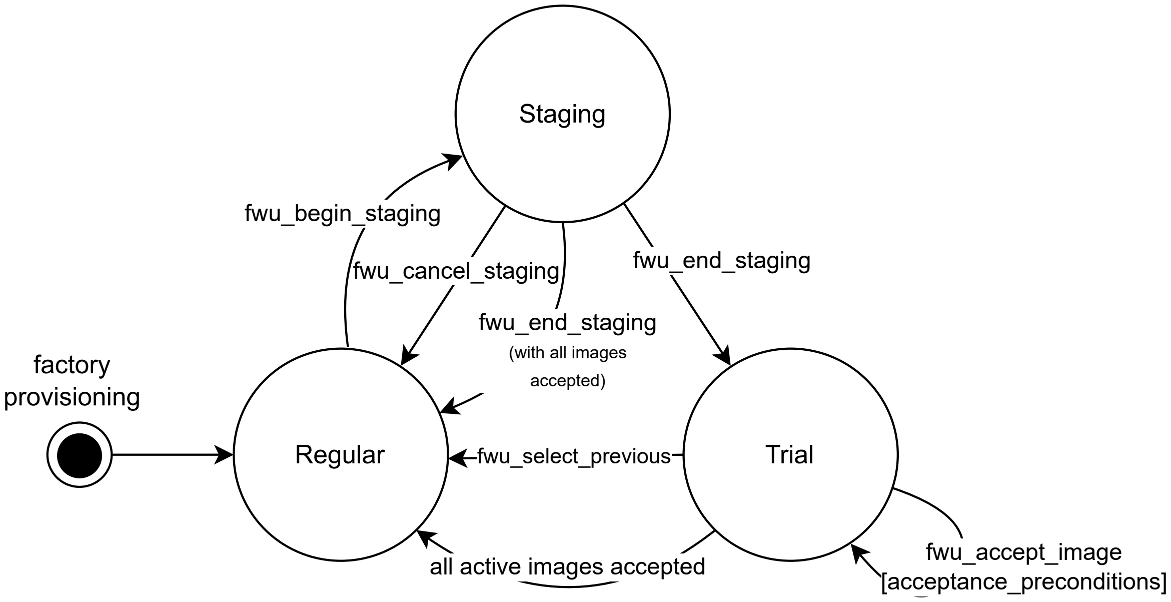

Figure 3.2: Firmware Store FSM

The diagram in Figure 3.2 depicts the state machine of a generic Firmware Store. The Firmware Store transitions between states (that are visible from the Client) as a direct result of the Client invoking primitives from the Firmware Store update ABI.

The Update Agent may impose that an image is activated prior to being accepted by a Client (see acceptance_preconditions guard in Figure 3.2). This is a per-image platform-specific policy, discoverable from the client_permission field in Table 3.3, in the Image Directory.

The state transitions occur at the following boundaries:

- Regular to Staging:
  - when a fwu_begin_staging call returns successfully (see 3.4.2 *ABI definition*).
- Staging to Trial:
  - when the call fwu_end_staging returns successfully.
- Trial to Regular:
  - once all firmware images in the active bank are accepted.

A detailed description of the Staging state is provided in 3.2.1.1 *Staging state*, the Trial state is covered in 3.2.1.2 *Trial state*.

3.2.1.1 Staging state

New firmware 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 firmware image context is open, a sequence of fwu_write_stream calls transmit the firmware image to the Update Agent. The sequence diagram of the FW image transfer is shown in Figure 3.3.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

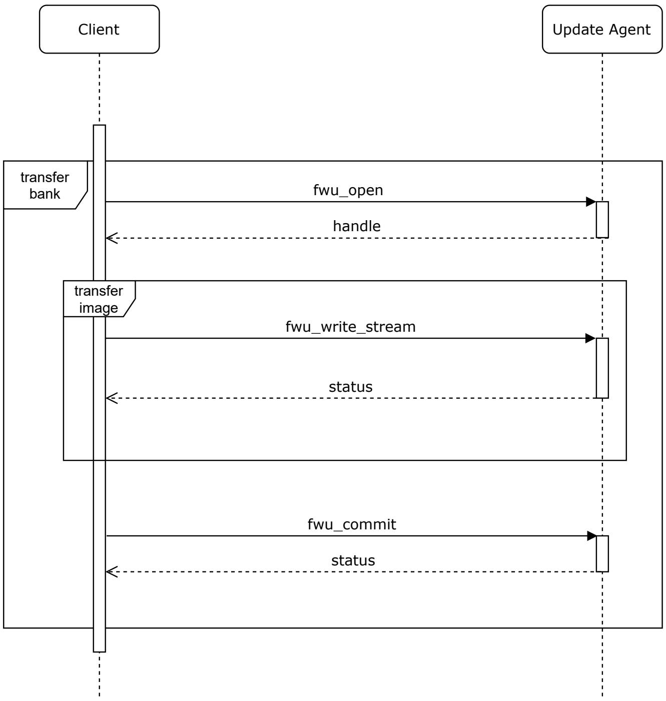

Figure 3.3: Staging procedure

The Update Agent can authenticate the staged firmware images before committing those to the Firmware Store. This optional procedure is performed at the fwu_commit call. The image authentication procedure is detailed in 3.3.3.1 *Image authentication*.

The image authentication may fail, this is communicated to the Client by the fwu_commit call returning a AUTH_FAIL status code, see 3.4.2.8 *fwu_commit* for further details.

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.

3.2.1.2 Trial state

The system is in the Trial state if any of the firmware images in the active bank are pending acceptance, see 3.3.3 *Protocol-updatable images*. A Client can instruct the images to be accepted during the fwu_commit call. Hence, from a Client point of view, the Trial state is optional.

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

Anti-rollback counter values must be updated once the Firmware Store transitions to the Regular state.

A platform design may allow public keys to be enrolled in-band using the update ABI defined in 3.4.2 *ABI definition*, or via a separate out-of-band mechanism (e.g. a BMC). The details of key enrollment are currently out of scope of this specification and are considered IMPLEMENTATION DEFINED. A key installation procedure is performed with the intent of permanently replacing a previous key. A newly installed public key must be used during a boot in the Trial state to authenticate the firmware images signed with its private pair. Any previous public key that has been superseded cannot be discarded until the Trial state terminates correctly. In the advent of a Trial state failure, the previous public key must be reinstated.

While booting in the Trial state, the trusted boot procedure must check that a firmware image meets the version requirements of a subsequent Regular state boot. If the firmware 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.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

3.3 Firmware Store management

3.3.1 Image directory

The Client discovers details on the firmware images, handled by the Update Agent, via the image directory. The Client reads the image directory, by opening the file with GUID *fwu_directory_guid* and reading from it, using the ABI defined in 3.4.2 *ABI definition*.

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 firmware bank that booted the platfrom.

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 information held entirely by the Update Agent.

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

Table 3.2: image_directory version 2

field	offset (bytes)	size (bytes)	Description
directory_version	0h	4h	the version of the fields in the img_info_entry array. Must be set to 2 for the data structure definined in this document.
img_info_offset	4h	4h	the offset of the img_info_entry array relative to the start of this data structure.
num_images	8h	4h	the number of entries in the img_info_entry array.
correct_boot	Ch	4h	boolean stating if the platform booted with the active bank.
img_info_size	10h	4h	the size, in bytes, of an entry in the img_info_entry array.
reserved	14h	4h	
img_info_entry[ ]	18h	–	array of Table 3.3 elements

The directory_version field determines the version of the img_info_entry.

Table 3.3: img_info_entry version 2

field	offset (bytes)	size (bytes)	Description
img_type_guid	0h	10h	GUID identifying the image type

field	offset (bytes)	size (bytes)	Description
client_permissions	10h	4h	bitfield specifying the access permissions that the Client has on the image: • [31:3] : MBZ • [2] : acceptance preconditions – 0 : the image can be accepted at any point in the Trial state. – 1 : the image must be activated before it can be accepted. • [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 (dictated by an anti-rollback counter, details are IMPLEMENTATION DEFINED).
img_version	1Ch	4h	the image version in the $bank_{boot\_index}$.
accepted	20h	4h	the acceptance status of the image in the $bank_{boot\_index}$.
reserved	24h	4h	MBZ

3.3.2 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 that 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.

3.3.3 Protocol-updatable images

The protocol-updatable firmware images are transferred from the Client to the Update Agent using the ABI defined in 3.4.2 *ABI definition*. The firmware image format is IMPLEMENTATION DEFINED.

A firmware image, in a firmware bank, can have either an accepted or unaccepted status.

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 3.4.2 *ABI definition*.
- fwu_image_accept: the Client changes the accepted status of an image in the $bank_{active\_index}$ to be accepted, see 3.4.2 *ABI definition*.

Some firmware images require activation before the Client is allowed to accept them. A Client can discover if this is the case from bit 2 of the client_permission field on the Image Directory.

3.3.3.1 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 Firmware Store, 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 firmware image is bound to a specific anti-rollback counter. The image to anti-rollback counter binding is IMPLEMENTATION DEFINED.

To be allowed execution in the platform, any firmware image must have a version that is greater or equal than its associated anti-rollback counter.

The image authentication procedure is composed of the following checks:

- firmware image creator authenticity check, the procedure is IMPLEMENTATION DEFINED.
- verification that the firmware image version is greater than the 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.

3.3.4 Telemetry

The Update Agent keeps track of the telemetry related to Firmware Store updates. The telemetry values are reset after a successful call to fwu_begin_staging.

The quantities tracked in the telemetry are:

- authentication work: the units of authentication work performed since the last successful call to fwu_begin_staging.
- image copy work: the units of image copying work performed since the last successful call to fwu_begin_staging.
- erase work: the units of erase work performed, on the non-volatile memory where the Firmware Store is located, since the last successful call to fwu_begin_staging.
- write work: the units of write work performed, on the non-volatile memory where the Firmware Store is located, since the last successful call to fwu_begin_staging.

The progress on each unit can be computed as the ratio between the units of work and respective total field. For instance, the authentication progress (*auth_progress*) can be computed as:

$auth\_progress = \frac{auth\_work}{total\_auth}$

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

3.3.4.1 Telemetry shared buffer

The Update Agent shares the telemetry with the Update Client via a shared memory buffer. The telemetry is laid out in the buffer following the layout specified in Table 3.4.

The usage of a shared buffer, for the telemetry exchange, allows for concurrent telemetry production and consumption on separate PEs. The telemetry provided is a crude progress estimate. Different telemetry fields can hold progress snapshots obtained at slightly different time instants.

The memory attributes and physical address of the memory shared buffer are IMPLEMENTATION DEFINED and must be agreed upon between the Update Agent and the Update Client. The memory attributes and the physical address of the shared buffer can be communicated to an OS via the HAFW ACPI table (see Chapter B1 *High Availability Firmware ACPI table*).

The base address of the telemetry buffer must be 64KiB aligned. To prevent erroneous telemetry being read, read and write accesses to the fields in the telemetry buffer should use the load and store instruction with the same size as the field.

Table 3.4: Firmware Store telemetry version 1

Field	offset (bytes)	size (bytes)	Description
telemetry_version	0h	4h	The version of the telemetry data structure.
auth_work	4h	2h	The authentication work units performed since the last fwu_begin_staging call.
total_auth	6h	2h	The total authentication work units to be performed.
img_copy_work	8h	2h	The blob copying work units performed since the last fwu_begin_staging call.
total_img_copy	Ah	2h	The total image copying work units to be performed.
erase_work	Ch	2h	The erase work units performed since the last fwu_begin_staging call.
total_erase	Eh	2h	The total erase work units to be performed.
write_work	10h	2h	The write work units performed since the last fwu_begin_staging call.
total_write	12h	2h	The total write work units to be performed.
state	14h	1h	The state of the Firmware Store. Can be one of: • 0: “Regular” • 1: “Staging” • 2: “Trial”

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

3.4 Firmware Store Update ABI

3.4.1 Transport layer

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

Prior to calling any Firmware Store update function, the Client must have a shared buffer with the Update Agent to carry the Firmware Store update ABI messages.

This buffer can be statically provisioned by means of a carve-out, agreed by the Client and Update Agent in an IMPLEMENTATION DEFINED manner.

Alternatively the Client can set a shared buffer dynamically by following the procedure listed in 3.4.1.1 *Dynamic shared buffer setup phase*.

3.4.1.1 Dynamic shared buffer 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_guid* 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.

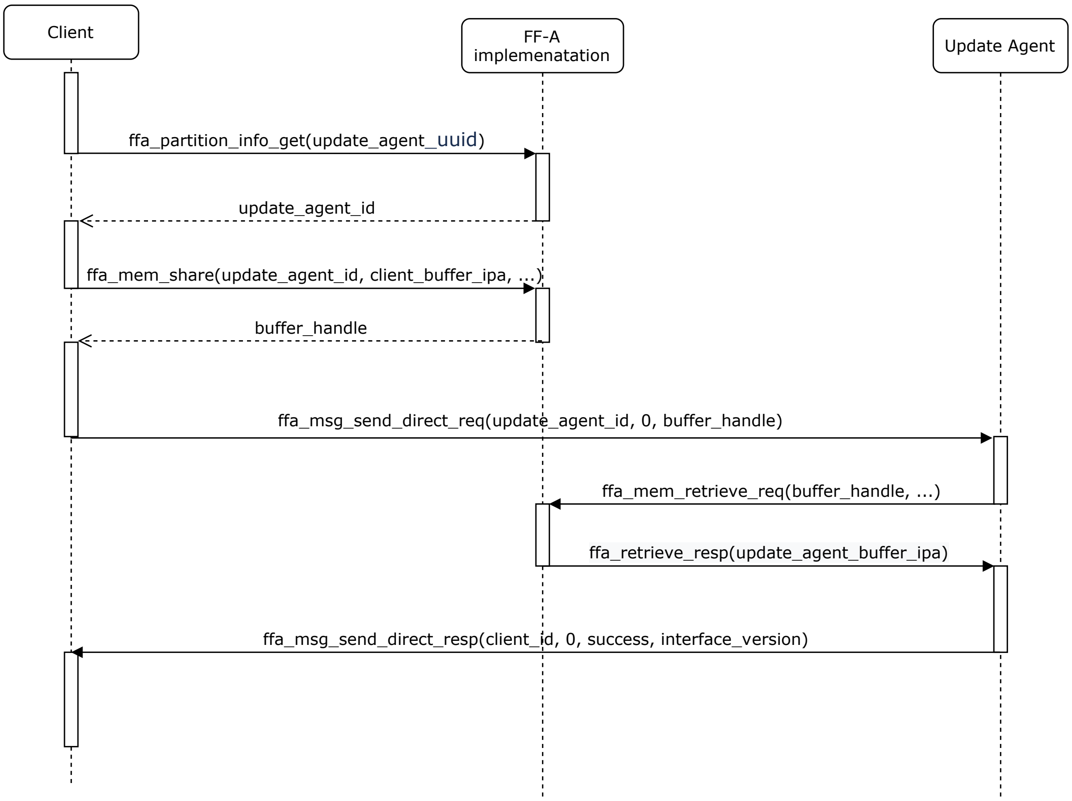
Client
FF-A implemenatation
Update Agent
ffa_partition_info_get(update_agent_uuid)
update_agent_id
ffa_mem_share(update_agent_id, client_buffer_ipa, ...)
buffer_handle
ffa_msg_send_direct_req(update_agent_id, 0, buffer_handle)
ffa_mem_retrieve_req(buffer_handle, ...)
ffa_retrieve_resp(update_agent_buffer_ipa)
ffa_msg_send_direct_resp(client_id, 0, success, interface_version)

Figure 3.4: Transport layer setup

After a successful completion of the setup phase, the Firmware Store update ABI calls can be issued. In case of

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

failure in the setup phase, the Client must assume the Firmware Store update protocol to be unavailable.

3.4.2 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 an 8 byte boundary.

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

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

3.4.2.1 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. Every entry in function_presence[] contains a 2-byte integer that denotes the function id.

If the Update Agent implements a function, identified by a given function_id, that function_id must be an entry of the function_presence array. The entries of the function_presence array must be unique and defined in ascending function_id order.

3.4.2.1.1 Arguments

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

3.4.2.1.2 Returns

field	offset (bytes)	size (bytes)	description
status	0	4	• SUCCESS
service_status	4	2	the status of the service provider. • 0: operative – the service provider is fully operative, the remainder fields are applicable. • -1: init_error – the service provider failed to initialize for an unspecified reason. • all other values are reserved.
version_major	6	1	the ABI major version, set to 1 for the ABI definition on this document.
version_minor	7	1	the ABI minor version, set to 0 for the ABI definition on this document.
off_function_presence	8	2	the offset (in bytes) of the function_presence array relative to the start of this data structure.
num_func	10	2	the number of entries in the function_presence array.
max_payload_size	12	8	the maximum number of bytes that a payload can contain.
flags	20	4	flags listing the update capabilities. • flags[0] – 1 : partial update supported – 0 : partial update not supported • flags[31:1] : Reserved, must be zero.
vendor_specific_flags	24	4	Vendor specific update capabilities flags.
function_presence[]	28	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.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

3.4.2.2 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.

It is IMPLEMENTATION DEFINED if this call is disallowed when the platform did not boot with the active bank.

3.4.2.2.1 Arguments

field	offset (bytes)	size (bytes)	description
function_id=16	0	4	
reserved	4	4	Reserved, must be zero.
vendor_flags	8	4	Vendor specific staging flags.
partial_update_count	12	4	The number of elements in the update_guid array. If this field is greater than 0, then update_guid[] contains an indication of the images the Client intends to update. The Client is not mandated to provide this indication. If this field is 0 both partial and full updates are allowed.
update_guid[]	16	partial_update_count.16	An array of image type GUIDs that the Update Client will update during the Staging state.

3.4.2.2.2 Returns

field	offset (bytes)	size (bytes)	description
status	0	4	• FWU_SUCCESS • FWU_DENIED: The Firmware Store is in the Trial state or the platform did not boot correctly (and the platform only allows updates when executing from the active bank). • FWU_BUSY: The Client is temporarily prevented from entering the Staging state. • FWU_UNKNOWN: One of more GUIDs in the update_guid field are unknown to the Update Agent.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

3.4.2.3 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.

On successful completion of this call, the Update Agent must update the active bank to the firmware bank that was just updated. The details of Firmware Store bank management are IMPLEMENTATION DEFINED.

3.4.2.3.1 Arguments

field	offset (bytes)	size (bytes)	description
function_id=17	0	4	

3.4.2.3.2 Returns

field	offset (bytes)	size (bytes)	description
status	0	4	• FWU_SUCCESS • FWU_DENIED: The system is not in a Staging state. • FWU_BUSY: There are open image handles. • FWU_AUTH_FAIL: At least one of the updated images fails to authenticate. • FWU_NOT_AVAILABLE: the Update Agent does not support partial updates, and the Client has not updated all the images.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

3.4.2.4 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.

3.4.2.4.1 Arguments

field	offset (bytes)	size (bytes)	description
function_id=18	0	4	

3.4.2.4.2 Returns

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

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

3.4.2.5 fwu_open

The open call returns a handle to the image with GUID=image_guid. 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 GUID, only the last returned handle is valid.

3.4.2.5.1 Arguments

field	offset (bytes)	size (bytes)	description
function_id=19	0	4	
image_type_guid	4	16	GUID of the image to be opened
op_type	20	1	The operation that the Client will perform on the image. This field takes the following values: • 0: open the stream for reading, • 1: open the stream for writing, • all other calues reserved.

3.4.2.5.2 Returns

field	offset (bytes)	size (bytes)	description
status	0	4	• FWU_SUCCESS: Call completed correctly. Remaining return fields are valid • FWU_UNKNOWN: image type with GUID=image_type_guid does not exist • FWU_DENIED: An image cannot be openned for write outside of ths Staging state • FWU_NOT_AVAILABLE: the Update Agent does not support the op_type for this image.
handle	4	4	staging context identifier

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

3.4.2.6 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.

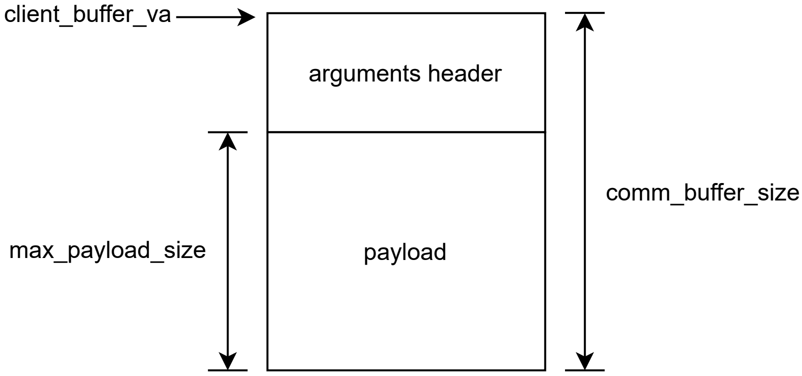

Figure 3.5: fwu_write_stream arguments in shared buffer

3.4.2.6.1 Arguments

field	offset (bytes)	size (bytes)	description
function_id=20	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

3.4.2.6.2 Returns

field	offset (bytes)	size (bytes)	description
status	0	4	• FWU_SUCCESS • FWU_UNKNOWN: unrecognized handle • FWU_OUT_OF_BOUNDS: less than data_len bytes available in the image. • FWU_NO_PERMISSION: The image cannot be written to. • FWU_DENIED: The system is not in a Staging state

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

3.4.2.7 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: $0 \leqslant \mathit{read\_bytes} < \mathit{max\_payload\_size}$, where $\mathit{max\_payload\_size}$ = $\mathit{comm\_buffer\_size}$ - offset_of(read_stream_return, payload).

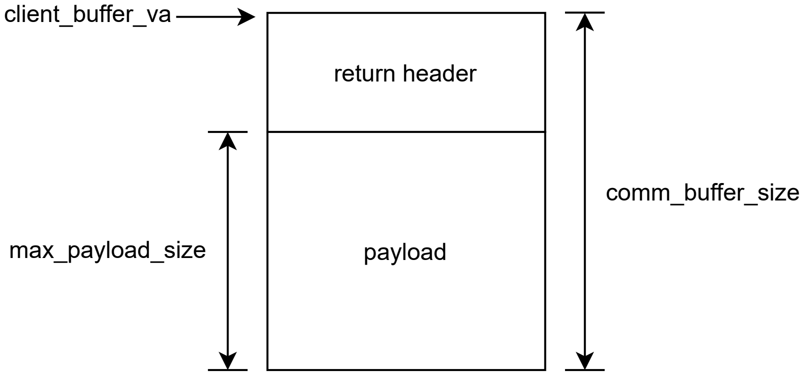

Figure 3.6: fwu_read_stream returns in shared buffer

3.4.2.7.1 Arguments

field	offset (bytes)	size (bytes)	description
function_id=21	0	4	
handle	4	4	The handle of the context being read from.

3.4.2.7.2 Returns

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

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

3.4.2.8 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 acceptance status to:

- "not accepted": if acceptance_req > 0;
- "accepted": if acceptance_req = 0;

The Client passes the *max_atomic_len* hint argument, specifying the length of time (nanoseconds) that the Client can withstand 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* nanoseconds 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 also 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. For any subsequent fwu_commit call following a RESUME return status, the acceptance_req argument is ignored by the Update Agent.

At the end of a successful fwu_commit call, the Update Agent guarantees that the firmware image has been persisted to the Firmware Store on the current firmware bank being updated.

Note: An Update Agent implementation is allowed to persist parts of the blob during the fwu_write_stream call – this is IMPLEMENTATION DEFINED.

If the image was open for writing and the Client has not written any data to it, the Update Agent must erase the current contents of the image.

3.4.2.8.1 Arguments

field	offset (bytes)	size (bytes)	description
function_id=22	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.

3.4.2.8.2 Returns

field	offset (bytes)	size (bytes)	description
status	0h	4	• FWU_SUCCESS • FWU_UNKNOWN: unrecognized handle. • FWU_AUTH_FAIL: image closed, authentication failed. • FWU_RESUME: the Update Agent yielded, the Client must invoke the call again. • FWU_DENIED: the image can only be accepted after activation, acceptance_req must be > 0.
progress	4h	4	• Units of work already completed by the Update Agent.
total_work	8h	4	• Units of work the Update Agent must perform until fwu_commit returns successfully.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

3.4.2.9 fwu_accept_image

The call sets the status of the firmware image, with type = image_type_guid, to “accepted” in the active firmware bank. This call can only be invoked if the system booted correctly (with $bank_{active\_index}$).

3.4.2.9.1 Arguments

field	offset (bytes)	size (bytes)	description
function_id=23	0	4	
reserved	4	4	Reserved, must be zero.
image_type_guid	8	10h	

3.4.2.9.2 Returns

field	offset (bytes)	size (bytes)	description
status	0	4	• FWU_SUCCESS • FWU_UNKNOWN: image with type=image_type_guid is not managed by the Update Agent. • FWU_DENIED: the system has not booted with the active bank, or the image cannot be accepted before being activated.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

3.4.2.10 fwu_select_previous

The call rolls back the firmware to the previous active firmware. The Update Agent returns DENIED if the previous active firmware cannot boot the platform (e.g. the previous active bank does not contain valid firmware, or the firmware has a version that is lower than the current anti-rollback counters).

A platform can opt to only allow this call when:

- the Firmware Store is in the Trial state, **or**
- the platform failed to boot with the active bank.

3.4.2.10.1 Arguments

field	offset (bytes)	size (bytes)	description
function_id=24	0	4	

3.4.2.10.2 Returns

field	offset (bytes)	size (bytes)	description
status	0	4	• FWU_SUCCESS • FWU_DENIED: the previous active bank cannot boot the platform, or the Firmware Store is not in the Trial state, or the platform booted with the active bank.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

3.4.2.11 Return status

status	value
FWU_SUCCESS	0
FWU_UNKNOWN	-1
FWU_BUSY	-2
FWU_OUT_OF_BOUNDS	-3
FWU_AUTH_FAIL	-4
FWU_NO_PERMISSION	-5
FWU_DENIED	-6
FWU_RESUME	-7
FWU_NOT_AVAILABLE	-8

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

Part A

A/B Firmware Store design guidance

This appendix provides guidance on how a platform can deploy an A/B Firmware Store design. This appendix provides guidance and as such the aspects described in this Appendix are **not mandatory** for compliance with the specification.

The guidance specifies:

- how an A/B Firmware Store can be implemented, with a GPT layout and a metadata data structure,
- how the platform boot process determines the firmware bank to use,
- guidance on the Update Agent implementation to perform bank management during Firmware Store updates.

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 firmware 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 3.1 *Firmware Store design* for more information.

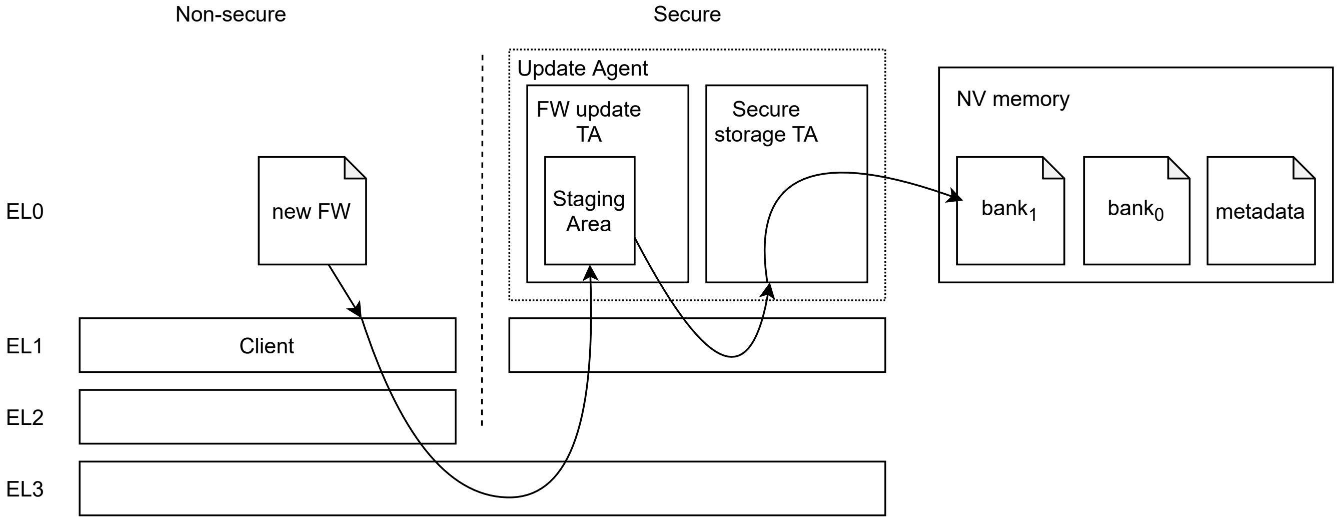

Figure 1: System diagram

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

Chapter A1
Firmware bank selection

The immutable or secondary stages select the firmware bank, containing the *protocol-updatable* images, to boot the platform with. The immutable or *secondary* stages must be able to read and interpret the image metadata (see Table A3.6).

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

A1.1 Platform Boot

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 boot attempt by, for instance, inspecting the watchdog state. The mechanism to determine a failed 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 platform 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*, AND $bank_{previous_active_index}$ is valid or accepted (see Table A3.6), 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 A3.2.1 *Firmware update metadata*.

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* and the metadata version must be propagated to the Update Agent. The mechanism to propagate these values to the Update Agent is IMPLEMENTATION DEFINED.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

Chapter A2
Recovery Mode

The platform recovery mode must be able to correctly update the firmware update metadata (See A3.2.1 *Firmware update metadata*).

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

Chapter A3
A/B Firmware Store design

On A/B Firmware Stores, there exits an *active* and an *update* bank.

The *active_index* is maintained by the Update Agent in the metadata, see A3.2.1 *Firmware update metadata*.

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 platform when an updated bank fails to properly boot.

All bank indices take values in the $\{0, \ldots, \#banks\text{-}1\}$ range.

The initialization of the firmware 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 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 A1.1 *Platform Boot*.

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

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

A3.1 A/B Firmware Store state machine

Like in the general design, at any given time the Firmware Store can be in one of the following states:

- Regular
- Staging
- Trial

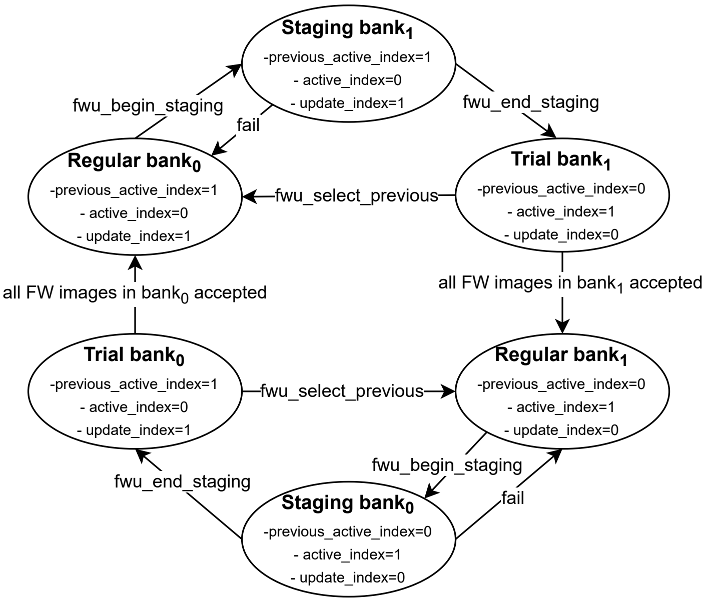

Figure A3.1: High level A/B Firmware Store FSM

The diagram in Figure A3.1 depicts the state machine of a particular Firmware Store implementation. In this example the Firmware Store has 2 different firmware banks ($bank_0$ and $bank_1$). For more information on the firmware banks see 3.1 *Firmware Store design*.

The following state variables are defined for the A/B Firmware Store:

1. *active_index*: integer indicating which firmware bank is currently active. The variable is kept in the metadata region, see Table A3.6. Its value is updated, by the Update Agent, during the handling of a fwu_end_staging call (see 3.4.2 *ABI definition*).
2. *previous_active_index*: integer indicating which firmware bank was active prior to the last successful update to the Firmware Store. The variable is kept in the metadata region, see Table A3.6. Its value is updated, by the Update Agent, during the handling of a fwu_end_staging call (see 3.4.2 *ABI definition*).
3. *update_index*: integer indicating which firmware 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 constraint:

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

4. image accepted status: Field recorded per-image and per-bank (see Table A3.8). The accepted status of all images in the $bank_{active\_index}$ determine if the Firmware Store is in the Trial state, see 3.2.1.2 *Trial state*.

A3.1.1 A/B Firmware Store Staging state

The firmware image authentication procedure, before committing images to the Firmware Store, is:

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

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

The Update Agent overwrites the images in $bank_{update\_index}$. The Update Agent must set bank_state[update_index] ← 0xFF (invalid) before any content in the $bank_{update\_index}$ is overwritten. This ensures that a bootloader, following the guidance in A1.1 *Platform Boot*, would not attempt to boot the platform with a potentially corrupted bank.

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

- update *previous_active_index*, see A4.1 *fwu_end_staging*.
- update *active_index*, see A4.1 *fwu_end_staging*.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

A3.2 Firmware Store management

A3.2.1 Firmware update metadata

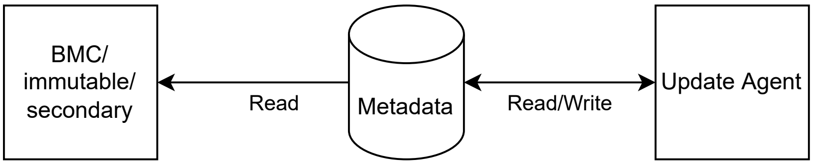

Figure A3.2: Metadata usage for Update Agent and early bootloader

The metadata is a collection of fields, maintained by the Update Agent, as defined in Table A3.6. The metadata serves primarily as an information exchange channel between the Update Agent and the early stage bootloader.

As seen in Figure A3.2, the metadata is read by the early stage bootloader (BMC, immutable or secondary). The information in the metadata instructs the early stage bootloader about the bank to boot the platform with (*active_index*). The Update Agent in turn maintains the metadata. The Update Agent stores in the metadata the bank indices (active_index and previous_active_index), it also restores the metadata (in the unlikely event it gets corrupted).

The Update Agent records the state of the different banks in the bank_state array. The entry *n* of the bank_state array relates to $bank_n$. The bank states are the following:

- invalid: one or more images are corrupted or empty.
- valid: all images in the bank are intact, but one or more are unaccepted.
- accepted: all images in the bank are accepted and intact.

Note

If the metadata contains a Firmware Store description (fw_store_desc, see Table A3.3), for a bank to be in the accepted state then all firmware images in that bank must be marked as accepted. This means that all of the img_bank_info data structures corresponding to that bank must have the field *accepted* set to 1.

The early stage bootloader should never attempt to load a bank in invalid state.

If the active bank has a *valid* state, then the Firmware Store is in the Trial state. If the active bank is in the *accepted* state, it follows that the Firmware Store must be in the Regular state.

For any banks, other than the active bank, the *accepted* and *valid* bank states are equivalent from the early stage bootloader point of view.

The non-volatile 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 A3.6

The metadata is versioned using a 4 byte integer – *version* field in Table A3.6.

The metadata size is determined by the metadata_size field.

Note

On version 1 of the metadata, the values *#images* and *#banks* were assumed to be platform-specific constants and were known by the entities that interact with the metadata. Also, version 1 did not include a metadata_size field. The metadata size was derived, independently by any entity that interacts with the metadata, via the linear function defined in Table A3.1.

On version 2 of the metadata, the metadata size is specified in the metadata_field, maintained by the Update Agent. Additionally, version 2 of the metadata restrict the maximum number of firmware banks to 4.

The metadata size, as a function of metadata version, is shown in Table A3.1.

Table A3.1: Metadata size per version

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

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]).

The metadata representation is replicated to ensure reliable operation.

Metadata management at serialization by the Update Agent: When the Update Agent introduces changes to the Metadata, it must update the replicas in sequence. The representation of the two metadata replicas must be kept in a disjoint set of non-volatile memory blocks.

Metadata management at early system boot: the *immutable* or *secondary* stage must use an intact metadata. The metadata replicas are inspected, using the procedure described in A3.2.4 *Metadata integrity check*, to ensure that they are intact.

Metadata management at Update Agent initialization: During its initialization, the Update Agent must check both metadata replicas for corruption (see A3.2.4 *Metadata integrity check*). If one of the metadata replicas is found to be corrupted, the Update Agent overwrites the corrupted metadata with the intact replica.

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.

A3.2.2 Metadata Version 2

Table A3.2: Metadata version 2

field	offset (bytes)	size (bytes)	Description
crc_32	0h	4h	
version	4h	4h	The version of the metadata structure. Must be 2 for the data structure defined in this document.
active_index	8h	4h	

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

field	offset (bytes)	size (bytes)	Description
previous_active_index	Ch	4h	
metadata_size	10h	4h	The size in bytes of the complete metadata structure.
descriptor_offset	14h	2h	The offset, from the start of this data structure, where the fw_store_desc starts (if one exists). If the fw_store_desc does not exist, then this field must be set to 0.
reserved	16h	2h	Reserved, must be zero.
bank_state[4]	18h	4h	The state of each bank. Each entry is an 8-bit value, the entry index is the index of the bank it relates to. Each bank_state entry can take one of the following values: • 0xFF: invalid – One or more images in the bank are corrupted or were partially overwritten. • 0xFE: valid – The bank contains a valid set of images, but some images are in an unaccepted state. • 0xFC: accepted – all of the images in the bank are valid and have been accepted. Note: a platform may have *#banks* < 4. The Update Agent must ensure bank_state[idx]=0xFF, for any idx such that *#banks* $\leqslant idx < 4$.
reserved	1Ch	4h	Reserved, must be zero.
fw_store_desc (Table A3.3)	20h	–	The data structure described in Table A3.3. This data structure is optional and it is only present if descriptor_offset > 0.

Note

A platform may choose not to carry a fw_store_desc (Table A3.3) data structure in the metadata. This is indicated by *descriptor_offset* > 0.

Table A3.3: Firmware Store description 2

field	offset (bytes)	size (bytes)	Description
num_banks	0h	1h	The number of firmware banks in the Firmware Store. Must not exceed 4 for this version of the data structure.
reserved	1h	1h	Reserved, must be zero.
num_images	2h	2h	The number of entries in the img_entry array.
img_entry_size	4h	2h	The size in bytes of the Table A3.7 data structure. Must be set to (20h + num_banks.18h) for the data structure defined in this document.
bank_info_entry_size	6h	2h	The size in bytes of the Table A3.8 data structure. Must be set to 18h for the data structure defined in this document.
img_entry [num_images]	8h	num_images.img_entry_size	array of aggregate in Table A3.7

Table A3.4: Metadata image entry version 2 (img_entry)

field	offset (bytes)	size (bytes)	Description
img_type_guid	0h	10h	GUID identifying the image type
location_guid	10h	10h	the GUID of the storage volume where the image is located
img_bank_info[]	20h	bank_info_entry_size.num_banks	the properties of images with img_type_guid in the different FW banks. Type described in Table A3.5

Table A3.5: Image properties in a given FW bank version 2 (img_bank_info)

field	offset (bytes)	size (bytes)	Description
img_guid	0h	10h	the guid 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 A3.6. The metadata contains an array of image entries (defined in Table A3.7) with *#images* elements.

A3.2.3 Metadata version 1

Table A3.6: Metadata version 1

field	offset (bytes)	size (bytes)	Description
crc_32	0h	4h	
version	4h	4h	
active_index	8h	4h	
previous_active_index	Ch	4h	
img_entry [#images]	10h	#images.(20h + #banks.18h)	array of aggregate in Table A3.7

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

Table A3.7: Metadata image entry version 1 (img_entry)

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. Type described in Table A3.8

Table A3.8: Image properties in a given FW bank version 1 (img_bank_info)

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)

A3.2.4 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:
        metadata_size <- metadata.metadata_size

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

    if crc != metadata.crc_32:
        return False

    return True
```

A3.2.4.1 Metadata integration with GPT

It is recommended that the layout of the Firmware Store is defined by a GPT [2].

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

The platform may possess different Firmware Stores (where firmware images are kept at rest). All firmware images of the same type should be located in the same Firmware Store. The location_guid of each image type should match the DiskGUID [2] of the medium the image is located on.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

Chapter A4
State variable updates by the Update Agent

For an A/B Firmware Store using a metadata, the Update Agent should perform bank management following the design specified in the current Appendix. The following sections specify how the Update Agent should change the variable of some state variables while handling specific ABI calls.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

A4.1 fwu_end_staging

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*.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

A4.2 fwu_accept_image

The call sets the firmware image acceptance status in the metadata (img_bank_info[active_index].accepted <- 1), for the image with type = image_type_guid.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

A4.3 fwu_select_previous

The Update Agent performs the following actions while handling this call:

- the *active_index* is updated: *active_index* ← *previous_active_index*.
- the *previous_active_index* is updated: IMPLEMENTATION DEFINED assignment.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

Part B
HAFW ACPI table

Chapter B1
High Availability Firmware ACPI table

The high availability firmware (HAFW) table describes aspects of the firmware update implementation to the OS.

The table describes the interrupt that is delivered to the OS once the Firmware Store has been updated. The OS must be notified when the Firmware Store has been updated, whether for in-band or out-of-band firmware update flows. To that end, the table lists the interrupt id, in the *gsiv* field, denoting the SPI interrupt ID that is generated when the Firmware Store has been updated. The entity that induces the interrupt is IMPLEMENTATION DEFINED, and must have direct visibility to the Firmware Store.

Lastly, the table allows the discovery of the Firmware Store update telemetry buffer. The table lists the base address of the buffer, as well as the memory attributes used by the Update Agent.

Table B1.1: High Availability Firmware Update ACPI table.

Field	Offset (bytes)	Size (bytes)	Description
signature	0h	4h	See ACPI System Description Table Header [4].
length	4h	4h	See ACPI System Description Table Header [4].
revision	8h	1h	The revision of this table, set to 1 for the table described in this document.
checksum	9h	1h	See ACPI System Description Table Header [4].
oem_id	Ah	6h	See ACPI System Description Table Header [4].
oem_table_id	10h	8h	See ACPI System Description Table Header [4].
oem_revision	18h	4h	See ACPI System Description Table Header [4].

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

	Offset	Size	
Field	(bytes)	(bytes)	Description
creator_id	1Ch	4h	See ACPI System Description Table Header [4].
creator_revision	20h	4h	See ACPI System Description Table Header [4].
gsiv	24h	4h	The interrupt id that is used to notify the OS when the Firmware Store has been updated.
mem_attr	28h	2h	The memory attributes of the telemetry buffer mappings. The Update Agent maps the buffer with the attributes listed in this field. The field values are: • mem_attr[7:0]: MAIR, this field holds the value of the MAIR_ELx used by the Update and Activation Agent to map the telemetry buffer, see [5] for the MAIR_ELx definition. Note that the MAIR defined the memory type. Hence: • mem_attr[3:2]: memory type – 0 – device memory – 1 – normal memory If device memory • mem_attr[15:8]: reserved, must be zero If normal memory: • mem_attr[9:8]: shareability – 0 – Non-shareable – 2 – Outer Shareable – 3 – Inner Shareable – all other values are reserved • mem_attr[15:10]: reserved, must be zero.
reserved	2Ah	6h	Reserved, must be zero.
telemetry_address	30h	8h	The base address of the telemetry buffer for the updates to the Firmware Store.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

Part C
In-band updates on systems with a Platform Controller

In some systems the firmware store is managed by a Platform Management Controller (PCtr). Examples of a PCtr are the BMC or the eRoT.

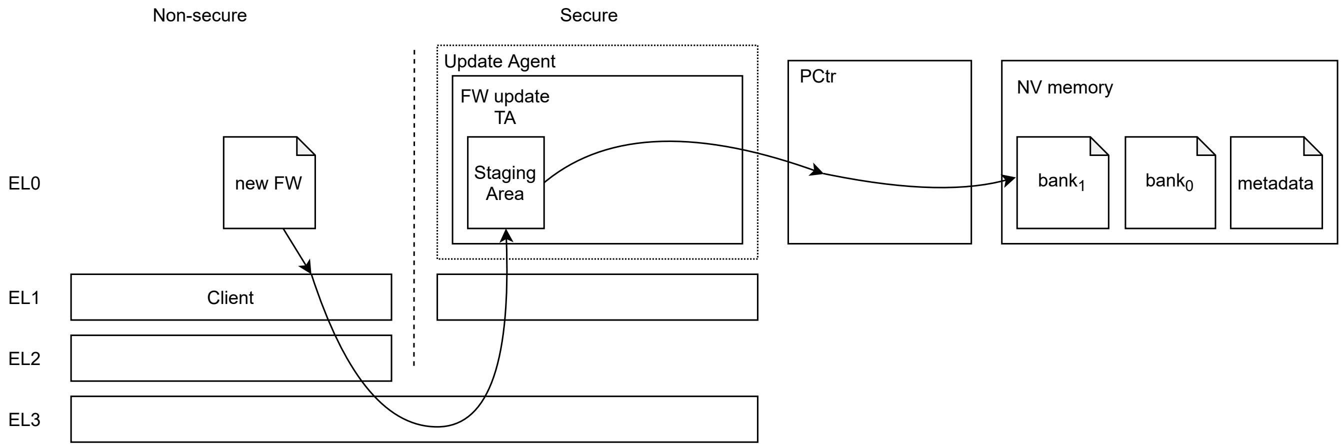

Figure 1: System diagram with PCtr

The PCtr can restrict the Application Processor (AP) processor from accessing the firmware Store.

The AP may thus be unable to directly update the firmware images in the Firmware Store.

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

	B1	B2
AP has direct R/W access to Firmware Store	Y	N
AP accesses the Firmware Store indirectly via the PCtr	Y	Y

- B1: AP has read and write access to the entire Firmware Store.
- B2: AP can only indirectly read and write to the NV storage by delegating to the PCtr.

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

In B2 the AP must send the firmware images to the PCtr using the Firmware Store update ABI previously defined.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

Chapter C1
ABI implementation in B2

The ABI implementation between the PCtr and the AP requires:

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

Chapter C2
AP and PCtr synchronization in B1 and B2

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

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

The synchronization mechanism requires:

- an event triggered by the AP and delivered to the PCtr, which the PCtr must acknowledge.

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

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

The Staging state terminates:

- if the AP resets;
- if the AP explicitly calls fwu_cancel_staging;
- if the AP explicitly calls fwu_end_staging.

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

If the AP takes too long in the Staging state, the PCtr can send an IMPLEMENTATION DEFINED termination event. The termination event signals to the AP that:

- the PCtr can resume writing to the Firmware Store;
- the AP must cease any direct accesses or that indirect write requests, via the PCtr, will be denied.

Once the PCtr has sent the termination event it can resume writing to NV storage immediately.

Chapter C3
AP boot

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

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

Chapter C4
Example AP to PCtr interaction via PLDM type 5 messaging

For platforms adopting model B2, the AP must communicate with the PCtr. The platform may choose to implement the communication between the AP and the PCtr by means of MCTP messaging over USB or I3C. In the firmware update context the communication between the Update Agent in the AP and the PCtr can be performed using PLDM type 5 messages.

The Client sets the system in the Staging state. After all the firmware images are exchanged between the Client and the Update Agent, the Update Agent crafts a firmware update package, as defined in Section 7 of the PLDM for Firmware Update [6].

Once all the firmware images are either communicated to the firmware devices or written to the Firmware Store the PCtr send a AcceptFirmware message.

The Update Agent in the AP terminates the exchange by invoking ActivateFirmware.

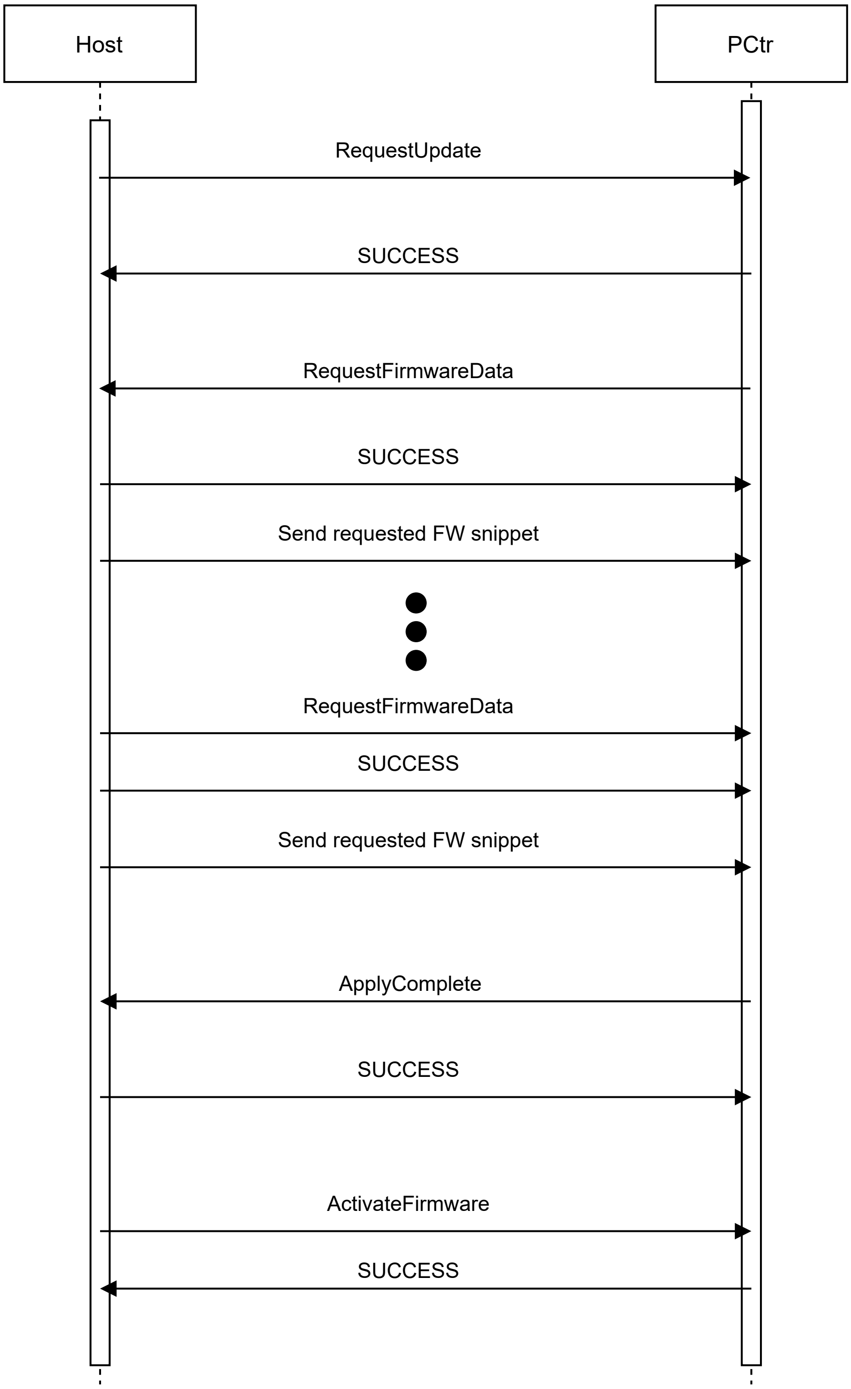

Figure C4.1: Information exchange between the AP and the PCtr during an image update

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

Part D
Update Agent in the Normal World

Some firmware image types can reside in a Firmware Store controlled by the Normal World. For these firmware image types, the Client may execute from within the context that has read/write access to the Firmware Store. In that case, the Client takes the role of the Update Agent and is responsible for writing the firmware images to the Firmware Store.

Chapter D1
State machine

The Firmware Store 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 firmware images.
- Trial to Regular: Once the Client has marked all images in the $bank_{active\_index}$ as accepted in the metadata.

The Firmware Store must not be updated while in the Trial state.

Once the Client initializes, it sets the *update_index* variable respecting the following constraints:

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

After writing each firmware 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.

Once the Client has updated all the firmware images it must set the following state variables:

- *previous_active_index* $\leftarrow$ *active_index*
- *active_index* $\leftarrow$ *update_index*

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

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

Chapter D2
Firmware directory information

The Client must be able to obtain the data otherwise provided by the firmware directory (see 3.3.1 *Image directory*).

The following fields are present in the metadata:

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

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

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

Part E

Security Risk Analysis

Chapter E1
Trust and information flows

The threat model and security properties of the FW update protocol are described in this appendix. Two distinct platform models are considered. The platform models are distinguished by the entity that can write to the mediums where the assets are kept in. The two platform models are the following:

1. The Client execution context does not have direct access to any of the assets.
2. The Client execution context has direct read/write access to some of the assets.

The threats, assets, security goals and assumptions are common to both platform designs. The mitigations are discussed separately for each platform model.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

E1.1 Assets

The following list shows the platform assets:

- A1: FW images
- A2: Rollback counter values
- A3: FWU Metadata
- A4: Chain of trust

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

E1.2 Security goals

The FW update framework must achieve the following goals:

- G1: the Firmware update framework cannot be used to corrupt the boot procedure of the current system:
  - G1.1 FWU metadata cannot be corrupted by the Client;
  - G1.2 Active FW bank cannot be corrupted by the Client.
- G2 FW images may be inaccessible to the Client.
- G3 The trusted boot procedure cannot be sidestepped or subvertible by the Client.

Additionally, for a platforms that must ensure only authentic images are written to the Firmware Store, the following goal applies:

- G4 Firmware images must be authenticated before being committed to flash by the Update Agent.

Note: achieving the goals listed above may be impossible on systems where the Update Agent and Client exist in the same security domain, unless schemes such as flash locking before OS runtime are used.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

E1.3 Model assumptions

The threat model assumes an attacker with the following capabilities:

- Malicious Client (SW execution at EL1/EL2).
- Light physical attacks (e.g. SPI/I2C probe, ability to unplug device from power source).

The threat model below assumes that:

- A trusted boot procedure is implemented, thus preventing images that fail to authenticate from executing on the platform.
- The Update Agent may optionally authenticate the images before writing these to flash. The implementation of the image authentication prior to flash commit is a platform vendor choice.

Logging of installation attempts is currently considered out of scope for this specification.

E1.4 Threats

Threat ID	STRIDE type [7]	Attack type	Attacked asset	Description
T1	Spoofing	SW	FW store	Attacker installs unauthenticated FW images to gain execution capability in the system.
T2	Tampering	SW	FW store	Attacker downgrades a FW image to exploit a vulnerability of a previous FW version.
T3	Tampering	SW	Anti-rollback counter	Attacker decrements an anti-rollback counter to enable a revoked FW image to execute on the platform.
T4	Tampering	SW	Chain of trust	Attacker alters CoT to enable unauthenticated images to execute.
T5	Tampering	Physical	FWU metadata, FW store	Attacker resets platform during firmware update to leave system in an inconsistent/exploitable state.
T6	Information disclosure	SW	FW store	Attacker without permissions accesses data in FW store.
T7.1	Denial of service	SW	FW store	Attacker overwrites the FW store to prevent parts of the system from becoming online.
T7.2	Denial of service	Physical	FW store	Attacker overwrites the FW store to prevent parts of the system from becoming online.
T8	Denial of service	SW	FWU metadata	Attacker alters the FWU metadata to prevent anti-rollback counter increments.
T9.1	Elevation of privilege	SW	FWU metadata	Attacker swaps image GUIDs in the FWU metadata to enable an authenticated image to execute at higher exception level.
T9.2	Elevation of privilege	Physical	FWU metadata	Attacker swaps image GUIDs in the FWU metadata to enable an authenticated image to execute at higher exception level.

E1.5 Platform models

E1.6 NV memory controlled by the Secure World

The platform trust relationships are shown in Figure E1.1. The Client trusts and sends information to the Update Agent. The Update Agent sends information to the Client, via the FW directory file. The Update Agent distrusts the Client.

Both the Update Agent and the Client trust the *immutable* and *secondary* bootloader stages.

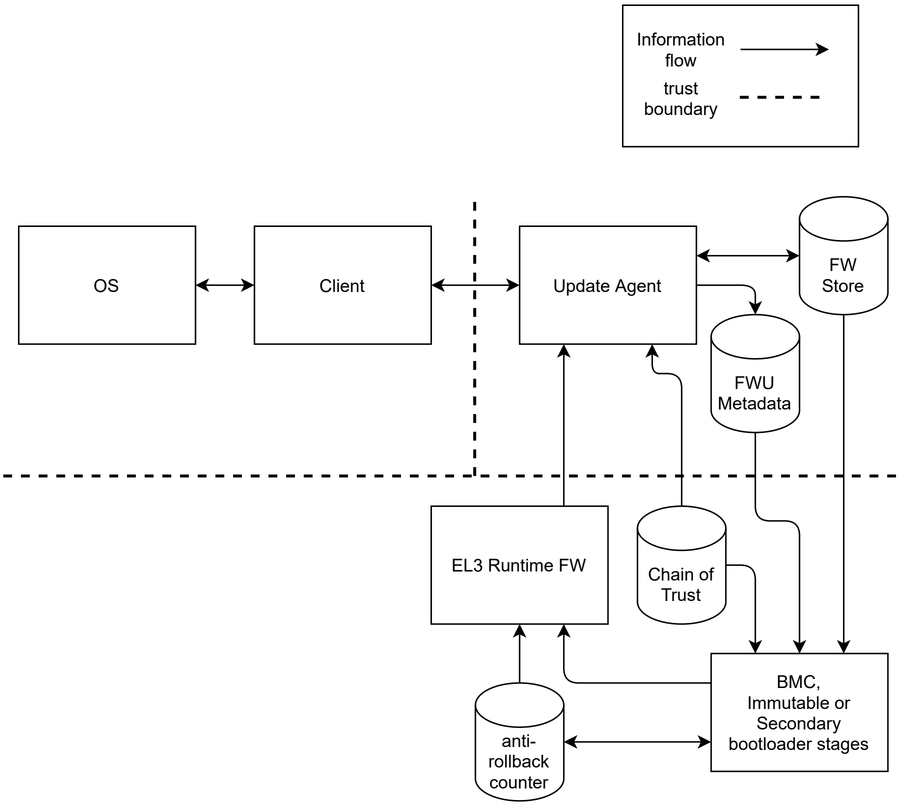

Figure E1.1: System diagram of the Secure World controlled NV storage model

E1.6.1 Mitigations

Threat ID	Mitigation	Notes
T1	FW images can be authenticated before being written to flash. Otherwise the image will fail the authentication check in the mandatory trusted boot procedure.	–

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

Threat ID	Mitigation	Notes
T2	The platform owner must have updated the anti-rollback counter value to prevent a vulnerable image from executing on the platform. The Update Agent detects the FW image downgrade attempt before writing to flash. Otherwise the image will fail the authentication check in the mandatory trusted boot procedure.	–
T3	The mechanism to change the anti-rollback counter is IMPLEMENTATION DEFINED. The anti-rollback counter is controlled by the Immutable/ Secondary or EL3 runtime FW and thus is the responsibility of these implementations to safeguard the correct anti-rollback counter behaviour.	**transferred:** the mitigation is the responsibility of the Immutable/Secondary and Secure World runtime FW implementations.
T4	The CoT is not be writable to using the FWU ABI. The CoT is placed in a NV memory location inaccessible to the Client.	–
T5	The trusted boot procedure ensures only authenticated images execute on the platform. The rules around FWU metadata update ensure that inconsistent FWU metadata are corrected before usage.	A glitch in the FW update will lead to 1) the FWU metadata getting corrupted or 2) some of the installed FW images getting partially updated. 1) The FWU metadata is replicated and hence there will be an intact FWU metadata that can be used to correct the corrupted one. If the power failure occurred between the correct update of one of the replicas but before the update of the other replica, then it is IMPLEMENTATION DEFINED which replica is used to overwrite the other. 2) The trusted boot procedure ensure that all images are authenticated prior to execution. A partially updated image will lead to an authentication failure with a very high likelihood. A failure to boot with the updated bank leads to a boot from the previous active which is not affected by the power failure during the update process.
T6	The Update Agent validates the Client permission to access the image before outputting any information to the Client. If the attacker has physical access, then the attacker is able to read the content of the flash. Security sensitive data must be encrypted to mitigate this attack.	–

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

Threat ID	Mitigation	Notes
T7.1	The Update Agent validates the Client permission to write to the image. The Update Agent authenticates the images before these are written to flash. If the image cannot be authenticated, then it still holds true that the Client can only update a single bank. If the platform has multiple banks, the current active bank will always be intact and available as a fallback.	The complete updated "certificate CoT" may not be available at the time of image write. In that case the image cannot be authenticated.
T7.2	**Unmitigated**	An attacker with physical access will be able to alter the FW store.
T8	The FWU metadata is outside of the Client trust boundary, only accessible to the Update Agent. The FWU ABI definition does not allow the Client to alter the FWU metadata.	–
T9.1	The FWU metadata is outside of the Client trust boundary, only accessible to the Update Agent. The FWU ABI definition does not allow the Client to alter the FWU metadata.	–
T9.2	The trusted boot procedure and the intact CoT will detect the image GUID swap and lead to a failed boot.	T9.2 mitigation leads unmitigated DoS.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

E1.7 NV memory controlled by the Normal World

The platform trust relationships are shown in Figure E1.2.

In this platform model the FW store and the FWU metadata are kept in a NV memory that the can be written to from the Normal World. The Chain of Trust can also be partially placed in the same medium as the FW store.

The Update Agent and the Client are in the same trust domain. The Client and the Update Agent trust each other.

Both the Update Agent and the Client trust the *immutable* and *secondary* bootloader stages.

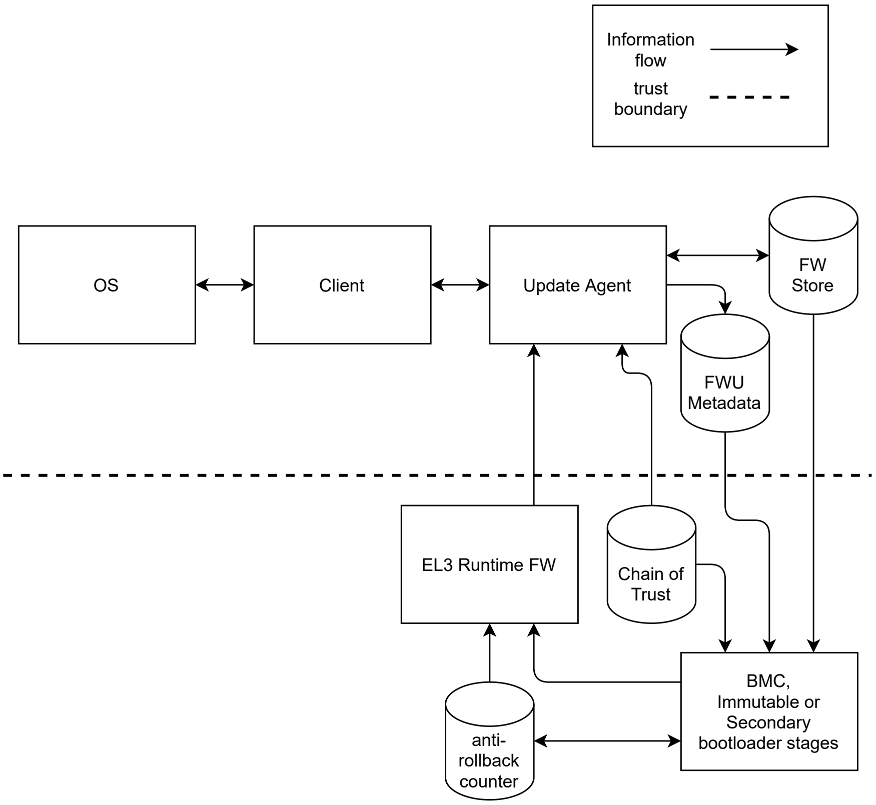

Figure E1.2: System diagram of the Normal World controlled NV storage model

Threat ID	Mitigation	Notes
T1	Trusted boot detects unauthenticated image, boot fails.	T1 mitigation does not mitigate against DoS attack
T2	The platform owner must have updated the anti-rollback counter value to prevent a vulnerable image from executing on the platform. Trusted boot detects the anti-rollback condition violation	T2 mitigation does not mitigate against DoS attack

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

Threat ID	Mitigation	Notes
T3	The mechanism to change the anti-rollback counter is IMPLEMENTATION DEFINED. The anti-rollback counter is controlled by the Immutable/Secondary or EL3 runtime FW and thus is the responsibility of these implementations to safeguard the correct anti-rollback counter behaviour.	**transferred:** mitigation is the responsibility of the Immutable/Secondary and EL3 runtime FW implementations
T4	CoT is authenticated must be authenticated by the Immutable/Secondary stage using the platform root of trust.	**transferred:** mitigation is the responsibility of the Immutable/Secondary implementations.
T5	The trusted boot procedure ensures only authenticated images execute on the platform. The rules around FWU metadata update ensure that inconsistent FWU metadata are corrected before usage.	A glitch in the FW update will lead to 1) the FWU metadata getting corrupted or 2) some of the installed FW images getting partially updated. 1) The FWU metadata is replicated and hence there will be an intact FWU metadata that can be used to correct the corrupted one. If the power failure occurred between the correct update of one of the replicas but before the update of the other replica, then it is IMPLEMENTATION DEFINED which replica is used to overwrite the other. 2) The trusted boot procedure ensure that all images are authenticated prior to execution. A partially updated image will lead to an authentication failure with a very high likelihood. A failure to boot with the updated bank leads to a boot from the previous active which is not affected by the power failure during the update process.
T6	Sensitive FW images can be installed and stored in an encrypted form to preserve confidentiality.	**transferred:** Security sensitive images must be encrypted when in the FW store.
T7.1	**Unmitigated**	The FW store is in the same trust domain as the Client.
T7.2	**Unmitigated**	An attacker with physical access will be able to alter the FW store.
T8	**Unmitigated**	The FWU metadata is in the same trust domain as the Client.
T9.1	The trusted boot procedure and the intact CoT will detect the image GUID swap and lead to a failed boot.	T9.1 mitigation leads unmitigated DoS.
T9.2	The trusted boot procedure and the intact CoT will detect the image GUID swap and lead to a failed boot.	T9.2 mitigation leads unmitigated DoS.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential

Part F

UEFI end-to-end example design

A particular platform may chose to implement the FW update client in Normal World FW which implements the UEFI interfaces.

The FW update Client takes shape both in the FMP (or multiple FMPs) and UpdateCapsule implementations withing UEFI.

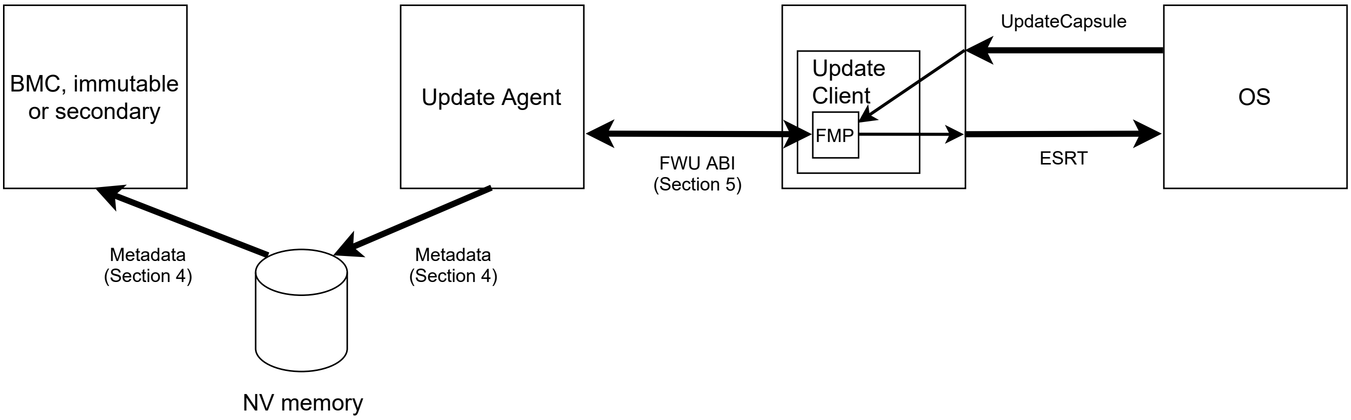

Figure 1: End-to-end UEFI example design

The following table lists the different interfaces in the system and the agents that interact over the interface.

Interface	Provider	Consumer	Description
FWU ABI	Update Agent	Update Client (FMP + UpdateCapsule)	Set of primitives used to transmit FW images from the Client to the Update Agent in Secure World.
FWU Metadata	Update Agent	BMC, Immutable or Secondary bootloader stages	The datastructure describing the FW bank structure, the acceptance status of each image and the active bank selection. The BMC, Immutable or Secondary stage bootloaders use the FWU metadata information to determine which FW images must be loaded from the FW Store.
Update Capsule	UEFI implementation	OS	The interface over which the OS passes a FMP Capsule containing the FW to be updated.
ESRT	UEFI Implementation	OS	A UEFI configuration table used to present the different image types and their versions present in the system. The table is created before ExitBootServices and retrieved by the OS loader.

A FW update involving the UEFI implementation involves the following steps:

1. The OS communicates the set of images to be updated by passing an FMP formatted capsule via the UpdateCapsule interface.
2. The UEFI implementation receives the capsule, passing the different images to the responsible FMP. The FMP communicates the FW images to the Update Agent.

3. The UpdateCapsule implementation tracks all the images being installed by the responsible FMPs. After all the images in the Capsule are installed, the UpdateCapsule implementation signals the completion of the Staging state.
    1. If the Capsule contained a partial set of images, the Update Agent should copy the non-updated images from the active bank, or keep the image in the update bank untouched if it has the same version or same digest as the version in the active bank.
    2. At this point the full CoT is always available, the Update Agent can perform an optional authentication of all images just updated. The Update Agent returns an error code if the authentication fails.
    3. The Update Agent performs the required changes in the Metadata.
4. The UpdateCapsule implementation returns confirming correct termination of the Staging state.
5. The OS resets the platform.
6. The platform boots. While there are unaccepted images in the active bank, the platform is in the Trial state.
    1. The UEFI implementation constructs the ESRT indicating the acceptance status of each FW image.

Copyright © 2022, 2023 Arm Limited or its affiliates. All rights reserved.
Non-confidential